Korešpondenčný seminár z programovania
XXXIV. ročník, 2016/17
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola zimnej časti

Mário

1. Zuzkino sudoku

(max. 6 b za popis, 4 b za program)

Vzhľadom na to, že každý vstup obsahoval iba 9×9 čísel (čiže každý vstup bol malý), mohli ste v tejto úlohe robiť všeličo. Mohli ste rekurziou vyriešiť sudoku, mohli ste pre každý riadok zisťovať, čo v ňom chýba...

Na vyriešenie tejto úlohy však nebolo nič z toho potrebné a stačilo spraviť jediné pozorovanie: **v každom vyriešenom sudoku sa nachádza každé číslo (1-9) práve deväťkrát**. Raz v každom stĺpci a stĺpcov je 9. Raz v každom riadku a riadkov je 9. Raz v každom bloku a blokov je 9.

Preto nám bude stačiť prečítať vstup, spočítať, koľko je v zadaní jednotiek, dvojok, ..., deviatok a vypísať čísla $9 - pocet_jednotiek, 9 - pocet_dvojok, \ldots, 9 - pocet_deviatok$.

Na pamätanie počtov jednotlivých čísel môžeme použiť 9 premenných, ale krajšie riešenie je použiť pole `pocet_cisel[]`, ktoré si inicializujeme na samé nuly. Potom načítame zo vstupu 81 čísel a zakaždým zvýšime dané políčko poľa.

Listing programu (C++)

```
#include <cstdio>

int main(){
    int pocet_cisel[10] = {0};

    for(int i = 0; i < 9 * 9; i++){
        int c;
        scanf(" %d", &c);
        pocet_cisel[c]++;
    }

    for(int i = 1; i <= 9; i++)
        printf("%d\n", 9 - pocet_cisel[i]);

    return 0;
}
```

Veľmi podobným spôsobom si môžeme v poli pamätať, koľko čísel i treba doplniť. Pole inicializujeme na samé deviatky a pri čítaní vstupu odčítame jednotku za každý výskyt čísla.

Listing programu (Python)

```
treba_doplnit = [9] * 10

for i in range(9):
    cisla_v_riadku = map(int, input().split())
    for cislo in cisla_v_riadku:
        treba_doplnit[cislo] -= 1

print("\n".join(map(str, treba_doplnit[1:])))
```

Časová zložitosť

Hoci vstup je konštantnej (vždy rovnakej) veľkosti, a mohli by sme povedať, že program pracuje v konštantnom čase $O(1)$[1], tak informatívnejšie by bolo povedať niečo takéto: Ak si dĺžku štvorca sudoku označíme ako n, potom náš algoritmus pracuje v čase $O(n^2)$.

Z takéhoto tvrdenia sa dá usúdiť, že rovnakým algoritmom by ste vedeli vyriešiť úlohu, v ktorej by bolo sudoku veľkosti 18×18, no trvalo by to 4-krát dlhšie.

Všimnite si rozdiel medzi pojmami *program* a *algoritmus*. Hoci vyššie uvedené programy by vstupy veľkostí 18×18 nevyriešili, rovnaká myšlienka, postup – algoritmus – sa dá použiť aj na riešenie vstupov rôznych veľkostí. Preto má zmysel hovoriť o časovej zložitosti algoritmu aj v tejto úlohe.

[1] program beží rovnako dlho pre všetky vstupy 9×9 a pre iné vstupy nebude pracovať správne

Pamäťová zložitosť

V našich riešeniach využívame len jedno pole veľkosti 10 (v riešení, ktoré je napísané v Pythone, si ešte pamätáme jeden riadok vstupu) a niekoľko jednoduchých premenných. Mohli by sme teda, takisto ako v časti o časovej zložitosti, povedať, že pamäťová zložitosť je konštantná od veľkosti vstupu, $O(1)$. Informatívnejšie by ale bolo uviesť pamäťovú zložitosť lineárnu od dĺžky strany sudoku, $O(n)$.

MikX

2. Zabudnutá chata

(max. 6 b za popis, 4 b za program)

Predstavme si, že **vieme** po akej sume sa budú vedúci skladať. Pomôže nám to? Na chatu chcú ísť všetci vedúci, ktorí môžu, a teda ak sa skladajú po 5 eurách, pôjdu všetci tí, ktorí majú aspoň 5 eur, ak po 7-mich, tak tí, čo majú aspoň 7.

Nech teda každý, kto môže, zaplatí sumu $S = 5$ eur. Ak postupne prejdeme limity všetkých vedúcich, vieme spočítať, koľko vedúcich na to má. Ostáva nám skontrolovať, či takto dokopy zaplatia za celú chatu. Ak hej, tak máme výsledok, ak nie, tak na chatu nepôjde nikto.

My ale túto sumu S nepoznáme. Nevadí, vieme **vyskúšať všetky** prípustné hodnoty S a pre každú z nich zistiť, koľko vedúcich tak môže ísť na chatu. Stačí nám priebežne si pamätať a upravovať najväčší počet zúčastnených vedúcich, a na konci toto číslo vypísať.

Posledná vec, ktorú musíme domyslieť, je to, aké hodnoty S máme skúšať:

- $0 \leq S \leq c$ – nedáva zmysel, aby pani domáca dostala od každého vedúceho viac ako je celková cena chaty
- $0 \leq S \leq max(s_i)$ – nemôžeme od chudákov vedúcich chcieť, aby každý zaplatil viac ako má ten *najbohatší* z nich

Listing programu (Python)

```
n, c = [int(x) for x in input().split()]
limity  = [int(x) for x in input().split()]

max_pocet_veducich = 0

for kolko_plati_kazdy in range(0, min(c, max(limity))+1):
    pocet_veducich = 0

    for limit_veduceho in limity:
        if kolko_plati_kazdy <= limit_veduceho:
            pocet_veducich += 1

    if pocet_veducich * kolko_plati_kazdy >= c:
        max_pocet_veducich = max(max_pocet_veducich, pocet_veducich)

print(max_pocet_veducich)
```

Keďže sa pre každú hodnotu S pozrieme na finančný limit každého z n vedúcich, je časová zložitosť tohto riešenia $O(min(c, max(s_i)) \cdot n)$.

Pamätať si stačí finančné limity vedúcich spolu so zopár premennými – pamäťová zložitosť je preto $O(n)$.

Hodnoty c aj s_i môžu byť však **príliš veľké** (až 10^9) a preto musíme vymyslieť algoritmus, ktorý od nich buď nebude závisieť, alebo bude závisieť menej ako lineárne.

Skúšajme menej možností

V predošlom riešení sme robili veľa vecí navyše. Keď overujeme sumu S, tak počet vedúcich, ktorí ju vedia zaplatiť sa predsa zmení až vtedy, keď S nadobudne hodnotu finančného limitu nejakého ďalšieho vedúceho. **Stačí nám teda postupne prekontrolovať len** $S = s_i$, kde s_i označuje finančný limit i-teho vedúceho.

Toto riešenie sa pre každý limit pozrie na všetkých vedúcich a teda sme prišli na riešenie s časovou zložitosťou $O(n^2)$. Takto sa na vstupoch dali získať 2 body.

Takéto riešenie dostaneme, ak jednoducho prepíšeme v predošlom programe riadok s for-cyklom na
`for kolko_plati_kazdy in limity:`

Skúšajme ich v správnom poradí

Skúsme sa pozrieť na druhú fázu nášho algoritmu (pre dané S chceme rýchlo zistiť, koľko vedúcich má $s_i \geq S$). Predstavme si, že by limity boli **usporiadané podľa veľkosti**:

$$s_0 \leq s_1 \leq \cdots \leq s_{n-1}$$

Potom vieme predsa pre $S = s_i$ **hneď** povedať, koľko vedúcich má limit aspoň S – je ich presne `n-i`. To ale znamená, že nemusíme už nič overovať. Vyhrali sme, pretože triediť vieme v čase $O(n \log n)$. Použitie takéhoto

triedenia nám dokopy dá riešenie v celkovej časovej zložitosti $O(n \log n)$, ktoré na testovači získa plný počet bodov.

Pamäťová zložitosť tohto riešenia je $O(n)$ – potrebujeme si pamätať finančný limit každého vedúceho, bez toho by sme nevedeli triediť.

Listing programu (C++)

```
#include <vector>
#include <cstdio>
#include <algorithm>

using namespace std;

vector<long long> s;

int main() {

  long long n, c;

  scanf("%lld %lld", &n, &c);

  s.resize(n);
  for (int i=0; i<n; ++i)
    scanf("%lld", &s[i]);

  sort(s.begin(), s.end());

  for(int i=0; i<n; ++i) {

    // ak tento veduci nema limit, ze by sa podla neho zvladla zaplatit chata, nezaujima nas
    if ( (s[i]) * (n - i) < c)
      continue;

    // najvacsia odpoved, pretoze n-i bude uz len mensie
    printf("%lld\n", (n-i));
    return 0;
  }

  printf("0\n"); // nenasli sme riesenie
  return 0;

}
```

Listing programu (Python)

```
n, c = [int(x) for x in input().split()]
limity  = [int(x) for x in input().split()]

max_pocet_veducich = 0

# metoda sorted s reversed=True nam utriedi limity od najvacsieho
# metoda enumerate nam zo zoznamu limitov vytvori zoznam dvojic
# [(1, limit najbohatsieho), (2, limit druheho najbohastieho), ... (n, limit najchudobnejsieho)]
for pocet_veducich, najchudobnejsi in enumerate(sorted(limity, reverse=True), start=1):
    if pocet_veducich * najchudobnejsi >= c:
        max_pocet_veducich = max(max_pocet_veducich, pocet_veducich)

print(max_pocet_veducich)
```

BONUS: Riešenie s lineárnou časovou zložitosťou

- Pre každého vedúceho i si vyrátajme L_i – minimálny počet vedúcich, ktorí musia na chatu ísť, aby tam mohol ísť aj i-ty vedúci. Inak povedané, pre $S = s_i$[2] vyjadruje L_i minimálny počet vedúcich, ktorí sa po tejto sume musia skladať, aby zaplatili chatu[3]:

$$L_i = \left\lceil \frac{c}{s_i} \right\rceil$$

- Vytvorme si pole X, kde X_i bude reprezentovať počet vedúcich s $L = i$. Inak, X_i hovorí o počte vedúcich, ktorí vyžadujú i vedúcich, aby išli na chatu. Toto vieme vytvoriť už za rátania hodnôt L, ak vyrátame i-temu vedúcemu L_i, zvýšime hodnotu X_{L_i} o jedna.

- Na tomto poli (X) si spočítame prefixové sumy. Hodnota X_i nám teraz bude hovoriť o počte vedúcich, ktorí vedia na chatu ísť, ak pôjde **aspoň** i vedúcich (rozmyslite si).

- Teraz nám už len stačí toto pole prejsť a ak $i \leq X_i$, tak X_i vedúcich vie na chatu ísť. Nájdeme maximum.

[2] Keď chceme minimálny počet vedúcich, každý musí platiť najviac, ako sa dá, a to bude práve s_i – limit i-teho vedúceho

[3] preto je vo vzorci použitá horná celá časť zlomku, čo predstavuje najmenšie celé číslo, ktoré je väčšie alebo rovné ako zlomok (totiž keď zlomok povie, že treba 3.4 vedúceho, znamená to, že traja chatu nezaplatia, ale štyria už áno)

Časová aj pamäťová zložitosť je už spomínané $O(n)$, keďže len prechádzame pole a robíme na ňom aritmetické operácie.

Pozn.: Toto riešenie sme od vás nevyžadovali. Ak ste ho však vymysleli, môžete očakávať **bonusové body** za popis.

Listing programu (C++)

```
#include <vector>
#include <cstdio>

using namespace std;

vector<long long> s;
vector<long long> L;
vector<int> X;

int main() {

  long long n, c;

  scanf("%lld %lld", &n, &c);

  // okrajovy pripad, ked netreba platit, idu vsetci
  if (c == 0LL) {
    printf("%lld\n", n);
    return 0;
  }

  s.resize(n);
  for (int i=0; i<n; ++i)
    scanf("%lld", &s[i]);

  // ratame hodnoty L[i] a vytvarame X
  L.resize(n); X.resize(n+1, 0);
  for(int i=0; i<n; ++i) {

    // takyto veduci sa na chatu nedostane ...
    if (s[i] == 0LL)
      continue;

    // L[i] = horna cela cast (c / s[i])
    L[i] = (c / s[i]) + (c % s[i] > 0LL);

    // nezaujimaju nas hodnoty, ked niekto potrebuje viac veducich ako je k dispozicii
    if (L[i] <= n)
      ++X[ L[i] ];
  }

  // prefixove sumy
  for(int i=1; i<=n; ++i)
    X[i] += X[i-1];

  // a uz to len prejdeme a najdeme najvacsie
  int ret = 0;
  for(int i=0; i<=n; ++i)
    if (i <= X[i])
      ret = max(ret, X[i]);

  printf("%d\n", ret);
  return 0;

}
```

3. Zajačí problém

Denis a Baklažán
(max. 6 b za popis, 4 b za program)

Táto úloha sa dala riešiť rôznymi, viac, či menej zložitými spôsobmi (ak ste to chceli potiahnuť do extrému, mohli ste použiť trebárs topologické triedenie). Vzorové riešenie je však až zarážajúco jednoduché.

Vzorové riešenie

Máme čísla $1, 2, \ldots, n$ a potrebujeme ich zoradiť tak, aby spĺňali nerovnosti zo zadania. Pozrime sa na prvý znak vstupu. Môžu nastať dve možnosti:

- Ak je prvý znak vstupu $<$, znamená to, že prvé číslo postupnosti musí byť menšie ako druhé.

  V tomto prípade za prvé číslo môžeme vziať najmenšie z čísel, ktoré máme k dispozícii (teda číslo 1). To nám zaručí, že nech akokoľvek usporiadame zvyšné čísla, táto nerovnosť bude určite platiť.

- Ak je prvý znak $>$, prvé číslo musí byť väčšie ako druhé. V takom prípade môžeme za prvé číslo zobrať najväčšie možné číslo (teda n). Opäť budeme mať zaručené, že táto nerovnosť bude platiť bez ohľadu na to, ako usporiadame ostatné čísla.

V oboch prípadoch teda vieme určiť prvé číslo tak, aby nerovnosť určená prvým znakom vstupu zaručene platila. Ak sa nám potom podarí usporiadať zvyšné čísla tak, aby boli splnené zvyšné nerovnosti, máme vyhrané – všetky nerovnosti už budú platiť.

Ako zoradiť zvyšných $n-1$ čísel tak, aby spĺňali zvyšné nerovnosti? Môžeme použiť tú istú myšlienku. Tento raz sa pozrieme na druhý znak vstupu. Ak je to $<$, potom môžeme za druhé číslo postupnosti zvoliť najmenšie z $n-1$ čísel, ktoré nám zostali a aj táto nerovnosť bude zaručene splnená (bez ohľadu na to, ako zoradíme zvyšných $n-2$ čísel). Ak bude druhý znak vstupu $>$, potom za druhé číslo postupnosti vezmeme najväčšie možné číslo.

Takto budeme pokračovať, až kým neprečítame celý vstup. Keď spracujeme všetkých $n-1$ znakov vstupu, budeme mať určených prvých $n-1$ čísel postupnosti. Za posledné číslo postupnosti vezmeme to jediné, čo nám zostane. Vzniknutá postupnosť bude určite spĺňať všetky nerovnosti na vstupe – prvú nerovnosť bude spĺňať vďaka tomu, ako sme zvolili prvé číslo postupnosti, druhú vďaka spôsobu voľby druhého čísla postupnosti, tretiu vďaka voľbe tretieho čísla atď.

Implementácia

Otázkou zostáva, ako tento algoritmus dobre implementovať. Asi najťažšou otázkou je, ako si pamätať zoznam doteraz nepoužitých čísel. Mohli by sme si ich pamätať v nejakom poli, ale ide to aj jednoduchšie. Stačí si uvedomiť, že zoznam nepoužitých čísel je v každom okamihu nášho algoritmu nejaká súvislá postupnosť prirodzených čísel: na začiatku je to $1, 2, \dots, n$ a vždy keď z neho niečo vyhadzujeme, tak je to buď najväčšie, alebo najmenšie číslo (teda aj po vyhodení zostane súvislý). Preto si stačí v dvoch premenných pamätať najmenšie a najväčšie číslo v zozname.

Výslednú postupnosť by sme si mohli pamätať v poli. To ale tiež nie je nutné, keďže ju môžeme počas konštruovania rovno vypisovať.

Listing programu (C++)

```
#include <string>
#include <iostream>

using namespace std;

int main(){

    string vstup;
    cin >> vstup;

    int minimum = 1, maximum = (int)vstup.size() + 1;

    for (int i = 0; i < (int)vstup.size(); i++)
    {
        if(vstup[i]=='<')
            cout << minimum++ << " ";
            // vyraz 'premenna++' najprv vrati hodnotu premennej
            // (tato hodnota sa teda vypise) a potom ju o 1 zvysi
        else
            cout << maximum-- << " ";
    }

    cout << minimum++ << "\n";


    return 0;
}
```

Zložitosť

Časová zložitosť algoritmu je $O(n)$ – lineárna od počtu znakov na vstupe. Pamäťová zložitosť tohto algoritmu je tiež $O(n)$, keďže je potrebné načítať vstup do poľa. Zvyšok operácií využíva už iba premenné *minimum* a *maximum*. Všimnime si, že jediný dôvod, prečo potrebujeme čítať celý vstup už na začiatku je, že potrebujeme vedieť aký je dlhý (aby sme vedeli, aké najväčšie číslo má byť vo výslednej permutácii). Keby sme na začiatku na vstupe dostali číslo n a až potom postupnosť znakov $<$ a $>$, úloha by sa dala riešiť v konštantnej pamäti – vstup by sme čítali po znakoch a vždy by sme si pamätali len znak, ktorý práve spracúvame.

Baška

4. Zabudnuté algoritmy

(max. 9 b za popis, 6 b za program)

Jednoduché riešenie

Skúsme najprv vymyslieť priamočiare riešenie. Budeme si pre každú úlohu pamätať, ktoré algoritmy na ňu potrebujeme vedieť a na akej úrovni. Vždy, keď sa Hodobox niečo naučí, prejdeme všetky úlohy a spočítame

tie, pre ktoré už vie všetky algoritmy dosť dobre.

Pre jednoduchosť si pri každom algoritme pre každú úlohu okrem toho, na akú úroveň ho potrebujeme vedieť, zapamätáme aj to, ako dobre ho vieme – **znalosť algoritmu**. Na začiatku je to nula pre každý algoritmus každej úlohy. Postupne, ako sa bude Hodobox algoritmy učiť, budeme tieto čísla zvyšovať.

Po načítaní zoznamov algoritmov pre jednotlivé úlohy začneme postupne spracovávať Hodoboxove akcie. Pri každej akcii prejdeme každú z n úloh a spočítame, koľko z nich vie vyriešiť. To vieme pre jednu úlohu spočítať jednoducho: Postupne sa pozrieme na všetky potrebné algoritmy, a overíme, či ich už Hodobox vie na dostatočne dobrej úrovni. Ak nájdeme algoritmus, ktorý má potrebnú úroveň ostro vyššiu ako znalosť, tak sa ho Hodobox ešte musí učiť.

Popri tomto prechádzaní vo všetkých úlohách aj zvyšujeme znalosť algoritmu, ktorý sa Hodobox v tejto akcii naučil.

Akú má toto riešenie časovú a pamäťovú zložitosť?

Časová zložitosť pre načítanie vstupu závisí od jednotlivých počtov algoritmov. Pre úlohu i načítanie trvá $O(1 + a_i)$. Jednotka je tu pre načítanie čísla a_i. To je podstatné hlavne ak $a_i = 0$. Na celkové načítanie vstupu potrebujeme načítať všetky úlohy, a tak čas potrebný na načítanie celého vstupu bude súčet časov pre jednotlivé úlohy: $\sum_{i=1}^{n}(1 + a_i) = \sum_{i=1}^{n} 1 + \sum_{i=1}^{n} a_i = n + \sum_{i=1}^{n} a_i$. Ak si označíme $\sum a_i = A$, tak načítanie vstupu má časovú zložitosť $O(n + A)$.

Koľko trvá spracovanie jednej Hodoboxovej akcie? Prechádzame každú úlohu, a pre ňu každý algoritmus, čo je znova súčet počtov algoritmov pre všetky úlohy. Teda aj spracovanie jednej akcie trvá $O(n + A)$. Spracovanie všetkých akcií bude trvať $O(p(n + A))$.

Celkovo je časová zložitosť $O(n + A + p(n + A)) = O(p(n + A))$.

Pamäťová zložitosť je $O(n + A)$, keďže si musíme zapamätať všetky úlohy a algoritmy, ktoré sú pre ne potrebné.

Listing programu (C++)

```
#include<cstdio>
#include<vector>

using namespace std;

struct algoritmus {
    int cislo;
    // Na koľko ho Hodobox musí vedieť.
    int potrebna_uroven;
    // Na koľko ho Hodobox vie.
    int znalost;
};

int main() {
    int n;
    scanf(" %d", &n);
    // Tu si budem pre každú úlohu pamätať algoritmy, ktoré na ňu potrebujem vedieť.
    vector< vector< algoritmus > > ulohy;

    for (int i = 0; i<n; ++i) {
        int a;
        scanf(" %d", &a);
        vector< algoritmus > algoritmy(a);
        for (int j = 0; j<a; ++j) {
            int x,y;
            scanf(" %d %d", &x, &y);
            algoritmy[j].cislo = x;
            algoritmy[j].potrebna_uroven = y;
            // Na začiatku vie Hodobox všetky algoritmy na nula.
            algoritmy[j].znalost = 0;
        }
        ulohy.push_back(algoritmy);
    }

    int p;
    scanf(" %d", &p);
    for (int i = 0; i<p; ++i) {
        int c, u;
        scanf(" %d %d", &c, &u);

        // Pre každú úlohu sa pozrieme, či už vie všetky potrebné algoritmy dostatočne dobre.
        int pocet_uloh = 0;
        for (int j = 0; j<n; ++j) {
            // Predpokladáme že ju vie vyriešiť.
            bool viem = true;

            // Prejdeme cez všetky algoritmy potrebné na túto úlohu.
            for (unsigned k = 0;  k < ulohy[j].size(); k++) {

                // Ak je toto algortimus, ktorý sa práve Hodobox naučil o
                // 'u' lepšie, tak si pripočítame, že ho už vie o 'u' lepšie.
                if (ulohy[j][k].cislo == c) {
                    ulohy[j][k].znalost += u;
```

```
            }

            // Zistíme, či už má Hodobox dostatočnú znalosť algoritmu 'k' na úlohu 'j'.
            if (ulohy[j][k].znalost < ulohy[j][k].potrebna_uroven) {
                viem = false;
            }
        }
        // Ak vie všetky algoritmy dostatočne, tak zvýšime počet úloh.
        if (viem) pocet_uloh++;
    }
    printf("%d\n", pocet_uloh);
  }
  return 0;
}
```

Skoro vzorové riešenie

Predošlé riešenie funguje, ale pozrime sa, či by vyriešilo úlohu aj pre najväčšie vstupy. Tam môžu premenné p, n aj A nadobúdať hodnoty až po 10^6. Keďže časová zložitosť je $O(p(n+A))$, bolo by to rádovo až $10^6 \cdot (10^6 + 10^6)) = 2 \cdot 10^{12}$ operácií, no bežný počítač zvláda za sekundu približne "len" 10^9 operácií. Pokúsme sa teda navrhnúť efektívnejší algoritmus.

Pozrime sa na to, čo sme počítali v predošlom riešení zbytočne veľakrát.

Prvou takou vecou bolo, že sme po každej akcii prešli všetky úlohy, a spočítali tie, ktoré vie Hodobox vyriešiť. Po jednej akcii sa ale výsledok (počet riešiteľných úloh) nezmení o veľa. Mohli by sme mať teda jedno **počítadlo: počet riešiteľných úloh**. V každej akcii by sme toto počítadlo len zvýšili o úlohy, ktoré sa stali riešiteľnými naučením tohoto algoritmu a potom by sme hodnotu tohto počítadla vypísali ako výsledok. Zavedenie počítadla nám samo osebe ešte nezlepší zložitosť, ale je to prvým krokom k lepšiemu riešeniu.

Druhá vec, ktorú sme robili pri každej akcii, bola, že sme prešli všetky algoritmy každej úlohy. V jednej akcii sa ale Hodobox naučil len jeden algoritmus, ktorý bol potrebný len pre pár úloh. Čo ak by sme si **pre každý algoritmus pamätali všetky úlohy, ktoré ho potrebujú**? Potom by sme pri spracovávaní Hodoboxovej akcie prešli naozaj iba tie úlohy, pri ktorých sa niečo zmenilo.

Zoznamy úloh pre každý algoritmus si budeme pamätať v dvojrozmernom poli dvojíc[4]. i-ty prvok tohto poľa bude jednorozmerné pole takýchto dvojíc: (úroveň algoritmu i (potrebná na vyriešenie úlohy), číslo úlohy).

Keďže každý prvok nášho poľa (zoznam) patrí jednému algoritmu a my vieme, že najväčšie číslo algoritmu je 10^6, tak veľkosť poľa si môžme zvoliť $10^6 + 1$.[5]

V inom riešení by sme si počas načítavania vstupu mohli pole naťahovať. V premennej max_alg by sme si udržiavali najväčšie nájdené číslo algoritmu a vždy, keď by sme našli algoritmus s číslom x väčším než max_alg, tak by sme naše pole rozšírili na dĺžku $x + 1$.

Pre každý algoritmus i máme teda zoznam úloh, pre ktoré potrebujeme algoritmus i vedieť lepšie. Mohli by sme teda povedať, že je to **zoznam úloh, ktoré sú blokované algoritmom** i. Každá úloha je na začiatku blokovaná všetkými svojimi algoritmami. Keď si Hodobox zlepší znalosť algoritmu i, mohli by sme tento zoznam prejsť a odstrániť tie úlohy, pre ktoré už vie Hodobox algortimus dostatočne dobre.[6]

Na vstupe vždy dostaneme len číslo u, o ktoré sa Hodoboxova znalosť algoritmu i zvýšila. Na to, aby sme vedeli povedať, aká je nová úroveň znalosti algoritmu (v predošlej poznámke sme ju označovali y) si **pre každý algoritmus chceme pamätať jeho aktuálnu úroveň** v poli. Na začiatku vie Hodobox každý algoritmus na nula a pri každej akcii upravíme znalosť jedného algoritmu (`y += u`). Toto pole pre úrovne môžeme spraviť veľkosti $10^6 + 1$, alebo veľkosti $max_alg + 1$.

Poslednou otázkou zostáva: Kedy vie Hodobox vyriešiť novú úlohu? Vtedy, ak už úloha nie je v žiadnom zozname – vtedy, keď nie je blokovaná žiadnym algoritmom. Pre každú úlohu by sme teda mohli mať navyše v jednej premennej uložené, **koľkými algoritmami je blokovaná**. Na začiatku tu bude počet všetkých algoritmov, ktoré sú pre ňu potrebné. Pri akciách budeme toto číslo znižovať, keď budeme danú úlohu vyhadzovať zo zoznamov. Keď počet blokácií dosiahne hodnotu 0, znamená to, že sa úloha stala riešiteľnou a tak zvýšime naše globálne počítadlo riešiteľných úloh.

Poďme si teda riešenie zhrnúť. Máme jedno globálne počítadlo riešiteľných úloh a pre každý algoritmus máme zoznam úloh s úrovňami znalosti potrebnými na ich vyriešenie. Navyše si ešte v samostatných poliach pamätáme aktuálne úrovne znalostí algoritmov a počet blokácií každej úlohy.

Pozrime sa na to, čo sa stane v jednej akcii. Nech si Hodobox v tejto akcii zlepšil úroveň algoritmu i o u.

[4] Dvojica je implementovaná v `C++` ako `pair`. Prípadne by ste si mohli spraviť vlastný `struct`. V `Python`-e viete využiť k-ticu pod názvom `tuple`.

[5] Keďže polia sú indexované od 0, v poli veľkosti 10^6 by algoritmus s číslom 10^6 už nemal svoj zoznam úloh. Preto pridávame +1.

[6] Nech si Hodobox zlepšil znalosť algoritmu i z x na y. Odstránime úlohy, ktoré algoritmus i potrebovali na úroveň a, takú že platí $x < a \leq y$.

- Pozrieme sa do poľa znalostí algoritmov a zvýšime aktuálnu znalosť algoritmu i o u. Nech je táto nová hodnota y.
- Ďalej sa chceme postupne pozrieť na všetky úlohy v zozname algoritmu i. Postupne ich všetky prejdeme a budeme vyhadzovať tie, pre ktoré už vie Hodobox algoritmus dosť dobre (tie, ktoré vyžadujú znalosť nanajvýš y).
  - Popri vyhadzovaní chceme každej vyhodenej úlohe znižovať počet blokácii o jedna.
  - Keď zistíme, že úlohe klesol počet blokácií na 0, zvýšime počítadlo úloh, ktoré už vie Hodobox vyriešiť, o jedna.
- Po prejdení všetkých úloh v zozname vypíšeme počítadlo riešiteľných úloh.

Teraz by sme už mali vedieť naprogramovať toto riešenie.

Posledný krok k vzorovému riešeniu

Vyhadzovanie úloh sme chceli robiť tak, že prejdeme celý zoznam, a vždy, keď nájdeme úlohu, ktorá algoritmus i potrebuje na úroveň najviac y, tak ju zo zoznamu odstránime.

Mohlo by sa nám ale stať, že by sme v každej akcii vyhadzovali len jednu úlohu z konca zoznamu, prípadne, ešte horšie, že naučením sa algoritmu by sme neodblokovali (neodstránili) žiadnu úlohu. V každej takejto akcii by sme prešli celý zoznam napriek tomu, že výsledok (počet riešiteľných úloh) sa vôbec nezmenil.

Chceli by sme teda v každom zozname prejsť len tie úlohy, ktoré mali šancu stať sa riešiteľnými, mali šancu na odblokovanie. Prvá úloha, ktorú by sme chceli zo zoznamu vyhodiť je tá, ktorá si vyžaduje najnižšiu znalosť algoritmu i. Ak nevieme odstrániť tú, môžeme rovno skončiť, lebo žiadna iná úloha sa určite neodblokuje.

Stačí teda, ak si **úlohy blokované algoritmom i usporiadame podľa potrebnej znalosti algoritmu**. Na konci zoznamu bude úloha, ktorá vyžaduje najnižšiu znalosť. Keď sa Hodobox zlepší v algoritme i, pozrieme sa na koniec zoznamu a ak už Hodobox vie algoritmus dostatočne, túto úlohu zo zoznamu vyhodíme a pokračujeme ďalšími úlohami – takými, ktoré potrebujú vyššiu znalosť algoritmu. Skracovanie zoznamu ukončíme, ak ďalšiu úlohu nevieme vyhodiť, teda ak Hodobox nevie algoritmus dostatočne pre žiadne ďalšie úlohy v zozname.

Hneď po načítaní vstupu si teda pre každý algoritmus usporiadame zoznam ním blokovaných úloh poľa potrebnej znalosti.[7]

Listing programu (C++)

```
#include <cstdio>
#include <vector>
#include <algorithm>

using namespace std;

bool compare(pair<int, int> a, pair<int, int> b) {
    return (a.second > b.second);
}

int main() {
    int n;
    scanf("%d", &n);
    // Počet aloritmov, potrebných na úlohy.
    vector< int > algs_count(n);

    // Pre každý algoritmus si budeme pamätať, ktoré úlohy ho potrebujú, a na akú úroveň.
    vector< vector< pair<int, int> > > problems;

    // V tejto premennej si budeme počítať, koľko algoritmov Hodobox zatiaľ vie.
    int uloh = 0;
    int max_alg = -1;
    for (int i = 0; i<n; ++i) {
        int a;
        scanf("%d", &a);
        algs_count[i] = a;
        for (int j = 0; j<a; ++j) {
            int x,y;
            scanf("%d %d", &x, &y);
            if (x > max_alg) {
                max_alg = x;
                problems.resize(max_alg+1, vector< pair<int, int> >());
            }
            problems[x].push_back({i, y});
        }
        // Ak na úlohu nie sú potrebné žiadne algoritmy, tak ju Hodobox vie hneď zozačiatku.
        if (a == 0) uloh++;
```

[7] Ak využívate `sort` v `C++` alebo v `Python`-e, tak si dajte pozor na to, že zoznam dvojíc sa primárne usporiadava podľa prvého prvku z dvojice. Oplatí sa teda mať v dvojiciach uloženú najprv potrebnú úroveň znalosti algoritmu a až potom číslo úlohy, alebo si napísať vlastnú porovnávaciu funkciu.

```
    }
    // Pre každý algoritmus, si usporiadam úlohu, ktorého potrebujú, podľa úrovne. Od najväčšej po najmenšiu.
    for (unsigned i = 0; i<problems.size(); ++i) {
        sort(problems[i].begin(), problems[i].end(), compare);
    }
    vector<int> level(max_alg+1, 0);
    int p;
    scanf("%d", &p);
    for (int i = 0; i<p; ++i) {
        int c,u;
        scanf("%d %d", &c, &u);
        // Aloritmus nie je potrebný na žiadnu úlohu.
        if (c > max_alg) {
            printf("%d\n", uloh);
            continue;
        }
        level[c] += u;
        while (!problems[c].empty() && problems[c].back().second <= level[c]) {
            // Našli sme problém, ktorý potreboval algoritmus "c".
            // Doteraz ale Hodobox nevedel tento algoritmus dostatočne.
            int problem = problems[c].back().first;
            problems[c].pop_back();
            algs_count[problem]--;
            // Ak sa počet algoritmov potrebných na problém zmenší na nula, tak už túto úlohu vie Hodobox vyriešiť.
            if (algs_count[problem] == 0) uloh++;
        }
        printf("%d\n", uloh);
    }
    return 0;
}
```

Ideálna časová zložitosť

Znova si označme súčet počtov algoritmov ako $A = \sum_{i=1}^{n} a_i$. Načítanie vstupu zvládneme určite v čase $O(A + n)$ (keďže A môže byť menšie ako n, treba v odhade uviesť aj n).

Potom si pre každý algoritmus musíme usporiadať zoznam jeho úloh. Potrebný čas bude $O(\sum_{i=1}^{n}(b_i \log b_i))$ kde b_i je počet úloh, ktoré potrebujú algoritmus i. Vieme, že platí $A = \sum b_i$, a pre každé i platí $b_i < A$, takže vieme predošlý súčet zjednodušiť:

$$\sum_{i=1}^{n}(b_i \log b_i) \leq \sum_{i=1}^{n}(b_i \log A) = (\sum_{i=1}^{n} b_i) \cdot \log A = A \log A$$

To znamená, že usporiadanie všetkých prvkov stíhame v $O(A \log A)$.

Čo sa týka spracovávania Hodoboxových akcií, tak v jednej akcii sme mohli odstrániť 0 až n úloh. Ak by sme zložitosť jednej akcie odhadli ako $O(n)$, v celkovej zložitosti by vystupoval člen $O(pn)$, čo je ale príliš veľké číslo.

Na počítanie zložitosti akcií sa teda treba pozrieť z pohľadu každej úlohy samostatne. Predstavme si prípad, že na konci bude Hodobox vedieť vyriešiť každú úlohu. Vezmime si teda jednu úlohu i, na ktorú treba a_i algoritmov. Ak ju Hodobox vie vyriešiť, tak sa niekedy musel naučiť každý z a_i algoritmov. Teda ak úloha i bola v a_i zoznamoch pre a_i rôznych algoritmov tak bola aj práve raz bola z každého z týchto zoznamov vyhodená (práve vtedy, keď sa naučil Hodobox daný algoritmus dostatočne). Môžeme teda povedať, že úloha i bola a_i krát "spracovaná".

Každé spracovanie úlohy – vyhodenie zo zoznamu, znížene počtu blokácií, zvýšenie počítadla riešiteľných úloh – vieme robiť v čase $O(1)$ (práve vďaka tomu, že úlohy vždy odstraňujeme z konca zoznamu). Sčítaním počtov spracovaní každej úlohy dostávame celkový počet spracovaní: $\sum_{i=1}^{n} a_i = A$. V každej akcii tiež zvyšujeme úroveň naučeného algoritmu a vypisujeme výsledok, a preto musíme do zložitosti započítať aj člen $O(p)$. Takže v najhoršom prípade, keď sa Hodobox naučí vyriešiť všetky úlohy, bude spracovanie akcií trvať $O(A + p)$.

V súčte je teda časová zložitosť $O(n + A \log A + p)$.

Naozajstná časová zložitosť

Možno ste si už všimli, že tu niečo nesedí. Veď naše veľké, dvojrozmerné pole má prvý rozmer max_alg, čo môže byť až 10^6 a toto pole predsa potrebujeme inicializovať. Každý z riadkov od 0 po max_alg musí byť vytvorený, aj keď bude prázdny. Na načítanie prvej časti vstupu budeme potrebovať čas až $O(A + n + max_alg)$.

Možno sa pýtate, prečo sa vôbec nad max_alg zamýšľame. Veď v najväčšom vstupe bude max_alg najviac 10^6, rovnako ako aj A či n. Pri malých vstupoch sa ale môže stať, že n bude 1, dokonca aj A bude 1, a predsa, ak na tú jednu úlohu potrebujeme vedieť práve jeden algoritmus s číslom 10^6, tak max_alg vyskočí na 10^6 a náš algoritmus bude potrebovať až rádovo 10^6 jednotiek pamäte a tiež rádovo 10^6 operácií na vytvorenie a vynulovanie daných polí. Ak by sme navyše mali algoritmy očíslované napríklad číslami 1 až 10^9, nemohli by sme si dovoliť používať takéto veľké pole.

Celková časová zložitosť bude teda $O(n + max_alg + A \log A + p)$, čo je v najhoršom prípade $O(A \log A + n + p + 10^6)$.

Pamäťová zložitosť

Načítanie vstupu nám zaberie $O(A)$ pamäte. Netreba ale zabudnúť aj na prázdne zoznamy pre čísla algoritmov, ktoré nikdy nepotrebujeme. Takže v skutočnosti to je $O(A + max_alg)$.

Okrem toho máme pole počtov blokácií jednotlivých úloh, ktoré je veľkosti n, a pole, kde si udržiavame aktuálnu znalosť algoritmov veľké max_alg+1.

Celková pamäťová zložitosť je teda $O(A + n + max_alg)$, v najhoršom prípade $O(A + n + 10^6)$.

Ako sa zbaviť závislosti na max_alg

Určite vás zaujíma, ako teda dosiahnuť časovú a pamäťovú zložitosť naozaj $O(A \log A + n)$ a $O(A + n)$, teda nezávislú od max_alg.

Naším problémom sú dve polia. Pole `problems` kde si držíme pre každý algoritmus zoznam úloh a pole `levels` kde si pre každý algoritmus držíme jeho aktuálnu úroveň. Obe majú veľkosť závislú od max_alg.

Možno niektorí z vás poznajú pojem **asociatívne pole**, `slovník` či `mapa`. Asociatívne pole je pole, ktorého prvky nie sú indexované pomocou postupnosti celých čísel, ale pomocou kľúčov. Kľúčom môže byť číslo, textový reťazec a iné. Medzi operácie ktoré vieme s mapou robiť patrí: pozrieť sa na hodnotu pod kľúčom, zmeniť hodnotu pod kľúčom, vytvoriť nový kľúč, prejsť všetky kľúče atď.

V našom prípade by sme teda zvolili mapu, kde kľúče budú celé čísla – konkrétne čísla algoritmov. Ale iba tých, ktoré naozaj potrebuje nejaká úloha. Pod týmto kľúčom by bol zoznam dvojíc: úloha a úroveň. Vždy, keď prvýkrát narazíme na algoritmus x, ktorý zatiaľ žiadna úloha nepotrebovala, tak vytvoríme v mape prázdny zoznam pod kľúčom x a pridáme tam aktuálnu úlohu. Ak sme už algoritmus predtým videli, tak len do jeho zoznamu pridáme úlohu.

To nám vyrieši problém s inicializáciou prázdnych zoznamov pre algoritmy, ktoré žiadna úloha nepotrebuje. Rovnako vieme aj prejsť iba všetky existujúce kľúče v mape, a tak sa vyhnúť usporiadavaniu prázdnych zoznamov pre tieto algoritmy. Podobne vieme mapu využiť aj na pole `levels`.

Okrem časovej zložitosti, použitie asociatívneho poľa zlepšuje aj pamäťovú zložitosť, keďže sme sa v prípade poľa `levels` zbavili núl pre nepotrebné algoritmy. Rovnako pre pole `problems` sme sa zbavili prázdnych zoznamov.

Obyčajná mapa so sebou prináša aj nevýhody, ktoré sa prejavia pri veľkom počte záznamov. Ak do nej chceme niečo vložiť, alebo sa pozrieť na nejakú hodnotu, tak to nie je v konštantnom čase, ale v $O(\log s)$, kde s je aktuálna veľkosť mapy.

Preto je lepšie v tomto prípade namiesto mapy použiť hashmapu, s ktorou sa pracuje väčšinou podobne ako s mapou, ale operácie sú v konštantnom čase, rovnako ako pri práci s poľom. Teda využitím hasmapy dostaneme naozaj čas. zložitosť $O(A \log A + n + p)$.

Porovnanie jednotlivých operácii môžeme vidieť v tabuľke, kde s predstavuje aktuálnu veľkosť mapy/hashmapy.

	Mapa	HashMapa
Vytvoriť kľúč	$O(\log s)$	$O(1)$
Vrátiť hodnotu pod kľúčom	$O(\log s)$	$O(1)$
Zmeniť hodnotu pod kľúčom	$O(\log s)$	$O(1)$
Prejsť všetky kľúče	$O(s)$	$O(s)$

Mapu môžete nájsť implementovanú v rôznych jazykoch. V `C++` sa mapa po anglicky nazýva `map`(dokumentácia) a hashmapa je `unordered_map` (dokumentácia). V `Python`-e máte hashmapu pod názvom `dict`(dokumentácia).

5. Obiehanie

Vlejd
(max. 10 b za popis, 5 b za program)

Jednoduché riešenie simuláciou

Najjednoduchší spôsob na riešenie tejto úlohy je simulovať celý turnaj postupne pre všetky počiatočné pozície Romankovho submitu.

Turnaj môžeme simulovať po úrovniach, pričom pre každú úroveň budeme mať jedno pole so submitmi, ktoré sa na túto úroveň dostali. Na začiatku si vytvoríme jedno pole so všetkými submitmi a Romankov submit vložíme na jednu pozíciu medzi tie ostatné a začneme simulovať. Pre dvojice submitov vedľa seba sa pozrieme na ich úžasnosti a do ďalšieho kola posunieme ten, ktorý je úžasnejší – úžasnosť víťaza uložíme do poľa ďalšej úrovne. Keď príde na rad Romankov submit, stačí zistiť, či je submit, s ktorým súperí, úžasnejší a teda či

musí Romanko obetovať horalku. Pozíciu Romankovho submitu na každej úrovni si vieme udržiavať napríklad v jednej premennej. Tento postup opakujeme, až kým nám nezostane v aktuálnej úrovni len jeden submit – ten Romankov.

Takto odsimulujeme celý turnaj pre všetky počiatočné pozície Romankovho submitu a pre každú simuláciu vypíšeme výsledok.

Ako rýchly je tento algoritmus? Potrebujeme $O(n)$-krát simulovať turnaj. Turnaj pozostáva z jednotlivých zápasov, presnejšie z porovnaní submitov. Vieme, že po každom porovnaní prestane byť jeden submit v turnaji, teda na vyradenie $n-1$ submitov potrebujeme spraviť $n-1$ porovnaní. Každý turnaj má teda $O(n)$ zápasov a toto riešenie má preto časovú zložitosť $O(n^2)$. Pri pozornej implementácií vieme dosiahnuť pamäťovú zložitosť $O(n)$. Naprogramovaním tohto postupu by ste dostali približne 3 body v praktickej časti.

Lepšie riešenie

Stará programátorská múdrosť hovorí, že ak chceme mať efektívny algoritmus, tak nesmieme nič počítať zbytočne a tiež nesmieme nič počítať dvakrát. Robíme niečo zbytočne? Simulujeme napríklad veľmi veľa zápasov, ktoré sa Romanka takmer vôbec netýkajú. Takisto veľmi veľa krát simulujeme presne rovnaké zápasy.

Najprv ale zaveďme trošku formálnejšie pojmy, aby sme sa mohli jednoduchšie vyjadrovať. Prečíslujme si submity ostatných ľudí ako a_0 až a_{n-2} (za číslovanie od 1 v zadaní sa ospravedlňujeme). Označme si pozície na začiatku ako p_0, p_1 až p_{n-1}.

Prvým pozorovaním zistíme, čo sa vlastne deje pri jednotlivých zápasoch. V každom zápase proti sebe súperia **najúžasnejšie submity z konkrétnych intervalov**.

- V prvom kole spolu súperia „reprezentanti" intervalov dĺžky 1 (p_0 proti p_1, p_2 proti p_3, atď.).
- V druhom kole sú to už reprezentanti intervalov dĺžky 2 (víťaz z p_0 až p_1 proti víťazovi z p_2 až p_3).
- V treťom kole sú to intervaly dĺžky 4 (víťaz z p_0 až p_3 proti víťazovi z p_4 až p_7).

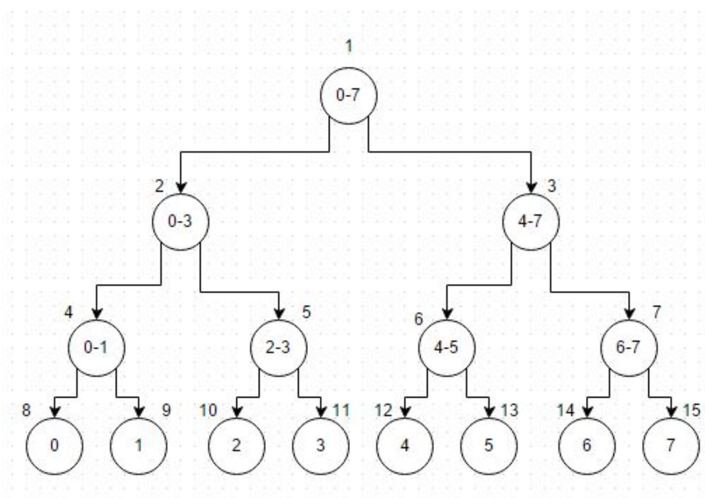
Podobne to ide aj pre ďalšie kolá. Kto ale môže byť reprezentantom pre jeden konkrétny interval? Vždy je to buď najúžasnejší submit z daného intervalu, alebo je to Romanko.

Druhým pozorovaním je, že to, **aký submit začína na pozícii p_i závisí len od umiestnenia Romankovho submitu**.

- Ak je Romankov submit naľavo od p_i, tak na p_i je submit a_{i-1}.
- Ak je Romankov submit napravo, tak na p_i je submit a_i.

Toto vieme povedať aj pre celé intervaly. Uvažujme interval p_x až p_y, na ktorom sa nenachádza Romankov submit. Potom sa na tomto intervale nachádzajú buď submity a_x až a_y (ak je Romankov submit napravo od tohoto intervalu), alebo a_{x-1} až a_{y-1} (ak je Romankov submit naľavo od tohoto intervalu).

Ako vieme tieto pozorovania využiť? Keď chceme zistiť, s kým musí Romankov submit v nejakom kole bojovať, nemusíme simulovať celý daný podstrom turnaja. Stačí nám len zistiť maximum z a_{x-1} až a_{y-1} alebo a_x až a_y. Na to vieme ľahko použiť maximový intervalový strom.



Každý vrchol stromu reprezentuje jeden interval (čísla v krúžkoch sú začiatok a koniec intervalu) a v každom vrchole si budeme pamätať jedno číslo. Vo vrcholoch na najspodnejšej úrovni budú úžasnosti jednotlivých submitov. Hodnoty zvyšných vrcholov budú vždy maximum z ich dvoch synov. V každom vrchole si budeme teda pamätať úžasnosť víťazného submitu na danom intervale.

Výhodou takéhoto stromu je to, že si ho nemusíme pamätať ako graf – vrcholy a hrany – ale dá sa dobre reprezentovať ako obyčajné pole P dĺžky $2n$. Na pozícii 1 bude koreň. Na pozíciach n až $2n-1$ budú listy. (Na obrázku sú indexy jednotlivých vrcholov v poli P napísané mimo krúžkov a $n = 8$.) To, čo je najkrajšie na tejto reprezentácii je, že vrchol x má synov na pozíciách $2x$ a $2x+1$ a otca na pozícii $\lfloor x/2 \rfloor$. Teda napr. vrchol 7 má synov 14 a 15 a otca 3. Vieme sa tak po strome rýchlo pohybovať a zistiť víťaza intervalu vo vrchole x ako $\max(P[2x], P[2x+1])$.

Intervalový strom vie vo všeobecnosti zistiť maximum na ľubovoľnom intervale v čase $O(\log n)$. Intervaly, ktorých maximá nás zaujímajú, majú ale vždy dĺžku mocninu dvojky a tak sú ich maximá práve čísla uložené vo vrcholoch intervalového stromu. Vďaka tomu vieme víťazov jednotlivých intervalov zistiť v konštantnom čase – pozretím sa na správny index poľa.

Potrebujeme ale dva intervalové stromy, jeden pre normálne intervaly – ako keby bol Romankov submit na konci (napravo) – a jeden pre posunuté – ako keby bol romankov submit na začiatku (naľavo).

Ako bude teda vyzerať celé riešenie? Najprv si predpočítame víťazov jednotlivých intervalov pomocou dvoch stromov. Ich výpočet nám potrvá $O(n)$. Najprv uložíme na koniec poľa P úžasnosti jednotlivých submitov a potom jedným prechodom od konca spočítame víťazov všetkých intervalov.

Ďalej budeme simulovať turnaje, ale tentokrát len zápasy Romankovho submitu. V každom kole vieme pomocou stromov zistiť úžasnosť submitu, s ktorým musí Romankov submit zápasiť (s víťazom niektorého intervalu). Kôl v turnaji je $O(\log n)$ a zisťovanie maxima nám trvá $O(1)$. Celý turnaj potrebujeme navyše odsimulovať pre $O(n)$ pozícií. Máme teda riešenie s časovou zložitosťou $O(n \log n)$ a s pamäťovou zložitosťou $O(n)$. Toto riešenie dokázalo dostať v praktickej časti plný počet bodov.

Listing programu (C++)

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

using namespace std;

vector<int> max1;
vector<int> max2;
int n, k;

int main(){
    // Nacitame a nainicializujeme
    cin >> n >> k;
    max1.resize(2*n, 0);
    max2.resize(2*n, 0);
    vector<int> submits(n-1);
    for(int i = 0; i < n - 1; i++){
        cin >> submits[i];
    }

    // Pripravime maximovy strom
    for(int i = 0; i < n-1; i++){
        max1[n+i+1] = submits[i];
        max2[n+i] = submits[i];
    }

    max1[n+0] = k;
    max2[n+n-1] = k;
    for(int i = n-1; i >= 0; i--){
        max1[i] = max(max1[2*i], max1[2*i+1]);
        max2[i] = max(max2[2*i], max2[2*i+1]);
    }

    // Prejdeme vsetky pozicie
    for(int i = 0; i < n; i++){
        int position = n+i;
        int answer = 0;

        while (position > 1){
            int new_position = position/2;
            if (new_position*2 == position){ // Romankov submit bol nalavo
                if (max1[position+1] > k)
                    answer += 1;
            }
            else{                            // Romankov submit bol napravo
                if (max2[position-1] > k)
                    answer += 1;
            }
            position = new_position;
        }

        cout << answer << ((i==n-1) ? '\n' : ' ');
    }
    return 0;
```

```
}
```

Optimálne riešenie

Riešenie však vieme ešte zefektívniť. Stále totiž robíme niektoré veci viackrát. Pozrime sa na posledný zápas. Koľkokrát sa spýtame na maximum z intervalu $p_0, p_{\frac{n}{2}}$? Vždy, keď Romankov submit začína v intervale $p_{\frac{n}{2}}, p_{n-1}$, čo je presne $\frac{n}{2}$ krát. Nevieme sa tohto počítania nejako zbaviť? Vieme!

Ak nám vadí, že sa na nejaký zápas pozeráme viackrát, pozrime sa naň len raz. Napríklad pre posledný zápas máme dve možnosti: buď bol Romankov submit na pozíciách z intervalu $A = (p_0, p_{\frac{n}{2}})$, alebo $B = (p_{\frac{n}{2}} + 1, p_{n-1})$. Ak bol na intervale A, tak vieme, ktoré submity boli na intervale B a teda vieme, s kým bude Romankov submit súperiť. To ale znamená, že pre všetky pozície z intervalu A vieme povedať, či bude musieť Romanko v danom zápase musieť obetovať horalku. Toto isté vieme zjavne urobiť pre pozície z intervalu B.

Následne môžeme samostatne riešiť prípady, keď bol Romankov submit v intervaloch $A = (p_0, p_{\frac{n}{4}})$ a $B = (p_{\frac{n}{4}} + 1, p_{\frac{n}{2}})$ a v podobných intervaloch v druhej polovici všetkých pozícií. To, že sú intervaly do seba tak pekne vnorené, navádza na použitie rekurzie.

Kým v predošlých riešeniach sme počítali výsledky zdola nahor – pre každú pozíciu Romankovho submitu sme simulovali súboje od prvého po posledný, teraz budeme úlohu riešiť zhora nadol. Pre všetky možné pozície Romankovho submitu najprv zistíme, v ktorých z nich musel obetovať horalku v poslednom súboji, v ďalšej úrovni rekurzie zistíme, na ktorých pozíciách musel obetovať horalku v predposlednom súboji, atď.

Pozrime sa na interval I pozícií (p_x, p_y). Tento interval si rozdeľme na úsek $A = (p_x, p_{\frac{x+y}{2}})$ a $B = (p_{\frac{x+y}{2}} + 1, p_y)$.

- Ak bol Romankov submit v intervale B, musel poraziť najúžasnejší submit z A. Jeho úžastnosť vieme ľahko nájsť v našom intervalovom strome, pretože na pozíciách $p_x, p_{\frac{x+y}{2}}$ budú submity $a_x, a_{\frac{x+y}{2}}$. V rekurzii si ako parameter posielame počet horaliek, ktoré musel Romanko zatiaľ obetovať. Tento počet zvýšime, ak je víťaz intervalu A úžasnejší ako Romankov submit, a zavoláme sa rekurzívne na interval B.
- Pokiaľ je Romankov submit z intervalu A, musíme si dávať trochu pozor. V B totiž budú submity $p_{\frac{x+y}{2}}, p_y - 1$, lebo Romanko je od nich naľavo. Pre tie máme druhý intervalový strom. Opäť porovnáme Romankov submit s víťazom B a rekurzívne sa zavoláme na A.
- Ak je náš interval I jednoprvkový, rekurziu ukončíme a v posielanom parametri sme dostali odpoveď pre danú pozíciu – koľko horaliek musel Romanko obetovať, ak začínal na pozícii $i, I = (p_i, p_i)$.

V riešení teda najprv zavoláme rekurzívnu funkciu na celý interval pozícií $(0, n-1)$ s nula obetovanými horalkami. Následne sa v každom volaní rozdvojíme na intervaly A a B.

Toto krásne riešenie má časovú zložitosť $O(n)$. Každé rekurzívne volanie spravíme v konštantnom čase, na každý interval sa totiž zavoláme práve raz a intervalov dĺžky 1 je n, intervalov dĺžky 2 je $n/2$, atď. a teda $n + n/2 + n/4 + \cdots + 2 + 1 = 2n - 1$ (n je mocninou dvojky).

Listing programu (C++)

```cpp
#include <iostream>
#include <algorithm>

using namespace std;

vector<int> ans;
vector<int> max1;
vector<int> max2;
int n, k;

void solve(int down, int up, int pozition, int horalky){
    if(down == up){ // Interval ubsahuje len jednu poziciu
        ans[down] = horalky;
        return ;
    }
    int middle = (down+up)/2;

    // Volanie na prvy interval (A)
    solve(down, middle, 2*pozition, horalky + ((max1[2*pozition+1] > k) ? 1 : 0));

    // Volanie na druhy insterval (B)
    solve(middle+1, up, 2*pozition + 1, horalky + ((max2[2*pozition] > k) ? 1 : 0));
}

int main(){
    cin >> n >> k;
    ans.resize(n,0);
    max1.resize(2*n, 0);
```

```
    max2.resize(2*n, 0);
    vector<int> submits(n-1);
    for(int i = 0; i < n-1; i++){
        cin >> submits[i];
    }
    for(int i = 0; i < n-1; i++){
        max1[n+i+1] = submits[i];
        max2[n+i] = submits[i];
    }

    max1[n+0] = k;
    max2[n+n-1] = k;
    for(int i = n-1; i >= 0; i--){
        max1[i] = max(max1[2*i], max1[2*i+1]);
        max2[i] = max(max2[2*i], max2[2*i+1]);
    }

    solve(0, n-1, 1, 0);

    cout << ans[0];
    for(int i = 1; i < n; i++){
        cout<<" "<<ans[i];
    }
    cout << endl;
    return 0;
}
```

Žaba

6. Odplata drepovaním

(max. 12 b za popis, 8 b za program)

Začnime tým, že si zopakujeme, o čom bolo zadanie a zavedieme trochu iné pojmy, ktoré budeme používať na popis situácie. V zadaní sme mali mriežku $r \times s$, v ktorej stáli účastníci. Každý účastník buď stál alebo čupel. Namiesto účastníkov však vyplňme našu mriežku 0 a 1. 0 bude predstavovať stojaceho účastníka a 1 čupiaceho.

Emo potom určoval riadky a stĺpce tejto mriežky a účastníci v príslušnom riadku alebo stĺpci si museli čupnúť ak stáli a naopak. Pri použití 0 a 1 to znamená, že vždy, keď určíme nejaký stĺpec alebo riadok, tak sa v ňom všetky 0 zmenia na 1 a naopak. Vieme, že Emo určil dokopy p_r riadkov a p_s stĺpcov a na konci bolo v mriežke k čupiacich účastníkov – teda k hodnôt 1.

Našou úlohou bolo vypočítať, koľkými možnosťami mohol za uvedených podmienok tento výsledok dosiahnuť. Už v zadaní je napísané, že pri počítaní možností nám nezáleží na tom, v akom poradí Emo riadky a stĺpce určoval. Uvedomme si, že táto podmienka je rozumná, lebo poradie nezmení to, ako vyzerá výsledná mriežka a koľko jednotiek obsahuje.

Pozorovanie

Čo teda ovplyvňuje to, ako vyzerá mriežka na konci? Kedy bude v i-tom riadku a j-tom stĺpci 1 a kedy tam bude 0?

Je jasné, že vybratie iného ako i-teho riadku alebo j-teho stĺpca nijak neovplyvňuje číslo na tomto políčku. Taktiež vieme, že na začiatku bola na tomto políčku 0. Vždy keď vyberieme i-ty riadok alebo j-ty stĺpec sa táto hodnota zmení na opačnú. Ak si označíme hodnotu r_i počet vybraní i-teho riadku a s_j počet vybraní j-teho stĺpca, tak ľahko nahliadneme, že hodnota na tomto políčku bude na konci $r_i + s_j \mod 2$. Teda parita počtu vybratí, ktoré ovplyvnili toto políčko. To nás vedie k veľmi zaujímavej úvahe. Zjavne je jedno, koľkokrát určíme niektorý riadok alebo stĺpec, dôležitá je len parita tohto počtu určení.

Poďme však s touto úvahou ešte ďalej. Nech bolo x_r riadkov a x_s stĺpcov vybraných nepárny počet krát. Koľko 1 bude vo výslednej mriežke? Zjavne na miestach kde sa pretína takýto nepárny riadok a stĺpec bude 0, lebo tieto dve vybratia sa anulujú. Takže 1 bude len na miestach, kde sa pretína nepárny riadok s párnym stĺpcom alebo naopak. Počet 1 vo výslednej mriežke teda bude $x_r \cdot (s - x_s) + (r - x_r) \cdot x_s$. Ak si teda určíme koľko riadkov a koľko stĺpcov bolo vybraných nepárny počet krát, vieme ľahko overiť, či takéto vybratie viedlo ku k jednotkám vo výslednej mriežke.

Avšak, všimnime si, že nás nezaujíma ani to, ktoré riadky a stĺpce to budú. Jediný dôležitý je ich počet. No a vzhľadom na veľkosť vstupu nie je problém vyskúšať všetky kombinácie počtov nepárnych riadkov a stĺpcov a zistiť, či je takáto kombinácia (dvojica – počet nepárnych riadkov, počet nepárnych stĺpcov) vyhovujúca a vytvorí potrebný počet jednotiek.

To, čo sme spravili doteraz nám však predsa vôbec nepomáha. Veď stále nevieme, koľko bolo všetkých možností. Získali sme však dôležitú vec. Všetky výsledné možnosti sme si rozdelili do pomerne málo (najviac $(r+1) \cdot (s+1)$) skupín podľa počtov nepárnych riadkov a stĺpcov. Pritom každá možnosť patrí do práve jednej takejto skupiny – stačí sa pozrieť, koľko jej riadkov a koľko jej stĺpcov bolo vybraných nepárny počet krát. Ak sa teda vrátime späť a podarí sa nám rýchlo vypočítať, koľko výsledných možností leží v danej skupine budeme vedieť pomerne rýchlo zistiť výsledok. Jednoducho sčítame veľkosti správnych skupín – tých, ktoré vedú k vytvoreniu k jednotiek.

Počítanie veľkosti jednej skupiny

Úloha sa teda zmenila: Koľko možností vybratia p_r riadkov a p_s stĺpcov vedie k tomu, že x_r riadkov a x_s stĺpcov vyberieme nepárny počet krát?

Pri tomto počítaní musíme postupne započítať všetko, čo sme zanedbali. To znamená výber konkrétnych x_r riadkov (a to isté pre stĺpce), ktoré majú byť vybrané nepárny počet krát a samotné počty vybratí každého riadku (resp. stĺpca).

Začnime tým prvým. Ak máme hodnotu x_r, musíme započítať všetky možnosti, kde je práve x_r riadkov vybraných nepárny počet krát. Ak ste už počuli o kombinačných číslach, tak by vám malo byť jasné, že tento počet možností je práve $\binom{r}{x_r}$. Lebo práve kombinačné číslo r nad x_r vyjadruje to, koľkými možnosťami vieme z r vecí (riadkov) vybrať x_r (tých nepárnych riadkov). Pre ľahší zápis zapisujme toto číslo $P(r, x_r)$.

Ak ste o kombinačných číslach ešte nepočuli, nič si z toho nerobte. Za chvíľu sa dostaneme k tomu, ako ich počítať bez ohľadu na to, či viete, čo to je. Zatiaľ však akceptujte ich existenciu a to, že sú presne hodnotou, ktorú hľadáme.

Ostáva nám ešte druhý krok zovšeobecnenia. Aj keď sme priradili každému riadku paritu, musíme túto výslednú paritu dosiahnuť nejakým konkrétnym vybratím týchto riadkov. Koľko je však týchto možností? Zjavne, každý z nepárnych riadkov musel byť vybraný aspoň raz. To znamená, že nám ostáva rozdeliť $p_r - x_r$ vybraní. Tieto vybratia však musia ísť vždy v páre. Jedno vybratie by totiž zmenilo paritu a to nechceme. Takže ak sa rozhodneme priradiť niektorému riadku ďalšie vybratie, musíme to spraviť rovno dvakrát. Máme teda $\frac{p_r - x_r}{2}$ párov vybratí a každý pár musíme priradiť niektorému riadku.

Opäť dostávame pomerne známu hodnotu – počet kombinácií s opakovaním $C(r, (p_r - x_r)/2)$. Keďže nám nezáleží na poradí, tak z r riadkov chceme vybrať $(p_r - x_r)/2$, niektoré riadky však môžeme aj opakovať – priradiť viacerým párom. Opakovanie znamená, že danému riadku nepriradíme len 2 vybratia (1 pár), ale 4, 6...

Naviac, pre kombinácie s opakovaním platí veľmi užitočná vlastnosť, že $C(n, k) = P(n + k - 1, k)$. To znamená, že ak vieme počítať kombinácie bez opakovania, vieme počítať aj tie s opakovaním.

Ak si teda zoberieme hodnoty x_r a x_s a tieto dve hodnoty vedú k požadovanému výsledku – vytvoria k jednotiek a taktiež x_r má rovnakú paritu ako p_r (párnym počtom vybratí riadku nemôžem vytvoriť nepárny počet nepárny počet krát vybraných riadkov) a x_s má rovnakú paritu ako p_s, tak počet možností v takejto množine spočítame ako $P(r, x_r) \cdot P(s, x_s) \cdot C(r, (p_r - x_r)/2) \cdot C(s, (p_s - x_s)/2)$.

Riešenie

Na vypočítanie hodnôt $P(a, b)$ a $C(a, b)$ použijeme Pascalov trojuholník. Ten si na začiatku behu programu spočítame pre dostatočne veľké hodnoty, teda prvých m riadkov, kde $m = max(2r, 2s, 2p_r, 2p_s)$. Potom už len pre každú z $(s + 1) \cdot (r + 1)$ skupín v konštantnom čase overíme, či vytvára k jednotiek a v konštantnom čase spočítame (pomocou Pascalovho trojuholníka), koľko možností patrí do danej skupiny. Celkovo tak dostaneme riešenie so zložitosťou $O(m^2 + r \cdot s) = O(m^2)$.

Kombinačné čísla

Možno si teraz hovoríte, že táto úloha je nefér pre niekoho, kto nikdy nepočul o kombinačných číslach. No aj keď je pre niekoho takého rozhodne ťažšia, zďaleka nie je neriešiteľná. Zabudnime preto na to, že existujú nejaké kombinačné čísla a riešme úlohu: Koľkými možnosťami vieme vybrať k prvkov z n možných, ak nám nezáleží na poradí a prvky sa nemôžu pri výbere opakovať?

Na začiatok môžeme vyriešiť nejaké jednoduché podproblémy. Napríklad ak je $k = n$ tak zjavne máme jedinú možnosť. Takisto ak je $k = 0$, tak ostáva jediná možnosť, nič nevybrať. Ak je $k = 1$, tak možností máme n, môžeme totiž vybrať ľubovoľný prvok. Zaujímavý je tiež prípad, keď $k > n$. V takom prípade je odpoveď 0, keďže neexistuje spôsob ako z n prvkov vybrať k.

Pozrime sa teraz na všeobecný prípad, keď máme n vecí, z ktorých chceme k vybrať a $k \geq 1$. Zoberme si teraz ľubovoľný z n prvkov, označme si ho x. Keďže nám nezáleží na poradí vyberania, jediné čo sa musíme pýtať je, či tento prvok vyberieme alebo nie. Všetky možnosti sa teda rozdelia na dve skupiny – možnosti, v ktorých prvok x vyberieme a možnosti, kde x nevyberieme.

Ak teda chceme zistiť hodnotu $P(n, k)$ vedeli by sme to vypočítať ako súčet možností v týchto dvoch skupinách. Skúsme teda vypočítať, koľko je takých možností, v ktorých x vyberieme. Tento prvok už použiť nemôžeme, takže nám ostáva $n - 1$ prvkov. Z nich ale potrebujeme vybrať už len $k - 1$ prvkov, keďže sme už vybrali prvok x. Táto hodnota je preto $P(n - 1, k - 1)$. A podobnú vlastnosť nájdeme aj v prípade, keď prvok x nevyberieme. Ostáva nám už len $n - 1$ prvkov (lebo o prvku x sme sa už rozhodli, že ho neberieme), z ktorých musíme vybrať ešte k. Takže máme hodnotu $P(n - 1, k)$.

Dostávame vzorec:

$$P(n,k) = P(n-1,k) + P(n-1,k-1)$$

Toto je pomerne dôležitý kombinatorický vzorec, ktorého aplikáciou vzniká napríklad Pascalov trojuholník. A práve tento trojuholník je to, čo chceme počítať. Postupne budeme počítať všetky hodnoty $P(n,k)$. Keďže na vypočítanie $P(n,k)$ potrebujeme hodnoty s menším n, budeme tieto hodnoty počítať od najmenších hodnôt n.

Na začiatku vieme, že $P(0,0) = 1$. Následne budeme počítať všetky kombinačné čísla s $n = 1$. Aby sme sa nesnažili vyberať záporný počet prvkov, tak si určíme, že $P(1,0) = 1$ a zvyšné hodnoty vypočítame vzorcom uvedeným vyššie. Toto všetko si budeme značiť do tabuľky, kde n je číslo riadku a k číslo stĺpca.

Po vypočítaní všetkých hodnôt pre $n = 1$ sa presunieme do riadku s $n = 2$ atď. Pričom vždy si v riadku pevne určíme hodnotu $P(n,0) = 1$ a potom na výpočet všetkých $1 \leq k \leq n$ použijeme odvodený vzorec. A keďže hodnoty, ktoré potrebujeme vo vzorci, už máme vypočítané, vypočítanie novej hodnoty nám zaberie konštantný čas. Časová zložitosť výpočtu Pascalovho trojuholníka je preto $O(n^2)$.

Ešte sme však nevyriešili, ako počítať kombinácie s opakovaním. Hlavne ak nepoznáme vzorec na prevod na kombinácie bez opakovania. Žiaden strach, použijeme úplne rovnakú úvahu. Naša úloha znie: Koľkými spôsobmi viem z n prvkov vybrať k, ak nám nezáleží na poradí, ale niektoré prvky môžeme vybrať aj viackrát.

Opäť si zoberieme prvok x, o ktorom sa budeme rozhodovať, či ho zoberieme alebo nie. Ak sa však rozhodneme, že prvok x nevyberieme, už nemáme možnosť ho zobrať neskôr. Ak ho chceme vybrať, hoci aj viackrát, musíme to spraviť teraz.

Ak teda prvok x nezoberieme, tak ako pred tým nám ostane $n-1$ prvkov, z ktorých musíme vybrať k. Ak však prvok x vyberieme, budeme vyberať tiež $k-1$ prvkov, na výber však budeme mať všetkých n prvkov, teda sa opäť môžeme rozhodnúť zobrať prvok x. To nám zaručí, že prvky vieme vyberať aj viac ako raz. A takisto nám to odvádza vzorec:

$$C(n,k) = C(n,k-1) + C(n-1,k-1)$$

Použitím rovnakej metódy ako pri počítaní Pascalovho trojuholníka vieme vypočítať aj tieto hodnoty s časovou zložitosťou $O(n^2)$. A nepotrebovali sme poznať zložité vzorce, stačilo vedieť správne použiť metódy dynamického programovania.

Posledná vec, ktorú treba spomenúť je, prečo sme na výpočet nepoužívali vzorec $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Problémom je, že výsledné číslo musíme modulovať, vrátiť len jeho zvyšok po delení 555 555 555. Ak by sme chceli teda používať faktoriály, tak by sme si museli vypočítať ich hodnoty modulo toto číslo, keďže 1500! sa nám do žiadnej číselnej premennej nezmestí (už s 20! by sme mali problém).

Problém ale potom nastáva pri delení. Modulo síce pekne funguje pri násobení alebo sčítaní a napríklad $a \cdot b \pmod m = (a \pmod m) \cdot (b \pmod m) \pmod m$ je správna rovnica, pre delenie takáto rovnica neplatí. Použitím delenia by sme preto dostali nesprávne čísla.

Listing programu (C++)

```
#include <cstdio>

#define MOD 555555555

using namespace std;

typedef long long ll;

ll P[2000][2000];
ll C[2000][2000];

int count( ll H, ll W, int Rcount, int Ccount, ll k ) {
    for(int i=0; i<1750; i++) P[i][0]=1;
    for(int i=1; i<1700; i++)
        for(int j=1; j<=i; j++)
            P[i][j] = (P[i-1][j] + P[i-1][j-1]) % MOD;

    for(int i=0; i<1750; i++){
        C[i][0]=1;
        C[1][i]=1;
    }
    for(int i=2; i<1700; i++)
        for(int j=1; j<1700; j++)
            C[i][j] = (C[i][j-1] + C[i-1][j]) % MOD;

    ll vysledok = 0;
    for(int i=0; i<=H; i++)
        for(int j=0; j<=W; j++) {
            ll pocet_jednotiek = (ll)j*H+(ll)i*W-2ll*(ll)i*(ll)j;
            if(pocet_jednotiek==k && i<=Rcount && j<=Ccount && i%2==Rcount%2 && j%2==Ccount%2) {
                ll poc=P[W][j]*P[H][i]; poc%=MOD;
                poc*=C[W][(Ccount-j)/2]; poc%=MOD;
                poc*=C[H][(Rcount-i)/2]; poc%=MOD;
                vysledok += poc; vysledok %= MOD;
```

```
            }
        }

    return vysledok;
}

int main() {
    ll r, s, k;
    int pr, ps;
    scanf("␣%lld␣%lld␣%d␣%d␣%lld␣", &r, &s, &pr, &ps, &k);
    printf("%d\n", count(r, s, pr, ps, k));
}
```

7. Obávaný skok syslí, horizontálno-zvislý

Hodobox
(max. 12 b za popis, 8 b za program)

Hrubá sila

Štedrí vedúci Klubu Slovenských Paraglidistov vám darujú jeden bod za riešenie, ktoré implementuje definíciu zo zadania. Stačí vyskúšať všetky trasy s s horami a zistiť, ktoré z nich vyhovujú podmienke v zadaní. Tak prečo to nevyužiť? Overíme si, či sme správne pochopili úlohu, a navyše budeme mať správne riešenie na otestovanie svojich riešení pre ďalšie sady.

Potrebujeme už len vymyslieť čo najpríjemnejší spôsob, ktorým trasy dĺžky s vyskúšame. Zamyslime sa, ako by si mohol za letu trasu zvoliť Syseľ? Vyberie si ľubovolnú horu, na ktorú sa na začiatku postaví a povie si, že chce skočiť ešte $(s-1)$-krát. Skočí s padákom, pričom letí nad všetkými horami napravo od tejto začiatočnej hory. Keď uvidí horu ktorá je aspoň o $m+1$ metrov nižšia ako tá z ktorej práve skočil, môže sa rozhodnúť že pristane práve na nej, alebo pokračuje v lete a rozhoduje sa nad ďalšou. Keď na nejakej hore pristane, opakuje sa tento proces znova – ale tentokrát už Syseľ túži skočiť len ďalších $(s-2)$-krát. Keď napokon pristane na niektorej z hôr a netúži už viac skákať, resp. túži skočiť ešte 0-krát, práve dokončil jednu trasu, ktorá mu vyhovuje.

Táto úvaha sa už celkom príjemne programuje pomocou rekurzie – skúsime 'skočiť' z každej hory, pamätajúc, že sme už dokončili trasu dlhú 1. Následne prejdeme všetky hory napravo, a pre každú horu nižšiu o viac ako m od vybranej hory zavoláme tú istú funkciu (akoby sme stáli na tejto hore) s tým, že sme už preleteli trasu dĺžky 2. Keď na nejakej hore pristaneme a zistíme, že sme už preleteli trasu s s horami, pripočítame k odpovedi 1.

Na záver malý detail – trás dĺžky s v pohorí s n horami môže byť najviac $\binom{n}{s}$. Toto číslo je v prvej sade najviac $\binom{20}{10} = 184\,756$, a teda nepresiahne hodnotu 10^9+7. Výsledok teda nemusíme modulovať (počítať zvyšok po delení 10^9+7).

Aká je naša pamäťová zložitosť? Stačí nám pamätať si výšky všetkých n hôr a zopár pomocných premenných. Naša rekurzívna funkcia sa v sebe zavolá najviac s-krát, a teda počet premenných, ktoré máme vytvorené v ľubovoľnom čase v jej všetkých volaniach bude úmerný s. Celková pamäťová zložitosť je teda $O(n+s)$.

Ako je to s časovou zložitosťou? Skúšame všetky platné trasy dĺžky s spomedzi n hôr. V najhoršom prípade všetky trasy dĺžky s vyhovujú Sysľovi, a bude ich teda $\binom{n}{s}$. Pre jednoduchší odhad zložitosti využijeme fakt, že všetkých podmnožín n prvkov je 2^n. Naše riešenie síce zaujímajú len s-prvkové množiny hôr v našom pohorí, ich počet však rastie asymptoticky rovnako rýchlo. Časová zložitosť je teda exponenciálna od n, s horným rádovým ohraničením $O(2^n)$.

Listing programu (C++)

```
#include <iostream>

using namespace std;

int hory[20],n,m,s,odpoved = 0;

//Simuluje Sysľa - skúša z hory 'pozicia' pristáť na niektorej ďalšej,
//alebo pripočíta odpoveď ak úspešne dokončil trasu dĺžky s.
void rekurzia(int pozicia,int kolko)
{
    if(kolko==0) // našli sme jednu platnú trasu
    {
        odpoved++;
        return;
    }

    for(int k=pozicia+1;k<n;++k) // skúsime skočit na každú ďalšiu horu
        if(hory[pozicia]-hory[k]>m)
            rekurzia(k,kolko-1);
}

int main()
{
    cin >> n >> m >> s;
    for(int i=0;i<n;++i) cin >> hory[i];
    //spočítame počet trás dĺžky s začínajúc horou 0,1,...,n-1
```

```
    for(int i=0;i<n;++i) rekurzia(i,s-1);

    cout << odpoved << "\n";
    return 0;
}
```

Postupne predlžujeme známe trasy

Označme výšku i–tej hory v smere na východ h_i. Pozrime sa bližšie na otázku, ktorú sme sa pri rekurzii pýtali: koľko platných trás dĺžky s začína v hore i? Ak by sme odpoveď chceli popísať slovne, je ich práve toľko, koľko platných trás dĺžky $s-1$ začína v horách $i+1, i+2, ..., n-1$, ktoré sú vysoké najviac $h_i - m - 1$. Keby sme pre každú horu k, pre ktorú platí $i < k$, vedeli povedať, koľko platných trás dĺžky $s-1$ v nej začína, ľahko by sme odpoveď spočítali. No a koľko takých trás existuje pre nejakú horu k? No predsa presne toľko, koľko platných trás dĺžky $s-2$ začína v takých horách $k+1, k+2, ..., n-1$, ktoré sú vysoké najviac $h_k - m - 1$.

Ale to je tá istá otázka ako predtým! Keby sme vedeli počet platných trás dĺžky d začínajúc v každej hore v pohorí, pre každú horu i by sme vedeli sčítať počty platných trás tých hôr, ktoré sú napravo od nás a dostatočne nízke, a tým našli počet platných trás dĺžky $d+1$ začínajúc v hore i. Túto hodnotu spočítame pre každé $0 \leq i \leq n-1$. Odteraz budeme označovať **počet platných trás dĺžky d začínajúc v hore i ako** $v(d,i)$.

Ak teda poznáme $v(d,i)$ pre nejaké d, vieme teraz spočítať počet platných trás dĺžky $d+1$ pre každú horu v pohorí! A kde začať? Poznáme pre nejaké d počet platných trás začínajúc v každej hore? Samozrejme - každá hora je sama o sebe platná trasa dĺžky 1. Vieme teda našou úvahou spočítať pre každú horu i počet platných trás dĺžky 2 ktoré v nej začínajú – $v(2,i)$. Jednoducho sčítame $v(1,j)$ všetkých dostatočne nízkych hôr napravo. Hodnoty $v(2,i)$ sú všetko, čo potrebujeme na vypočítanie trás dĺžky tri – $v(3,i)$, a tak ďalej. Tento výpočet zopakujeme teda $(s-1)$-krát a dostaneme pre každú horu počet platných trás dĺžky s, ktoré v nej začínajú. Tieto čísla sčítame, a máme hľadaný výsledok. Keďže nás zaujíma len jeho zvyšok po delení číslom $10^9 + 7$, všetky výpočty ním budeme modulovať.

Na záver tohto riešenia ešte vieme spraviť jedno pozorovanie: akonáhle sme pre každú horu zistili počet platných trás dĺžky $d+1$ ktoré v nej začínajú $(v(d+1,i),)$ predtým vypočítaná hodnota $v(d,i)$ nás už nebude nikdy zaujímať. Na celý výpočet nám teda budú stačiť dve polia, pričom budeme striedať ich význam. Pri prvom výpočte máme v prvom poli hodnoty $v(d,i)$, a do druhého počítame $v(d+1,i)$. Následne prvé pole vynulujeme, a spočítame doňho pre každú horu $v(d+2,i)$, využívajúc výsledky pre $d+1$ ktoré máme teraz uložené v druhom poli.

Dokopy si teda budeme pamätať výšky všetkých n hôr, a zároveň dve polia dĺžky n, v ktorých budeme striedavo vypočítavať pre každú horu počet v nej začínajúcich platných trás postupne väčšej dĺžky. Pamäť teda lineárne závisí od n a ničoho iného – $O(n)$.

Tú istú úvahu opakujeme $(s-1)$-krát – máme pre každú horu i vypočítanú hodnotu $v(d,i)$ pre nejaké d (na začiatku $v(1,i) = 1$), a pre každú horu budeme počítať $v(d+1,i)$. Ako vyzerá tento výpočet pre jednu horu? Prejdeme všetky hory východnejšie od nej, a pre každú dostatočne nízku horu k pripočítame k našej hore jej hodnotu $v(d,k)$. Toto robíme pre každú horu, ktorých je n, a počet kontrolovaných hôr napravo od nich takisto závisí lineárne od n. Vypočítanie hodnôt pre $v(d+1,i)$ z hodnôt $v(d,i)$ nám teda zaberá kvadratický čas od n, a tento výpočet opakujeme $(s-1)$-krát, teda lineárne od s. Celková časová zložitosť bude teda $O(s \cdot n^2)$.

Listing programu (C++)

```
#include <iostream>

using namespace std;
int main()
{
    int n,m,s,i,j,k,mod = 1000000007;
    cin >> n >> m >> s;
    int dp[2][n],hory[n];

    for(i=0;i<n;++i)
    {
        cin >> hory[i];
        dp[1][i] = 1; //každá hora je sama o sebe trasa dĺžky 1
    }

    for(i=1;i<s;++i)
    {
        int teraz = i%2, potom = (i+1)%2;
         //vynulujeme riadok kam budeme zapisovať, keďže v ňom sú súčty predošlých trás
        for(j=0;j<n;++j) dp[potom][j] = 0;
        for(j=0;j<n;++j)
        {
            //sčítame počet trás dĺžky i, začínajúc dostatočne nízkymi horami napravo
            for(k=j+1;k<n;++k)
            {
                if(hory[j]-hory[k]>m)
```

```
                {
                    dp[potom][j] += dp[teraz][k];
                    dp[potom][j] %= mod;
                }
            }

        }
    }

    int odpoved = 0;
    //sčítame počet trás dĺžky s začínajúc horami 0,1,...,n-1
    for(i=0;i<n;++i)
    {
        odpoved += dp[s%2][i];
        odpoved %= mod;
    }

    cout << odpoved << "\n";

    return 0;
}
```

Spočítavame rýchlejšie

Potrebujeme si zrýchliť naše riešenie. Zjavne potrebujeme z našej časovej zložitosti odstrániť faktor n^2, keďže pre $n = 100\,000$ je táto zložitosť neúnosná. Čo nám trvá n krokov? n-krát spracúvame nejakú horu i a počítame počet trás dĺžky $d+1$ (postupne $2, 3, ..., s-1, s$), ktoré v nej začínajú – $v(d+1, i)$. Ako vyzerá spracovanie jednej hory? Prejdeme všetky hory východnejšie od nej (teda hory k kde $i < k$. Tých je lineárne veľa od n), a pre tie hory, ktorých výška h_k je najviac $h_i - m - 1$ pripočítame k našemu číslu $v(d+1, i)$ počet trás dĺžky d začínajúc v hore k – $v(d, k)$.

Pozrime sa na hory, ktorých trasy vlastne chceme spočítať. Zoberme si všetky východnejšie hory od hory i, a usporiadajme si ich podľa výšky. Ktoré z nich nás zaujímajú? Tie dostatočne nízke! Presnejšie, tie hory, ktorých hodnoty $v(d, k)$ chceme spočítať, budú po usporiadaní tvoriť súvislý úsek hôr začínajúci najnižšou z nich, a končiaci najvyššou horou, ktorej výška h_k je stále menšia ako $h_i - m$ (toto môžu teda byť aj všetky východnejšie hory od hory i, prípadne ani jedna). Toto pozorovanie nám samo o sebe nepomôže – usporiadať východnejšie hory, ktorých je lineárne veľa od n, vieme najlepšie v čase $n \log n$. Pomôžeme si však, keď si uvedomíme že nás pri počítaní $v(d+1, i)$ vlastne nezaujíma, ktorá východnejšia hora prispieva koľkými trasami dĺžky d, ani to, kde tieto hory vlastne sú – stačí vedieť že su dostatočne nízke a východnejšie od nás.

Skúsme na to ísť teda trošku inak. Namiesto toho, aby sme zisťovali pre každú horu jej hodnotu $v(d, i)$, bude nám stačiť pre každú výšku h súčet $v(d, i)$ tých hôr i, ktoré majú výšku h. Ak toto vieme, pri počítaní $v(d+1, i)$ by nám stačilo spočítať tieto hodnoty pre výšky $1, 2, ..., h_i - m - 1$. Tieto hodnoty si pre hory dostatočne malej výšky môžeme rovno pamätať v pomocnom poli P na indexoch $1, 2, ..., 10^6$. Keď si teda náš algoritmus takto pozmeníme, stačilo by nám vedieť sčítavať čísla na intervale v poli P v lepšom ako lineárnom čase od veľkosti intervalov. Vieme to urobiť?

Odpoveď je: áno. Budeme však musieť použiť dátovú štruktúru, ktorá dokáže spracovávať (v našom prípade sčítavať) hodnoty na zvolenom intervale: intervalový strom. Ak ste o intervalovom strome nepočuli, prečítajte si nasledovný článok v kuchárke KSP.

Nám bude teraz stačiť skutočnosť, že s intervalovým stromom vieme zostrojiť funkciu $spocitaj(P, l, r)$, ktorá nám vráti súčet políčok v P od l po r, v logaritmickom čase od veľkosti stromu. Zároveň vieme zostrojiť aj funkciu $pridaj(P, i, y)$, ktorá na políčko P_i pripočíta hodnotu y taktiež v logaritmickom čase. Po zvýšení políčka i už bude funkcia spocitaj(P, l, r) vracať nový, zvýšený výsledok ak $l \leq i \leq r$,

Postupovať budeme teda rovnako ako pri pomalšom riešení s niekoľkými rozdielmi. Namiesto polí veľkosti n budeme mať intervalové stromy s o trochu viac ako 10^6 políčkami, pričom v intervale $[l, r]$ stromu S_d budeme mať hodnoty $v(d, i)$ začínajúc horami s výškami l až r. Počet platných trás dĺžky $d+1$ začínajúcich v hore i vypočítame pomocou $spocitaj(S_d, 0, h_i - m - 1)$ (nazvime výsledok y), a pridáme ho do stromu S_{d+1} pomocou $pridaj(S_{d+1}, i, y)$.

Ostáva jedna otázka – ako zaručiť, že práve spracovaná hora i neovplyvní počet spočítaných trás, keď budeme v ďalšom kroku spracovávať východnejšiu horu $i+1$? Stačí, ak po vypočítaní hodnoty $v(d+1, i)$ v S_{d+1} z nášho momentálne používaného stromu S_d zmažeme z políčka h_i počet trás, ktoré začínali v našej hore $v(d, i)$. Túto hodnotu si môžeme zapamätať v poliach veľkosti n (v riešení nazvané `zacina`). Keď spočítame počet trás dĺžky $d+1$, zapíšeme si do prislúchajúceho poľa túto hodnotu, a tú použijeme pri výpočte S_{d+2}.

Na záver prichádza rovnaké pozorovanie ako pri riešení dynamickým programovaním. Pri výpočte trás s dĺžkou $d+1$ do intervalového stromu S_{d+1} a $zacina_{d+1}$ využívame len výšky hôr a hodnoty v S_d a $zacina_d$. Takisto ako minule si teda vystačíme len s dvoma pármi štruktúr, ktoré budeme pri výpočtoch striedať.

Pamätáme si výšky všetkých hôr a počet trás končiacich v každej z nich v (dvoch) poliach `zacina` , ktoré majú počet prvkov lineárny od n. K tomu nám pribudli dva rovnako veľké intervalové stromy, ktorých veľkosť je

lineárna od najvyššej hory na vstupe. Ak by sme toto číslo mali uvedené na vstupe ako max (a teda rozlišovali sadu 4 od sád 1,2,3), naša pamäťová zložitosť by bola $O(n + max)$.

Namiesto spočítania trás začínajúc v menších horách napravo v čase lineárnom od n používame konštantne veľa volaní funkcii pridaj a spocitaj, ktorých časová zložitosť je $O(\log max)$. Výpočet robíme stále pre všetkých n hôr a opakujeme s-krát, postupne pre väčšie dĺžky trás. Vynulovanie polí **zacina** a intervalového stromu pri prestupovaní z dĺžky d na $d + 1$ robíme taktiež v lineárnom čase od ich veľkosti. Časová zložitosť je teda $O(s \cdot n \cdot \log max + max)$.

Listing programu (C++)

```
#include <bits/stdc++.h>

using namespace std;
struct interval
{ int l,r; };
//intervaly[x] nám povie ktoré najľavejšie a najpravejšie políčko je v podstrome x
vector<interval>intervaly;
vector<int>strom[2]; //intervalové stromy
int mod = 1000000007;

//prepočítajú sa hodnoty v strome 'ktory' na ceste z koreňa do 'kde'
void bubli(int ktory,int kde)
{
    if(!kde) return;
    strom[ktory][kde] = (strom[ktory][kde*2] + strom[ktory][kde*2+1]) % mod;
    bubli(ktory,kde/2);
}

//pridá intervalovému stromu 'ktory' na políčko 'kde' hodnotu 'v'
void pridaj(int ktory,int kde,int v)
{
    strom[ktory][kde] += v;
    //pridaním záporného čísla môžeme omylom prejsť k zápornému modulovanému súčtu
    if(strom[ktory][kde]<0) strom[ktory][kde] += mod;
    strom[ktory][kde] %= mod;
    bubli( ktory, (kde)/2 );
}

//spočíta súčet čísel v strome 'ktory' od políčka 'vlavo' po políčko 'vpravo'
int spocitaj(int vlavo,int vpravo,int ktory,int kde)
{
    if(vlavo>vpravo)
        return 0;

    //podstrom 'kde' vie súčet presne nášho intervalu
    if(intervaly[kde].l==vlavo && intervaly[kde].r==vpravo)
        return strom[ktory][kde];
    //v intervale lavého syna nás nič nezaujíma - zavoláme sa do pravého
    else if (intervaly[kde*2].r<vlavo)
        return spocitaj(vlavo,vpravo,ktory,kde*2+1);
    //v intervale pravého syna nás nič nezaujíma - zavoláme sa do ľavého
    else if (intervaly[kde*2+1].l>vpravo)
        return spocitaj(vlavo,vpravo,ktory,kde*2);
    //obaja synovia majú časť intervalu. Zavoláme sa na oboch
    else
        return (spocitaj(vlavo,intervaly[kde*2].r,ktory,kde) + spocitaj(intervaly[kde*2+1].l,vpravo,ktory,kde) ) % mod;

}
int main()
{
    //načítavame veľa čísel - zrýchlyme vstup a výstup
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);

    //MAX je najväčšia výška hory, velkost je počet listov v intervalovom strome
    int MAX = 1000000,n,m,s,i,j,velkost=1;
    cin >> n >> m >> s;

    //zacina[cislo%2][i] bude počet trás dĺžky 'cislo' začínajúc horou i
    int hory[n],zacina[2][n];
    for(i=0;i<n;++i)
    {
        cin >> hory[i];
        zacina[1][i] = 1; //každá hora je sama o sebe trasa dĺžky 1
    }

    while(velkost<MAX) velkost *= 2;

    strom[0].resize(velkost*2,0);
    strom[1].resize(velkost*2);

    //pripočítame trasu dĺžky 1 začínajúc s výškou hory[i]
    for(i=0;i<n;++i)
        pridaj(1,hory[i]+velkost,1);


    intervaly.resize(velkost*2);
    for(i=velkost;i<velkost*2;++i)
        intervaly[i].l = intervaly[i].r = i;
    for(i=velkost-1;i>0;i--)
    {
```

```
        //najlavejšie políčko podstromu i je najlavejšie políčko jeho ľavého syna
        intervaly[i].l = intervaly[i*2].l;
        //rovnaká úvaha pre najpravejšie políčko
        intervaly[i].r = intervaly[i*2+1].r;
    }

    for(i=1;i<s;++i)
    {
        int teraz = (i%2),potom = (i+1)%2;
        //vynulujeme strom do ktorého budeme počítať trasy dĺžky i+1
        for(j=1;j<velkost*2;++j)
            strom[potom][j] = 0;
        for(j=0;j<n;++j)
        {
            int index = hory[j]+velkost; //list predstavujúci výšku hory[j]
            //spočítame počet trás dĺžky i začínajúc v horách vysokých najviac hory[j]-m-1
            int sucet = spocitaj(velkost,index-m-1,teraz,1);
            pridaj(potom,index,sucet);
            zacina[potom][j] = sucet;
            /* odčítame zo stromu počet trás, ktoré prispievala naša hora,
            pretože berieme do úvahy len hory napravo od tej ktorú budeme počítať v ďalšiom kroku */
            pridaj(teraz,index,-zacina[teraz][j]);
        }

    }

    //počet trás dĺžky s začínajúc v ľubovoľnej výške je v koreni stromu[s%2]
    cout << strom[s%2][1] << "\n";

    return 0;
}
```

Ako na vysoké hory

S doterajším prístupom si na horách veľkosti nad 10^6 náš program vylámе zuby. Skúsme si tento problém s vysokými horami posunúť niekam inam – vytvorí nám to asi iný problém, a ten potom skúsime vyriešiť.

Zoraďme si teda hory podľa výšky. Prvej hore povieme, že je najnižšia, akoby s výškou 1. Nájdeme si druhú najnižšiu (ale nie s rovnakou výškou!). Tej povieme, že má výšku 2. Takto postupujeme, až každej hore priradíme novú výšku z rozsahu $[1, n]$. Tieto výšky sa nám už zmestia do intervalového stromu rovnako ako v minulom riešení. Hurá, hotovo!

Teda, skoro. V minulom riešení sme sa opierali o to, že z hory s výškou h_i vieme skočiť na všetky hory z výškou najviac $h_i - m - 1$, teda súčtom v intervalovom strome cez $spocitaj(S_d, 1, h_i - m - 1)$. Toto však už zďaleka nie je pravda. O ozajstných výškových rozdieloch hôr po prečíslovaní už nič nevieme. Hora s novou výškou 2 mohla byť len o jeden meter vyššia ako hora s novou výškou 1, ale rovnako pravdepodobne mohla byť vyššia o miliardu metrov. Po vyriešení veľkosti nášho intervalového stromu teda musíme vyriešiť iný problém – pri spracúvaní hory i potrebujeme efektívne povedať, ktorá je najvyššia hora ktorá má s h_i prevýšenie aspoň m. Označme si (novo pridelenú) výšku tejto hory inv_i.

Zoberme si naše hory, ktoré máme momentálne zoradené podľa (starej) výšky. Pozrime sa na tú, ktorá je v strede tohto poľa (nazvime ho C) – $C_{n/2}$. Ak je táto hora privysoká – $h_i - m \le C_{n/2}$ – určite sú aj hory napravo od nej ($C_{n/2+1}, ..., C_{n-1}$) príliš vysoké. Ak je však dostatočne nízka – $h_i - m > C_{n/2}$ určite sú aj všetky hory naľavo od nej ($C_0, ..., C_{n/2-1}$) dostatočne nízke. Určite už tušíte riešenie – v poli C v čase $O(\log n)$ binárne vyhľadáme najvyššiu horu, ktorá je dostatočne nízka, a zapíšeme si k príslušnej hore i hodnotu inv_i – novopriradenú výšku tejto hory (pripomíname, že tá je z rozsahu $[1, n]$). Toto zopakujeme pre každú horu, pridávajúc nami akceptovanú zložitosť $O(n \cdot \log n)$.

Už nám zafunguje naše riešenie rovnako ako v minulých sadách – pri počítaní $v(d+1, i)$ zavoláme spocitaj$(S_{d+1}, 0, inv_i)$ a nakoniec odčítame z S_d na pozící novej výšky hory i hodnotu $zacina_i$. Prečíslované výšky hôr si môžeme pamätať napríklad v hešovacej tabulke (`map` v C++) – jej zložitosť na vkladanie alebo pozeranie sa do nej je $O(\log k)$, kde k je počet prvkov v nej – prečíslovanie n hôr teda bude trvať najviac $O(n \log n)$ – a pozrieme sa do nej konštantne veľa krát pri spracovaní každej hory počas vypočítavania odpovede.

V pamäti máme teraz n výšok hôr, dve polia `zacina` a dva intervalové stromy (ktorých veľkosť po prečíslovaní výšok klesne na $O(n)$), a napokon pre každú horu záznam v hešovacej tabuľke a priradenú hodnotu inv_i. Celková pamäťová zložitosť nášho programu je teda $O(n)$.

Rovnako ako v predchádzajúcom riešení, práca s poliami `zacinaj`, intervalovými stromami a tentokrát aj s prvkami inv_i prebieha v lineárnom čase od ich veľkosti, ktorá je lineárna od n. Volania funkcii *spocitaj* a *pridaj*, ktoré obe používame $O(s \cdot n)$-krát, je $O(\log n)$ a teda s celkovou zložitosťou $O(s \cdot n \cdot \log n)$. Usporiadania poľa C a prečíslovanie výšok hôr pomocou hešovacej tabulky vieme v vykonať v $O(n \log n)$, zatiaľ čo $O(s \cdot n)$ nazretí do nej pri spracovávaní trás intervalovými stromami nám trvá $O(s \cdot n \cdot \log n)$. Horný odhad časovej zložitosti je teda $O(s \cdot n \cdot \log n)$.

Listing programu (C++)

```
#include <bits/stdc++.h>

using namespace std;
struct interval
{ int l,r; };
//intervaly[x] nám povie ktoré najľavejšie a najpravejšie políčko je v podstrome x
vector<interval>intervaly;
vector<int>strom[2]; //intervalové stromy
int mod = 1000000007;

//prepočítajú sa hodnoty v strome 'ktory' na ceste z koreňa do 'kde'
void bubli(int ktory,int kde)
{
    if(!kde) return;
    strom[ktory][kde] = (strom[ktory][kde*2] + strom[ktory][kde*2+1]) % mod;
    bubli(ktory,kde/2);
}

//pridá intervalovému stromu 'ktory' na políčko 'kde' hodnotu 'v'
void pridaj(int ktory,int kde,int v)
{
    strom[ktory][kde] += v;
    //pridaním záporného čísla môžeme omylom prejsť k zápornému modulovanému súčtu
    if(strom[ktory][kde]<0) strom[ktory][kde] += mod;
    strom[ktory][kde] %= mod;
    bubli( ktory, (kde)/2 );
}

int spocitaj(int vlavo,int vpravo,int ktory,int kde)
{
    //podstrom 'kde' vie súčet presne nášho intervalu
    if(intervaly[kde].l==vlavo && intervaly[kde].r==vpravo)
        return strom[ktory][kde];
    //v intervale lavého syna nás nič nezaujíma - zavoláme sa do pravého
    else if (intervaly[kde*2].r<vlavo)
        return spocitaj(vlavo,vpravo,ktory,kde*2+1);
    //v intervale pravého syna nás nič nezaujíma - zavoláme sa do ľavého
    else if (intervaly[kde*2+1].l>vpravo)
        return spocitaj(vlavo,vpravo,ktory,kde*2);
    //obaja synovia majú časť intervalu. Zavoláme sa na oboch
    else
        return (spocitaj(vlavo,intervaly[kde*2].r,ktory,kde) + spocitaj(intervaly[kde*2+1].l,vpravo,ktory,kde) ) % mod;

}
int main()
{

    //načítavame veľa čísel - zrýchlyme vstup a výstup
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);

    long long m;
    int n,s,i,j,velkost=1; //velkost bude počet listov intervalového stromu
    cin >> n >> m >> s;

    long long int hory[n],zacina[2][n],cisla[n+1],inv[n];
    for(i=0;i<n;++i)
    {
        cin >> hory[i];
        cisla[i] = hory[i];
        zacina[1][i] = 1;
    }
    //pridáme zarážku, ktorá bude tvoriť inverziu s každým prvkom
    cisla[n] = (long long)-1000000000000000047;
    sort(cisla,cisla+n+1);
    int pocet = 1;
    //prečíslujeme výšky hôr na rozumne malé čísla
    map<long long int,int> kompresia;
    kompresia[ cisla[0] ] = 0;
    for(i=1;i<=n;++i)
        if(cisla[i]!=cisla[i-1])
        {
            kompresia[ cisla[i] ] = pocet;
            pocet++;
        }

    //pre každú horu binárne vyhľadáme, ktorá v poradí hora s ňou tvorí dostatočne veľkú inverziu
    for(i=0;i<n;++i)
    {
        int lo = 0,hi = n+2;
        while(hi-lo>1)
        {
            int mid = (lo+hi)/2;
            if(hory[i]-m>cisla[mid]) lo = mid;
            else hi = mid;
        }
        inv[i] = kompresia[ cisla[lo] ];
    }

    while(velkost<pocet) velkost *= 2;

    strom[0].resize(velkost*2,0);
    strom[1].resize(velkost*2);
    //pripočítame trasu dĺžky 1 začínajúc s výškou hory[i]
    for(i=0;i<n;++i)
        pridaj(1,kompresia[ hory[i] ]+velkost,1);
```

```
    intervaly.resize(velkost*2);
    for(i=velkost;i<velkost*2;++i)
        intervaly[i].l = intervaly[i].r = i;
    for(i=velkost-1;i>0;i--)
    {
        //najlavejšie políčko podstromu i je najlavejšie políčko jeho ľavého syna
        intervaly[i].l = intervaly[i*2].l;
        //rovnaká úvaha pre najpravejšie políčko
        intervaly[i].r = intervaly[i*2+1].r;
    }

    for(i=1;i<s;++i)
    {
        int teraz = (i%2),potom = (i+1)%2;
         //vynulujeme strom do ktorého budeme počítať trasy dĺžky i+1
        for(j=1;j<velkost*2;++j)
            strom[potom][j] = 0;
        for(j=0;j<n;++j)
        {
            int index = kompresia[ hory[j] ] + velkost; //list predstavujúci výšku hory[j]
            /* spočítame počet trás dĺžky i začínajúc v horách nie vyšších ako tou,
               s ktorou tvorí hory[j] dostatočne veľkú inverziu */
            int sucet = spocitaj(velkost,inv[j]+velkost,teraz,1);
            pridaj(potom,index,sucet);
            zacina[potom][j] = sucet;
            /*odčítame zo stromu počet trás, ktoré vytvárala naša hora,
            pretože berieme do úvahy len hory napravo od tej ktorú budeme počítať v ďalšiom kroku */
            pridaj(teraz,index,-zacina[teraz][j]);
        }

    }

    //počet trás dĺžky s začínajúc v ľubovoľnej hore je v koreni stromu[s%2]
    cout << strom[s%2][1] << "\n";

    return 0;
}
```

8. Obmedzený pohyb

Baklažán
(max. 12 b za popis, 8 b za program)

Pretínanie úsečiek

Kľúčovou časťou tejto úlohy je zisťovanie, či sa dve úsečky pretínajú. Dve úsečky AB a CD sa pretínajú vtedy a len vtedy, keď úsečka AB pretína priamku CD a súčasne úsečka CD pretína priamku AB. To, že úsečka CD pretína priamku AB vlastne znamená, že body C a D ležia v opačných polrovinách vzhľadom na priamku AB. A to vieme ľahko skontrolovať pomocou vektorového súčinu[8] tak, že skontrolujeme, či majú súčiny $(B - A) \times (C - A)$ a $(B - A) \times (D - A)$ opačné znamienka (notáciou $Y - X$ tu myslíme vektor z X do Y). Keďže nás zaujíma iba pretínanie vo vnútorných (nie koncových) bodoch úsečiek, budeme vyžadovať, aby oba vektorové súčiny boli nenulové. To, či úsečka AB pretína priamku CD overíme úplne rovnako.

Listing programu (C++)

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

struct vektor {
  cislo x, y;

  vektor(cislo xx = cislo(0), cislo yy = cislo(0)) : x(xx), y(yy) {
  }

  vektor operator-(vektor prava_strana) {
    return vektor(x - prava_strana.x, y - prava_strana.y);
  }

  cislo operator*(vektor prava_strana) {      //vektorovy sucin
    return x * prava_strana.y - y*prava_strana.x;
  }
};

int signum(cislo a) {      //vrati "znamienko" cisla
  if(a < 0)return -1;
  if(a > 0)return 1;
  return 0;
}

struct usecka {
  vektor zaciatok, koniec;
};

bool pretina_priamku_usecky(usecka *u, usecka *p) {      //pretina usecka u priamku, na ktorej lezi usecka p?
  vektor v = p->koniec - p->zaciatok;
```

[8] viac o vektorovom súčine a jeho využití si môžete prečítať na https://www.ksp.sk/kucharka/skalarny_a_vektorovy_sucin

```
  return signum(v * (u->zaciatok - p->zaciatok)) * signum(v * (u->koniec - p->zaciatok)) == -1;
}

bool pretinaju_sa(usecka *u1, usecka *u2) {
  return pretina_priamku_usecky(u1, u2) && pretina_priamku_usecky(u2, u1);
}
```

Jednoduché riešenie

Keď už vieme v čase $O(1)$ overovať, či sa dve úsečky pretínajú, ľahko napíšeme riešenie s časovou zložitosťou $O(n \cdot m)$ – pre každú jamu sa pozrieme na všetky dosky a tie, ktoré pretína, vypíšeme.

Vzorové riešenie

Dá sa to však aj rýchlejšie, konkrétne v čase $O((n+m)\log(n+m))$. Použijeme techniku s názvom *zametanie*, ktorá sa v geometrii v rôznych obmenách používa pomerne často.

Pre jednoduchosť budeme chvíľu predpokladať, že na vstupe sa nevyskytujú zvislé úsečky. Nakoniec si ukážeme, ako sa vieme vysporiadať aj s nimi.

Vezmeme myslenú zvislú priamku (ďalej ju budeme volať zametacia priamka), ktorú na začiatku umiestnime dostatočne ďaleko naľavo, tak aby bola naľavo od všetkých úsečiek, ktoré sme dostali na vstupe. Postupne posúvame zametaciu priamku doprava, až kým ju nedostaneme napravo od všetkých úsečiek na vstupe. Počas tohto posúvania si vo vhodnej dátovej štruktúre udržiavame zoznam úsečiek, ktoré momentálne zametaciu priamku pretínajú, usporiadaný podľa y-ovej súradnice bodu pretnutia (teda úsečka, ktorej priesečník s našou priamkou je najnižšie, bude v zozname prvá a úsečka, ktorá zametaciu priamku pretína najvyššie bude posledná). Keďže v skutočnosti nás zaujímajú iba momenty, keď sa niečo sa deje s týmto zoznamom, stačí nám postupne odsimulovať všetky udalosti, pri ktorých sa zoznam mení (pohyb priamky medzi týmito udalosťami je aj tak nuda).

Zoznam sa bude meniť pri troch druhoch udalostí:

1. Zametacia priamka „narazila" na ľavý koniec nejakej novej úsečky. Pri tejto udalosti treba danú úsečku pridať do zoznamu na správne miesto, keďže odteraz bude našu priamku pretínať aj ona.

2. Zametacia priamka „narazila" na pravý koniec nejakej úsečky. Vtedy treba danú úsečku zo zoznamu vymazať, keďže odteraz už zametaciu priamku pretínať nebude.

3. Zametacia priamka „narazila" na priesečník dvoch úsečiek. Vtedy treba tieto dve úsečky v zozname vymeniť, keďže tá, ktorá doteraz pretínala zametaciu priamku nižšie ju bude odteraz pretínať vyššie a naopak.

Na to, aby sme udalosti mohli spracúvať, musíme, samozrejme, vedieť, kedy nastanú. Za týmto účelom budeme mať haldu (alebo inú prioritnú frontu), na ktorej vrchu bude prvok s najnižšou x-ovou súradnicou. Do tejto haldy budeme dávať všetky budúce udalosti, o ktorých už vieme, kedy nastanú. Z nej budeme vždy vedieť vybrať najbližšiu udalosť, ktorú máme spracovať.

Už pri načítaní vstupu vieme povedať, kedy (teda pri akej x-ovej súradnici zametacej priamky) nastanú udalosti prvých dvoch druhov. Tieto udalosti si teda môžeme rovno nasypať do haldy. S udalosťami tretieho typu bude trochu väčší problém, keďže na začiatku netušíme, kde sa jednotlivé úsečky pretínajú. Našťastie nás nič nenúti hádzať všetky tieto udalosti do haldy hneď na začiatku, môžeme ich tam pridávať „za behu". Jediným obmedzením je, že každú udalosť musíme do haldy pridať skôr, než nastane (inak by sme nevedeli, že ju máme odsimulovať).

Všimnime si, že keď zametacia priamka ide naraziť na priesečník dvoch úsečiek, tesne predtým sú tieto úsečky v zozname úsečiek pretínajúcich zametaciu priamku hneď vedľa seba. Ak by sme teda pri každej zmene zoznamu celý tento zoznam prešli a pre každú dvojicu susedných úsečiek skontrolovali, či sa niekedy v budúcnosti nepretnú (a ak áno, pridali by sme túto udalosť do haldy), určite by sme žiaden priesečník nevynechali. To by, samozrejme, bolo dosť neefektívne, keďže niektoré dvojice úsečiek by sme takto zbytočne kontrolovali veľakrát. V skutočnosti sa nám stačí po každej zmene v zozname pozrieť na tie dvojice úsečiek, ktoré pred zmenou neboli susedné a po zmene sú. Konkrétne:

1. Po pridaní úsečky do zoznamu sa treba pozrieť, či sa pretne so svojím spodným susedom (ak nejakého má) a či sa pretne s vrchným susedom.

2. Po zmazaní úsečky zo zoznamu sa treba pozrieť, či sa jej niekdajší susedia (ktorí odteraz susedia navzájom) pretnú.

3. Po výmene dvoch úsečiek sa treba pozrieť, či sa tieto dve úsečky pretnú so svojimi novými susedmi (tá, ktorá bola pred výmenou nižšie, má teraz nového horného suseda a tá, ktorá bola vyššie, má nového spodného suseda).

Pri spracovaní ľubovoľnej udalosti sa teda pozrieme na $O(1)$ (jednu alebo dve) novovzniknutých dvojíc susedných úsečiek. Keď o nejakej dvojici zistíme, že sa pretne, poznačíme si to. Ak sme navyše doteraz o tomto priesečníku nevedeli, pridáme ho do haldy. Takýmto spôsobom zaručene nevynecháme žiadnu udalosť, ani žiadnu nespracujeme dvakrát. Navyše popri tom nájdeme všetky priesečníky, čo je presne to, čo potrebujeme.

Zvislé úsečky

Ostáva ešte doriešiť niekoľko detailov. Prvým z nich sú zvislé úsečky. Tie môžeme ošetriť napríklad tak, že zavedieme štvrtý typ udalosti: „narazenie" zametacej priamky na zvislú úsečku. O týchto udalostiach vieme už po načítaní vstupu povedať, kedy nastanú. Spracovať ich môžeme tak, že v zozname úsečiek pretínajúcich zametaciu priamku binárne vyhľadáme prvú a poslednú úsečku, ktorá danú zvislú úsečku pretína a pre všetky úsečky v zozname medzi nimi si poznačíme priesečník. Zoznam pritom nemeníme, keďže hneď, ako sa zametacia priamka pohne, zoznam sa vráti do pôvodného stavu.

Súčasné udalosti

Druhým problémom je, čo budeme robiť, keď naraz nastane viac ako jedna udalosť. Odpoveď je jednoduchá: odsimulujeme ich jednu po druhej, v ľubovoľnom poradí. Keďže máme zaručené, že úsečky sa pretínajú vždy len vo vnútorných bodoch a v žiadnom bode sa nepretínajú viac ako dve úsečky, udalosti nastávajúce naraz sa nemôžu ovplyvňovať.

Poloha priesečníka

Tretí detail je, že počas zametania budeme občas potrebovať pre dané dve úsečky zistiť, kde (na akej x-ovej súradnici) sa pretnú. Už sme si ukázali, ako zistiť, *či* sa pretnú, neukázali sme si však, ako zistiť *kde* sa pretnú. Priesečník dvoch úsečiek (ak existuje) je priesečník priamok, na ktorých tieto úsečky ležia. Rovnice týchto priamok vieme zapísať v tvare $a_1x + b_1y + c_1 = 0$ a $a_2x + b_2y + c_2 = 0$. Ak (x, y) je ich priesečník, potom musia x a y spĺňať rovnice oboch priamok. Máme teda sústavu dvoch rovníc o dvoch neznámych, ktorej riešením dostaneme

$$x = \frac{b_1c_2 - b_2c_1}{b_2a_1 - b_1a_2}.$$

Ak máme úsečku zadanú jej koncovými bodmi $(x_1, y_1), (x_2, y_2)$, parametre a, b, c jej rovnice vieme dopočítať tak, aby jej oba z bodov (x_1, y_1) aj (x_2, y_2) vyhovovali a dostaneme niečo ako

$$a = y_2 - y_1,$$
$$b = x_1 - x_2,$$
$$c = -(ax_1 + by_1).$$

Dátová štruktúra zoznamu

Štvrtým (a najväčším) detailom je voľba vhodnej dátovej štruktúry pre zoznam úsečiek pretínajúcich zametaciu priamku. Vhodnou štruktúrou sa ukazujú byť vyvažované binárne vyhľadávacie stromy (v angličnite Binary Search Trees, skratka BST). Tie nám totiž umožňujú mať usporiadaný zoznam prvkov, v ktorom sa dá vyhľadávať, pridať prvok na ľubovoľnú pozíciu aj zmazať ľubovoľný prvok v čase $O(\log N)$. V C++ sú vyhľadávacie stromy implementované v `std::set` a táto implementácia sa dá v riešení použiť, je to však dosť bolestivé a silno neodporúčané. Problém je v tom, že `set`e treba povedať, ako má prvky porovnávať a to sa v našom prípade v čase mení – vždy, keď zametacia priamka prejde nejakým priesečníkom. Preto je veľmi náročné udržať `set` v konzistentnom stave.

Lepšia voľba je napísať si vlastný BST, na naše účely sa dobre hodí napríklad treap[9] s implicitnými kľúčmi. Táto dátová štruktúra sa správa v princípe ako obyčajné pole (teda prvky sú číslované indexami $0, 1, \ldots, N-1$) s tým rozdielom, že prečítanie prvku na danom indexe trvá čas $O(\log N)$ namiesto obvyklých $O(1)$ pri poli. Na oplátku nám však umožňuje nasledujúce operácie:

- Vymazanie prvku z daného indexu v čase $O(\log N)$ (ostatné prvky sa automaticky prečíslujú).

[9] viac o treapoch na https://www.ksp.sk/kucharka/treap

- Vloženie prvku na daný index v čase $O(\log N)$ (ostatné prvky sa automaticky prečíslujú).
- Ak si zapamätáme referenciu (pointer) na niektorý prvok, vieme neskôr v čase $O(\log N)$ zistiť, aký má index (aj ak sa index medzičasom zmenil). To sa nám obzvlášť hodí pri udalostiach 3. typu (vieme si zapamätať, ktoré dva prvky budeme chcieť vymeniť a keď daná udalosť nastane, vieme ich efektívne nájsť).
- Ak sú v nejakom momente všetky prvky zoznamu podľa niečoho usporiadané, vieme v ňom vďaka stromovej štruktúre binárne vyhľadávať v čase $O(\log N)$, rovnako ako v obyčajnom poli.

Presnosť

Posledný detail je presnosť výpočtov. Jednou z možností je nenechať nič na náhodu a všetko počítať v racionálnych číslach. Druhá, na implementáciu jednoduchšia možnosť je použiť reálnu aritmetiku. Vtedy ale treba použiť typ premennej s dostatočnou presnosťou (v C++ v závislosti od implementácie mohol stačiť `double`, alebo mohlo byť nutné použiť `long double`).

Odhad zložitosti

Zamyslime sa, koľko udalostí budeme musieť počas zametania spracovať. Keďže na vstupe je $n+m$ úsečiek, udalostí prvého druhu bude najviac $n+m$, rovnako udalostí druhého a štvrtého druhu. Keďže každá doska sa pretína s práve jednou jamou, úsečky na vstupe majú dokopy m priesečníkov, teda budeme mať m udalostí tretieho druhu. Dokopy teda máme $O(n+m)$ udalostí. Udalosti prvých troch druhov vieme spracúvať v čase $O(\log(n+m))$ (vybranie udalosti z haldy, konštantne veľa operácii so zoznamom úsečiek pretínajúcich zametaciu priamku, prípadné pridanie konštantného počtu udalostí na haldu). S udalosťami štvrtého druhu (zvislými úsečkami) je to trochu horšie. Ak má daná zvislá úsečka p priesečníkov, jej spracovanie môže trvať až $O((p+1)\log(n+m))$. Vieme však, že na vstupe je dokopy m priesečníkov, teda súčet p-čiek pre všetky zvislé úsečky dokopy je najviac m. To znamená, že spracovanie všetkých udalostí štvrtého druhu nám zaberie dokopy $O((n+m)\log(n+m))$ času. Spracovanie ostatných udalostí tiež trvá $O((n+m)\log(n+m))$, teda aj celé zametanie bude trvať $O((n+m)\log(n+m))$. Načítanie vstupu a nahádzanie udalostí do haldy trvá $O((n+m)\log(n+m))$ času, vypisovanie výstupu trvá tiež $O((n+m)\log(n+m))$ (lebo ho pred vypísaním potrebujeme triediť), celková časová zložitosť nášho algoritmu je teda $O((n+m)\log(n+m))$.

Pamäťová zložitosť je $O(n+m)$.

Listing programu (C++)

```
struct uzol;
typedef pair<uzol*, uzol*> puu;

//časť zdrojového kódu, ktorá je uvedená v predošlom listingu, už neuvádzame. Výnimkou
//je definícia štruktúry usecka, do ktorej sme nejaké veci pridali.

struct usecka {
  vektor zaciatok, koniec;
  int id;
  vector<usecka*> koho_pretinam;

  usecka(vektor z, vektor k, int i = -1) : zaciatok(z), koniec(k), id(i) {
    if(make_pair(zaciatok.x, zaciatok.y) > make_pair(koniec.x, koniec.y)) swap(zaciatok, koniec);
  }

  bool zvisla() {
    return zaciatok.x == koniec.x;
  }

  cislo y_na(cislo x) {
    return (x - zaciatok.x) * (koniec.y - zaciatok.y) / (koniec.x - zaciatok.x) + zaciatok.y;
  }
};

struct priamka {
  cislo a, b, c;

  priamka(usecka *u) {
    a = u->koniec.y - u->zaciatok.y;
    b = u->zaciatok.x - u->koniec.x;
    c = 0 - (a*u->zaciatok.x + b*u->zaciatok.y);
  }
};

cislo priesecnik_x(usecka *u1, usecka *u2) {
  priamka p1(u1), p2(u2);
  return (p1.b * p2.c - p2.b * p1.c) / (p2.b * p1.a - p1.b * p2.a);
}


int velkost_podstromu(uzol *u);

struct uzol {
```

```
  uzol *lavy, *pravy, *otec;
  usecka *hodn;
  int priorita;
  int podstrom;

  uzol(usecka *u = NULL) : hodn(u), lavy(NULL), pravy(NULL), otec(NULL), priorita(rand()), podstrom(1) {
  }

  void reinit() {
    lavy = pravy = otec = NULL;
    priorita = rand();
    podstrom = 1;
  }

  int update() {
    podstrom = 1 + velkost_podstromu(lavy) + velkost_podstromu(pravy);
  }

  void nastav_lavy(uzol *u) {
    lavy = u;
    if(u != NULL) u->otec = this;
    update();
  }

  void nastav_pravy(uzol *u) {
    pravy = u;
    if(u != NULL) u->otec = this;
    update();
  }

  int index() {
    if(otec == NULL) return velkost_podstromu(lavy);
    int oi = otec->index();
    if(otec->lavy == this) return oi - 1 - velkost_podstromu(pravy);
    return oi + 1 + velkost_podstromu(lavy);
  }
};

int velkost_podstromu(uzol *u) {
  return u == NULL ? 0 : u->podstrom;
}

puu split(uzol *koren, int kde) {
  if(koren == NULL)return puu((uzol*)NULL, (uzol*)NULL);
  if(velkost_podstromu(koren->lavy) >= kde) {
    puu pom = split(koren->lavy, kde);
    koren->nastav_lavy(pom.second);
    return puu(pom.first, koren);
  }
  puu pom = split(koren->pravy, kde - velkost_podstromu(koren->lavy) - 1);
  koren->nastav_pravy(pom.first);
  return puu(koren, pom.second);
}

uzol* merge(uzol *a, uzol *b) {
  if(a != NULL) a->otec = NULL;
  if(b != NULL) b->otec = NULL;
  if(a == NULL) return b;
  if(b == NULL) return a;
  if(a->priorita > b->priorita) {
    a->nastav_pravy(merge(a->pravy, b));
    return a;
  }
  b->nastav_lavy(merge(a, b->lavy));
  return b;
}

uzol* insert(uzol *koren, int kde, uzol *u) {
  if(koren == NULL)return u;
  if(koren->priorita < u->priorita) {
    puu pom = split(koren, kde);
    u->nastav_lavy(pom.first);
    u->nastav_pravy(pom.second);
    return u;
  }
  if(kde <= velkost_podstromu(koren->lavy)) {
    koren->nastav_lavy(insert(koren->lavy, kde, u));
    return koren;
  }
  koren->nastav_pravy(insert(koren->pravy, kde - 1 - velkost_podstromu(koren->lavy), u));
  return koren;
}

uzol* erase(uzol *koren, int kde, bool znic = 1) {
  if(koren == NULL)return NULL;
  int ls = velkost_podstromu(koren->lavy);
  if(kde < ls) {
    koren->nastav_lavy(erase(koren->lavy, kde, znic));
    return koren;
  }
  if(kde > ls) {
    koren->nastav_pravy(erase(koren->pravy, kde - 1 - velkost_podstromu(koren->lavy), znic));
    return koren;
  }
  uzol *res = merge(koren->lavy, koren->pravy);
  if(znic) delete koren;
  else koren->reinit();
  return res;
}
```

```
uzol* get(uzol *koren, int kde) {
  if(koren == NULL)return NULL;
  int ls = velkost_podstromu(koren->lavy);
  if(kde < ls)return get(koren->lavy, kde);
  if(kde > ls)return get(koren->pravy, kde - ls - 1);
  return koren;
}

int index_of(uzol *koren, int y, int x) {
  if(koren == NULL)return 0;
  cislo ry = koren->hodn->y_na(x);
  if(y > ry) return index_of(koren->pravy, y, x) + velkost_podstromu(koren->lavy) + 1;
  return index_of(koren->lavy, y, x);
}

enum typ_udalosti{prisla_usecka, odisla_usecka, zvisla_usecka, vymena_useciek};

struct udalost {
  typ_udalosti co;
  usecka *u;
  uzol *n1, *n2;

  udalost(typ_udalosti c, usecka* us = NULL, uzol* no1 = NULL, uzol* no2 = NULL) : co(c), u(us), n1(no1), n2(no2) {
  }
};

void pretni(usecka* u1, usecka* u2) {
  u1->koho_pretinam.push_back(u2);
  u2->koho_pretinam.push_back(u1);
}

void over_koliziu(uzol *n1, uzol *n2,
         priority_queue<pair<cislo, udalost* >, vector<pair<cislo, udalost*> >, greater<pair<cislo, udalost*> > >& halda) {
  if(n1 == NULL || n2 == NULL)return;
  usecka *u1 = n1->hodn, *u2 = n2->hodn;
  if(u1->koho_pretinam.size() > 0 && u2->koho_pretinam.size() > 0)return;
  if(pretinaju_sa(u1, u2)) {
    halda.push(pair<cislo, udalost*> (priesecnik_x(u1, u2), new udalost(vymena_useciek, NULL, n1, n2)));
    pretni(u1, u2);
  }
}

int main() {
  int n, m;
  scanf("%d %d",&n,&m);
  vector<usecka*> objekt;
  priority_queue<pair<cislo, udalost* >, vector<pair<cislo, udalost*> >, greater<pair<cislo, udalost*> > > halda;
  for(int i=0; i<n + m; i++) {
    int x1, y1, x2, y2;
    scanf("%d %d %d %d", &x1, &y1, &x2, &y2);
    usecka *u = new usecka(vektor(x1, y1), vektor(x2, y2), i-n);
    objekt.push_back(u);
    if(u->zvisla()) halda.push(pair<cislo, udalost*> (u->zaciatok.x, new udalost(zvisla_usecka, u)));
    else {
      halda.push(pair<cislo, udalost*> (u->zaciatok.x, new udalost(prisla_usecka, u)));
      halda.push(pair<cislo, udalost*> (u->koniec.x, new udalost(odisla_usecka, u)));
    }
  }

  uzol *metla = NULL;

  while(!halda.empty()) {
    pair<cislo, udalost*> akt = halda.top();
    halda.pop();
    cislo x = akt.first;
    udalost *ud = akt.second;
    switch(ud->co) {
      case prisla_usecka: {
        int y = ud->u->zaciatok.y;
        int ind = index_of(metla, y, x);
        uzol *cr = new uzol(akt.second->u);
        metla = insert(metla, ind, cr);
        uzol *prev = get(metla, ind-1);
        uzol *nxt = get(metla, ind+1);
        over_koliziu(prev, cr, halda);
        over_koliziu(cr, nxt, halda);
        break;
      }
      case odisla_usecka: {
        int y = ud->u->koniec.y;
        int ind = index_of(metla, y, x);
        metla = erase(metla, ind);
        uzol *prev = get(metla, ind-1);
        uzol *nxt = get(metla, ind);
        over_koliziu(prev, nxt, halda);
        break;
      }
      case zvisla_usecka: {
        int y1 = ud->u->zaciatok.y, y2 = ud->u->koniec.y;
        int i1 = index_of(metla, y1, x), i2 = index_of(metla, y2, x);
        for(int i=i1; i<i2; i++) {
          pretni(get(metla, i)->hodn, ud->u);
        }
        break;
      }
      case vymena_useciek: {
        uzol *n1 = ud->n1, *n2 = ud->n2;
        int i1 = n1->index(), i2 = n2->index();
```

```
        metla = erase(metla, i1, false);
        metla = erase(metla, i1, false);
        metla = insert(metla, i1, n1);
        metla = insert(metla, i1, n2);
        uzol *prev = get(metla, i1-1);
        uzol *nxt = get(metla, i2+1);
        over_koliziu(prev, n2, halda);
        over_koliziu(n1, nxt, halda);
        break;
      }
    }
  }

  for(int i=0; i<n; i++) {
    vector<int> pom;
    for(int j=0; j<objekt[i]->koho_pretinam.size(); j++) {
      pom.push_back(objekt[i]->koho_pretinam[j]->id+1);
    }
    sort(pom.begin(), pom.end());
    for(int j=0; j<pom.size(); j++) {
      printf("%d", pom[j]);
      if(j+1 < pom.size())printf(" ");
    }
    printf("\n");
    delete objekt[i];
  }
  return 0;
}
```