Korešpondenčný seminár z programovania
XXXV. ročník, 2017/18
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. kola letnej časti

Kubík (kubik@ksp.sk)

1. Treba začať kúriť

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

Úlohou bolo zo sáhovice dĺžky $L = 2^n$ nalámať také kusy, aby sme si vedeli vyskladať kôpku, resp. sáhovicu dĺžky k.

Najdôležitejším pozorovaním je fakt, že lámaním sáhovice na polovice vieme postupne dostať sáhovice všetkých dĺžok mocniny dvojky $(1, 2, 4, 8, \ldots)$.

Potrebujeme teda číslo k vyskladať iba z mocnín dvojky. Jeden z takýchto rozkladov dostaneme, ak sa pozrieme na zápis čísla k v dvojkovej sústave[1]. Za každú jednotku v tomto zápise vezmeme mocninu dvojky prislúchajúcu danému rádu. Napríklad ak $k = 22$, dostaneme:

$$k = 22 = 10110_2 = 2^4 + 2^2 + 2^1 = 16 + 4 + 2.$$

Takýto rozklad čísla na súčet mocnín dvojky má jednu peknú vlastnosť: každá mocnina dvojky sa v ňom vyskytuje nanajvýš raz. Ak by teda chcel Lomižaba nalámať sáhovicu na takéto kusy, mohol by to robiť nasledujúcim algoritmom:

0. Ak $k = L$, netreba nič lámať, môže skončiť.
1. Kým nemá kus dĺžky 1, láme vždy najkratší kus dreva, ktorý práve má.
2. Po skončení kroku 1 má po jednom kuse dreva dĺžok $\frac{L}{2}, \frac{L}{4}, \frac{L}{8}, \ldots, 8, 4, 2$ a dva kusy dĺžky 1. Keďže z každej mocniny dvojky má aspoň jeden kus, môže spomedzi nich vybrať tie, ktoré potrebuje.

Keďže na začiatku má najkratší (a jediný) kus dreva dĺžku $L = 2^n$ a každým zlomením sa najkratší kus skráti na polovicu, v kroku 1. tohto algoritmu potrebuje Lomižaba n lámaní. Ak však binárny zápis čísla k končí nulami (ich počet označme x), nepotrebujeme x najmenších mocnín dvojky. V kroku 1. nášho algoritmu teda môže Lomižaba prestať lámať už v momente, keď má najkratší kus dreva dĺžku 2^x (najmenšia mocnina dvojky, ktorú potrebujeme), čím ušetrí posledných x lámaní. Stačí mu teda $n - x$ lámaní.

Nedá sa to však nejakým iným spôsobom na menej? Ukážeme, že nie. Z toho, že binárny zápis čísla k končí x nulami, vyplýva, že číslo k je deliteľné 2^x. Z toho, že pred týmito x nulami je už jednotka, vyplýva, že k nie je deliteľné 2^{x+1}. Z mocnín dvojky (ostro) väčších ako 2^x sa dajú vyskladať iba násobky čísla 2^{x+1}, keďže všetky tieto mocininy sú násobkami 2^{x+1}. Na vyskladanie sáhovice dĺžky k teda nutne potrebujeme aspoň jeden kus dreva dlhý 2^x, alebo kratší. A na to, aby sme z kusu dĺžky $L = 2^n$ získali kus dĺžky 2^x (alebo kratší) potrebujeme aspoň $n - x$ lámaní.

Na vyriešenie našej úlohy nám teda stačí zistiť, koľkými nulami končí binárny zápis čísla k a vypísať n mínus tento počet. Ako tento počet núl zistíme? Keď je na konci čísla v binárnom zápise nula, znemaná to, že je deliteľné 2. Ak sú tamm dve nuly, tak je deliteľné 4. Vo všeobecnosti ak je tam x núl, tak dané číslo je deliteľné 2^x. Stačí nám teda zistiť, koľkokrát sa dá k bezo zvyšku vydeliť dvomi.

Listing programu (Python)

```
#!/usr/bin/env python3

t = int(input())
for _ in range(t):
    n, k = map(int, input().split())
    # počet nepotrebných lámaní
    nelam = 0

    # kontrolujeme, či sa na poslednej pozícii binárnej
    # reprezentácie k nachádza 0
    while k % 2 == 0:
        nelam += 1
        k //= 2
    print(n - nelam)
```

[1] https://sk.wikipedia.org/wiki/Dvojkov%C3%A1_%C4%8D%C3%ADseln%C3%A1_s%C3%BAstava

Listing programu (C++)

```
#include<cstdio>

int main(){
    int t;
    scanf("%d ",&t);
    for(int i=0; i<t; i++){
        long long n, k;
        scanf("%lld %lld ", &n, &k);
        long long nelam = 0;
        while(k%2==0){
            nelam ++;
            k/=2;
        }
        printf("%lld\n", n-nelam);
    }
}
```

Keďže binárny zápis čísla k má $\lfloor \log_2(k) \rfloor + 1$ cifier (bitov), časová zložitosť jednej otázky je $O(\log(k))$ - pretože v najhoršom prípade musíme skontrolovať $\log_2(k)$ bitov. Pamäťová zložitosť je $O(1)$.

Bonus pre fajnšmekrov: Existuje aj nasledujúce riešenie v $O(1)$:

Listing programu (C++)

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

using namespace std;

long long n, k;
int t;

int main() {
  scanf("%d", &t);
  while (t--) {
    scanf("%lld %lld", &n, &k);
    long long mnau = k-1;
    long long ans = __lg( k ^ (k & mnau) );

    printf("%lld\n", n - ans);
  }
  return 0;
}
```

Toto riešenie by bolo tiež logaritmické, no procesory majú inštrukciu pre funkciu `__lg` (dvojkový logaritmus – konkrétne pozícia najvyššej jednotky), ktorá umožnuje vypočítať túto hodnotu v $O(1)$. Upozornenie: funkcie s __ pred menom sú interné funkcie kompilátora, to znamená, že ich implementácia sa môže líšiť od kompilátora ku kompilátoru, a teda nie je ju moc dobré používať v kóde, ktorý bude reálne niekde bežať.

2. Uzimený večer

Aja (`aja@ksp.sk`)
(max. 12 b za popis, 8 b za program)

O farbách políčok

Políčka, ktoré majú mať rovnakú farbu ako políčko v ľavom hornom rohu, budeme v tomto vzoráku volať *biele* a políčka, ktoré majú mať opačnú farbu, budeme volať *čierne*. Ak si očíslujeme riadky a stĺpce šachovnice číslami 0 až $n-1$ tak, že ľavý horný roh má súradnice $(0,0)$ a pravý dolný roh má súradnice $(n-1, n-1)$, biele políčka budú mať párny súčet súradníc a čierne nepárny[2]. Ak teda chceme o nejakom políčku zistiť, či je biele alebo čierne, stačí sa pozrieť na súčet jeho súradníc.

Hrubá sila

Našim cieľom bolo nájsť čo najmenej políčok, ktoré treba zmeniť na to, aby bola šachovnica len dvojfarebná. Môžeme teda vyskúšať všetky možnosti – pre všetky možnosti, na ktorú farbu nakoniec prefarbíme biele políčka vyskúšame všetky možnosti, na ktorú farbu prefarbíme čierne políčka. Pre každú takúto dvojicu farieb si prejdeme celú šachovnicu a spočítame, koľko políčok by sme museli prefarbiť. Zapamätáme si tú možnosť, pre ktorú sme toho museli prefarbiť najmenej.

Možností, ktorú farbu použijeme na biele políčka je n^2, pre každú z nich je n^2-1 možných farieb pre čierne. Skúšame teda $n^2 \cdot (n^2-1)$ možností, čo je $O(n^4)$. Pre každú možnosť prechádzame celú šachovnicu veľkosti n^2, časová zložitosť algoritmu je teda $O(n^6)$. Musíme si pamätať celú šachovnicu a zopár premenných navyše, takže naša pamäťová zložitosť je $O(n^2)$.

[2] Pri číslovaní od jednotky to platí tiež.

Polohrubá sila

Aby sme nemuseli pre každú dvojicu farieb prechádzať celú šachovnicu, môžeme si pre každú z n^2 farieb spočítať, koľkokrát sa nachádza na bielom políčku a koľkokrát na čiernom políčku. Keď potom chceme pre nejakú dvojicu farieb zistiť, koľko políčok by sme museli prefarbiť, stačí nám od počtu všetkých políčok (n^2) odrátať počet bielych políčok, ktoré už majú správnu farbu a počet čiernych políčok, ktoré už majú správnu farbu.

Časová zložitosť sa nám zníži na $O(n^4)$. Pôvodnú šachovnicu si už pamätať nepotrebujeme, musíme si však pamätať počet výskytov na bielych a čiernych políčkach pre každú z n^2 farieb. Pamäťová zložitosť teda ostáva $O(n^2)$.

Listing programu (C++)

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

using namespace std;

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

    //pocty[x][0] = kolko krat bola farba x na 'bielom' policku, pocty[x][1] = na ciernom
    vector< vector<int> > pocty(n*n,{0,0});

    for(int i=0;i<n;++i)
    {
        for(int j=0;j<n;++j)
        {
            int x,farba = (i+j)%2;
            cin >> x;
            pocty[x][farba]++;
        }
    }

    int best = n*n-1;

    for(int biela=0;biela<n*n;++biela)
    {
        for(int cierna=0;cierna<n*n;++cierna)
        {
            if(biela == cierna)
                continue; //nemozeme celu sachovnicu premalovat rovnakou farbou

            best = min(best,n*n-pocty[biela][0]-pocty[cierna][1]);
        }
    }

    cout << best << "\n";

    return 0;
}
```

Vzorové riešenie

Jedna farba

Pozrime sa najprv na jednoduchší prípad, v ktorom by jedna farba šachovnice bola už opravená, napríklad biela, a my by sme potrebovali opraviť tú druhú.

Máme na šachovnici na čiernych políčkach rôzne farby a chceme vybrať jednu, na ktorú potom prefarbíme všetky ostatné čierne políčka. Z toho je zrejmé, že ak vyberieme farbu, ktorá sa na čiernych políčkach vyskytuje najviac, budeme potrebovať najmenej prefarbovania. Riešením by bolo zrátať pre každú farbu, koľkokrát sa na čiernych políčkach vyskytuje a následne vybrať farbu s maximálnym výskytom (ale nie tú, ktorou sú vyfarbené biele políčka). Stačí potom ostatné políčka prefarbiť na ňu.

Celá šachovnica

Ak by sme chceli predchádzajúce riešenie použiť na celú šachovnicu, narazíme na jeden problém. Mohlo by sa nám stať, že pre biele aj čierne políčka šachovnice vyberieme rovnakú farbu. To by ale potom nebola šachovnica, lebo by bola celá jednofarebná. Tento problém vyriešime tak, že si pre biele aj čierne políčka nájdeme dve najčastejšie sa vyskytujúce farby. Ak sa nám potom stane, že najčastejšie sa vyskytujúca farba je pre oba typy políčok rovnaká, pre jeden z nich využijeme druhú najčastejšiu farbu.

Máme pritom dve možnosti: buď vezmeme najlepšiu farbu pre biele políčka a druhú najlepšiu pre čierne, alebo najlepšiu pre čierne a druhú najlepšiu pre biele. Vyskúšame teda obe z nich a vyberieme si tú lepšiu.

Implementácia

Najprv chceme hľadať najčastejšiu farbu pre biele aj čierne políčka. Mohli by sme pre každú farbu prejsť celú

šachovnicu a spočítať, koľkokrát sa vyskytuje na oboch typoch políčok. Ak by ale malo každé políčku inú farbu, trvalo by nám to až $O(n^2 \cdot n^2)$. Vytvoríme si preto pole veľkosti n^2, v ktorom si budeme pre jednotlivé farby pamätať, na koľkých bielych políčkach sa vyskytujú. Také isté pole si vytvoríme aj pre výskyty na čiernych políčkach. Teraz nám stačí jedno prejdenie šachovnice. Pri každom políčku sa pozrieme, či je to biela alebo čierna pozícia a potom do vhodného prvku príslušného poľa prirátame jedna. Po prejdení celej šachovnice máme zrátané výskyty všetkých farieb na oboch typoch pozícií.

Potrebujeme teraz nájsť dve najväčšie hodnoty v oboch poliach. Mohli by sme naše polia utriediť a potom použiť najväčšie hodnoty. To by nám ale trvalo až $O(n^2 \log n^2)$. Namiesto toho nám stačí obe naše polia raz prejsť. Pritom si budeme v dvoch premenných pamätať, aké su dve najčastejšie farby a pri prechádzaní poľa to postupne aktualizovať. Takto nám nájdenie najčastejších farieb bude trvať už len $O(n^2)$.

Potom nám už len stačí na základe početnosti najčastejších a druhých najčastejších farieb vybrať, akou farbou ofarbíme čierne políčka a akou biele. Počet políčok, ktoré treba prefarbiť, zrátame ako n^2 mínus počet políčok (bielych aj čiernych), ktoré už majú správnu farbu.

V riešení najprv načítame celú šachovnicu, pričom si rátame počty farieb na bielych a čiernych pozíciách. To nám trvá čas $O(n^2)$. Následne hľadáme najčastejšie farby jedným prejdením v dvoch poliach veľkosti n^2 a to nám bude tiež trvať $O(n^2)$. Celková časová zložitosť bude preto $O(n^2)$. Keď sa pozrieme na pamäť, pamätáme dve polia pre výskyty farieb, ktoré majú veľkosť $2n^2$. Výsledná pamäťová zložitosť je teda $O(n^2)$.

Listing programu (C++)

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

using namespace std;

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

    //pocty[x][0] = kolko krat bola farba x na 'bielom' policku, pocty[x][1] = na ciernom
    vector< vector<int> > pocty(n*n+1,{0,0});

    for(int i=0;i<n;++i)
    {
        for(int j=0;j<n;++j)
        {
            int x,farba = (i+j)%2;
            cin >> x;
            pocty[x][farba]++;
        }
    }

    //best[0][x] = najcastejsie cislo na polickach farby x, best[1][x] = druhe najcastejsie

    int best[2][2] = { {n*n,n*n}, {n*n,n*n} };

    for(int i=0;i<n*n;i++)
    {
        for(int farba=0;farba<2;++farba)
        {
            if(pocty[i][farba] > pocty[ best[farba][0] ][farba])
            {
                best[farba][1] = best[farba][0];
                best[farba][0] = i;
            }
            else if (pocty[i][farba] > pocty[ best[farba][1] ][farba])
            {
                best[farba][1] = i;
            }
        }
    }

    int usetrim;
    if(best[0][0] != best[1][0])
        usetrim = pocty[best[0][0]][0] + pocty[best[1][0]][1];
    else
        usetrim = max(pocty[best[0][0]][0]+pocty[best[1][1]][1],
                      pocty[best[0][1]][0]+pocty[best[1][0]][1]);
    cout << n*n - usetrim << endl;

    return 0;
}
```

3. Hurá na Oravu

Denis (denis@ksp.sk)
(max. 12 b za popis, 8 b za program)

Našou úlohou bolo nájsť súčet čísel všetkých staníc, na ktorých mohol Andrej skončiť, ak sa riadil pokynmi na mape, na ktorej boli pôvodne iba dva typy pohybu – vľavo a vpravo. Vypočítať číslo novej stanice bolo pritom veľmi jednoduché. Ak Andrej stál pri stanici s číslom k a pohol sa doľava, dostal sa do stanice $2k$,

ak išiel doprava, tak do stanice $2k + 1$. V úlohe bol však jeden zádrheľ. Keďže niektoré pokyny na mape boli nečitateľné (*), museli sme pri nich brať do úvahy obe možnosti.

Simulácia

Na začiatok môžeme skúsiť odsimulovať Andrejov pohyb po lyžiarskom stredisku. Pokyny z mapy budeme spracovávať jeden po druhom a celý čas si budeme pamätať čísla všetkých staníc, na ktorých sa Andrej mohol nachádzať. Na začiatku bol Andrej v stanici číslo 1.

Pokyny sa spracovávajú jednoducho. Ak je na mape `L`, tak prejdeme cez všetky čísla a vynásobíme ich 2, pri `P` k nim okrem zdvojnásobenia pripočítame 1. Jediný problém je znak *, pri ktorom z každej stanice vedú dve možnosti. Preto si zapamätáme obe z nich, ak bol Andrej v stanici k tak po znaku * mohol byť aj v stanici $2k$ aj v $2k + 1$. Po spracovaní všetkých znakov z mapy jednoducho spočítame výsledné čísla staníc, čím dostaneme hľadanú odpoveď.

Problémom je však pamäťová a časová zložitosť. Pri každom znaku * zdvojnásobujeme počet čísel, ktoré si musíme pamätať. Preto ak by na mape boli samé *, čo sa stať môže, tak pamäťová aj časová zložitosť je $O(2^n)$.

Listing programu (C++)

```
#include <iostream>
#include <vector>
using namespace std;

const int MOD = 1000000007;

int main() {
    char vstup;
    vector<long long> stanice = {1};    // pociatocna stanica
    while(cin >> vstup) {
        if(vstup == 'L') {
            for(int i = 0; i < stanice.size(); i++) {
                stanice[i] = (2 * stanice[i]) % MOD;
            }
        }
        else if(vstup == 'P') {
            for(int i = 0; i < stanice.size(); i++) {
                stanice[i] = (2 * stanice[i] + 1) % MOD;
            }
        }
        else {
            int size = (int)stanice.size();    // vo for cykle menim velkost pola stanice, preto nemozem pouzit .size()
            for(int i = 0; i < size; i++) {
                stanice[i] = (2 * stanice[i]) % MOD;
                stanice.push_back((stanice[i] + 1) % MOD);
                }
        }
    }
    long long int vysledok = 0;
    for(int i = 0; i < stanice.size(); i++) {
        vysledok = (vysledok + stanice[i]) % MOD;
    }
    cout << vysledok << "\n";
}
```

Optimalizácia

Keď sa snažíme zlepšiť nejaké riešenie, vždy sa oplatí zamyslieť sa nad tým, či **nerobíme niečo zbytočne**. Hľadaným výsledkom je súčet všetkých staníc, kde sa Andrej môže nachádzať. Nepotrebujeme však vedieť, ktoré konkrétne stanice to sú. Možno si ich teda nemusíme pamätať.

Predstavme si, že Andrej môže byť v staniciach s číslami p, q, r a s. Ak pôjde doľava (`L`), tak bude môcť byť v staniciach $2p$, $2q$, $2r$ a $2s$, ktorých súčet je dvakrát väčší ako súčet predchádzajúcich čísel. Ak si teda budeme pamätať iba jediné číslo – súčet všetkých možných staníc – tak ho pri znaku `L` vieme ľahko upraviť.

Pri znaku `P` majú nové stanice súčet

$$(2p + 1) + (2q + 1) + (2r + 1) + (2s + 1) = 2(p + q + r + s) + 4$$

Súčet sa opäť zdvojnásobil, naviac sa k nemu ale pripočítal počet staníc, v ktorých Andrej mohol byť. Vidíme, že pamätať si iba súčet nestačí, musíme si k nemu zapamätať aj počet staníc, v ktorých Andrej môže byť. Pomocou týchto dvoch čísel už vieme spracovať aj znak `P`.

Ostal nám znak *. Pri ňom by bol súčet nových staníc

$$2p + 2q + 2r + 2s + (2p + 1) + (2q + 1) + (2r + 1) + (2s + 1) = 4(p + q + r + s) + 4$$

Súčet sa teda zoštvornásobil a naviac sa k nemu pripočítal počet staníc. To však nie je problém pomocou našich dvoch hodnôt vypočítať. Vieme preto rýchlo spracovať aj znak *. Pri ňom však nemôžeme zabudnúť, že počet staníc, v ktorých Andrej môže byť, sa zdvojnásobí.

Naše riešenie teda opäť spracováva mapu znak po znaku, tentokrát si však pamätá iba dve hodnoty – súčet a počet staníc, v ktorých sa Andrej môže nachádzať. Ukázali sme si, že pri každom znaku vieme tieto dve čísla jednoduchým spôsobom upraviť na novú hodnotu. A aby sme nepracovali s príliš veľkými číslami, tak ich po každej operácii modulujeme číslom 1 000 000 007.

Pamäťová zložitosť nášho riešenia bude konštantná ($O(1)$), pretože si pamätáme iba dve premenné. Časová bude lineárna od dĺžky mapy, teda $O(n)$, pretože každý znak vieme spracovať v konštantnom čase úpravou najviac dvoch čísel.

Listing programu (C++)

```
#include <iostream>
using namespace std;

const int MOD = 1000000007;

int main() {
    char vstup;
    long long int sucet = 1, pocet = 1;
    while(cin >> vstup) {
        if(vstup == 'L') {
            sucet = (2 * sucet) % MOD;
        }
        else if(vstup == 'P') {
            sucet = (2 * sucet + pocet) % MOD;
        }
        else {
            sucet = (4 * sucet + pocet) % MOD;
            pocet = (2 * pocet) % MOD;
        }
    }
    cout << sucet << "\n";
}
```

Andrej (ajo@ksp.sk)

4. Átriové problémy

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

Ak si odmyslíme príbeh, našou úlohou je načítať n čísel a potom nájsť na nejakých úsekoch súčin týchto čísel, vždy modulo nejaké iné číslo.

Prvá vec, ktorá by programátorovi mala napadnúť v úlohe, kde sa vyskytuje nejaký typ otázok, je, čo si môžeme predpočítať pred ich samotným spracovávaním. Začnime ale na začiatok riešením hrubou silou, ktoré si nič nepotrebuje predpočítať.

Bruteforce

Na začiatku načítame čísla na vstupe. Potom postupne načítavame otázky a každú z nich hneď po načítaní zodpovieme. Pre každú otázku prejdeme zadaný úsek zľava doprava a postupne násobíme čísla v úseku, pričom si medzivýsledok priebežne modulujeme.

Takéto riešenie má časovú zložitosť na jednu otázku až $O(n)$, keďže existujú otázky, ktoré nás donútia prejsť celé pole. Pre q otázok to teda dáva celkovú časovú zložitosť $O(n \cdot q)$ a pamäťovú $O(n)$, kedže si pamätáme iba čísla na vstupe.

Listing programu (C++)

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

using namespace std;

int main()
{
    // optimalizacia I/O
    ios_base::sync_with_stdio(false);

    int n, q;
    long long odp;
    cin >> n >> q;

    vector<int>cisla(n);

    //nacitame si cisla
    for(int i=0;i<n;++i) cin >> cisla[i];

    int l, r, m;
    for(int f=0;f<q;++f)
    {
        cin >> l >> r >> m;
        --l;

        //prechadzame pole, nasobime odp a zaroven modulujeme
```

```
        odp = 1;
        for(int i=l;i<l+r;++i) odp = (odp*cisla[i]) % m;

        cout << odp << "\n";
    }

    return 0;
}
```

Toto riešenie funguje v praxi veľmi dobre, ale na ozaj veľkú sadu stačiť nebude. Na vzorové riešenie sa ešte budeme musieť zamyslieť. Ako sme spomenuli, podozrivé nám môže byť, že sme si nič nepredpočítavali pred samotným spracovávaním otázok.

Prefixové súčty a prečo sú veľké čísla zlé

Ak vám pojem *prefixové súčty* nie je známy, pred čítaním zvyšku vzoráku sa vám oplatí sa s ním zoznámiť, napríklad v kuchárke[3].

Prefixové súčty riešia príbuzný problém, kde namiesto súčinu čísel v úseku chceme zistiť ich súčet. Uvedomme si ale, že tu nefunguje niečo ako "prefixové súčiny". Nevieme si pre každé políčko x, od 1 po n, vypočítať $k_1 \cdot k_2 \cdot \ldots \cdot k_x$, nakoľko takéto číslo môže byť naozaj veľké.

Čo je také zlé na veľkých číslach? A aké veľké čísla by to boli? V poradí i-ty súčin je súčinom prvých i čísel, každé z nich je od 1 do 100. Na jeho uschovanie by sme preto potrebovali až rádovo $i \cdot \log 100$ bitov. Všetky prefixové súčiny by teda zaberali až $O(n^2)$ pamäte, a zaplnenie tejto pamäte by trvalo až $O(n^2)$ času. Navyše, pri implementácii v C++ by sme si museli implementovať vlastný typ pre veľké čísla a aritmetiku na ňom, nakoľko `long long` nestačí.

Dokonca, problém s nimi nie je len ten, že zaberajú veľa miesta. Ak by sme chceli sčítať dve čísla dĺžky i, trvalo by nám to až $O(i)$ času. A ich súčin by trval ešte dlhšie. Vidíme teda, že veľké čísla sú zlé, pretože operácie s nimi trvajú dlho.

V úlohe je veľmi dôležité, že nás zaujíma zvyšok súčinu po delením nejakým relatívne malým číslom $m \leq 10^9$. Ak by nás zaujímal celý súčin, dostali by sme veľké číslo. Už len jeho výpis by trval dlho.

Vo zvyšku textu sa veľkým číslam vyhneme tak, že všetky medzivýsledky budeme modulovať m. Nebojte sa, vždy, keď tak učiníme, na to upozorníme. Priebežným modulovaním si zaručíme, že budeme vždy narábať s číslami do 10^9. Operácie s nimi teda budú zaberať konštantne veľa času.

Vzorové riešenie

Keď čítame úlohu, môžeme sa snažiť nájsť niečo, čo nám udrie do očí, akúsi slabinu úlohy. Tu si všimneme, že čísla na vstupe nie sú veľké, sú do 100. Toto by sme snáď mohli využiť. Ak dostaneme nejaký úsek, zjavne v ňom bude najviac 100 rôznych čísel. Ich počty môžu byť veľké, avšak nikdy týchto čísel nebude veľa rôznych. Predstavme si, že poznáme počty výskytov jednotlivých čísel. Teda, že pre každý úsek a každé číslo x vieme rýchlo povedať, koľkokrát sa číslo x nachádza v danom úseku. Ako vieme toto využiť? Ak máme takúto informáciu, stačí nám vynásobiť jednotlivé čísla umocnené na počty ich výskytov, pričom to berieme modulo m, a dostaneme našu odpoveď. Ako rýchlo však vieme umocniť x na i-tu, modulo m?

Zrekapitulujme si to. Máme dva podproblémy, ktoré chceme vedieť riešiť. Po prvé, chceme vedieť pre ľubovoľný úsek rýchlo povedať, koľkokrát sa v ňom ktoré číslo nachádza. Po druhé, chceme vedieť rýchlo umocniť x na i-tu modulo m. Začnime riešením prvého.

Ako zistiť počet výskytov jednotlivých čísiel na intervale? Predstavme si, že sa pýtame na počet výskytov čísla 42. Každé číslo v poli buď je 42 a prispeje do výsledku $+1$, alebo nie je 42 a prispeje 0. Vytvorme si teda pomocné pole dĺžky n, kde si na pozíciách, kde boli v pôvodnom poli 42-ky, budeme pamätať jednotky a na ostatných pozíciách nuly. Počet 42-jek v nejakom úseku v pôvodnom poli je rovný súčtu čísel v rovnakom úseku v našom pomocnom poli. A súčty čísel v úsekoch vieme rýchlo počítať pomocou prefixových súčtov.

Nás ale nezaujíma odpoveď len pre 42, ale pre všetky čísla od 1 po 100. Tak budeme mať 100 pomocných polí s prefixovými súčtami – jedno pre každé z týchto čísel. Predpočítať si ich nám zaberie $O(100 \cdot n) = O(n)$ času. Následne vieme v konštantnom čase zistiť počet výskytov nejakého čísla v ľubovolnom úseku.

Vrhnime sa na druhý podproblém: ako sa dá rýchlo umocniť x na i-tu, modulo m?

Jedným cyklom by sme to vedeli zaiste v $O(i)$. Avšak, čo ak je i naozaj veľké, tak ako v našom prípade? Využijeme techniku opakovaného umocňovania na druhú. Tá sa dá elegantne implementovať za pomoci rekurzie[4]. Ak máme umocniť x^i a i je nepárne, tak si rekurzívne vypočítame x^{i-1} a vrátime túto hodnotu vynásobenú x, modulo m. Rozdiel oproti pomalému umocňovaniu uvidíme v prípade, keď i je párne. Vtedy si vieme rekurzívne vypočítať $x^{\frac{i}{2}}$. Ak označíme túto hodnotu *pom*, x^i dostaneme zadarmo ako $pom \cdot pom$ alebo pom^2, modulo m.

[3] https://www.ksp.sk/kucharka/prefixove_sumy/
[4] https://www.ksp.sk/kucharka/rekurzia/#wiki-toc-6-chvostova-rekurzia

Akú ma toto časovú zložitosť? Môžme nahliadnuť, že aspoň každý druhý krok klesne i na polovicu. Koľko krát môže takto klesnúť? Nanajvýš $O(\log i)$-krát, čo je aj výsledná časová zložitosť tohto rekurzívneho umocňovania. V kontexte našej úlohy je počet výskytov nejakého čísla v nejakom úseku vždy nanajvýš n. Časová zložitosť bude teda $O(\log n)$.

Aká je celková časová zložitosť nášho algoritmu? Spočítanie si prefixových súčtov nám trvá $O(n)$. Následne, spracovanie každej z q otázok trvá $O(100 \cdot \log n) = O(\log n)$. Dokopy máme časovú zložitosť $O(n + q \cdot log(n))$. Toto riešenie si potrebuje pamätať prefixové súčty polí dĺžky n, z čoho vyplýva pamäťová zložitosť $O(n)$.

Listing programu (C++)

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

using namespace std;

// funkcia, ktora umocni "x" na "n"-tu a to modulo "mod_m" v case O( log(n) )
long long umocni(long long x, long long n, long long mod_m)
{
    // vsetko na 0-tú je 1
    if(n == 0)return 1;

    // ak je n nepárne
    if( (n%2) == 1) return (x*umocni(x, n-1, mod_m)) % mod_m;

    // ----
    // sem sa dostaneme iba ak je n párne
    // ----

    //ak je n párne, zisti hodnotu x^(n/2) a uloz ju do pom
    long long pom = umocni(x, n/2, mod_m);

    return (pom*pom) % mod_m;
}

int main()
{
    // optimalizacia I/O
    ios_base::sync_with_stdio(false);

    long long n, q;
    cin >> n >> q;

    vector<long long> cisla(n);

    for(long long i=0;i<n;++i) cin >> cisla[i];

    vector<vector<long long> > prefix (n+1, vector<long long> (101, 0));

    for(long long i=1;i<=n;++i)
    {
        prefix[i] = prefix[i-1];
        ++prefix[i][cisla[i-1]];
    }

    long long l, r, m, odp;
    for(long long f=0;f<q;++f)
    {
        cin >> l >> r >> m;

        --l;
        r += 1;

        odp = 1;
        for(long long i=0;i<101;++i)
        {
            if( prefix[r][i] - prefix[l][i] > 0)
            {
                odp *= umocni(i, prefix[r][i] - prefix[l][i], m);
                odp %= m;
            }
        }
        cout << odp << "\n";
    }

    return 0;
}
```

Dávid (davidb@ksp.sk)

5. Zajovo umenie

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

Mišov program si náhodne vyberá, ktorá štvrtina obrázku bude biela a ktorá čierna. Našou úlohou je zistiť, ako si má program vyberať tak, aby bol vygenerovaný obrázok čo najviac podobný obrázku na vstupe.

Môžeme si to predstaviť tak, že necháme bežať program a vždy, keď si má náhodne vybrať, podstrčíme mu to "správne" umiestnenie čierneho a bieleho štvorca. Takýmto spôsobom vieme jednoducho vytvoriť výsledný obrázok, ktorý potrebujeme vypísať na výstup. Ostáva nám zistiť, ktoré je to "správne" rozloženie štvorcov.

Ako nájsť najlepšie rozloženie?

Aby sme našli najlepšie rozloženie štvorca so stranou n, potrebujeme zistiť, kam umiestniť biely a čierny štvorec. Koľko na to máme možností? Vyberáme zo štyroch štvrtín štvorca. Jedna má byť biela, jedna čierna, a ostatné dve budú vyfarbené rekurzívne, ako keby to boli samostatné obrázky so stranou $\frac{n}{2}$ (ak sú väčšie ako 1 štvorček). Znamená to, že vyfarbíme jeden zo štyroch štvorcov na bielu a potom jeden zo zvyšných troch na čiernu, čo je spolu $4 \cdot 3 = 12$ možností. Zvyšné dva štvorce sa vyfarbia rekurzívne.

Pre každú možnosť potrebujeme zistiť, ako veľmi by sa líšil výsledný obrázok od zadaného (ďalej skóre). Aby sme vedeli rýchlo zistiť skóre jednotlivých štvrtín, použijeme niekoľko trikov.

Pre skóre bieleho štvorca potrebujeme vedieť, koľko jednotiek sa nachádza v príslušnom štvorci na zadanom obrázku. V našom obrázku sú len nuly a jednotky, čiže počet jednotiek vo štvorci je vlastne súčet všetkých čísel vo štvorci. S tým nám pomôžu dvojrozmerné prefixové súčty[5]. Ich výpočet nám trvá čas lineárny od veľkosti obrázku, a potom sme schopní v konštantnom čase zodpovedať otázky tvaru: "Aký je súčet čísel v tomto obdĺžniku?"

Pre čierny štvorec potrebujeme vedieť, koľko núl sa nachádza v príslušnom štvorci. Počet núl vo štvorci je počet jeho políčok mínus počet jednotiek v ňom, a obe z týchto hodnôt už vieme vypočítať.

Ostáva nám zistiť, aké skóre majú štvorce, ktoré vyplníme rekurzívne. Ak sú tieto štvorce 1×1, skóre je vždy 0, lebo si môžeme zvoliť farbu. Ak sú väčšie, zistíme to rekurzívne. Použijeme teda rovnaký postup, ako keď zisťujeme skóre veľkého štvorca (vyskúšame všetkých 12 možností).

Toto riešenie je ale pomalé, dostali by sme za neho 2 body.

Prečo je to pomalé?

V prvom kroku máme 12 možností ktoré skúšame. Z týchto 12 možností je každý menší štvorec 3-krát čierny, 3-krát biely a 6-krát určovaný rekurzívne. Rekurzívnych volaní je teda spolu 24.

V každom z týchto rekurzívnych volaní sa deje to isté. Na druhej úrovni už máme $24 \cdot 24 = 576$ volaní na $4 \cdot 4 = 16$ štvorcov, pre ktoré chceme zistiť skóre. Očividne robíme niečo zbytočne viac krát. Predídeme tomu jednoducho tak, že vždy, keď poprvýkrát zrátame skóre niektorého štvorca, si jeho skóre zapamätáme. Neskôr, keď budeme chcieť zistiť jeho skóre, sa iba pozrieme na zapamätanú hodnotu. Aký dopad bude mať táto malá zmena? Pozrime sa napríklad opäť na druhú úroveň. Budeme na nej mať už len 16 rekurzívnych volaní, a zvyšných $576 - 16 = 560$ volaní už vyhodnotíme v konštantnom čase tak, že sa pozrieme na zapamätanú hodnotu.

Tento postup sa volá memoizácia. Ďalšiu ukážku použitia memoizácie nájdete napríklad v úlohe Optimálna šifrovačka[6].

Časová zložitosť

Pri úlohách, v ktorých je vstup dvojrozmerný (tabuľka, mriežka), máme dve možnosti, ako zapísať časovú zložitosť. Buď ju môžeme uvádzať zložitosť od skutočného počtu čísel na vstupe (veľkosti vstupu) alebo od dĺžky strany štvorca. My budeme uvádzať zložitosť v závislosti od čísla $N = n \cdot n$, teda od veľkosti vstupu.

Na začiatku si predrátame prefixové súčty obrázku na vstupe, pomocou ktorých neskôr budeme zisťovať skóre bielych a čiernych štvorcov v konštantnom čase. Na to potrebujeme $O(N)$ času. Zaujímavejšie je to s našou rekurzívnou funkciou.

Vďaka memoizácii sa rekurzívne volanie pre každý štvorec vykoná iba raz. Pri každom ďalšom volaní len vráti už uložené riešenie. Počet volaní tejto funkcie je teda rovnaký, ako počet rôznych štvorcov, na ktoré sa funkcia zavolá.

Na začiatku máme štvorec veľkosti N. V ďalšom kroku máme 4 štvorce veľkosti $\frac{N}{4}$, potom 16 štvorcov veľkosti $\frac{N}{16}$, a tak ďalej až po N štvorcov s veľkosťou 1. Dostávame postupnosť $1 + 4 + 16 + 64 + ... + N$ štvorcov všetkých veľkostí.

Aby sme ukázali, že súčet tejto postupnosti je $O(N)$, porovnáme ju s postupnosťou $1 + 2 + 4 + 8 + ... + N = 2N - 1$. Vidíme, že naša postupnosť obsahuje len niektoré jej členy a teda náš súčet bude určite menší. Z toho vyplýva, že celkový počet štvorcov, a teda aj počet volaní funkcie, je $O(N)$.

Teraz potrebujeme ešte zistiť, ako dlho trvá výpočet jedného volania. Potom to vynásobíme počtom volaní a dostaneme zložitosť celého výpočtu. Aby sme zistili zložitosť iba jedného volania, budeme predpokladať, že všetky volania, ktoré bude potrebovať, sú už vyrátané.

Čo robí jedno volanie? Zisťujeme v ňom skóre menších štvorcov a podľa nich vyberieme najlepšie z 12 možných rozložení. Zisťovanie skóre pre biely a čierny štvorec je vďaka prefixovým súčtom v $O(1)$, skóre ostatných štvorcov zistíme opätovným volaním našej funkcie, čo môžeme kvôli rekurzii považovať za $O(1)$.

[5] https://www.ksp.sk/kucharka/2d_prefixove_sumy
[6] https://www.ksp.sk/ulohy/riesenia/1098/

Spolu bude teda výpočet celého rozloženia trvať $O(N) \cdot O(1) = O(N)$.

Na koniec nám ostáva len skonštruovať obrázok na výstup. Na to stačí priamočiara simulácia Mišovho programu (keď už vieme, ktoré štvorce majú byť biele a ktoré čierne). Pri takejto simulácii iba vypĺňame tabuľku veľkosti N, na čo potrebujeme opäť $O(N)$ času.

Celková časová zložitosť je teda $O(N)$. Pamäťová tiež, lebo si pamätáme obrázok veľkosti $O(N)$ a skóre už vyrátaných štvorcov, ktorých je $O(N)$.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <unordered_map>

using namespace std;

vector<string> zadanie;
vector<string> riesenie;
vector<vector<int>> prefix_sum;

// memoizácia, indexujeme stredom štvorca
vector<vector<int> > mem; //najlepšie skóre pre štvorec
vector<vector<array<int, 2>>> memk; //ako sme dostali najlepšie skóre

int dx[] = {0, 0, 1, 1}; //tieto polia nám neskôr ušetria veľa podmienok
int dy[] = {0, 1, 0, 1};

int diff_zeros(int a, int b, int c, int d){
    return prefix_sum[c][d] + prefix_sum[a][b] - prefix_sum[a][d] - prefix_sum[c][b];
}

int diff_ones(int a, int b, int c, int d){
    return (c - a) * (d - b) - (prefix_sum[c][d] + prefix_sum[a][b] - prefix_sum[a][d] - prefix_sum[c][b]);
}

//hladá najlepšie rozloženie
int diff(int a, int b, int c, int d){
    if(c - a == 1)
        return 0;
    int sx = a + (c - a) / 2 - 1,
        sy = b + (d - b) / 2 - 1;
    if(mem[sx][sy] != -1) //výsledok pre tento štvorec už máme zrátaný
        return mem[sx][sy];

    else{
        int l = (c - a) / 2; //strana menšieho štvorca
        int m = c * d; //najlepšie skóre pre tento štvorec
        array<int, 2> k; //ako dostaneme najlepšie skóre
        //vyskúšame 12 možností
        for(int x = 0; x < 4; x++){
            for(int y = 0; y < 4; y++){
                if(x == y) continue;
                int t = diff_zeros(a + dx[x] * l, b + dy[x] * l, c - (1 - dx[x]) * l, d - (1 - dy[x]) * l) +
                        diff_ones(a + dx[y] * l, b + dy[y] * l, c - (1 - dx[y]) * l, d - (1 - dy[y]) * l);

                for(int z = 0; z < 4; z++){
                    if(x == z || y == z) continue;
                    t += diff(a + dx[z] * l, b + dy[z] * l, c - (1 - dx[z]) * l, d - (1 - dy[z]) * l);
                }

                if(t < m){
                    m = t;
                    k = {x, y};
                }
            }
        }

        mem[sx][sy] = m;
        memk[sx][sy] = k;
        return m;
    }
}

void fill_zeros(int a, int b, int c, int d){
    for(int i = a; i < c; i++)
        for(int j = b; j < d; j++)
            riesenie[i][j] = '0';
}

void fill_ones(int a, int b, int c, int d){
    for(int i = a; i < c; i++)
        for(int j = b; j < d; j++)
            riesenie[i][j] = '1';
}

//konštruuje výsledok z predrátaného rozloženia memk
void fill(int a, int b, int c, int d){
    if(c - a > 1){
        int sx = a + (c - a) / 2 - 1,
            sy = b + (d - b) / 2 - 1;
        array<int, 2> kde = memk[sx][sy];
        int x = kde[0], y = kde[1];
        int l = (c - a) / 2;
        fill_zeros(a + dx[x] * l, b + dy[x] * l, c - (1 - dx[x]) * l, d - (1 - dy[x]) * l);
```

```
        fill_ones(a + dx[y] * l, b + dy[y] * l, c - (1 - dx[y]) * l, d - (1 - dy[y]) * l);
        for(int z = 0; z < 4; z++){
            if(x == z || y == z) continue;
            fill(a + dx[z] * l, b + dy[z] * l, c - (1 - dx[z]) * l, d - (1 - dy[z]) * l);
        }
    }
}

int main(){
    int n;
    cin >> n;
    zadanie.resize(n);
    riesenie.resize(n);
    prefix_sum.resize(n + 1);
    prefix_sum[0].resize(n + 1, 0);
    for(int i = 1; i <= n; i++){
        prefix_sum[i].resize(n + 1);
        prefix_sum[i][0] = 0;
        cin >> zadanie[i - 1];
        riesenie[i - 1] = zadanie[i - 1];
        for(int j = 1; j <=n; j++){
            prefix_sum[i][j] = prefix_sum[i - 1][j] + prefix_sum[i][j - 1] -
                               prefix_sum[i - 1][j - 1] + zadanie[i - 1][j - 1] - '0';
        }
    }
    mem.resize(n, vector<int>(n, -1));
    memk.resize(n, vector<array<int, 2>>(n));
    int ans = diff(0, 0, n, n);
    fill(0, 0, n, n);
    cout << ans << endl;
    for(int i = 0; i < n; i++){
        cout << riesenie[i] << endl;
    }
}
```

Emo (siegrift@ksp.sk)
(max. 12 b za popis, 8 b za program)

6. Ipeľ sa vylieva

Našou úlohou bolo pre daný plán mesta zistiť, koľko políčok ostane nezaplavených, keď sa Ipeľ vyleje. Ako už býva zvykom, pod úlohami týchto typov sa ukrýva grafové riešenie.

Prvé riešenie

Toto riešenie je založené na jednoduchej myšlienke, že pre každé políčko skúsime nájsť cestu, ktorou by ho vedel Ipeľ zaplaviť. Ak takáto cesta neexistuje, tak je dane políčko nezaplaviteľné. Ako overíme, či je nejaké políčko zaplaviteľné? Jednoducho, spustíme z neho DFS (prehľadávanie do hĺbky)[7]. Políčko je zaplaviteľné, ak sa počas DFS-ka dostaneme mimo mapu.

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> pii;

inline void magic_optimalization(){ios_base::sync_with_stdio(false);cin.tie(0);cout.tie(0);}

// v programe mozeme pozuivat integery, lebo itak vyriesime iba 1.sadu
int n, m, k;
vector<vector<int>> g;
vector<vector<bool>> vis;
int xp[] = {1, -1, 0, 0}, yp[] = {0, 0, -1, 1};

bool isOk(int x, int y) {
  try {
    g.at(x).at(y);
    return true;
  } catch (const out_of_range &error) {
    return false;
  }
}

void clearVisited() {
  vis.resize(n, vector<bool>(m));
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
      vis[i][j] = false;
    }
  }
}

bool dfs(int x, int y) {
  // ak sme mimo mapy, tak sme nasli vodu
  if (!isOk(x, y)) return true;
  // ak sme tu boli, alebo policko je budova, vodu sme nenasli
  if(vis[x][y] || g[x][y] == 1) return false;
  vis[x][y] = true;
```

[7] https://www.ksp.sk/kucharka/dfs/

```
  // preiterujeme susedov, ak existuje cesta mimo mapu z nejakeho
  // z nich, tak existuje aj z daneho vrchola
  for (int i = 0; i < 4; ++i) {
    if(dfs(x+xp[i], y+yp[i])) return true;
  }
  return false;
}

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

  // nacitame mapu
  g.resize(n, vector<int>(m));
  for(int i = 0; i < k; i++) {
    int x, y;
    cin >> x >> y;
    g[x][y] = 1;
  }

  // skusime z kadeho policka vyjst mimo mapu
  int unflood = 0;
  for (int i = 0; i < n; ++i) {
    for(int j = 0; j < m; ++j) {
      if(g[i][j] == 0) {
        clearVisited();
        if (!dfs(i, j)) unflood++;
      }
    }
  }

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

Takéto riešenie má však zložitosť až $O(n^2 \cdot m^2)$, pretože políčok je $n \times m$ a DFS má v tomto prípade zložitosť $O(n \cdot m)$. *(Zamyslite sa, že ani pri použiti BFS[8] sa časová zložitosť nezlepší.)*

Vylepšenie pôvodného riešenia

Prvé riešenie nebolo zlé, bolo len príliš pomalé. Vieme ho však zrýchliť dostatočne na to, aby sme vyriešili až 3 sady vstupov (v daných sadách sa mapa zmestí do pamäte).

Treba si uvedomiť, že je dosť zbytočné robiť DFS furt z každého políčka. Ak predsa voda zaleje nejaké políčko, tak zaleje aj všetkých jeho susedov, a ich susedov... V grafovej terminologií sa tomu hovorí komponent súvislosti. Stačí mať jedno pole `visited`, a pre každé voľné políčko nájsť celý jeho komponent. Všetky políčka tohto komponentu si potom označíme za spracované a z nich už ďalšie prehľadávanie spúšťať nebudeme. Ak sme počas hľadania komponentu vyšli mimo mapu, vieme, že celý komponent bude nakonci zaliaty vodou.

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> pii;

inline void magic_optimalization(){ios_base::sync_with_stdio(false);cin.tie(0);cout.tie(0);}

// v programe mozeme pozuivat integery, lebo poslednu sadu itak nevyriesime
int n, m, k;
vector<vector<int>> g, vis;
int xp[] = {1, -1, 0, 0}, yp[] = {0, 0, -1, 1};

bool isOk(int x, int y) {
  try {
    g.at(x).at(y);
    return true;
  } catch (const out_of_range &error) {
    return false;
  }
}


// zistime, ci sme pocas bfs nasli vodu a kolko policok ma dana komponenta
int foundWater, found;
void bfs(int startX, int startY) {
  foundWater = found = 0;
  queue<pii> q;
  q.push({startX, startY});

  while (q.size()) {
    pii p = q.front(); q.pop();
    int x = p.first, y = p.second;
    // ak sme mimo mapy, tak sme nasli vodu
    if (!isOk(x, y)) {
      foundWater = true;
    } else {
      // ak sme na policku ktore je navstivene alebo budova, vratme sa na zaciatok cyklu
      if(vis[x][y] || g[x][y] == 1) continue;
```

[8] https://www.ksp.sk/kucharka/bfs/

```
      // inak si zaznacme, ze sme nasli nove policko, a pridajme susedne policka do radu
      vis[x][y] = true;
      found++;
      for (int i = 0; i < 4; ++i) {
        q.push({x+xp[i], y+yp[i]});
      }
    }
  }
}

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

  // nacitajme mapu
  g.resize(n, vector<int>(m));
  for(int i = 0; i < k; i++) {
    int x, y;
    cin >> x >> y;
    g[x][y] = 1;
  }

  // spustime bfs na kazdom volnom policku
  vis.resize(n, vector<int>(m));
  int unflood = 0;
  for (int i = 0; i < n; ++i) {
    for(int j = 0; j < m; ++j) {
      if(g[i][j] == 0 && !vis[i][j]) {
        bfs(i, j);
        if(!foundWater) unflood += found;
      }
    }
  }

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

Takéto pozorovanie nám vylepší zložitosť na $O(n \cdot m)$, pretože každé políčko mapy spracujeme v DFS najviac raz.

Plný počet bodov

Na zisk plného počtu bodov sa musíme vysporiadať s tým, že mapa sa nám nezmestí celá do poľa. Tu si veľmi nepomôžeme a musíme vymyslieť niečo nové. Kľučom je pozorovanie, že budov celkom málo ($\leq 10^6$).

Každý riadok mesta, ktorý obsahuje nejaké budovy, si rozdelíme na intervaly voľných políčok, ktoré budú určené dvoma budovami. Napríklad, ak máme v riadku budovy na indexoch $0, 13, 72$, tak si spravíme intervaly $[1, 12]$, $[14, 71]$. Každý interval je buď celý zaplavený, alebo celý nezaplavený. Tieto intervaly budeme brať ako vrcholy a vytvoríme si z nich graf. Dva intervaly (vrcholy) sú spojené hranou, ak sú na dvoch po sebe idúcich riadkoch a majú prienik. Pre smrteľníkov to znamená, že ak sa voda dostane do jedného vrchola, dostane sa aj do druhého.

Ak sú medzi dvoma nasledujúcimi riadkami s budovami nejaké prázdne riadky, celú túto skupinu prázdnych riadkov budeme reprezentovať jedným vrcholom, ktorý bude spojený so všetkými intervalmi z riadku nad ním a z riadku pod ním.

Okej, už to skoro máme, teraz stačí spustiť prehľadávanie zo všetkých krajných intervalov. Hmm, ale ako to spraviť tak, aby sme sa nezbláznili? Najjednoduchšie je do každého riadka pridať virtuálne budovy na pozície $-\infty$ a ∞. Tým nám na začiatku a na konci riadka vzniknú intervaly reprezentujúce kraj mapy. Z týchto intervalov a z intervalov v prvom a poslednom zastavanom riadku potom môžeme spustiť DFS, ktorým zistíme, ktoré intervaly budú zaplavené a ktoré nie.

Listing programu (C++)

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

inline void magic_optimalization(){ios_base::sync_with_stdio(false);cin.tie(0);cout.tie(0);}

ll n, m, k;
vector<bool> vis;
vector<vector<int>> g;

struct Node {
  ll x, y1, y2;
  Node(ll _x, ll _y1, ll _y2) {
    x = _x;
    y1 = _y1;
    y2 = _y2;
  }
};

bool intersects(Node n1, Node n2) {
  // zmenime otvorene intervaly na zatvorene
```

```
  int a1 = n1.y1 + 1, a2 = n1.y2 - 1;
  int b1 = n2.y1 + 1, b2 = n2.y2 - 1;
  // zistime, ci nemame nahodou neplatny interval
  if (a1 > a2 || b1 > b2) return false;
  // zistime ci sa intervaly prekryvaju
  if (max(a1, b1) <= min(a2, b2)) return true;
  return false;
}

void dfs(int start) {
  stack<int> s; s.push(start); vis[start] = true;
  while(s.size()) {
    int u = s.top(); s.pop();
    for (int v: g[u]) {
      if (!vis[v]) {
        s.push(v);
        vis[v] = true;
      }
    }
  }
}

int main() {
  magic_optimalization();
  cin >> n >> m >> k;
  // nacitaj a utried zabrany a
  // zisti najjuznejsiu a najsvernejsiu budovu
  vector<pair<ll, ll>> blocked;
  unordered_set<ll> added;
  ll north_x = ULONG_MAX, south_x = -1;
  for (int i = 0; i < k; ++i) {
    ll x, y; cin >> x >> y;
    blocked.push_back({x, y});
    // pridame specialne budovy, ktore nam vytvoria vrchol od ktoreho zacneme zaplavovat
    for (ll j = x-1; j <= x+1; ++j) {
      if (!added.count(j)) {
        blocked.push_back({j, -1});
        blocked.push_back({j, m});
        added.insert(j);
      }
    }
    north_x = min(north_x, x);
    south_x = max(south_x, x);
  }
  sort(blocked.begin(), blocked.end());

  // vytvor pomocny graf
  vector<Node> nodes;
  int block = 0;
  while (block < blocked.size()) {
    ll x = blocked[block].first, y = blocked[block].second;
    while (block < blocked.size() && blocked[block].first == x) {
      nodes.push_back(Node(x, y, blocked[block].second));
      y = blocked[block].second;
      block++;
    }
  }

  // vytvor graf
  g.resize(nodes.size());
  int u = 0, v = 0;
  while(u < nodes.size()) {
    // preiterujeme mensi alebo rovnaky riadok mapy
    while (v < nodes.size() && nodes[v].x <= nodes[u].x) v++;
    // zaujima iba prvy vacsi riadok a iba pokym je
    // medzi prienikom danych intervalov prazdne miesto
    while (v < nodes.size() && nodes[u].x + 1 == nodes[v].x) {
      //cout << "T: "<< u << " " << v << endl;
      if (intersects(nodes[u], nodes[v])) {
        g[u].push_back(v);
        g[v].push_back(u);
      }
      // bud zvysime v alebo zvysime u
      if (nodes[u].y2 < nodes[v].y2) break;
      else v++;
    }
    u++;
  }

  // spusti dfs z kadeho krajneho intervalu
  vis.resize(nodes.size());
  for (int i = 0; i < nodes.size(); ++i) {
    if (nodes[i].x == south_x + 1 ||
      nodes[i].x == north_x - 1 ||
      nodes[i].y1 == -1 ||
      nodes[i].y2 == m) {
      dfs(i);
    }
  }


  // spocitaj nezaplavene uzemia
  ll unflood = 0;
  for (int i = 0; i < nodes.size(); ++i) {
    if (!vis[i]) {
      // obcas sme zapocitali aj zabranu (x, x)
      unflood += max(0LL, nodes[i].y2 - nodes[i].y1 - 1);
    }
```

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

Počet vrcholov v našom grafe (intervalov) je priamo úmerný počtu budov (nanajvýš trikrát väčší) a dá sa ukázať, že počet hrán tiež. Časová zložitosť nášho algoritmu je teda $O(k \log k)$, keďže graf dokážeme po utriedení budov zostrojiť v lineárnom čase a DFS navštívi každý vrchol najviac raz. Pamäťová zložitosť je $O(k)$.

7. Marťan a snehová búrka

Hodobox (hodobox@ksp.sk)
(max. 12 b za popis, 8 b za program)

Stará dobrá hrubá sila

Ako inak – úloha sa dá riešiť aj hrubou silou. Pre každé pristávacie miesto si pamätáme zoznam KSPákov v jeho výsledku a najlepší čas, za ktorý títo KSPáci k nemu pribehnú – na začiatku si tento čas inicializujeme na nekonečno. Potom jednoducho prejdeme celý zoznam KSPákov, keď natrafíme na niekoho s rovnakým časom príchodu ako doterajší najlepší, pridáme si ho do zoznamu, a ak na niekoho s lepším časom, čas si zmeníme naň, zoznam premažeme a pridáme tam len tohto najnovšieho.

Pre každé z q pristávacích miest prejdeme celý zoznam n KSPákov, spravíme konštantne veľa operácií a niekdy premažeme doterajší zoznam – do tohto zoznamu však vždy pridáme nanajvýš n KSPákov, teda pre jedno pristávacie miesto spravíme $O(n)$ operácií, čiže dokopy dostávame časovú zložitosť $O(nq)$. Toto postačuje na prvé dve sady, s dostatočne rýchlym jazykom a implementáciou aj na tretiu. Pamätáme si všetkých n KSPákov a otázky vieme riešiť po jednej, pričom v rámci jej riešenia si pamätáme zoznam dĺžky nanajvýš n, čiže máme pamäťovú zložitosť $O(n)$.

Listing programu (C++)

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

using namespace std;

int main()
{

    int n,q;
    cin >> n >> q;

    vector<pair<int,double> > KSPaci(n);
    for(int i=0;i<n;++i)
        cin >> KSPaci[i].first >> KSPaci[i].second;

    while(q--)
    {
        int y;
        cin >> y;
        vector<int> odpoved;
        double najrychlejsi = 2000000000;

        for(int i=0;i<n;++i)
        {
            double cas = abs(KSPaci[i].first-y)/KSPaci[i].second;

            if(cas == najrychlejsi)
                odpoved.push_back(i+1);
            else if (cas < najrychlejsi)
            {
                najrychlejsi = cas;
                odpoved = {i+1};
            }

        }

        cout << odpoved.size();
        for(int KSPak:odpoved)
            cout << " " << KSPak;
        cout << "\n";
    }

    return 0;
}
```

Viete povedať, ktorá časť tejto implementácie je 'pomalá'?

Poznámka o výstupe

Jedna otázka, ktorá vám mohla po prečítaní zadania skrsnúť, je ako veľký môže vlastne byť výstup. Keby totiž skoro každý KSPák mohol mať najlepší čas na skoro každom mieste pristátia, museli by sme vypísať až $O(nq)$ čísel, čo by sme teda vo väčších sadách nestihli bez ohľadu na to, ako by sme hľadali odpoveď.

Takéto niečo však nemôže nastať vďaka podmienkam v zadaní, to jest *pristávacie miesta sú navzájom rôzne, a Denys nikdy nepristane na niektorom KSPákovi*, a *dvojice* x_i, v_i *sú navzájom rôzne*. Toto nám vyplynie z pozorovania, ktoré spravíme popri vymýšľaní vzorového riešenia.

Jednosmerné riešenie

Pozrime sa teda na ľahšiu verziu úlohy – naše riešenie musí mať síce oveľa lepšiu časovú zložitosť, nemusíme sa však starať o to, že KSPáci sa ku každému pristátiu hrnú z oboch strán. V momente, keď Denys pristane, všetci KSPáci sa jednoducho rozbehnú doprava ako rýchlo vládzu.

Ak si vypočítame pozíciu každého predbehnutia medzi dvoma KSPákmi, úlohu môžeme riešiť jednoduchým zametacím algoritmom. Postupne pôjdeme po pristávacej dráhe zľava doprava a budeme si pritom udržiavať poradie KSPákov, v akom prejdú cez našu aktuálnu pozíciu. Na začiatku sa nachádzame naľavo od najľavejšieho KSPáka a naše poradie je zatiaľ prázdne (lebo cez našu pozíciu nikdy nijaký KSPák neprejde). Vždy, keď prechádzame cez štartovaciu pozíciu nejakého KSPáka, pridáme si ho do nášho poradia, konkrétne na čelo (keďže na danej pozícii začína, bude tam ako prvý)[9]. Vždy, keď prejdeme cez pozíciu, kde nejaký KSPák predbehne iného, vymeníme si týchto dvoch KSPákov v našom poradí. Keď prechádzame cez pristávacie miesto, zapíšeme si ako odpoveď KSPáka, ktorý je momentálne prvý v poradí (a v prípade, že sa na tej istej súradnici udejú aj nejaké predbehnutia, zapíšeme sem aj ostatných s rovnakým časom).

Problém je však v tom, že predbehnutí medzi KSPákmi môže byť až $\frac{n\cdot(n-1)}{2}$, keby boli KSPáci zoradení zľava doprava zostupne podľa rýchlosti. Uvedomme si ale, že drvivá väčšina predbehnutí v skutočnosti vôbec nie je dôležitá. Ak totiž ľubovoľného KSPáka niekto predbehne, môžeme ho už úplne ignorovať – predbehol ho nejaký rýchlejší KSPák, takže už nikdy vo vedení nebude. To znamená, že pri simulácii predbehnutia môžeme namiesto vymenenia dvoch KSPákov v poradí úplne vyhodiť predbehnutého KSPáka z poradia. Pre každého KSPáka tak budeme simulovať iba prvé predbehnutie, keď bol predbehnutý iným KSPákom. A takýchto predbehnutí je nanajvýš n. Tu si všimnime, že okrem KSPáka, ktorý bol sám na čele preteku, sa do odpovede na otázku dostane iný len vtedy, keď práve v tom momente predbehne ešte nepredbehnutého KSPáka. Keďže takýchto predbehnutí je $O(n)$, celkový počet KSPákov v odpovediach je $O(n+q)$ (dokonca tesný odhad je práve $n+q$).

Teraz však máme problém, že nevieme, kde tieto dôležité predbehnutia nastanú. Nemôžeme si ich len tak vypočítať na začiatku, lebo nevieme, medzi ktorými dvojicami KSPákov nastanú a skúšať všetky dvojice by bolo príliš pomalé. Namiesto toho to budeme robiť postupne. Okrem poradia KSPákov si budeme pamätať ešte zoznam všetkých pozícií, kde niektorý KSPák z nášho poradia predbehne KSPáka, ktorý je momentálne v poradí **tesne** pred ním. Ak sa nám tento zoznam podarí udržiavať aktuálny, vždy keď bude treba odsimulovať nejaké predbehnutie, toto predbehnutie budeme mať v zozname, teda o ňom budeme vedieť.

Vždy, keď pridávame KSPáka na čelo poradia, vypočítame si pozíciu, kde ho predbehne KSPák, ktorý bol na čele doteraz (ak ho predbehne) a pridáme si ju do zoznamu predbehnutí. Keď simulujeme, že nejaký KSPák A predbehol KSPáka B, toto predbehnutie vyhodíme zo zoznamu. Okrem toho sa pozrieme, či KSPák A nepredbehne niekedy neskôr aj KSPáka, ktorý bol doteraz v poradí tesne pred B (označme ho C) a ak áno, pridáme si do zoznamu predbehnutí pozíciu, kde ho predbehne. Nakoniec sa ešte pozrieme, či sme v zozname nemali, že B predbehne C a ak áno, vymažeme toto predbehnutie zo zoznamu (keďže už nie je aktuálne).

Pozíciu, kde nejaký KSPák X predbehne KSPáka Y vypočítame jednoducho ako $x_Y + \frac{x_Y - x_X}{v_X - v_Y} \cdot v_Y$.

Všimnime si, že so zoznamom predbehnutí manipulujeme, iba keď pridávame KSPáka, alebo keď simulujeme predbehnutie, teda dokopy maximálne $2n$ krát.

Už si musíme len vybrať vhodné dátové štruktúry. Zoberme si poradie KSPákov. Od neho potrebujeme q krát pristúpiť k vedúcemu KSPákovi, $O(n)$ krát pridať KSPáka na začiatok poradia a $O(n)$ krát vyriešiť predbehnutie, ktoré zahŕňa zmazanie jedného prvku (predbehnutého KSPáka) a nájdenie KSPáka, ktorý sa ocitne tesne pred predbiehajúcim KSPákom. Na túto úlohu sa hodí spájaný zoznam, ktorý všetky tieto operácie dokáže robiť v konštantnom čase.

Ak sa nám však nechce implementovať spájaný zoznam, môžeme namiesto neho použiť usporiadanú množinu, ktorá je v mnohých jazykoch už implementovaná, napríklad `set` v C++. Prvky pritom budeme usporiadavať podľa štartovacej pozície (keďže v poradí máme iba nepredbehnutých KSPákov, sú usporiadaní rovnako, ako boli na začiatku). Táto dátová štruktúra je o logaritmus pomalšia než spájaný zoznam, časovú zložitosť nám však nepokazí, lebo iné časti algoritmu sú pomalšie.

Ďalšiu dátovú štruktúru si musíme zvoliť na efektívne zistenie, ktorú udalosť máme spracovať ďalšiu – prechod na ďalšieho KSPáka, konkrétne predbehnutie, alebo odpovedanie na pristávacie miesto. Tieto udalosti chceme riešiť usporiadané podľa pozície, na ktorej sa udejú, a musíme do nej $O(n+q)$ krát udalosť pridať, a teda aj $O(n+q)$ krát vybrať a zmazať nasledujúcu udalosť. Toto vieme vykonať prioritnou frontou (napr. haldou), ako je `priority_queue` v C++. V tejto prioritnej fronte teda budeme mať aj náš zoznam predbehnutí. Z tohoto

[9] z KSPákov začínajúcich na rovnakej pozícii môžeme všetkých, až na najrýchlejšieho, ignorovať

zoznamu sme však občas potrebovali aj mazať (keď nejaké predbehnutie prestalo byť aktuálne), čo bežná halda nepodporuje. To vyriešime nasledujúcou fintou: neaktuálne prvky nebudeme mazať, ale pri výbere z prioritnej fronty budeme kontrolovať, či je vyberaný prvok ešte aktuálny. Druhá možnosť je použiť ako prioritnú frontu usporiadanú množinu.

Budeme teda $O(n+q)$ krát pridávať/mazať prvok z prioritnej fronty ktorá bude mať $O(n+q)$ prvkov, toto nám zaberie $O((n+q)\log(n+q))$ času a $O(n+q)$ pamäte. Taktiež budeme musieť $O(n)$ krát pridávať a vyhadzovať KSPákov zo spájaného zoznamu s poradím, čo nám zaberie $O(n)$ času (alebo, ak použijeme usporiadanú množinu, $O(n\log n)$ času) a $O(n)$ pamäte. Dokopy teda celé naše riešenie zaberie $O((n+q)\log(n+q))$ času a $O(n+q)$ pamäte.

Listing programu (C++)

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

using namespace std;

struct bezec
{
    int x,id;
    double v;
};

struct otazka
{
    int y,id;
    vector<int> odpoved;
};

struct udalost
{
    double kde;
    int typ; // 1 = novy KSPak, 2 = predbehnutie, 3 = pristatie
    int id,id2; // id = id, id2 je id predbehnuteho KSPaka v pripade typ = 2
};

struct najskor
{
    bool operator()(const udalost&a,const udalost&b) const
    {
        if(a.kde != b.kde)
            return a.kde > b.kde; // najlavejsie udalosti najprv

        return a.typ < b.typ;
    }
};

struct zostupne
{
    bool operator()(const bezec a,const bezec b) const
    {
        return a.x > b.x;
    }
};

void vyries(vector<bezec> &KSPaci,vector<otazka> &pristatia)
{
    priority_queue<udalost,vector<udalost>,najskor> udalosti;
    set<bezec,zostupne> nepredbehnuti;

    //predbehnutych KSPakov vyskrtneme, a uz ich budeme ignorovat
    vector<bool> zaujimavi(KSPaci.size(),true);

    int posledne_pristatie = -1; //

    for(auto KSPak:KSPaci)
        udalosti.push({KSPak.x,1,KSPak.id,-1});

    for(auto pristatie:pristatia)
        udalosti.push({pristatie.y,3,pristatie.id,-1});

    while(!udalosti.empty())
    {
        udalost event = udalosti.top();
        udalosti.pop();
        int id = event.id;

        if(event.typ == 1) // novy KSPak
        {
            // z KSPakov startujucich na rovnakej pozicii chceme len rychlejsieho
            if(!nepredbehnuti.empty() && nepredbehnuti.begin()->x == KSPaci[id].x)
            {
                int id2 = nepredbehnuti.begin()->id;
                if(KSPaci[id2].v > KSPaci[id].v)
                {
                    zaujimavi[id] = false;
                    continue;
                }
                else
```

```
                {
                    nepredbehnuti.erase(KSPaci[id2]);
                    zaujimavi[id2] = false;
                }
            }

            // ak existuje nepredbehnuty KSPak, je rychlejsi, tak si zapiseme kedy nas predbehne

            if(!nepredbehnuti.empty() && nepredbehnuti.begin()->v > KSPaci[id].v)
            {
                int id2 = nepredbehnuti.begin()->id;
                double predbehne = KSPaci[id].x + (KSPaci[id].x - KSPaci[id2].x) /
                                   (KSPaci[id2].v - KSPaci[id].v) * KSPaci[id].v;
                udalosti.push({predbehne,2,id2,id});
            }

            nepredbehnuti.insert(KSPaci[id]);
        }
        else if (event.typ == 2) // predbehnutie
        {
            int id2 = event.id2;

            if(zaujimavi[id] == false)
                continue; //pozostatok KSPaka, ktoreho uz niekto iny predbehol

            if(zaujimavi[id2] == false)
                continue; //toto sa moze stat v pripade, ze sme mali predbehnut KSPaka
                // ktoreho hned na to 'predbehol' rychlejsi KSPak na rovnakej suradnici
                // v tom pripade uz mame zapisane aj predbehnutie s tym rychlejsim, a budeme riesit to

            if(posledne_pristatie != -1)
            {
                // skusime zapisat do odpovede aj tychto KSPakov, ak maju rovnaky cas ako ten najlepsi

                double najlepsi = (pristatia[posledne_pristatie].y -
                                  (*nepredbehnuti.begin()).x ) / (*nepredbehnuti.begin()).v;
                double my = (pristatia[posledne_pristatie].y - KSPaci[id].x) / KSPaci[id].v;

                if(my == najlepsi)
                {
                    pristatia[posledne_pristatie].odpoved.push_back(id);
                    pristatia[posledne_pristatie].odpoved.push_back(id2);
                }
            }

            // zahodime predbehnuteho KSPaka
            zaujimavi[id2] = false;
            nepredbehnuti.erase(KSPaci[id2]);

            auto it = nepredbehnuti.find(KSPaci[id]);
            // tento KSPak je na cele, nema koho predbehnut
            if(it == nepredbehnuti.begin()) continue;
            it--;

            id2 = it->id;
            if(KSPaci[id2].v < KSPaci[id].v)
            {
                double predbehne = KSPaci[id2].x + (KSPaci[id2].x - KSPaci[id].x) /
                                   (KSPaci[id].v - KSPaci[id2].v) * KSPaci[id2].v;
                udalosti.push({predbehne,2,id,id2});
            }
        }
        else
        {
            posledne_pristatie = id;
            if(!nepredbehnuti.empty())
            {
                pristatia[id].odpoved.push_back(nepredbehnuti.begin()->id);
            }
        }
    }
}

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);

    int n,q;
    cin >> n >> q;

    if(n <= 3000 && q <= 3000)
    {
        //riesenie hrubou silou
        return 0;
    }

    vector<bezec> KSPaci(n);
    vector<otazka> pristatia(q);

    for(int i=0;i<n;++i)
    {
        KSPaci[i].id = i;
        cin >> KSPaci[i].x >> KSPaci[i].v;
    }

    for(int i=0;i<q;++i)
    {
```

```
        pristatia[i].id = i;
        cin >> pristatia[i].y;
    }

    vyries(KSPaci,pristatia);

    for(auto &pristatie:pristatia)
    {
        sort(pristatie.odpoved.begin(),pristatie.odpoved.end());
        auto it = unique(pristatie.odpoved.begin(),pristatie.odpoved.end());
        pristatie.odpoved.resize(distance(pristatie.odpoved.begin(),it));

        cout << pristatie.odpoved.size();
        for(auto KSPak:pristatie.odpoved)
            cout << " " << KSPak+1;
        cout << "\n";
    }

    return 0;

}
```

A už len čelom vzad

No a čo nám od tohto riešenia chýba k verzii, keď sú KSPáci hala-bala rozmiestnení medzi pristátiami? Naše riešenie si vlastne vie poradiť s ľubovoľným vstupom, ak by naša otázka bola *"pre každé pristávacie miesto povedz tých KSPákov, ktorí by sa k nemu dostali najrýchlejšie, keby všetci bežali doprava"*. Stačí nám teda úlohu vyriešiť ešte raz, ale v opačnom smere – teraz všetci KSPáci budú bežať len doľava, a KSPákov, pristátia a predbehnutia budeme spracovávať v poradí sprava doľava. Ešte poznamenáme, že o dosť príjemnejšie na implementáciu ako skopírovanie celého kódu a menenie dátových štruktúr, aby teraz zoraďovali udalosti naopak ako predtým, je napísať si jednosmerné riešenie ako funkciu, spustiť ju raz na danom vstupe a následne celý vstup otočiť (všetkým KSPákom zmeníme súradnice z x_i na $10^9 - x_i$ a pristávacím miestam z y_i na $10^9 - y_i$). Potom našu funkciu zavoláme ešte raz, a máme vyhraté. Treba len mierne pozmeniť zapisovanie odpovede – keď ideme KSPáka zapísať, najprv sa pozrieme, či už pre dané pristávacie miesto nemáme už nejakú odpoveď, a ak áno, aký čas majú KSPáci, ktorí tam boli predtým zapísaní.

Časová aj pamäťová zložitosť ostáva rovnaká – naše riešenie len použijeme dva krát.

Listing programu (C++)

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

using namespace std;

struct bezec
{
    int x,id;
    double v;
};

struct otazka
{
    int y,id;
    vector<int> odpoved;
    double cas = 100000001;
};

struct udalost
{
    double kde;
    int typ; // 1 = novy KSPak, 2 = predbehnutie, 3 = pristatie
    int id,id2; // id = id, id2 je id predbehnuteho KSPaka v pripade typ = 2
};

struct najskor
{
    bool operator()(const udalost&a,const udalost&b) const
    {
        if(a.kde != b.kde)
            return a.kde > b.kde; // najlavejsie udalosti najprv

        return a.typ < b.typ;
    }
};

struct zostupne
{
    bool operator()(const bezec a,const bezec b) const
    {
        return a.x > b.x;
    }
};

void skus_odpovedat(bezec &KSPak,otazka &pristatie)
```

```
{
    double cas = abs(pristatie.y - KSPak.x) / KSPak.v;

    if(pristatie.cas > cas)
    {
        pristatie.odpoved = {KSPak.id};
        pristatie.cas = cas;
    }
    else if (pristatie.cas == cas)
        pristatie.odpoved.push_back(KSPak.id);

}

void vyries(vector<bezec> &KSPaci,vector<otazka> &pristatia)
{
    priority_queue<udalost,vector<udalost>,najskor> udalosti;
    set<bezec,zostupne> nepredbehnuti;

    //predbehnutych KSPakov vyskrtneme, a uz ich budeme ignorovat
    vector<bool> zaujimavi(KSPaci.size(),true);

    int posledne_pristatie = -1; //

    for(auto KSPak:KSPaci)
        udalosti.push({KSPak.x,1,KSPak.id,-1});

    for(auto pristatie:pristatia)
        udalosti.push({pristatie.y,3,pristatie.id,-1});

    while(!udalosti.empty())
    {
        udalost event = udalosti.top();
        udalosti.pop();
        int id = event.id;
        if(event.typ == 1) // novy KSPak
        {
            // z KSPakov startujucich na rovnakej pozicii chceme len rychlejsieho
            if(!nepredbehnuti.empty() && nepredbehnuti.begin()->x == KSPaci[id].x)
            {
                int id2 = nepredbehnuti.begin()->id;
                if(KSPaci[id2].v > KSPaci[id].v)
                {
                    zaujimavi[id] = false;
                    continue;
                }
                else
                {
                    nepredbehnuti.erase(KSPaci[id2]);
                    zaujimavi[id2] = false;
                }
            }

            // ak existuje nepredbehnuty KSPak, je rychlejsi, tak si zapiseme kedy nas predbehne
            if(!nepredbehnuti.empty() && nepredbehnuti.begin()->v > KSPaci[id].v)
            {
                int id2 = nepredbehnuti.begin()->id;
                double predbehne = KSPaci[id].x + (KSPaci[id].x - KSPaci[id2].x) /
                                   (KSPaci[id2].v - KSPaci[id].v) * KSPaci[id].v;
                udalosti.push({predbehne,2,id2,id});
            }
            nepredbehnuti.insert(KSPaci[id]);
        }
        else if (event.typ == 2) // predbehnutie
        {
            int id2 = event.id2;

            if(zaujimavi[id] == false)
                continue; //pozostatok KSPaka, ktoreho uz niekto iny predbehol

            if(zaujimavi[id2] == false)
                continue; //toto sa moze stat v pripade, ze sme mali predbehnut KSPaka
                // ktoreho hned na to 'predbehol' rychlejsi KSPak na rovnakej suradnici
                // v tom pripade uz mame zapisane aj predbehnutie s tym rychlejsim, a budeme riesit to

            if(posledne_pristatie != -1)
            {
                // skusime zapisat do odpovede aj tychto KSPakov, ak maju rovnaky cas ako ten najlepsi

                skus_odpovedat(KSPaci[id],pristatia[posledne_pristatie]);
                skus_odpovedat(KSPaci[id2],pristatia[posledne_pristatie]);
            }

            // zahodime predbehnuteho KSPaka
            zaujimavi[id2] = false;
            nepredbehnuti.erase(KSPaci[id2]);

            auto it = nepredbehnuti.find(KSPaci[id]);
            // tento KSPak je na cele, nema koho predbehnut
            if(it == nepredbehnuti.begin()) continue;
            it--;

            id2 = it->id;
            if(KSPaci[id2].v < KSPaci[id].v)
            {
                double predbehne = KSPaci[id2].x + (KSPaci[id2].x - KSPaci[id].x) /
                                   (KSPaci[id].v - KSPaci[id2].v) * KSPaci[id2].v;
                udalosti.push({predbehne,2,id,id2});
            }
        }
```

```
        else
        {
            posledne_pristatie = id;
            if(!nepredbehnuti.empty())
            {
                skus_odpovedat(KSPaci[nepredbehnuti.begin()->id],pristatia[id]);
            }
        }
    }
}

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);

    int n,q;
    cin >> n >> q;
    vector<bezec> KSPaci(n);
    vector<otazka> pristatia(q);

    for(int i=0;i<n;++i)
    {
        KSPaci[i].id = i;
        cin >> KSPaci[i].x >> KSPaci[i].v;
    }

    for(int i=0;i<q;++i)
    {
        pristatia[i].id = i;
        cin >> pristatia[i].y;
    }

    // vyriesime pripad, ked KSPaci bezia doprava
    vyries(KSPaci,pristatia);

    // otocime vstup

    for(auto &KSPak:KSPaci)
        KSPak.x = 1000000000 - KSPak.x;

    for(auto &pristatie:pristatia)
        pristatie.y = 1000000000 - pristatie.y;

    // vyriesime pripad, ked KSPaci bezia dolava

    vyries(KSPaci,pristatia);

    for(auto &pristatie:pristatia)
    {
        sort(pristatie.odpoved.begin(),pristatie.odpoved.end());
        auto it = unique(pristatie.odpoved.begin(),pristatie.odpoved.end());
        pristatie.odpoved.resize(distance(pristatie.odpoved.begin(),it));

        cout << pristatie.odpoved.size();
        for(auto KSPak:pristatie.odpoved)
            cout << " " << KSPak+1;
        cout << "\n";
    }

    return 0;
}
```

Samo (samo@ksp.sk)
(max. 12 b za popis, 8 b za program)

8. Ako Jemo Etku spoznal

Potom, ako ste si spolu s Jemom prešli strastiplnou cestou na druhú stranu kopca, našli slušné bývanie a napriek obrovskému hladu stretli jeho vyvolenú, ste dostali neľahkú úlohu. Mnohým z vás však táto úloha musela prísť povedomá. A tak stačilo spomenúť si na jej riešenie a následne ho trošku vytúniť, aby riešilo aj túto úlohu.

Knapsack

Áno, táto úloha bola len jemné zovšeobecnenie úlohy, ktorá je vo svete známa pod názvom knapsack problem[10]. Zadanie znie nasledovne: "Máme n kryptomien, pre každú z nich poznáme jej cenu v Tesku a jej hodnotu na trhu. Akú najväčšiu hodnotu vieme nakúpiť za p peňazí?"

Túto úlohu vyriešime dynamickým programovaním. Vypočítame si dvojrozmerné pole, kde v $D[i][j]$ je uložená najväčšia hodnota, akú vieme nakúpiť, ak máme k dispozícii j peňazí a môžeme nakupovať len prvých i typov kryptomien. Potom odpoveď na našu otázku je uložená v $D[n][p]$.

Ako vypočítame hodnoty v jednotlivých políčkach poľa? Predstavme si, že máme j peňazí a nakupujeme z prvých i typov kryptomien. Aké sú možnosti pre i-tu kryptomenu? Buď ju nekúpime a výsledok je rovnaký, ako keď nakupujeme iba z prvých $i-1$ kryptomien. Alebo ju kúpime (samozrejme, iba ak máme na to dosť peňazí).

[10] https://en.wikipedia.org/wiki/Knapsack_problem

Vtedy chceme kúpiť najväčšiu hodnotu z prvých $i-1$ kryptomien, pričom už máme k dispozícii len $j - cena[i]$ peňazí.

Teda pokiaľ máme dosť peňazí na kúpenie i-tej kryptomeny ($j \geq cena[i]$), tak platí

$$D[i][j] = \max(D[i-1][j], D[i-1][j - cena[i]] + hodnota[i]).$$

Inak $D[i][j] = D[i-1][j]$. Základný prípad je jednoduchý, keďže za 0 peňazí a ani z 0 kryptomien si nič nekúpiš. Vieme teda, že $D[0][j] = D[i][0] = 0$.

Priamočiare využitie

Toto vieme využiť v našej úlohe tak, že vždy, keď príde otázka, zrátame si výsledok knapsacku pre kryptomeny od l po r a vypíšeme odpoveď. Akú to má celé časovú zložitosť? Pre každú otázku potrebujeme zrátať celé dvojrozmerné pole knapsacku odznova. Toto pole môže mať v najhoršom prípade veľkosť až $n \times p$. Teda celková časová zložitosť je $O(q \cdot n \cdot P)$, kde P je najväčšie p_i.

Listing programu (C++)

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

using namespace std;

int n,q,l,r,b;
vector<int> c,h;

int main() {
  ios_base::sync_with_stdio(false);
  cin>>n>>q;
  c.resize(n);h.resize(n);
  for(int i=0;i<n;i++)cin>>c[i]>>h[i];
  while(q--){
    cin>>l>>r>>b;l--;
    vector<vector<int> > D(r-l+1, vector<int>(b+1, 0));
    for(int i=1;i<=b;i++){
      for(int j=1;j<D.size();j++){
        //skus kupit j+l-ty typ
        D[j][i]=D[j-1][i];
        if(i>=c[j+l-1])D[j][i]=max(D[j][i], D[j-1][i-c[j+l-1]]+h[j+l-1]);
      }
    }
    cout <<D.back().back()<<endl;
  }
  return 0;
}
```

Vzorové riešenie

Všimnime si, že keď zrátame knapsack pre nejaký interval kryptomien (od l po r), tak musíme zrátať celú tabuľku D. Zaujímavé je ale to, že v i-tom riadku tabuľky D vlastne uvažujeme len prvých i kryptomien (kryptomeny od l po $l+i-1$). A teda keď už máme vyrátané D, tak máme odpovede zrátané aj pre každý prefix tohto intervalu. Teda pre každý interval tvaru $\langle l; l+i \rangle$, kde $0 \leq i \leq r-l$. Pokiaľ zrátame všetky tieto knapsacky odzadu, tak máme dokonca zrátaný knapsack aj pre každý jeho sufix (intervaly tvaru $\langle l+i; r \rangle$).

Postavme si teraz nad našimi kryptomenami strom s rovnakou štruktúrou, ako má intervalový strom[11] – teda binárny strom, kde je každému vrcholu priradený nejaký interval, pričom koreňu je priradené celé pole, každému z jeho synov jedna polovica poľa, ich synom štvrtiny poľa atď. V každom vrchole nášho stromu zrátajme knapsack na intervale kryptomien, ktorý tento vrchol pokrýva, v oboch smeroch.

Akú výhodu nám to dá? Ľubovoľný interval sa teraz dá rozdeliť na dve časti (o chvíľu si ukážeme ako), pričom pre obe z nich už máme knapsack predrátaný niekde v našom strome. Ak by sme vedeli, koľko peňazí máme minúť v prvej a koľko v druhej časti, odpoveď na otázku by sme zrátali v konštantom čase sčítaním vhodných dvoch čísel, ktoré už máme predpočítané. To síce nevieme, ale môžeme vyskúšať všetky možnosti, ktorých je $p+1$. Poďme si teda zhrnúť naše riešenie.

Predstavme si, že nám príde otázka, že chceme výsledok knapsacku na intervale od l po r. Skúsme nájsť nejaké intervaly v intervaláči, ktoré nám v tomto pomôžu. Začnime v koreni. Pokiaľ celý náš hľadaný interval leží len v pravom alebo len v ľavom synovi, tak nás vlastne tá druhá časť intervaláču nezaujíma a môžeme sa presunúť do syna, v ktorom sa náš interval nachádza. Toto opakujeme až dôjdeme do vrchola, kde toto opakovať nemôžeme. To môže znamenať, že sme v liste, vtedy je hľadaný interval dĺžky 1 a odpoveď je jednoduchá. Buď si vieme dovoliť kúpiť kryptomenu a odpoveď je jej hodnota, alebo je odpoveď 0. Iná možnosť je, že časti nami hľadaného intervalu ležia v oboch synoch. Dokonca vieme, že náš hľadaný interval je vlastne zlepením nejakého sufixu ľavého syna a nejakého prefixu pravého syna. Toto už ale máme predpočítané, teda jediné, čo ostáva, je

[11] https://www.ksp.sk/kucharka/intervalovy_strom/

nakombinovať odpoveď z týchto dvoch intervalov. Jediný problém je, že nevieme, koľko peňazí chceme minúť v ľavom synovi a koľko v pravom. Tak to urobíme jednoducho—keďže dokopy chceme minúť p peňazí, tak vyskúšame všetky možnosti, ako rozdeliť tieto peniaze do synov. Potom už len zoberieme maximum a vypíšeme.

Ako sme teda zlepšili časovú zložitosť? Najprv si predpočítame celý intervalový strom. Všimnime si, že na každej úrovni tohto stromu zrátame niekoľko knapsackov, dokopy pre n prvkov s tým, že minieme maximálne p peňazí. Strom má $\log n$ úrovní, a teda zrátať celý strom vieme v čase $O(n \cdot p \cdot \log n)$. Pri spracovávaní konkrétnej otázky najprv potrebujeme nájsť vhodný vrchol, pričom pri každom kroku tohto hľadania klesneme v hĺbke stromu o 1. Teda vrchol vieme nájsť v $O(\log n)$, a na záver vyskúšame všetky možnosti rozdelenia peňazí do ľavého a pravého intervalu, čo bude trvať $O(p)$. Celková časová zložitosť je teda $O(n \cdot p \cdot \log n + q \cdot (\log n + p))$. Čo sa týka pamäťovej zložitosti, tak si musíme pamätať celý náš intervalový strom, čo je $O(n \cdot p \cdot \log n)$.

Listing programu (C++)

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

using namespace std;

int n,q,l,r,b, maxP=2000;
vector<int> c,h;
vector<vector<vector<int> > > D,E;
void calculateKnapsnack(int vrchol, int odkial, int pokial) {
  D[vrchol].resize(pokial-odkial+1, vector<int>(maxP+1,0));
  E[vrchol].resize(pokial-odkial+1, vector<int>(maxP+1,0));
  for(int i=1;i<=maxP;i++){
    for(int j=1;j<D[vrchol].size();j++){
      D[vrchol][j][i]=D[vrchol][j-1][i];
      if(i>=c[j+odkial-1])D[vrchol][j][i]=max(D[vrchol][j][i], D[vrchol][j-1][i-c[j+odkial-1]]+h[j+odkial-1]);
    }
  }
  for(int i=1;i<=maxP;i++){
    for(int j=1;j<E[vrchol].size();j++){
      E[vrchol][j][i]=E[vrchol][j-1][i];
      if(i>=c[pokial-j])E[vrchol][j][i]=max(E[vrchol][j][i], E[vrchol][j-1][i-c[pokial-j]]+h[pokial-j]);
    }
  }
  if(pokial-odkial<2)return;
  int stred=(odkial+pokial)/2;
  calculateKnapsnack(2*vrchol, odkial, stred);
  calculateKnapsnack(2*vrchol+1, stred, pokial);
}

pair<int,int> najdivrchol(int vrchol, int odkial, int pokial){
  int stred=(odkial+pokial)/2;
  if(l>stred)return najdivrchol(2*vrchol+1, stred, pokial);
  if(r<stred)return najdivrchol(2*vrchol, odkial, stred);
  return {vrchol,stred};
}

int main() {
  ios_base::sync_with_stdio(false);
  cin>>n>>q;
  int listy=1;
  while(listy<n)listy*=2;
  D.resize(2*listy);
  E.resize(2*listy);
  c.resize(listy,0);h.resize(listy,0);
  for(int i=0;i<n;i++)cin>>c[i]>>h[i];
  //ulozene v poli [cislo vrchola v intervalaci][pocet spracovanych][pocet penazi]
  calculateKnapsnack(1,0,listy);
  while(q--){
    cin>>l>>r>>b;l--;
    pair<int,int> interval=najdivrchol(1,0,listy);
    int vsledok=0;
    for(int i=0;i<=b;i++){
      vsledok=max(vsledok, D[2*interval.first+1][r-interval.second][i]+E[2*interval.first][interval.second-l][b-i]);
    }
    cout <<vsledok<<endl;
  }
  return 0;
}
```