Korešpondenčný seminár z programovania
XXXIII. ročník, 2015/16
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. série letnej časti

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

1. Zásuvky

Máme predlžovačku s n zásuvkami, do ktorej chceme vsunúť k normálnych úzkych zástrčiek a m širokých zástrčiek, ktoré po zasunutí zaberú miesto aj v susedných zásuvkách. Ako sa však dalo všimnúť v ukážkovom príklade, dve široké zástrčky vieme zastrčiť tak, že je medzi nimi len jedna voľná zásuvka.

Je jasné, že hlavný problém je, ako pozastrkávať široké zástrčky tak, aby zabrali čo najmenej miesta. Zdá sa, že vhodné miesto pre širokú zástrčku je na kraji predlžovačky. Ak ju totiž dáme na kraj, zaberie iba dve zásuvky. Z jednej strany totiž prečnieva ponad koniec predlžovačky. Jediná susedná zásuvka krajnej zásuvky bude určite zablokovaná, no do tretej zásuvky v poradí však môžeme dať aj úzku aj širokú zástrčku. Zbavili sme sa teda jednej širokej zástrčky a našu predlžovačku sme si skrátili na $n - 2$ zásuviek.

Túto myšlienku sa nám však oplatí zopakovať opäť, až kým nevyčerpáme všetky široké zástrčky. Tie umiestnime čo najviac na jeden kraj predlžovačky, čo najbližšie k sebe. Každá zaberie dve zásuvky a preto nám zostane $n - 2m$ zásuviek, do ktorých chceme vložiť k úzkych zástrčiek. Je jasné, že to sa bude dať spraviť iba ak je $k \leq n - 2m$.

Takéto riešenie nie je problém naprogramovať, stačí predsa v čase $O(1)$ overiť túto jedinú podmienku. Teda takmer. Špeciálny prípad nastane, ak bude $k = 0$, teda máme iba široké zástrčky. Vtedy nám postačuje $2m - 1$ zásuviek. Takéto riešenie nám už dá na testovači plný počet bodov.

Listing programu (C++)

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

int main() {
    long long n, m, k;
    cin >> n >> m >> k;
    n -= 2 * m;
    if (k == 0) {
        n++;
    }
    if (n < k) {
        cout << "nie\n";
    } else {
        cout << "ano\n";
    }
}
```

Riešenie, ktoré sme vymysleli, nazývame *pažravé*[1]. Všimli sme si, že je výhodné dať *jednu* širokú zástrčku na kraj predlžovačky. *Pažravo* sme ale usúdili, že je výhodné dať *všetky* široké zástrčky k sebe na kraj predlžovačky. Platí to však vždy? To musíme dokázať v popise.

Najlepšie sa takéto tvrdenie dokazuje nasledovne. Zoberieme si nejaké ľubovoľné platné rozmiestnenie zástrčiek do predlžovačky. Ukážeme, že takéto rozmiestnenie vieme upraviť na naše rozmiestnenie, ktoré má široké zástrčky vedľa seba na (ľavom) kraji predlžovačky.

Prvé, čo spravíme s ľubovoľným platným rozmiestnením, bude, že posunieme všetky zástrčky čo najviac doľava, aby sme odstránili zbytočné medzery (samozrejme, nejaké zásuvky zostanú voľné, lebo budú zakryté širokými zástrčkami). Následne, ak toto rozmiestnenie ešte nevyzerá ako naše riešenie, tak niektoré dve široké zástrčky nie sú pri sebe. Medzi nimi je teda jedna, alebo viac úzkych zástrčiek.

Ak „`o`" označuje prázdnu zásuvku, „`u`" zásuvku s úzkou zástrčkou a „`s`" zásuvku so širokou zástrčkou, tak kus predlžovačky, kde niečo takéto vznikne, môže vyzerať ako „`souuoso`". Nič však nepokazíme, ak tieto zástrčky prepojíme do tvaru „`sosouuo`". Akurát sme odstránili problém širokých zástrčiek, ktoré neboli vedľa seba. Opakovaním tohto postupu naozaj vytvoríme rovnaké riešenie ako náš program – široké zástrčky na kraji predlžovačky.

Vidíme teda, že ľubovoľné riešenie vieme prerobiť na nami navrhnuté riešenie. Preto je takéto riešenie určite správne.

[1] Po anglicky greedy.

Dávid

2. Zapisovanie trpaslíkov

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

Hrubá sila

Riešenie hrubou silou vyzerá tak, že postupujeme presne podľa zadania. Postupne si vygenerujeme každú permutáciu cifier zadaného čísla a potom všetky takto vytvorené čísla sčítame. V jazyku `C++` sa to dá robiť pomocou funkcie `next_permutation()`, ktorá generuje ďalšiu permutáciu poľa. V Pythone sa na to dá použiť knižnica `itertools`. Ak si počet cifier zadaného čísla označíme k, tak takéto riešenie má časovú zložitosť $O(k!)$ (k faktoriál) a na testovači by malo získať 2 body.

Listing programu (Python)

```
from itertools import permutations

n = int(input())

for i in range(n):
    c = input()
    print(sum(int(''.join(p)) for p in permutations(c)))
```

Optimálne riešenie

Najprv potrebujeme zistiť, koľko rôznych permutácií k cifier vlastne existuje. Ak chceme z k cifier vyrobiť číslo, dôležité je, v akom poradí ich za seba naukladáme. Na prvé miesto môžeme dať ľubovoľnú z cifier, takže máme k možností. Na druhé miesto už nemáme k možností, ale iba $k-1$. Jedna cifra (aj keď nevieme ktorá) totiž leží na prvej pozícii. Preto máme pre druhé miesto iba $k-1$ možností, čo je dokopy pre prvé dve miesta $k \cdot (k-1)$ možností.

Neprekvapivo, pre tretie miesto máme $k-2$ možností a tak ďalej. Na poslednom, k-tom mieste máme len 1 možnosť, lebo nám zostala posledná nezaradená cifra. Dokopy máme teda $k \cdot (k-1) \cdot (k-2) \ldots 2 \cdot 1$ možností. Takýto súčin čísel od 1 po k zvykneme tiež označovať $k!$ (k faktoriál).

Označme si cifry nášho čísla zľava doprava ako $a_k\ a_{k-1} \ldots a_2\ a_1$. Toto číslo si teda môžeme zapísať aj ako:

$$10^{k-1} \cdot a_k + 10^{k-2} \cdot a_{k-1} + \ldots + 10^1 \cdot a_2 + 10^0 \cdot a_1$$

Je jasné, že to, koľko zaváži daná cifra, je dané aj jej poradím v čísle. Skúsme teraz vypočítať, akú hodnotu pridá cifra a_1 do súčtu všetkých permutácií. Ak dáme cifru a_1 na prvú pozíciu zľava, tak nám do súčtu pridá $10^{k-1}a_1$. Potrebujeme už len zistiť, v koľkých permutáciách bude a_1 na prvom mieste. Keď si však takýmto spôsobom zafixujeme prvú cifru, ostane nám $k-1$ cifier, ktoré chceme uložiť na $k-1$ pozícií. Máme teda $(k-1)!$ rôznych čísel, kedy je a_1 na prvom mieste.

Ak dáme cifru a_1 na druhú pozíciu zľava, bude pridávať do súčtu hodnotu $10^{k-2}a_1$ a opäť $(k-1)!$ krát, lebo zafixovaním druhej pozície nám opäť ostane $k-1$ cifier na $k-1$ pozícií. To isté bude platiť pre ľubovoľné miesto, kam našu cifru a_1 uložíme. Ak toto všetko sčítame, dostaneme celkovú hodnotu, ktorú do súčtu permutácií pridá cifra a_1 a táto hodnota bude:

$$10^{k-1}a_1(k-1)! + 10^{k-2}a_1(k-1)! + \ldots + 10^1a_1(k-1)! + 10^0a_1(k-1)!$$

Po vyňatí a_1 a $(k-1)!$ získame jednoduchší tvar:

$$a_1(k-1)!(10^{k-1} + 10^{k-2} + \ldots 10 + 1)$$

Táto istá úvaha však platí aj pre ľubovoľnú cifru, nie len pre a_1. Keďže sú cifry od seba nezávislé, celkový súčet permutácií bude určite rovný súčtu hodnôt, ktoré pridajú jednotlivé cifry. Výsledný súčet všetkých permutácií preto bude

$$a_1(k-1)!(10^{k-1}+10^{k-2}+\ldots 10+1)+a_2(k-1)!(10^{k-1}+10^{k-2}+\ldots 10+1)+\ldots+a_k(k-1)!(10^{k-1}+10^{k-2}+\ldots 10+1)$$

čo môžeme opäť upraviť na oveľa jednoduchší tvar:

$$(a_1 + a_2 + \ldots + a_k)(k-1)!(10^{k-1} + 10^{k-2} + \ldots 10 + 1)$$

Výsledok teda vieme vypočítať ako súčin troch členov. Prvý z nich je súčet cifier, druhý je $(k-1)!$, čo je $(k-1)(k-2)\ldots 1$ a tretí súčet mocnín 10. Každý z týchto členov vieme vypočítať v čase $O(k)$, čo je teda

výsledná časová zložitosť nášho programu. Pamäťová zložitosť je dokonca konštantná $O(1)$, keďže cifry zadaného čísla vieme počítať postupne jeho delením 10.

Listing programu (C++)

```cpp
#include <iostream>

using namespace std;

int main(){
    int n;
    cin >> n;
    for (int i=0; i<n; i++){
        long long a;
        cin >> a;

        int pocet_cifier = 0;
        int sucet_cifier = 0;
        while(a > 0){
            pocet_cifier++;
            sucet_cifier += a % 10; // ziskame poslednu cifru cisla a
            a /= 10;                // odstranime poslednu cifru cisla a
        }

        long long faktorial=1;
        for (int j=1; j<pocet_cifier; j++)
            faktorial *= (long long)(j);

        long long jednotkove_cislo = 0;
        for (int j=0; j<pocet_cifier; j++)
            jednotkove_cislo = 10LL * jednotkove_cislo + 1LL;

        cout << faktorial * jednotkove_cislo * sucet_cifier << endl;
    }
}
```

Listing programu (Python)

```python
from math import factorial as f
n = int(input())

for i in range(n):
    c = input()
    print(sum(map(int, c)) * int('1' * len(c)) * f(len(c) - 1))
```

Zygro

3. Záchrana princeznej

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

Predstavíme si dve riešenia: riešenie hrubou silou a optimálne riešenie.

Hrubá sila $O(n^2)$

Príklad vieme vyriešiť veľmi jednoducho vyskúšaním všetkých možných začiatkov záchrany: Spočítame, koľko hodín práce by musel Jimi preskočiť, keby začal so záchranou práve v i-tu hodinu. Zo všetkých i vyberieme také, kedy je počet preskočených hodín práce minimálny.

Počet preskočených hodín spočítame tak, že v cykle prejdeme vstupné pole od i-tej hodiny, pričom si budeme počítať počet prejdených `P` a `H`. Ak dosiahneme počet `H` rovný c, dosiahli sme potrebný počet hracích hodín a počet `P` je počet obetovaných pracovných hodín. Stačí teda počet `P` porovnať s doterajším minimom a ak je menší, našli sme nové minimum.

Keď od hodiny i po koniec rozvrhu ostáva menej ako c hodín, vypíšeme najmenší počet vynechaných `P`.

Keďže pre každý začiatok môžeme prejsť až na koniec poľa, časová zložitosť je $O(n^2)$.

Optimálne riešenie $O(n)$

Môžeme si všimnúť, že v predošlom riešení robíme množstvo práce viackrát. Ak totiž poznáme počet `P`-čok (označme ho p_i) v úseku začínajúcom na i, tak vieme, že počet `P`-čok v úseku začínajúcom na $i+1$ nemusí byť veľmi odlišný. Mohli by sme teda zvýšiť i na $i+1$ a nezačať počítať `P` a `H` od $i+1$ ale rovno od $i+c+p_i$ (teda odtiaľ, kde sme skončili prechod poľa pre predošlé i), kým nedosiahneme potrebný počet `H`-čok.

My budeme postupovať podobne, akurát teraz nebudeme skúšať všetky začiatky, ale prejdeme cez všetky konce.

Budeme sa teda vždy pozerať na časť poľa, ktorá je určená začiatkom a koncom[2]. Tiež si udržiavame počet `P` a počet `H` v tejto časti poľa.

[2] Prístup, ktorým vyriešime túto úlohu sa nazýva „dvaja bežci" – jeden ukazuje na začiatok, druhý na koniec a postupne nimi prejdeme pole.

Začneme so začiatkom aj koncom na prvom políčku poľa. V každom kroku cyklu posunieme koniec úseku o jedno políčko ďalej. Kým je počet `H` v našom úseku väčší alebo rovný c, posúvame začiatok úseku doprava. Tým dosiahneme najkratší úsek s týmto koncom, ktorý má aspoň c `H`-čok. Keďže je najkratší, má aj minimálny počet `P`-čok. Počet `P`-čok porovnáme s medzivýsledkom a zapamätáme si menší.

Po prechode poľom vypíšeme minimum.

Každé políčko poľa sme navštívili raz koncom a najviac raz začiatkom, teda celková časová zložitosť je $O(n)$.

Listing programu (C++)

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

using namespace std;

int main () {
    int c;
    string vstup;

    cin >> c;
    cin >> vstup;
    int n = vstup.size();

    int zaciatok = 0;
    int H = 0, P = 0;
    int vysledok = n;

    for (int koniec = 0; koniec < n; koniec++) {
        if (vstup[koniec] == 'P') {
            P++;
        } else {
            H++;
        }

        while (H >= c && zaciatok < koniec) {
            vysledok = min(P, vysledok);
            if (vstup[zaciatok] == 'P') {
                P--;
            } else {
                H--;
            }
            zaciatok++;
        }
    }
    cout << vysledok << endl;
    return 0;
}
```

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

4. Žiarivý zádrhel

Hrubá sila

Najjednoduchšia vec, ktorá sa dá spraviť, je postupne vyskúšať sčítať všetky k-tice čísel od 1 po n, až kým nenarazíme na niektorú so súčtom s a tú vypísať. V Pythone sa takéto riešenie hrubou silou dá veľmi jednoducho naprogramovať pomocou knižnice `itertools`.

Listing programu (Python)

```
from itertools import combinations

n, k, s = map(int, input().split())

for k_tica in combinations(range(1, n + 1), k):
    if sum(k_tica) == s:
        print('␣'.join(map(str, k_tica)))
        break
else:
    # else sa za for-cyklom vykoná vtedy, ak cyklus dobehne do konca
    # bez prerušenia break-om
    print(-1)
```

V najhoršom prípade prejdeme všetký k-tice, ktorých je $\binom{n}{k}$ a každú spracujeme v čase $O(k)$. Časovú zložitosť by sme mohli voľne zhora ohraničiť ako $O(k \cdot n^{n/2})$ alebo ako $O(k \cdot 2^n)$ (ak zhora ohraničíme kombinačné číslo veľkosťou súčtu n-tého riadku Pascalovho trojuholníka $\binom{n}{k} \leq 2^n$).

Optimálne riešenie

Ako prvú vec si môžeme všimnúť, že úloha by sa nám výrazne zjednodušila, keby každý súčet mohol byť vybraný ako súvislý úsek čísel. Súčet takejto aritmetickej postupnosti vieme zrátať v konštantnom čase (pre

súčet čísel $1, 2, \ldots n$ existuje známy vzorec $\frac{n(n+1)}{2}$) a takýchto úsekov je menej. Teda riešenie prechádzajúce všetky takéto postupnosti by malo výrazne lepšiu časovú zložitosť ($O(n+k)$). A nielen to, už zo súčtu by sme rovno vedeli povedať, ktoré čísla v tejto postupnosti budú.

Takáto postupnosť sa však nedá nájsť vždy. Napríklad pre vstup (n, k, s) `6 3 11` je riešenie `2 3 4` primálo a `3 4 5` priveľa. Možno ale existuje postupnosť čísel s takýmto súčtom, ktorá je skoro súvislá.

Na náš príklad sa vieme pozrieť aj takto: keď z `2 3 4 5` vynecháme najväčšie číslo, súčet zvyšku bude primalý. Ak však vezmeme to najmenšie, súčet zvyšku bude priveľký. Nevieme vynechať nejaké číslo v strede tak, aby nám to sedelo? Vieme, a dokonca takéto čislo naozaj vieme vynechať vždy. Vynechávaním postupne čísel $2, 3, 4, 5$ sa nám zmenšuje súčet úseku a tento súčet dosiahne všetky hodnoty medzi $3 + 4 + 5 = 12$ a $2 + 3 + 4 = 9$.

Z postupnosti teda potrebujeme vynechať jedno číslo: (súčet postupnosti $- s$).

Už len potrebujeme nájsť takúto postupnosť, z ktorej vieme jedno číslo vynechať a dostaneme súčet s.

Označme si prvé číslo tejto postupnosti ako a. Vieme, že táto postupnosť bude vyzerať ako $a, a+1, \ldots a+k$. Chceme, aby súčet $a + (a+1) + \ldots + (a+k-1) \leq s$, a pritom a bolo čo najväčšie:

$$a \cdot k + (0 + 1 + \ldots + (k-1)) \leq s$$

$$a \leq \frac{(s - (0 + 1 + \ldots + (k-1)))}{k}$$

Keďže chceme aby a bolo celé číslo, potrebujeme zaokrúhliť pravú stranu rovnice nadol a dostávame priamo vzorec na jeho výpočet. Zaokrúhlenie za nás bude robiť celočíselné delenie k na záver.

Zhrňme si teda riešenie. Najprv si vyrátame, ktorá súvislá postupnosť dĺžky $k+1$ nás zaujíma. Z tejto postupnosti potom vynecháme jedno číslo tak aby súčet zvyšku bol presne s a postupnosť vypíšeme bez jedného čísla.

Časová zložitosť tohto algoritmu je $O(k)$ – nájsť postupnosť $k+1$ čísel vieme v konštantnom čase, rovnako aj nájdenie vhodného čísla na vynechanie. Musíme však vypísať k čísel.

Algoritmus sa dá implementovať s pamäťovou zložitosťou $O(1)$. V Pythone sa však rýchlejšie vypisuje po celých riadkoch. Preto si najprv vytvoríme celú postupnosť a potom ju naraz vypíšeme – implementácia teda používa $O(k)$ pamäte.

Listing programu (Python)

```
n, k, s = map(int, input().split())

# zratame si najvacsi a najmensi dosiahnutelny sucet
low = sum(range(1, k + 1))
high = sum(range(n - k + 1, n + 1))

if s < low or s > high:
    print(-1)
    exit(0)

# najmensie cislo, ktore chceme zobrat v suvislej postupnosti
# low = 1 + 2 + ... + k-1 + k, k vsak odratat nechceme
a = (s - low + k) // k

# sucet [a,a+1..a+k], teda k+1 cisel, z ktorych 1 chceme vynechat
b = (a * (k + 1)) + (k * (k + 1) // 2)

res = [x for x in range(a, a + k + 1) if s + x != b]

print (' '.join(map(str, res)))
```

5. Otepľovanie v Absurdistane

Baklažán
(max. 7 b za popis, 8 b za program)

Prehľadávanie do šírky

V tomto riešení budeme hojne používať štandardný algoritmus *prehľadávanie do šírky* (po anglicky *breadth-first search*, skratka BFS). O tomto algoritme existuje veľa materiálov všelikde po internete[3], preto ho v tomto vzorovom riešení nebudeme rozoberať.

[3] napríklad https://ksp.mff.cuni.cz/tasks/25/cook4.html, https://people.ksp.sk/~baklazan/UFO-Prask_2016_jar/zbornik_Prask.pdf (od strany 20), ale aj https://en.wikipedia.org/wiki/Breadth-first_search

Naivné riešenie

Asi najjednoduchšie riešenie, ktoré nám napadne (ak dostatočne dobre poznáme BFS) je simulovať postupne jednotlivé roky zatápania Absurdistanu a každý rok označiť novozatopené oblasti tak, ako by to urobila Kartografická Spoločnosť Patriotov (KSP). Každý rok pri tom môžeme postupovať nasledovne:

1. Spustíme BFS, ktorému povolíme prehľadávať iba zatopené políčka, naraz zo všetkých políčok, ktoré boli zatopené už minulý rok[4]. Keď v tomto BFS z políčka A objavíme dosiaľ neobjavené novozatopené políčko B, zaznačíme si, že B patrí do rovnakého jazera ako A. Takýmto spôsobom bude každé novozatopené políčko priradené tomu jazeru, ktoré sa k nemu „dostalo najskôr", teda jazeru, ku ktorému bolo dané políčko po vode najbližšie (čo je presne to, čo podľa zadania chceme).

   Ostáva ešte doriešiť, ako zabezpečiť, aby v prípade remízy vyhralo jazero s nižším číslom. To sa dá vyriešiť napríklad nasledovne:

   - BFS neimplementujeme s frontou, ale budeme ho robiť po kolách. V každom kole vezmeme všetky políčka, ktoré boli objavené v predchádzajúcom kole (špeciálne v prvom kole vezmeme všetky políčka zatopené už minulý rok) a pozrieme sa na všetkých ich novozatopených susedov. Tých z nich, ktorých sme doteraz neobjavili, si zaznačíme ako objavených v tomto kole. V prvom kole teda objavíme novozatopené políčka, ktorých vzdialenosť od najbližšieho políčka zatopeného v minulých rokoch je 1, v druhom kole objavíme políčka vzdialené 2 atď.
   - Ak v nejakom kole objavíme jedno políčko P viackrát (z rôznych smerov), pričom políčka, z ktorých sme ho objavili, patria do rôznych jazier, políčko P priradíme tomu z nich, ktoré má najmenšie číslo.

2. Následne prejdeme cez všetky políčka obdĺžnika v rovnakom poradí ako KSP (teda zhora nadol, zľava doprava). Vždy, keď nájdeme novozatopené políčko R, ktoré ešte nemá určené číslo jazera, ktorému patrí (teda sme našli novovzniknuté jazero), vezmeme najmenšie číslo, ktoré sme pri označovaní jazier ešte nepoužili a týmto číslom označíme políčko R. Následne spustíme z políčka R BFS idúce iba po novozatopených políčkach. Všetky políčka, ktoré toto BFS objaví, označíme rovnakým číslom jazera ako R.

Odhad zložitosti

Prvá časť nášho postupu (rozširovanie už existujúcich jazier) je BFS na grafe s najviac $r \cdot s$ vrcholmi a $2 \cdot r \cdot s$ hranami, teda má zložitosť $O(r \cdot s)$. V druhej časti musíme prejsť cez celú mapu, čo nás bude stáť $O(r \cdot s)$ času a okrem toho ešte robíme nejaké BFS-ká. Keďže tieto BFS-ká púšťame v každom novovzniknutom jazere iba raz (keď objavíme prvé jeho políčko), navštívia dokopy každé novozatopené políčko neoznačené v kroku 1 práve raz, teda dokopy nám budú trvať nanajvýš $O(r \cdot s)$ času. Odsimulovanie jedného roka nám teda bude trvať $O(r \cdot s)$ času a potrebujeme odsimulovať H rokov, preto časová zložitosť celého algoritmu je $O(H \cdot r \cdot s)$.

Pamäťová zložitosť je $O(r \cdot s)$, nakoľko pre každé políčko si pamätáme iba konštantne veľa údajov a počas BFS máme v zozname vrcholov určených na prehľadanie vždy najviac $r \cdot s$ vrcholov.

Rýchlejšie riešenie

Naše riešenie sa ešte dá podstatne zrýchliť. Stačí si uvedomiť, že veľa vecí počítame zbytočne.

Prvá zbytočná vec, ktorú robíme, je, že aj v rokoch, keď sa nič neudialo (žiadne políčko nebolo zatopené) sa snažíme označovať novozatopené políčka a strávime tým $O(r \cdot s)$ času. Prvou optimalizáciou by teda mohlo byť na začiatku si nejako poznačiť, v ktorých rokoch sa nič neudialo a tieto roky vynechať. To môžeme urobiť napríklad nasledovne:

1. Na začiatku si vyrobíme pomocné pole dĺžky $H + 1$, kde si budeme pre každý rok pamätať, či sme videli nejaké políčko, ktoré bude zatopené v danom roku.
2. Následne postupne prejdeme všetky políčka obdĺžnika a pre každé políčko sa pozrieme, v ktorom roku bude zatopené (akú má nadmorskú výšku). K príslušnému roku si potom v našom poli poznačíme, že sa v ňom niečo udeje.

Toto celé nám bude trvať čas $O(r \cdot s + H)$ ($+H$ je tam preto, lebo aj vytvorenie (vynulovanie) poľa stojí nejaký čas) a urobíme to za celý algoritmus iba raz, na začiatku. Následne budeme úlohu riešiť rovnako ako predtým, teda simulovať situáciu rok po roku. Na začiatku simulovania jedného roka sa ale najprv pozrieme, či sa v danom roku niečo zmení a ak nie, daný rok preskočíme. Takto teda roky, keď sa niečo deje, odsimulujeme v čase $O(r \cdot s)$ a roky, keď sa nič nedeje, v čase $O(1)$. Časová zložitosť nášho algoritmu je teda $O(r \cdot s \cdot h + H)$

[4] Toto vieme jednoducho dosiahnuť tak, že na začiatku do fronty vložíme všetky zatopené políčka.

(tentoraz je tam to $+H$ preto, lebo aj v rokoch, keď sa nič neudialo, sme minuli $O(1)$ času), kde h je počet rokov, keď sa niečo stane. Keďže rôznych nadmorských výšok môže byť na mape najviac $r \cdot s$, platí $h \leq r \cdot s$, preto je zložitosť nášho algoritmu v najhoršom prípade $O(r \cdot s \cdot \min(H, r \cdot s) + H)$. Zlepšili sme sa teda v prípadoch, keď je H ďaleko väčšie ako $r \cdot s$ a v prípadoch, keď je na mape málo rôznych nadmorských výšok.

Vidíme, že v prípadoch, keď je na mape veľa rôznych nadmorských výšok, sme si veľmi nepolepšili. V takýchto prípadoch bude totiž veľa rokov, v ktorých sa zatopí iba maličké územie, ale my kvôli nemu urobíme veľa práce. To ale tiež vieme zlepšiť. V prvom kroku simulácie jedného roka spúšťame BFS zo všetkých doteraz zatopených políčok. Mnohé z nich ale nemusia susediť so žiadnym novozatopeným políčkom a preto ich prehľadaním nič neobjavíme.

V skutočnosti by nám teda stačilo BFS spustiť iba z tých políčok, ktoré susedia s niektorým novozatopeným políčkom. Týchto políčok môže byť najviac štyrikrát viac ako novozatopených políčok, keďže každé novozatopené políčko susedí s najviac štyrmi, preto by nám BFS trvalo iba $O(Z_i)$ času, kde i je poradové číslo daného roku a Z_i je počet políčok zatopených v tomto roku.

V druhom kroku simulácie jedného roka prechádzame celú mapu a hľadáme neoznačené novozatopené políčka, čo nám trvá $O(r \cdot s)$ času. Ak by sme namiesto toho prechádzali iba cez novozatopené políčka a ostatné preskakovali, trvalo by nám to iba $O(Z_i)$ času.

Nakoniec z niektorých políčok púšťame BFS. Všetky tieto BFS-ká ale dokopy prehľadajú iba všetky neoznačené novozatopené políčka, ktorých je najviac $O(Z_i)$, teda dokopy trvajú najviac $O(Z_i)$ času. Vylepšený algoritmus teda môže vyzerať nasledovne:

1. Úplne na začiatku si vytvoríme zoznam $H + 1$ zoznamov: pre každý rok zoznam políčok, ktoré budú v tomto roku zatopené, v poradí, v akom ich navštívi KSP (teda po riadkoch zhora nadol, v rámci riadka zľava doprava).
2. Prejdeme celú mapu, pričom každé políčko zapíšeme do zoznamu prislúchajúcemu roku, v ktorom bude dané políčko zatopené.
3. Následne simulujeme rok po roku, pričom simulácia jedného roka vyzerá nasledovne:
    1. Pozrieme sa na zoznam políčok zaplavených v tomto roku. Ak je prázdny, daný rok môžeme preskočiť. Ak nie, pre všetky políčka z tohto zoznamu sa pozrieme na všetkých ich susedov. Z tých susedov, ktorí boli zatopení už skôr, spustíme BFS, v ktorom poprirađujeme novozatopené políčka ich jazerám.
    2. Prejdeme znova zoznam políčok zatopených v danom roku a vždy keď nájdeme nejaké ešte nepriradené políčko, označíme toto políčko novým číslom jazera a spustíme z neho BFS, kde pooznačujeme aj zvyšok tohto novovzniknutého jazera.

Rok číslo i takto odsimulujeme v čase $O(Z_i)$, teda dokopy nám všetky simulácie zaberú $O(Z_1 + Z_2 + \cdots + Z_{H+1}) = O(r \cdot c)$ času. Celý algoritmus teda pobeží v čase $O(r \cdot c + H)$.

Listing programu (C++)

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

using namespace std;

int r, s, h;
vector< vector< pair<int, int > > > zaplavime;  // zaplavime[kedy] = zoznam policok
vector< vector<int> > vyska;                    // sentinely maju vysku h+1, uz vyriesene -1
vector< vector<int> > jazero;                   // sentinely su NEZAPLAVENE
int NEZAPLAVENE;
int cislo_jazera = 0;

int dx[] = {-1, 0, 1, 0};
int dy[] = {0, -1, 0, 1};

vector<int> zaplav_susedov(vector<int> &v, int uroven){
    vector<int> susedia;
    for(int i=0; i<v.size(); i+=2){
        int y = v[i], x = v[i+1];
        for(int s=0; s<4; s++){
            int ny = y + dy[s];
            int nx = x + dx[s];
            if (vyska[ny][nx] == uroven && jazero[ny][nx] > jazero[y][x]){
                susedia.push_back(ny);
                susedia.push_back(nx);
                jazero[ny][nx] = jazero[y][x];
            }
        }
    }

    vector<int> unikatni_susedia;
    for(int i=0; i<susedia.size(); i+=2){
```

```
        int y = susedia[i], x = susedia[i+1];
        if (vyska[y][x] != -1){
            vyska[y][x] = -1;
            unikatni_susedia.push_back(y);
            unikatni_susedia.push_back(x);
        }
    }
    return unikatni_susedia;
}

void zaplav_hladinu(int hladina){
    // najdeme tych, co susedia so zaplavenymi
    vector<int> na_zaplavenie;
    for(int i=0; i<zaplavime[hladina].size(); i++){
        int y = zaplavime[hladina][i].first;
        int x = zaplavime[hladina][i].second;
        for(int s=0; s<4; s++){
            int ny = y + dy[s];
            int nx = x + dx[s];
            if (jazero[ny][nx] != NEZAPLAVENE){
                na_zaplavenie.push_back(ny);
                na_zaplavenie.push_back(nx);
            }
        }
    }

    while(na_zaplavenie.size() > 0)
        na_zaplavenie = zaplav_susedov(na_zaplavenie, hladina);

    // najdeme tych, co sme este nezaplavili
    for(int i=0; i<zaplavime[hladina].size(); i++){
        int y = zaplavime[hladina][i].first;
        int x = zaplavime[hladina][i].second;
        if (jazero[y][x] == NEZAPLAVENE){
            jazero[y][x] = cislo_jazera;
            cislo_jazera++;
            vyska[y][x] = -1;
            vector<int> na_zaplavenie;
            na_zaplavenie.push_back(y);
            na_zaplavenie.push_back(x);
            while(na_zaplavenie.size() > 0)
                na_zaplavenie = zaplav_susedov(na_zaplavenie, hladina);
        }
    }
}

int main(){
    scanf("␣%d␣%d␣%d", &r, &s, &h);
    NEZAPLAVENE = (r+2)*(s+2);
    zaplavime.resize(h+1);
    vyska.resize(r+2, vector<int>(s+2, h+1));
    jazero.resize(r+2, vector<int>(s+2, NEZAPLAVENE));

    for(int i=1; i<r+1; i++)
        for(int j=1; j<s+1; j++){
            int v;
            scanf("␣%d", &v);
            vyska[i][j] = v;
            zaplavime[v].push_back(make_pair(i, j));
        }

    for(int hladina=0; hladina<=h; hladina++)
        zaplav_hladinu(hladina);

    for(int i=1; i<r+1; i++)
        for(int j=1; j<s+1; j++)
            printf("%d%c", jazero[i][j], (j==s) ? '\n' : '␣');
    return 0;
}
```

Rasťo

6. Oddaľovanie roboty

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

Najprv si ukážeme, ako sa dal riešiť ľahší problém, kedy chceme stihnúť všetky úlohy. Potom si ukážeme, ako sa dal riešiť zložitejší problém, kedy môžeme nejaké úlohy vynechať.

Ľahšia úloha – riešenie jednoduchou myšlienkou

Intuitívne by sa mohlo zdať, že keď chceme maximalizovať čas flákania sa na začiatku, tak chceme riešiť úlohy čo najneskôr, teda čo najbližšie k ich deadlajnu. Ináč povedané, ak má úloha i deadline d_i a jej splnenie trvá t_i času, tak by sme ju ideálne chceli začať plniť v čase $d_i - t_i$.

Bohužiaľ toto sa nedá vždy urobiť. Napríklad, ak máme úlohu, ktorú treba splniť do času 6 a jej plnenie trvá 3 jednotky času a úlohu, ktorej deadline je 5 a máme na ňu tiež 3 jednotky času. Podľa vyššie spomenutej stratégie chceme prvú úlohu začať v čase 3 a skončiť v čase 6 a druhú úlohu chceme začať v čase 2 a skončiť v čase 5. To je ale problém, lebo nevieme robiť na dvoch úlohách naraz.

Nevieme teda obe úlohy splniť tesne pred deadlajnom. Máme na výber z dvoch možností: buď splníme tesne pred deadlajnom prvú, alebo druhú úlohu. Náš cieľ je flákať sa čo najdlhšie na začiatku, takže bude lepšie, ak

splníme tesne pred deadlajnom úlohu, ktorá má neskorší deadline. Čiže druhú úlohu začneme plniť v čase 0, skončíme v čase 3, a potom robíme prvú úlohu, ktorú dokončíme v čase 6 – tesne pred deadlajnom.

Ak si to zosumarizujeme, tak sme vlastne urobili dve pozorovania. Prvé pozorovanie je, že chceme riešiť úlohu čo najbližšie k jej deadlajnu. Druhé pozorovanie je, že občas sa to nedá, keď sa intervaly plnenia úloh prekrývajú. Tento prípad vieme riešiť tak, že úlohu, ktorej deadline je neskoršie, budeme končiť presne v čase deadlajnu a intervaly pre ostatné úlohy trocha poposúvame doľava (na časovej osi).

Ľahšia úloha – algoritmus

Táto stratégia sa dá zovšeobecniť do algoritmu. Najprv utriedime všetky úlohy podľa deadlajnu zostupne, a potom prechádzame cez takto utriedené úlohy. Počas tohto prechodu si udržiavame čas x – kedy sme začali plniť poslednú úlohu v našom poradí. Zistime, kedy začneme plniť i-tu úlohu v našom poradí:

- Ak $d_i \geq x$, úlohu nemôžeme začať plniť tesne pred deadlajnom, lebo v čase d_i už riešime jednu z nasledujúcich úloh. Začneme preto túto úlohu plniť už v čase $x - t_i$.
- Inak začneme úlohu plniť v čase $d_i - t_i$ (tesne pred deadlajnom).

Potom nastavíme x na novú hodnotu a pokračujeme ďalšou úlohou. To, čo nám na záver zostane v premennej x je náš výsledok – čas, kedy sme začali plniť prvú úlohu.

Časová zložitosť tohto riešenia je $O(n \log n)$, kde n je počet úloh. Triedenie je najpomalšia operácia. Zvyšok je iba prechod poľom.

Pamäťová zložitosť je $O(n)$, lebo si musíme pamätať všetky úlohy.

Ľahšia úloha – dôkaz správnosti

Pri takýchto greedy algoritmoch je dôležité si zdôvodniť, prečo náš algoritmus skutočne nájde optimálne riešenie.

Zoberme si **ľubovoľné poradie úloh, ktoré dáva najlepšie riešenie**, teda nám dovolí flákať sa na začiatku najdlhšie a porovnajme toto riešenie **s riešením, ktoré vytvorí náš algoritmus**. Chceme ukázať, že naše riešenie nám dovolí sa flákať na začiatku aspoň tak dlho ako optimálne riešenie.

Zoberme si z optimálnej postupnosti úlohu i, ktorá má deadline najneskôr a presuňme ju na koniec postupnosti úloh. Takto dosiahneme to, že túto úlohu budeme končiť presne v čase jej deadlajnu. Ak sme v optimálnom poradí v čase od $d_i - t_i$ riešili inú úlohu, musíme posunúť interval jej riešenia doľava (na časovej osi) a takisto aj intervaly ostatných predošlých úloh.

Dôležité je všimnúť si, že intervaly neposunieme viac doľava než bol pôvodný začiatok intervalu riešenia úlohy i. Tým, že sme presunuli úlohu i na koniec sme totiž využili časový interval medzi deadlajnom predposlednej úlohy a deadlajnom poslednej úlohy, i. Vďaka tomu nemusíme posúvať intervaly, pre úlohy ktoré sme v optimálnom riešení začali riešiť pred i, a teda touto operáciou určite nezhoršíme optimálne riešenie.

Teraz si zoberme úlohu s druhým najneskorším deadlajnom a presuňme ju na predposlednú pozíciu. Znova platia podobné argumenty, že si týmto nezhoršíme naše riešenie. Tento postup budeme opakovať pre všetky intervaly. Takto dostaneme poradie úloh, v akom by ich vykonával náš algoritmus, čiže vlastne vieme prerobiť ľubovoľné optimálne riešenie na naše riešenie a nezhoršiť ho. To znamená, že naše riešenie musí byť jedno z optimálnych.

Listing programu (C++)

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

using namespace std;

int main() {
    int N, K;
    scanf("␣%d␣%d", &N, &K);

    vector<pair<int, int>> ulohy(N);
    for (int i = 0; i < N; ++i) {
        scanf("␣%d␣%d", &ulohy[i].first, &ulohy[i].second);
    }

    // utried ulohy zostupne podla deadlinu.
    sort(ulohy.begin(), ulohy.end(), greater<pair<int, int>>());

    int zaciatok_poslednej_ulohy = INT_MAX;
    for (const pair<int, int>& uloha : ulohy) {
        int deadline = uloha.first;
        int trvanie = uloha.second;
        if (deadline > zaciatok_poslednej_ulohy) {
```

```
            if (zaciatok_poslednej_ulohy - trvanie < 0) {
                printf("Neda_sa_to_stihnut.\n");
                return 0;
            }
            zaciatok_poslednej_ulohy -= trvanie;
        } else {
            zaciatok_poslednej_ulohy = deadline - trvanie;
        }
    }

    printf("%d\n", zaciatok_poslednej_ulohy);
    return 0;
}
```

Pôvodná úloha

Teraz sa pozrime na úlohu pre $k > 0$. V tomto prípade už nejde riešiť túto úlohu greedy spôsobom. Ak by sme napríklad použili stratégiu, ktorou sme riešili prípad $k = 0$, tak sa nám nie vždy musí oplatiť zobrať úlohu s najneskorším deadlajnom. Ak práca na poslednej úlohe trvá veľmi dlho, oplatí sa nám ju vynechať a namiesto nej splniť ostatné úlohy.

Greedy algoritmy sa vyznačujú tým, že sú veľmi rýchle. Keď si poriadne pozrieme obmedzenia v zadaní úlohy, tak zistíme, že postačí aj pomalšie riešenie. Na riešenie použijeme techniku zvanú dynamické programovanie. Pri dynamickom programovaní si najprv jednoducho definujeme problém, a potom sa pomocou menších podproblémov snažíme vyskladať ten väčší. Najprv riešime malé problémy a z nich skladáme riešenia pre väčšie a väčšie problémy.

Náš problém je definovaný postupnosťou úloh a počtom úloh, ktoré chceme vynechať. Ako podproblém si teda definujeme, že chceme zistiť, ako najdlhšie sa môžeme flákať, ak chceme z prvých i úloh vynechať najviac j. Pričom stále chceme spracúvať úlohy v takom istom poradí, ako sme to robili v riešení pre ľahší problém – od najneskoršieho deadlajnu po najskorší. Riešenie pre podproblémy si budeme ukladať do tabuľky P, aby sme ich nemuseli rátať viackrát. Pričom na $P[i][j]$ **je uložené číslo, ako najdlhšie sa môžeme flákať ak chceme vynechať najviac j úloh z prvých i.**

Začnime triviálnym prípadom, keď chceme z prvých nula úloh nestihnúť najviac j úloh. V takom prípade sa môžeme nekonečne dlho flákať, takže do prvého riadku dáme samé nekonečná.

Zoberme si prípad, keď $i > 0$. V takom prípade máme dve možnosti. V optimálnom riešení $P[i][j]$ sa nachádza alebo nenachádza úloha i:

- Ak sa v riešení nenachádza úloha i, tak je toto riešenie zhodné s riešením pre $P[i-1][j-1]$. Čiže v takom prípade sa môžeme flákať $P[i-1][j-1]$ času.
- Ak sa v tomto riešení nachádza úloha i, tak odobratím i dostaneme optimálne riešenie pre podúlohu $P[i-1][j]$. Toto sa dá jednoducho dokázať sporom. Ak by to nebolo optimálne riešenie, tak potom odobratím úlohy i dostaneme lepšie riešenie pre podproblém $P[i-1][j]$, čo je spor.

  Otázkou ešte je, kedy začneme (a skončíme) plniť úlohu i. Buď ju skončíme skončíme v čase deadlajnu, alebo v čase, keď začneme prvú úlohu z riešenia $P[i-1][j]$. Ak je deadline pred začiatkom prvej úlohy v riešení pre $P[i-1][j]$, tak musíme úlohu i začať v čase $d_i - t_i$. Ak už počas d_i riešime jednu z neskorších úloh, začneme i riešiť v čase $P[i-1][j] - t_i$ a skončíme v čase $P[i-1][j]$.

Z oboch týchto možností si vyberieme tú, ktorá nám dovolí sa najviac flákať.

$$P[i][j] = \max(P[i-1][j-1], \min(d_i - t_i, P[i-1][j] - t_i))$$

Túto tabuľku si počítame po riadkoch a finálny výsledok sa nachádza na políčku $P[N][K]$ – ako najdlhšie sa môžeme flákať ak chceme vynechať najviac K úloh z prvých N.

Časová zložitosť tohto riešenia je $O(nk)$ a pamäťová tiež. Môžeme si všimnúť, že na vypočítanie nového riadku v tabuľke potrebujeme iba predchádzajúci riadok, takže vieme pamäťovú zložitosť vylepšiť na $O(k)$ tým, že si budeme pamätať iba posledné dva riadky tabuľky.

Listing programu (C++)

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

using namespace std;

int main() {
    int N, K;
```

```
    scanf("%d%d", &N, &K);

    vector<pair<int, int>> ulohy(N);
    for (int i = 0; i < N; ++i) {
        scanf("%d%d", &ulohy[i].first, &ulohy[i].second);
    }

    // utried ulohy zostupne podla deadlinu.
    sort(ulohy.begin(), ulohy.end(), greater<pair<int, int>>());

    vector<vector<int>> dp(N + 1, vector<int>(K + 1));
    fill(dp[0].begin(), dp[0].end(), INT_MAX);
    for (int i = 1; i <= N; ++i) {
        for (int j = 0; j <= K; ++j) {
            int deadline = ulohy[i - 1].first;
            int trvanie = ulohy[i - 1].second;
            // zaporne cislo znamena, ze sa to neda.
            int urob_ulohu = min(deadline - trvanie, dp[i - 1][j] - trvanie);
            int vynechaj_ulohu = j > 0 ? dp[i - 1][j - 1] : -1;
            dp[i][j] = max(vynechaj_ulohu, urob_ulohu);
        }
    }

    if (dp[N][K] < 0)
        printf("Neda sa to stihnut.\n");
    else
        printf("%d\n", dp[N][K]);
    return 0;
}
```

Mišo

7. Obézni brankári

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

Táto úloha na prvý pohľad vyzerala ako nejaká komplikovaná kombinátorika. V skutočnosti to tak vôbec nebolo, išlo len o efektívne spočítanie všetkých možností vyplnenia obdĺžnika. Poďme sa najprv pozrieť na spôsob, ako tieto možnosti spočítať hrubou silou. (Odporúčam nepreskakovať riešenie hrubou silou, sú v ňom veľmi dôležité myšlienky potrebné pre pochopenie vzorového riešenia.)

Hrubá sila – vypĺňanie po jednotlivých políčkach

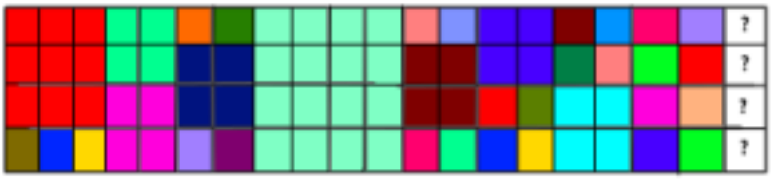
Poviete si: „Poďme vygenerovať všetky možnosti!", otázka však je ako. Jeden zo spôsobov by mohol byť, že si zoberieme prázdnu bránku a budeme do nej postupne všetkými možnosťami umiestňovať štvorce. Vždy, keď je bránka plná, započítame toto rozloženie do výsledku. Musíme ale nájsť taký spôsob vytvárania (prehľadávania) všetkých rozložení, aby sme každé rozloženie navštívili len raz.

Jeden zo spôsobov by mohol byť taký, že postupne prechádzame políčka bránky zhora dole, zľava doprava. Ak nájdeme nevyplnené políčko, skúsime sem vložiť štvorce všetkých možných veľkostí tak, aby na tomto políčku ležali ľavým horným rohom (vždy skúsime vložiť 1 štvorec a potom rekurzívne zavoláme funkciu, čo vyplní zvyšok bránky). Takto vždy zapĺňame políčka len smerom dole a doprava od našej pozície a všetky políčka smerom hore a doľava od nás sú už určite vyplnené. Pri vytváraní rozložení nevytvoríme žiadne viackrát (lebo každé dve sa líšia veľkosťou brankára aspoň na jednom políčku – ľavom hornom rohu nejakého brankára) a takisto vytvoríme každé.

Toto riešenie sa ľahko vymyslí, aj implementuje, no navštívi všetky rozloženia, ktorých je exponenciálne veľa.

Hrubá sila – vypĺňanie po stĺpcoch

Čo ak by sme mali už vygenerované všetky bránky veľkosti $4 \times n$? Dokážeme z týchto bránok vygenerovať všetky bránky veľkosti $4 \times (n+1)$? Odpoveď znie, áno, a budeme to robiť nasledovne. Predstavme si, že máme už vygenerované konkrétne rozloženie veľkosti $4 \times n$:



Oblasť jednej farby je jeden brankár. Otázniky v poslednom stĺpci reprezentujú nový stĺpec, ktorý chceme pridať (teraz budeme pridávať brankárov do stĺpca tak, aby boli zarovnaní podľa pravého kraja – teda aby sme presne vyplnili obdĺžnik $4 \times (n+1)$). Je jasné, že v pôvodnom rozložení musíme aj niečo zmeniť, inak by sme mohli predsa pridať len samých jednotkových brankárov. Avšak, ak budeme meniť ľubovoľne, môžme dôjsť k problému generovania rovnakého rozloženia viackrát.

Do stĺpca s otáznikmi umiestníme ľubovoľne veľkých brankárov, ktorí sa tam zmestia. V pôvodnom rozložení budú však môcť prekryť (a teda nahradiť) **len jednotkových brankárov**. Na našom obrázku máme napravo celkom pekné množstvo jednotkových brankárov. Jedinému brankára, ktorého si nemôžeme dovoliť do stĺpca s otáznikmi umiestniť je brankár veľkosti 4×4, pretože ten by musel prekryť tyrkysového brankára 2×2 a vieme, že takého prekryť nemôžeme.

Vždy je najviac osem možností, ako umiestniť brankárov do stĺpca s otáznikmi (odporúčam si ich vypísať na papier), pre každú bránku veľkosti $4 \times n$ je teda celkom ľahko možné vygenerovať najviac 8 bránok veľkosti $4 \times (n+1)$.

Ukážme si ešte, že takýmto spôsobom vygenerujeme každé rozloženie a žiadne nevygenerujeme dvakrát. Zoberme si nejaké konkrétne rozloženie $4 \times n$ a každého brankára, ktorý sa dotýka pravého okraja bránky nahraďme jednotkovými brankármi a potom odstráňme posledný stĺpec. Týmto spôsobom dostaneme rozloženie veľkosti $4 \times (n-1)$, z ktorého bolo naše rozloženie veľkosti $4 \times n$ vygenerované. Vidíme teda, že každé rozloženie $4 \times n$ vieme vygenerovať, ak už máme vygenerované všetky rozloženia veľkosti $4 \times (n-1)$. Každé dve rozloženia veľkosti $4 \times n$ sa budú líšiť buď v poslednom stĺpci, alebo v rozloženiach $4 \times (n-1)$, z ktorých boli vygenerované.

Budeme si teda postupne generovať rozloženia pre stále dlhšiu a dlhšiu bránku, až sa nakoniec dostaneme po našu cieľovú šírku bránky. Toto riešenie je však pomerne pomalé. Z každej bránky $4 \times n$ vygenerujeme prinajhoršom až 8 bránok veľkosti $4 \times (n+1)$ a tak časová zložitosť môže byť až exponenciálna (aj pamäťová, ak si udržiavame všetky doteraz vygenerované rozloženia).

Poďme sa teda pozrieť na to, ako toto riešenie výrazne zefektívniť. (Pre porozumenie ďalším riešeniam je však veľmi dôležité v prvom rade porozumieť aktuálnemu.)

Brankári si udržujú líniu – lineárne riešenie

V predchádzajúcom riešení môžeme spraviť nasledujúce pozorovanie: to, akých brankárov môžeme pridať do posledného stĺpca s otáznikmi závisí len a len od počtu jednotkových brankárov úplne napravo v každom riadku. Napríklad, ak pravý okraj nejakej bránky vyzerá takto (prázdne políčka bez otáznikov znamenajú hocijakých nejednotkových brankárov):

tak tam vieme doplniť nejakých jednotkových, dvojkových a dole aj trojkového brankára.

Úplne všetky rozloženia, ktoré končia takto, sa dajú do otáznikov doplniť rovnakými spôsobmi. Koľko je spôsobov, ktorými môžu rozloženia končiť? Keďže to závisí len od počtu jednotkových brankárov úplne napravo v každom riadku, a záleží nám len na posledných troch (lebo ďalších už nezvládame prekryť), tak možností je 4^4 (štyri riadky a v každom $0-3$ jednotkových brankárov).

Slabina predchádzajúceho riešenia hrubou silou bola, že sme si pamätali úplne všetky rozloženia, aj tie, ktoré sa končili rovnako. Pekným vylepšením teda bude, že namiesto toho si budeme pamätať len počet rozložení pre každé možné ukončenie. Tým sa nám zlepší priestorová zložitosť na konštantnú, časová sa však nezmení.

Na to musíme spraviť ešte jedno vylepšenie. Zaveďme si malú terminológiu: *stavom* budeme nazývať ľubovoľnú z tých 256 možností, ktorými môže končiť rozloženie. Čo ak by sme si pre každý stav dopredu vypočítali, na aké ďalšie stavy sa môže zmeniť doplnením stĺpca s otáznikmi? Predpočítať to je pomerne jednoduché. Všetky prechody medzi rozloženiami si môžeme uložiť do veľkej tabuľky 256×256, kde j-te políčko v i-tom riadku bude určovať, koľkými spôsobmi sa dá dostať zo stavu číslo j do stavu číslo i (prečo nie naopak, z i do j uvidíme čoskoro). Nazvime si túto tabuľku M. (Situáciu si môžete predstaviť aj tak, že vytvoríme graf, kde stavy sú vrcholy a prechody medzi stavmi sú orientované hrany. i-ty riadok tabuľky hovorí, z ktorých vrcholov (stavov) sa dá dostať do stavu i a číslo na pozícii (i, j) hovorí koľko hrán vedie z j do i.)

V ďalšom jednorozmernom poli veľkosti 256 si budeme pamätať počty rozložení pre každý z 256 stavov. Nazvime si toto pole u. Na začiatku máme len jeden stav $(0, 0, 0, 0)$. Vždy, keď už máme spočítané počty stavov pre šírku bránky n, tak pre šírku $n+1$ spočítame ľahko. Označme si nové pole počtov stavov ako v. Toto pole vypočítame jednoducho ako $v_i = \sum_{j=0}^{255} u_j \cdot M_{i,j}$. Inými slovami, počet nových rozložení v stave i vypočítame

tak, že spočítame všetky možnosti, ako sa doňho vieme dostať zo starých rozložení. Keďže tie máme uložené, ako počty v jednotlivých stavoch a pre každý stav vieme podľa tabuľky M, kam sa z neho vieme dostať, tak nám stačí vždy vynásobiť počet rozložení v každom stave j s počtom možností, ako sa z tohoto stavu dostať do stavu i a celé to sčítať.

Celkový výsledok bude súčet počtov rozložení vo všetkých stavoch.

Týmto spôsobom dostaneme pekné lineárne riešenie. Výpočet riešenia pre každú ďalšiu šírku nám zaberie len konštantný čas, keďže počet stavov je konštantný a nijako nezávisí od vstupu.

Znížme počet stavov

Riešenie je síce lineárne od veľkosti n, no konštanta, ktorou n násobíme je $\left(4^4\right)^2 = 256^2 = 65536$, teda toto riešenie je prakticky použiteľné pre n rádovo 100.

Ak si ale maticu M vypíšeme na výstup, zistíme, že je pomerne riedka. Inými slovami, veľa stavov je úplne nedosiahnuteľných. Ako sa týchto stavov zbaviť? Jedno z riešení by bolo spraviť prehľadávanie do šírky na grafe, ktorý matica reprezentuje a takým spôsobom nájsť všetky stavy dosiahnuteľné zo stavu $(0, 0, 0, 0)$.

Existuje však aj jednoduchšie riešenie. Každý riadok v matici M určuje spôsob, ako vypočítať nový stav z predchádzajúcich stavov. Pokiaľ je celý riadok v matici nulový, tak stav, ktorý reprezentuje tento riadok nikdy nenadobudne žiadne rozloženie. Takýto stav je nedosiahnuteľný. Odstránime teda všetky nulové riadky a im prislúchajúce stĺpce. Týmto spôsobom nám môžu vzniknúť ďalšie nulové riadky a teda odstránime aj tie a tiež im prislúchajúce stĺpce. Tento proces opakujeme, až kým sa v matici nenachádzajú žiadne nulové riadky. Takto sa nám obrovská matica 256×256 zredukuje na celkom príjemnú maticu 30×30, čiže naše riešenie vyrieši úlohu aj pre n rádovo 10000. Konštanty sú niekedy podstatné :)

Smer čierna matematika (ale vôbec nie až tak)

Máme lineárne riešenie, ale to zjavne nestačí, keďže šírka bránky môže dosahovať pomerne závratné hodnoty. Keď sa však pozrieme na to, čo vlastne robíme v našom lineárnom riešení, tak nie je to nič iné, ako násobenie vektora maticou! (V podstate, výpočet počtu nových rozložení zo starých pomocou tabuľky je tak častá matematická operácia, že ju ľudia dobre preskúmali a nazvali *súčinom matíc*. Použite Google a Wikipédiu, ak ste ešte o násobení dvoch matíc a o násobení vektora maticou nepočuli.)

Pozrime sa opäť na to, ako z poľa (vektora) u vypočítame pole (vektor) v. Náš vzorec bol:

$$v_i = \sum_{j=0}^{255} u_j \cdot M_{i,j}$$

Ale toto je predsa násobenie vektora maticou! Ľahko to prepíšeme na:

$$v = M \cdot u$$

Takto vypočítame počty stavov v bránke o jedna väčšej. Čo ak chceme vypočítať počty stavov v bránke o 2 väčšej? Jednoducho:

$$v = M \cdot M \cdot u$$

Tu sa nám začína rysovať celkom jednoduchý pattern. Ak chceme vypočítať počty stavov v bránke o n väčšej, tak to spravíme takto:

$$v = M \cdot M \cdot \cdots \cdot M \cdot v = M^n \cdot u$$

Hurá, tým sme sa naozaj posunuli ďalej, pretože matice vieme umocňovať v logaritmickom čase[5]! Takýmto spôsobom sme dostali riešenie s časovou zložitosťou $O(\log n)$ a pamäťovou $O(1)$.

Listing programu (C++)

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

using namespace std;

long long MOD;
```

[5]Napríklad ak $n = 14$, potom $M^n = M^{14} = M^8 + M^4 + M^2$ Stačí nám teda vypočítať mocniny M^{2^k} a vhodné mocniny sčítať. Ak si zapíšeme n v dvojkovej sústave $14_{10} = 1110_2$, vidíme, ktoré mocniny potrebujeme. Mocniny spočítame jednoducho, opakovaným násobením $M^{k+l} = M^k * M^l$

```
// datova struktura reprezentujuca maticu a peknymi operaciami
// podporuje len stvorcove matice, ale tie nam v tejto ulohe stacia
struct matrix {
    int size;
    vector<vector<long long>> fields;

    matrix(int size) {
        this->size = size;
        fields = vector<vector<long long>>(size, vector<long long>(size, 0));
    }

    // vygeneruje nam diagonalnu (jednotkovu) maticu
    static matrix diagonal(int size) {
        matrix d(size);
        for (int k = 0; k < size; ++k) {
            d.fields[k][k] = 1;
        }
        return d;
    }

    // vynasobi dve matice v case O(n^3), kde n je velkost matice
    matrix operator * (const matrix &other) const {
        matrix result(size);

        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                for (int k = 0; k < size; ++k) {
                    result.fields[i][j] += fields[i][k] * other.fields[k][j];
                    result.fields[i][j] %= MOD;
                }
            }
        }

        return result;
    }

    // vynasobi maticu s vektorom v case O(n^2), kde n je velkost matice
    vector<long long> operator * (const vector<long long> &v) const {
        vector<long long> result(v.size(), 0);

        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                result[i] += fields[i][j] * v[j];
                result[i] %= MOD;
            }
        }

        return result;
    }

    // logaritmicke umocnovanie matice
    // algoritmus je uplne rovnaky, ako ked umocnujeme normalne cisla v logaritmickom case
    matrix exp(long long n) const {
        matrix result = matrix::diagonal(size);
        matrix mul = *this;

        while (n > 0) {
            if (n & 1) result = mul * result;
            n >>= 1;
            mul = mul * mul;
        }

        return result;
    }
};

// struktura reprezentujuca stav praveho konca branky
// pocty jednotkovych brankarov uplne napravo
struct state {
    int rows[4];    // tu su ulozene samotne pocty jednotkovych brankarov v kazdom riadku

    // vyrobi stav z indexu stavu
    // stavy potrebujeme vediet indexovat, aby sme vedeli, v ktorom riadku matice sa nachadza
    static state from_index(int index) {
        state s;
        for (int i = 3; i >= 0; --i) {
            s.rows[i] = index % 4;
            index /= 4;
        }
        return s;
    }

    // opacna operacia, ako predchadzajuca funkcia
    // vrati index daneho stavu
    int index() {
        return rows[0] * 4 * 4 * 4 + rows[1] * 4 * 4 + rows[2] * 4 + rows[3];
    }

    // rekurzivne vypocitame, do akych vsetkych stavov sa priamo vieme dostat z daneho stavu
    // ak sa do jedneho stavu vieme dostat dvoma roznymi sposobmi, tak tento stav sa
    // vo vysledku bude nachadzat dva krat
    // v parametri done_rows si pamatame, kolko hornych riadkov z praveho stlpca sme uz doplnili
    vector<state> next_states(int done_rows = 0) {
        if (done_rows == 4) return { *this };   // uz sme doplnili cely pravy okraj

        vector<state> result;

        // vyskusame umiestnit vsetkych moznych brankarov na najvyssiu nezaplnenu poziciu
        for (int square_size = 1; square_size <= 4 && square_size <= 4 - done_rows; ++square_size) {
```

```
            bool fits = true;

            // overime, ci sa da dany brankar doplnit
            for (int row = done_rows; row < done_rows + square_size; ++row) {
                if (rows[row] < square_size - 1) {
                    fits = false;
                    break;
                }
            }

            // ak ano, tak ho doplnime a rekurzivne doplnime zvysok
            if (fits) {
                state partial_next_state = *this;
                if (square_size == 1) {
                    partial_next_state.rows[done_rows] = min(3, rows[done_rows] + 1);
                } else {
                    for (int row = done_rows; row < done_rows + square_size; ++row) {
                        partial_next_state.rows[row] = 0;
                    }
                }

                for (state next_state: partial_next_state.next_states(done_rows + square_size)) {
                    result.push_back(next_state);
                }
            }
        }

        return result;
    }
};

// tato funkcia nam spocita kompletnu maticu 256x256, tak, ako je popisana
// vo vzorovom rieseni
// j-te policko v i-tom riadku urcuje pocet sposobov, ako sa da zo stavu j
// dostat priamo do stavu i
matrix full_matrix() {
    matrix m(256);

    for (int state_index_a = 0; state_index_a < 256; ++state_index_a) {
        state state_a = state::from_index(state_index_a);
        for (state state_b: state_a.next_states()) {
            int state_index_b = state_b.index();

            m.fields[state_index_b][state_index_a] += 1;
        }
    }

    return m;
}

// tato funkcia odstrani z matice vsetky nulove riadky a im
// prisluchajuce stlpce
matrix reduce_matrix(matrix m) {
    vector<int> nonzero_states;

    // najprv si najdeme indexy vsetkych nenulovych riadkov
    for (int i = 0; i < m.size; ++i) {
        bool zero_row = true;
        for (int j = 0; j < m.size; ++j) {
            if (m.fields[i][j] != 0) {
                zero_row = false;
                break;
            }
        }

        if (!zero_row) nonzero_states.push_back(i);
    }

    matrix reduced(nonzero_states.size());

    // nasledne vybudujeme maticu, ktora obsahuje len tie nenulove riadky
    for (int i = 0; i < nonzero_states.size(); ++i) {
        for (int j = 0; j < nonzero_states.size(); ++j) {
            reduced.fields[i][j] = m.fields[nonzero_states[i]][nonzero_states[j]];
        }
    }

    return reduced;
}

int main() {
    matrix m = full_matrix();

    // kompletnu maticu redukujeme az kym sa neprestane zmensovat
    while (true) {
        int orig_size = m.size;

        m = reduce_matrix(m);
        if (m.size == orig_size) break;
    }

    // pociatocny vektor poctov stavov
    // na zaciatku je len jeden stav (0, 0, 0, 0)
    vector<long long> initial_counts(m.size, 0);
    initial_counts[0] = 1;

    long long n;
    cin >> n >> MOD;
```

```
    // vypocitame konecne pocty stavov v branke 4 x n peknym umocnenim matice
    vector<long long> final_counts = m.exp(n) * initial_counts;

    // a nakoniec scitame pocty konecnych stavov do samotneho vysledku
    long long result = 0;
    for (long long count: final_counts) {
        result += count;
        result %= MOD;
    }

    cout << result << endl;
}
```

Buj

8. Opitý za volantom

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

V prvej časti vzoráku si ukážeme, ako sa postupným zlepšovaním vieme dopracovať k optimálnemu riešeniu pomocou checkpointov. Ku koncu si ukážeme pomalšie, ale pomerne rýchle riešenie pomocou intervalového stromu a pre fajnšmekrov aj optimálne riešenie, ktoré používa karteziánsky strom.

Výstraha: Odporúčame vám čítať vzorák od začiatku. Jednotlivé sekcie na seba niekedy netriviálne nadväzujú. Pri odhadoch časových zložitostí sa budú vyskytovať hlavne písmenká n = počet obmedzení, q = počet otázok a o = veľkosť výstupu. Ak nebude uvedené inak, pamäťová zložitosť je $O(n)$.

Priamočiare $O(n + qn + o)$

Na začiatku načítame všetky dopravné obmedzenia a uložíme si ich v skoro ľubovoľnej dátovej štruktúre – pole, vektor, spájaný zoznam,... Čokoľvek, čo nám umožňuje prejsť všetkými prvkami.

Keď sa neskôr vodiči pýtajú na dopravné obmedzenia, ktoré sa ich týkajú, jednoducho pre každé dopravné obmedzenie zistíme, či sa vodič nachádza v jeho aktívnej zóne. Ak áno, obmedzenie si zapamätáme. Na konci všetky zapamätané dopravné obmedzenia vypíšeme v požadovanom formáte.

Pre každú otázku prechádzame zoznam všetkých obmedzení, čiže najpomalšia časť algoritmu si vyžaduje čas $O(qn)$.

Neskúšajme obmedzenia začínajúce za pozíciou vodiča $O(n + qn + o)$

Všimnime si, že nemá zmysel skúšať dopravné obmedzenia, ktorých začiatok je na ceste neskôr ako pozícia vodiča. Môžeme tak vyskúšať nasledujúce vylepšenie:

Na začiatku si všetky obmedzenia utriedime podľa pozície ich začiatku. Na otázky potom vieme odpovedať efektívnejšie – ak je panikáriaci vodič na pozícii p, postupne overujeme obmedzenia v poradí, ako sú utriedené. Keď narazíme na obmedzenie začínajúce na $l > p$, vieme, že ďalej nemá zmysel skúšať.

Ak všetky obmedzenia pokrývajú len krátke úseky cesty a vodič spanikáril skoro na konci cesty, tento algoritmus začne pekne poporiadku testovať obmedzenia, ktoré začínajú na začiatku cesty. Väčšina obmedzení však vďaka ich zanedbateľnej dĺžke nebude aktívna. V najhoršom prípade tak stále môžeme potrebovať pre každú otázku prechádzať väčšinu obmedzení, hoci výstup môže byť pomerne malý.

Neskúšajme obmedzenia začínajúce za a končiace pred pozíciou vodiča $O(n^2 + q + o)$

Čo ak by si ale algoritmus pamätal len obmedzenia, ktoré boli aktívne iba malý kúsok pred vodičovou pozíciou p? Povedzme, že tie, čo boli aktívne na pozícii $q \leq p$. Potom by algoritmus nemusel skúšať obmedzenia končiace pred pozíciou q. Stačí mu vyskúšať len obmedzenia aktívne na q a obmedzenia začínajúce v intervale $(q, p\rangle$.

Keď predchádzajúcu myšlienku dotiahneme do totálneho extrému, dostaneme algoritmus, ktorý po načítaní dopravných obmedzení pre každú pozíciu vypočíta, ktoré obmedzenia sú na nej aktívne.

Ako to spravíme? Tak, ako to robia ľudia za volantom:

Na cestu si umiestnime značky dvoch typov – začiatok dopravného obmedzenia a jeho koniec. Pri každej značke si tiež pamätáme poradové číslo obmedzenia, ku ktorému sa vzťahuje. Dopravné značky si utriedime podľa ich pozícií. Následne simulujeme jazdu od začiatku cesty po jej koniec, pričom si udržujeme množinu aktívnych obmedzení. Keď sa posunieme o 1 pozíciu vpred, pozrieme sa na všetky dopravné značky na novej pozícii, a podľa nich upravíme množinu aktívnych obmedzení – pridáme alebo odstránime obmedzenie z množiny. Následne si pre danú pozíciu uložíme všetky aktívne obmedzenia.

Celkové množstvo toho, čo si algoritmus zapamätá, môže ale byť až $O(n^2)$ – napríklad, keď máme n obmedzení, a i-te z nich je aktívne na intervale $\langle i, i + n)$. Množinu aktívnych obmedzení si tak stačí pamätať v jednoduchom poli, kde pridanie a odobranie záznamu bude trvať $O(n)$. Ak by sme nepotrebovali ukladať aktívne obmedzenia pre každú pozíciu, vieme množinu upravovať v čase $O(\log n)$, keď množinu implementujeme

ako binárny vyhľadávací strom (`std::set` v C++). Túto optimalizáciu budeme používať v ďalších zlepšeniach algoritmu.

Keď už pre každú pozíciu vieme, ktoré obmedzenia sú na nej aktívne, každú otázku vieme spracovať v čase $O(1 + \text{veľkosť výstupu})$. Pamäťová zložitosť je $O(n^2)$.

Niečo prijateľnejšie $O(n\sqrt{n} + q\sqrt{n}\log n + o\log n)$

Nebudeme si ukladať aktívne obmedzenia pre každú pozíciu, ale iba pre niektoré. Opäť simulujeme jazdu od začiatku cesty po jej koniec. Pozíciu, pre ktorú sa rozhodneme uložiť si aktívne obmedzenia, nazveme **checkpoint**. Ako prvý checkpoint zvolíme začiatok cesty. Keď sa potom posúvame vpred, štandardne si udržujeme množinu aktívnych obmedzení. Ak sme od posledného checkpointu videli aspoň $\sqrt{n}$ značiek, uložíme si pre túto pozíciu aktívne obmedzenia a prehlásime ju za nový checkpoint. Pre každý checkpoint si pamätáme najviac n obmedzení a ľahko vidno, že checkpointov je rádovo $\sqrt{n}$. Preto toto predspracovanie vieme spraviť v čase $O(n\sqrt{n})$.

Ako odpovedáme na otázky? Keď je panikáriaci vodič na pozícii p, nájdeme najbližší checkpoint, ktorý je pred alebo na pozícii p. To vieme spraviť napríklad tak, že si pamätáme zoznam všetkých checkpointov utriedený podľa ich pozície, a binárne v ňom vyhľadáme v čase $O(\log\sqrt{n}) = O(\log n)$.

Všetky obmedzenia aktívne v checkpointe vložíme do množiny (v čase $O(a\log n)$, kde a je počet týchto obmedzení). Následne sa posúvame vpred, kým nie sme za pozíciou p a podľa značiek upravujeme množinu. Vieme, že značiek bude najviac $\sqrt{n}$, takže toto spravíme v čase $O(\sqrt{n}\log n)$. Nakoniec ešte vypísanie časti výstupu trvá $O(v)$.

Dokopy potrebujeme na jednu otázku čas $O(\log n + a\log n + \sqrt{n}\log n + v)$. Podľa nasledujúceho pozorovania to prevedieme do krajšieho tvaru, v ktorom už nebude vystupovať a.

Pozorovania o lokálnosti

Nech na pozícii p je aktívnych a obmedzení, a na úseku $\langle p, q\rangle$ je s značiek. Potom na pozícii q je aktívnych aspoň $a - s$ a najviac $a + s$ obmedzení.

Cestou od najbližšieho checkpointu k pozícii vodiča narazíme na najviac $\sqrt{n}$ značiek, teda $a - \sqrt{n} \le v$, odkiaľ dostávame horný odhad pre a: $a \le v + \sqrt{n}$. Dosadením dostaneme odhad času na jednu otázku $O(\sqrt{n}\log n + v\log n)$. Pamäťová zložitosť je $O(n\sqrt{n})$.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <set>
#include <cmath>
#include <algorithm>
using namespace std;

void vloz_do_setu (vector<int>& src, set<int>& dest) {
    for (unsigned i=0; i<src.size(); i++) {
        dest.insert(src[i]);
    }
}

void zmaz_zo_setu (vector<int>& src, set<int>& dest) {
    for (unsigned i=0; i<src.size(); i++) {
        dest.erase(src[i]);
    }
}

void uloz_do_vektora (set<int>& src, vector<int>& dest) {
    for (set<int>::iterator it = src.begin(); it != src.end(); it++) {
        dest.push_back(*it);
    }
}

int n, len;
vector<vector<int> > zac;   // znacky zaciatku obmedzenia na pozicii
vector<vector<int> > kon;   // znacky konca obmedzenia na pozicii
vector<int> nextnonempty;   // najblizsia dalsia pozicia, na ktorej su znacky
vector<vector<int> > check;
vector<int> checkpoint_positions;

void spocitaj_nextnonempty(){
    nextnonempty.resize(len);
    nextnonempty[len-1] = len;
    for (int i=len-2; i>=0; i--) {
        if (zac[i+1].size() > 0 || kon[i+1].size() > 0)
            nextnonempty[i] = i+1;
        else
            nextnonempty[i] = nextnonempty[i+1];
    }
}

void spocitaj_checkpointy(){
```

```
    int sqn = ceil(sqrt(n));
    check.resize(len);

    set<int> aktivne;
    vloz_do_setu(zac[0], aktivne);
    zmaz_zo_setu(kon[0], aktivne);
    uloz_do_vektora(aktivne, check[0]);
    checkpoint_positions.push_back(0);

    int pocet_znaciek = 0;
    for (int i=1; i<len; i++) {
        vloz_do_setu(zac[i], aktivne);
        zmaz_zo_setu(kon[i], aktivne);
        pocet_znaciek += zac[i].size() + kon[i].size();
        if (pocet_znaciek >= sqn) {
            uloz_do_vektora(aktivne, check[i]);
            checkpoint_positions.push_back(i);
            pocet_znaciek = 0;
        }
    }
}

set<int> zisti_aktivne_obmedzenia(int vodic){
    if (vodic < 0 || vodic >= len)
        return set<int>();

    int pos = *(upper_bound(checkpoint_positions.begin(), checkpoint_positions.end(), vodic) - 1);
    set<int> aktivne;
    vloz_do_setu(check[pos], aktivne);
    for (pos = nextnonempty[pos]; pos <= vodic; pos = nextnonempty[pos]) {
        vloz_do_setu(zac[pos], aktivne);
        zmaz_zo_setu(kon[pos], aktivne);
    }
    return aktivne;
}

int main () {
    cin >> n;
    len = 2*n + 1;
    zac.resize(len); kon.resize(len);

    for (int i=0; i<n; i++) {
        int a, b;
        cin >> a >> b;
        if (b <= a) continue;
        zac[a].push_back(i);
        kon[b].push_back(i);
    }

    spocitaj_nextnonempty();
    spocitaj_checkpointy();

    int q;
    cin >> q;
    int maska = 0;
    for (int i=0; i<q; i++) {
        int vodic;
        cin >> vodic;
        vodic ^= maska;

        set<int> aktivne = zisti_aktivne_obmedzenia(vodic);

        cout << aktivne.size();
        for (set<int>::iterator it = aktivne.begin(); it != aktivne.end(); it++) {
            cout << " " << *it;
            maska ^= *it;
        }
        cout << "\n";
    }

    return 0;
}
```

Lepšie? Lepšie. $O(n\sqrt{n} + q\sqrt{n} + o)$

Zamyslime sa nad tým, prečo sme mali v predchádzajúcom riešení v časovej zložitosti tak veľa logaritmov. Pri každej otázke na pozíciu p sme museli zostrojiť množinu, a potom ju upravovať. Pritom nás ale vôbec nezaujíma obsah množiny na pozíciách iných ako p. Ide to teda spraviť aj lepšie – zostrojíme si zoznam všetkých kandidátov – obmedzení, ktoré by mohli byť aktívne na p. Tých je $O(a + \sqrt{n})$. Pre každého kandidáta vieme v konštantom čase povedať, či je aktívny na pozícii p.

Dostaneme tak oveľa prijateľnejší čas na jednu otázku $O(\log n + a + \sqrt{n} + v) = O(\sqrt{n} + v)$.

Celé sa to dá v $O(n + q + o)$

Na prvý pohľad vyzerá táto časová zložitosť nedosiahnuteľne – veď na upravovanie množiny aktívnych obmedzení potrebujeme $O(n \log n)$ času a hľadanie najbližších checkpointov nám trvá $O(q \log n)$! Ukážeme si teda najprv, ako tieto veci vieme robiť rýchlejšie.

Pri udržiavaní množiny, nás nezaujíma jej obsah na pozíciách, ktoré nie sú checkpoint. Predstavme si teda, že poznáme všetky aktívne obmedzenia pre checkpoint na pozícii p, a nasledujúci checkpoint položíme na pozíciu

q. Ako vypočítame množinu aktívnych obmedzení pre q? Tak, ako sme v predchádzajúcom riešení odpovedali na otázky – vyskúšame všetky obmedzenia aktívne na p, a všetky obmedzenia začínajúce v $(p, q\rangle$.

Časová zložitosť zostrojenia jedného checkpointu je teda $O(a + b)$, kde a je počet obmedzení aktívnych v predchádzajúcom checkpointe a b je počet obmedzení začínajúcich v $(p, q\rangle$. Keď takto zostrojíme všetky checkpointy, dostaneme časovú zložitosť $O(c + n)$, kde c je počet obmedzení zapamätaných v checkpointoch (súčet a-čok).

Nakoniec, hľadanie najbližšieho checkpointu vieme robiť rýchlejšie vďaka tomu, že všetky pozície sú od 0 po $2n$. Pre každú pozíciu zistíme, ktorý checkpoint je najbližšie. Ak označme najbližší checkpoint pre pozíciu x ako `P[x]`, ľahko vidno, že ak pozícia p je checkpoint, tak `P[p] = p`, inak `P[p] = P[p-1]`. Stačí nám teda raz vypočítať pole `P` v $O(n)$.

A stačí na to táto jednoduchá úprava

Všetky prerekvizity už máme, a dostávame sa tak k jadru celého riešenia – k rozumnému umiestňovaniu checkpointov. Prvý checkpoint umiestnime štandardne na začiatok cesty. Ako umiestňujeme nasledujúce checkpointy? Označme a počet obmedzení v predchádzajúcom checkpointe. Vydáme sa z neho smerom dopredu, pričom si počítame počet značiek, ktoré sme cestou videli. Keď tento počet presiahne $\frac{a}{2}$, našu pozíciu prehlásime za nasledujúci checkpoint.

A to je celé! Iba drobnou úpravou odmocninového riešenia dostaneme optimálne lineárne. Patrí sa ale dokázať, že je to naozaj lineárne.

Zamyslime sa nad tým, koľko si toho v checkpointoch pamätáme. V prvom checkpointe si pamätáme všetky obmedzenia začínajúce na pozícii 0, ktorých je najviac n. Pre každý ďalší checkpoint máme z pozorovania o lokálnosti nasledujúci odhad:

Nech predchádzajúci checkpoint na pozícii p má a aktívnych obmedzení, a aktuálny checkpoint je na pozícii q a má b aktívnych obmedzení. Na intervale $(p, q\rangle$ je $s \geq \frac{a}{2}$ značiek. Potom $b \leq a + s \leq 3s$.

Teda počet zapamätaných obmedzení v aktuálnom checkpointe je rádovo **najviac toľko, koľko značiek sme stretli cestou z predchádzajúceho checkpointu**. Každá značka sa ale zrejme nachádza najviac v jednom úseku `(pozícia checkpointu, pozícia nasledujúceho checkpointu>`. Značiek je $O(n)$, takže celkovo si vo všetkých checkpointoch pamätáme $O(n)$ obmedzení a tak vieme spočítať všetky obmedzenia v checkpointoch v čase $O(n)$.

Nakoniec sa zamyslíme, ako dlho nám trvá odpovedať na otázku. Nech je panikáriaci vodič na pozícii p a najbližší predošlý checkpoint je na q. Označme a počet obmedzení v q. Vieme, že na intervale $(q, p\rangle$ je $s < \frac{a}{2}$ značiek. Podľa pozorovaní o lokálnosti vieme, že počet obmedzení v p je aspoň $a - s > a - \frac{a}{2} = \frac{1}{2}a$. Ak teda označíme veľkosť výstupu pre túto otázku v, platí $v > \frac{1}{2}a$. Počet skúšaných kandidátov je $a+s \leq \frac{3}{2} < 3v = O(v)$. Takže na každú otázku vieme odpovedať v čase $O(1 + v)$.

Potrebujeme teda čas $O(n)$ na predpočítanie a čas $O(q + o)$ na odpovedanie – dokopy $O(n + q + o)$.

Listing programu (C++)

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

struct Obmedzenie {
    int l, r, por;

    Obmedzenie(int l, int r, int por) : l(l), r(r), por(por) {}
    Obmedzenie(){}
    bool aktivne (int vodic) { return (vodic >= l && vodic < r); }
};

void pridaj_aktivne_obmedzenia(int vodic, vector<Obmedzenie>& kandidati, vector<Obmedzenie>& dest) {
    for (unsigned i=0; i<kandidati.size(); i++)
        if (kandidati[i].aktivne(vodic))
            dest.push_back(kandidati[i]);
}

int n, len;
vector<int> nextnonempty;
vector<int> znacky_na_pozicii, predosly_checkpoint;
vector<vector<Obmedzenie> > obmedzenia_od_pozicie, obmedzednia_v_checkpointe;

vector<Obmedzenie> obmedzenia_na_pozicii(int pozicia, int checkpoint){
    vector<Obmedzenie> obmedzenia;
    pridaj_aktivne_obmedzenia(pozicia, obmedzednia_v_checkpointe[checkpoint], obmedzenia);
    for (int j = nextnonempty[checkpoint]; j<=pozicia; j = nextnonempty[j]) {
        pridaj_aktivne_obmedzenia(pozicia, obmedzenia_od_pozicie[j], obmedzenia);
    }
    return obmedzenia;
}

void spocitaj_nextnonempty(){
    nextnonempty.resize(len);
```

```
    nextnonempty[len-1] = len;
    for (int i=len-2; i>=0; i--) {
        if (znacky_na_pozicii[i+1] > 0)
            nextnonempty[i] = i+1;
        else
            nextnonempty[i] = nextnonempty[i+1];
    }
}

void spocitaj_checkpointy(){
    obmedzednia_v_checkpointe.resize(len);
    predosly_checkpoint.resize(len);

    obmedzednia_v_checkpointe[0] = obmedzenia_od_pozicie[0];
    predosly_checkpoint[0] = 0;

    int pocet_znaciek = 0;
    for (int i=1; i<len; i++) {
        pocet_znaciek += znacky_na_pozicii[i];
        int predosly = predosly_checkpoint[i-1];

        if (pocet_znaciek >= (int)obmedzednia_v_checkpointe[predosly].size() / 2) {
            obmedzednia_v_checkpointe[i] = obmedzenia_na_pozicii(i, predosly);
            pocet_znaciek = 0;
            predosly_checkpoint[i] = i;
        }
        else {
            predosly_checkpoint[i] = predosly;
        }
    }
}

int main () {
    cin >> n;
    len = 2*n + 1;
    znacky_na_pozicii.resize(len, 0);
    obmedzenia_od_pozicie.resize(len);

    for (int i=0; i<n; i++) {
        int a, b;
        cin >> a >> b;
        if (b <= a) {
            continue;
        }
        obmedzenia_od_pozicie[a].push_back(Obmedzenie(a, b, i));
        znacky_na_pozicii[a]++;
        znacky_na_pozicii[b]++;
    }

    spocitaj_nextnonempty();
    spocitaj_checkpointy();

    int q;
    cin >> q;
    int maska = 0;
    for (int i=0; i<q; i++) {
        int vodic;
        cin >> vodic;
        vodic ^= maska;

        vector<Obmedzenie> ans(0);
        if (vodic >= 0 && vodic < len){
            ans = obmedzenia_na_pozicii(vodic, predosly_checkpoint[vodic]);
        }

        cout << ans.size();
        for (unsigned i=0; i<ans.size(); i++) {
            cout << "␣" << ans[i].por;
            maska ^= ans[i].por;
        }
        cout << "\n";
    }

    return 0;
}
```

Čestné uznania...

...získavajú nasledujúce riešenia. Prvé je založené na intervalovom strome, beží v čase $O(n \log n + q \log n + o)$ a bolo obľúbené medzi riešiteľmi. Druhé je takisto optimálne, a založené na karteziánskom strome.

Riešenie intervalovým stromom

Na pozícii l si budeme pamätať všetky obmedzenia, ktoré začínajú na pozícii l. Nad takýmto poľom zostrojíme nasledovný intervalový strom – v každom intervale si pamätáme všetky obmedzenia, ktoré majú začiatok v tomto intervale. Navyše si ich pamätáme usporiadané podľa pozície konca obmedzenia. Takýto intervalový strom vieme ľahko zostrojiť – ak už vieme, aké obmedzenia začínajú v intervale $\langle l, s)$ a v akom poradí končia, a rovnakú informáciu máme aj pre interval $\langle s, r)$, tak tieto dva intervaly vieme spojiť v lineárnom čase do intervalu $\langle l, r)$. (Podobne, ako to robí *merge sort*.)

Keď chceme odpovedať vodičovi na pozícii p, chceme vlastne nájsť všetky obmedzenia začínajúce v $\langle 0, p +$

1), ktorých koncový bod je $> p$. Rozložíme teda štandardne interval $\langle 0, p+1)$ na $O(\log n)$ intervalov, ktoré zodpovedajú vrcholom v našom strome. Pre každý z týchto intervalov vieme, ktoré obmedzenia v ňom začínajú, a v akom poradí končia. Tak začneme tými, ktoré končia najneskôr, a postupne ich skúšame, kým nenájdeme prvé obmedzenie končiace na $r \leq p$. Ďalej vieme, že nemá zmysel skúšať.

Časová zložitosť jednej odpovede je teda $O(\log n + v)$ – potrebujeme $O(\log n)$ času na rozklad na jednotlivé intervaly, a v každom spravíme aspoň jednu operáciu. Čas na zostrojenie intervaláča je $O(n \log n)$ – dokopy máme čas $O(n \log n + q \log n + o)$. Pamäťová zložitosť je $O(n \log n)$ – na každej úrovni intervalového stromu si totiž pamätáme všetky obmedzenia, a úrovní je $O(\log n)$.

Karteziánske riešenie

Veríme, že čitateľ si sám doplní znalosti o tom, čo to je karteziánsky strom, a ako sa zostrojuje. Najlepšie predtým, než sa pustí do čítania tohto riešenia.

Každé dopravné obmedzenie môžeme interpretovať ako bod v rovine – ak začína na pozícii l a končí na r, tak mu bude zodpovedať bod so súradnicami l, r. Predstavme si pre začiatok, že žiadne dve obmedzenia nezačínajú ani nekončia na tej istej pozícii. Nech je vodič na pozícii p, a už máme zostrojený karteziánsky strom nad bodmi s x-súradnicou $\leq p$. Ako nájdeme odpoveď?

Jednoducho – začneme v koreni karteziánskeho stromu, spracujeme ho, a prípadne sa posunieme do jeho ľavého a pravého syna. Čo to znamená „spracovať" a „posunúť" ? Vieme, že body v ľavom aj v pravom podstrome majú y-súradnicu menšiu ako spracovávaný vrchol. Takže ak aktuálne spracovávaný vrchol nevyhovuje (zodpovedá nejakému obmedzeniu l, r pre $r \leq p$), tak nemá zmysel pokračovať so synom. Ak vyhovuje, tak ho pridáme do zoznamu aktívnych obmedzení, a pokračujeme so synmi. Ľahko vidno, že takto spravíme rádovo $O(1 + v)$ operácii.

Vidíme teda, že by sme chceli nejaký perzistentný karteziánsky strom. To vieme dosiahnuť nasledovne – štandardne pomocou zásobníku zostrojujeme karteziánsky strom nad poľom. Každý vrchol niekedy pridávame do stromu. Pozrime sa na tento moment – pridávaný vrchol je najpravejší vrchol stromu, takže môže mať iba ľavého otca. Toho si v danom vrchole zapamätáme.

Ako teraz odpovedáme panikáriacemu vodičovi na pozícii p ? Predpokladajme, že existuje obmedzenie začínajúce na p, a označme jemu zodpovedajúci vrchol a_0. Spracujeme jeho ľavý podstrom, a nájdeme ľavého otca a_0. Nech to je a_1, spracujeme ľavý podstrom a_1, nájdeme ľavého otca a_1, a opakujeme. A tak ďalej, až kým nenarazíme na niekoho, kto už ľavého otca nemá.

Ľahko vidno, že takto sme naozaj spracovali náš strom v takom stave, v akom bol po pridaní vrcholu p, p. Konštrukcia stromu z poľa trvá $O(n)$, a odpovedanie trvá dokopy $O(q+o)$ – dokopy $O(n+q+o)$. Ešte potrebujeme rozšíriť naše riešenie o technické detaily. Konkrétne, keď obmedzenia môžu začínať aj končiť na rovnakej pozícii, a keď nemusí pre každú pozíciu p existovať obmedzenie začínajúce na p. Prenechávame čitateľovi.

Listing programu (C++)

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

struct Node {
    vector<pair<int, int> > obm; // konce a id
    // na pozicii 0 je najpravejsi koniec, a dalej zostupne

    // pridame do zoznamu obmedzeni
    void push (pair<int, int> par) {
        obm.push_back(par);
    }
    // ziskame obmedzenia konciace po p
    void get (int p, vector<int>& ans) {
        for (unsigned i = 0; i < obm.size(); i++) {
            if (obm[i].first <= p) {
                break;
            }
            ans.push_back(obm[i].second);
        }
    }
    // vrati y suradnicu reprezentanta -- najvacsia y suradnica
    int y () {
        if (obm.empty()) {
            return -1;
        }
        return obm[0].first;
    }
};

struct CarTree {
    vector<Node> V; // pre kazdu poziciu si pamatame obmedzenia zacinajuce na nej
    vector<int> lsyn, rsyn, lotec; // lavy/pravy syn, lavy otec v strome

    CarTree (vector<pair<int, int> >& obm) {
        // obmedzenia podla ich konca
```

```
        vector<vector<pair<int, int> > > sorted;
        for (unsigned i = 0; i < obm.size(); i++) {
            int kon = obm[i].second;
            while ((int)sorted.size() <= kon) { // zvacsime sorted, aby bol koniec vnutri rozsahu
                sorted.push_back(vector<pair<int, int> >());
            }
            sorted[kon].push_back(make_pair(obm[i].first, i)); // zaciatok obmedzenia a jeho id
        }
        // pomocou sorted naplnime V
        for (int i = (int)sorted.size() - 1; i >= 0; i--) {
            for (unsigned j = 0; j < sorted[i].size(); j++) {
                // najprv dostatocne zvacsime V
                int zac = sorted[i][j].first;
                while ((int)V.size() <= zac) {
                    V.push_back(Node());
                }
                // vlozime
                int id = sorted[i][j].second;
                V[zac].push(make_pair(i, id));
            }
        }
        // pridame virtualne obmedzenia
        for (unsigned i = 0; i < V.size(); i++) {
            V[i].push(make_pair(i, -1));
        }
        // nakoniec zostrojime samotny strom -- lsyn, rsyn, lotec
        vector<pair<int, int> > S; // stack, pamatame si (max y sur, x sur)
        for (unsigned i = 0; i < V.size(); i++) {
            int last = -1; // posledny vybrany zo stacku
            while (!S.empty() && S.back().first < V[i].y()) {
                last = S.back().second;
                S.pop_back();
            }
            if (!S.empty()) {
                int j = S.back().second;
                rsyn[j] = i;
                lotec.push_back(j);
            }
            else {
                lotec.push_back(-1);
            }
            lsyn.push_back(last);
            rsyn.push_back(-1);
            S.push_back(make_pair(V[i].y(), i));
        }
    }
    // spracuje zadany vrchol
    void spracuj (int v, int p, vector<int>& ans) {
        if (v < 0 || v >= (int)V.size()) {
            return;
        }
        if (V[v].y() <= p) {
            return;
        }
        V[v].get(p, ans);
        spracuj(lsyn[v], p, ans);
        spracuj(rsyn[v], p, ans);
    }
    // odpovie na otazku
    void query (int p, vector<int>& ans) {
        int i = p;
        if (i >= (int)V.size()) {
            i = (int)V.size() - 1;
        }
        if (i < 0) {
            return;
        }
        while (i != -1) {
            V[i].get(p, ans);
            spracuj(lsyn[i], p, ans);
            i = lotec[i];
        }
    }
};

int main () {
    int n;
    cin >> n;
    vector<pair<int, int> > obm; // obmedzenia
    for (int i = 0; i < n; i++) {
        int a, b;
        cin >> a >> b;
        obm.push_back(make_pair(a, b));
    }
    CarTree ct(obm);

    int q;
    cin >> q;
    int maska = 0;
    for (; q>0; q--) {
        int t;
        cin >> t;
        t ^= maska;
        vector<int> ans;
        ct.query(t, ans);
        cout << ans.size();
        for (unsigned i=0; i<ans.size(); i++) {
            cout << " " << ans[i];
            maska ^= ans[i];
```

```
        }
        cout << "\n";
    }

    return 0;
}
```