Korešpondenčný seminár z programovania
XL. ročník, 2022/23
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. kola zimnej časti

Marcel

1. Dalo by sa?!

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

Našou úlohou bolo spočítať počet zrážok, ktoré nastanú. Vieme, že za zrážku sa ráta všetko, kde idúce auto narazí do niečoho. Vieme tiež, že všetky autá po zrážke ostanú stáť.

Pomalé riešenie

Prvou možnosťou je pre každé auto, ktoré sa niektorým smerom hýbe (doprava alebo doľava) sa pozrieť, či má do čoho naraziť. Naraziť má do čoho vtedy, keď sa tým smerom, ktorým sa hýbe, nachádza auto, ktoré stojí, alebo smeruje opačne. Takže pre každé auto sa pozrieme, či má do čoho naraziť a spočítame tieto zrážky.

Pre každé auto (ktorých je n), pozrieme prinajhoršom všetky ostatné autá (ktorých je n), takže spolu urobíme $O(n^2)$ operácií.

Časová zložitosť je $O(n^2)$, a pamäťová $O(n)$, lebo okrem vstupného poľa si nič nemusíme pamätať.

Listing programu (Python)

```
auta = input()

zrazky = 0

for i in range(len(auta)):
        if auta[i]=='>':
                for j in range(i+1, len(auta)):
                        if auta[j]!='>':
                                zrazky+=1
                                break
        elif auta[i]=='<':
                for j in range(i-1, -1, -1):
                        if auta[j]!='<':
                                zrazky+=1
                                break

print(zrazky)
```

Listing programu (C++)

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

using namespace std;

int main(){
        string auta;
        cin >> auta;

        long long zaciatok, koniec, zrazky=0;

        for(long long i=0; i<=auta.size(); i++){
        int smer = 0;
```

```
        if(auta[i]=='<')
            smer = -1;
        if(auta[i]=='>')
            smer = 1;

        long long j=i+smer;
        while(smer!=0 && j>=0 && j<auta.size()){
            if(auta[i]!=auta[j]){
                zrazky++;
                break;
            }
            j += smer;
        }
        }

        cout<<zrazky<<endl;

        return 0;
}
```

Vzorové riešenie

Je pomerne jasné, že ak auto, ktoré je na ľavej strane, smeruje doľava, tak do ničho nenarazí. Rovnako je to aj na pravej strane s autom, ktoré smeruje doprava. Môžeme si uvedomiť, že to ale neplatí len pre autá, ktoré sú úplne na kraji, ale aj pre všetky autá, ktoré sú naľavo (tým myslíme také, že naľavo od nich sú len autá, ktoré smerujú doľava), a smerujú doľava. Analogicky to platí aj pre autá, ktoré sú napravo. Napríklad, ak máme autá:

<<<>=<>>

Tak vieme povedať, že ľavé 3 autá a pravé 2 autá určite do ničoho nenarazia, iba pekne za sebou odídu preč.

To znamená, že nás nezaujíma súvislý úsek áut, ktoré sú naľavo a smerujú doľava, a súvislý úsek áut napravo, ktoré smerujú doprava.

Z nášho príkladu vyššie nás teda zaujímajú iba tieto autá:

>=<

Čo vieme povedať, o zrážkach, ktoré nastanú v tejto strednej časti?

Vieme, že každé auto, ktoré nestojí musí do niečoho naraziť. Prečo? Lebo jediná iná možnosť je, že by odišlo preč, a nikdy do ničoho nenarazilo, ale to sa nemôže stať, keďže také autá sú naľavo, a smerujú doľava, alebo napravo a smerujú doprava, a tie ignorujeme.

Vzorové riešenie teda najprv zistí, ktorý úsek áut nás presne zaujíma, teda zistí aký dlhý je úsek áut, ktoré sú naľavo a smerujú doľava. To isté urobí aj pre pravú stranu. Následne pre každé auto v strednej časti, ktoré nestojí, pripočíta 1 k výslednému počtu zrážok, a vypíše tento výsledok.

Toto riešenie prejde celé pole práve raz, a teda jeho časová zložitosť je $O(n)$. Pamätať si stále musíme len celé pole, teda pamäťová zložitosť je tiež $O(n)$.

Listing programu (Python)

```
auta = input()

for zaciatok in range(len(auta)):
    if auta[zaciatok]!='<':
        break

for koniec in range(len(auta)-1, -1, -1):
    if auta[koniec]!='>':
        break

zrazky = 0

for i in range(zaciatok, koniec+1):
```

```
    if auta[i]!='=':
            zrazky += 1

print(zrazky)
```

Listing programu (C++)

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

using namespace std;

int main(){
        string auta;
        cin >> auta;

        long long zaciatok, koniec, zrazky=0;

        for(zaciatok=0; zaciatok<auta.size(); zaciatok++){
                if(auta[zaciatok]!='<')
                        break;
        }

        for(koniec=auta.size()-1; koniec>=0; koniec--){
                if(auta[koniec]!='>')
                        break;
        }

        for(long long i=zaciatok; i<=koniec; i++){
        if(auta[i]!='=')
            zrazky++;
        }

        cout<<zrazky<<endl;

        return 0;
}
```

Vzorovejšie riešenie (v konštantnej pamäti)

Existuje aj riešenie v konštantnej pamäti. Jeho kľúčovou myšlienkou je, že ak ideme po vstupom poli áut, tak vždy keď nájdeme auto, ktoré smeruje doľava (<), tak vieme, že všetky autá naľavo od neho, ktoré idú doprava do niečoho narazia. V prípade, že nájdeme auto, ktoré smeruje doprava, tak si iba musíme poznačiť, že sme ho niekde videli. No a v prípade, že nájdeme auto, ktoré stojí na mieste, tak vieme, že všetky autá, čo sme videli, a smerovali doprava tiež majú do čoho naraziť.

Kaja

2. Aerolinkové ceny

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

Na začiatok sa pozrime na to, aké informácie vlastne budeme potrebovať. Môžeme si všimnúť, že pre ľubovoľný let A B platí, že hodnota mesta A sa za tento let započíta raz do celkového súčtu cien letov nezávisle od mesta B, rovnako pre mesto B sa jeho hodnota započíta raz, bez ohľadu na mesto A. Pri zisťovaní celkovej ceny všetkých letov nám teda v skutočnosti nezáleží na tom, medzi ktorými mestami sa uskutočňujú jednotlivé lety, ale iba na tom, koľko existuje letov z jednotlivých miest.

To znamená, že hodnota mesta bude započítaná do celkového súčtu toľkokrát, koľko letov sa z neho uskutočňuje. Preto najväčší možný súčet všetkých letov získame tak, že mestám s najväčším počtom ciest priradíme najväčšie hodnoty.

Vzorové riešenie

Na vyriešenie tejto úlohy nám pri načítavaní vstupu stačí pre každý výskyt nejakého mesta navýšiť počet letov tohto mesta. Následne tieto počty utriedime vzostupne a priradíme mestám v tomto poradí postupne hodnoty od 1 po n. Celkový súčet potom dostaneme ako súčet hodnôt priradených mestám vynásobených počtom ich letov.

Keďže si potrebujeme pamätať iba počty letov pre n miest, tak pamäťová zložitosť bude $O(n)$. Načítanie vstupu je v $O(n+m)$, utriediť pole o veľkosti n vieme v $O(n \cdot \log n)$. Sčítanie celkovej sumy všetkých letov vieme spraviť v jednom cykle v $O(n)$. Celková časová zložitosť je teda $O(m + n \cdot \log n)$.

Listing programu (Python)

```
n, m = map(int, input().split())
pocet_letov = [0 for _ in range(n)]

for _ in range(m):
    a, b = map(int, input().split())
    pocet_letov[a] += 1
    pocet_letov[b] += 1

pocet_letov.sort()
cena_letov = 0
for i in range(n):
    cena_letov += pocet_letov[i] * (i + 1)

print(cena_letov)
```

Listing programu (C++)

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

int main() {
        int n, m;
        cin >> n >> m;
        vector<int> pocet_letov(n, 0);
        for (int i = 0; i < m; i++)
        {
            int a, b;
            cin >> a >> b;
            pocet_letov[a]++;
            pocet_letov[b]++;
        }

        sort(pocet_letov.begin(), pocet_letov.end());
        long long int cena_letov = 0;
        for (int i = 0; i < n; i++) cena_letov += pocet_letov[i]*(i+1);
        cout << cena_letov << endl;
        return 0;
}
```

Rýchlejšie riešenie

V skutočnosti ale existovalo aj lepšie riešenie v prípade využitia counting sortu. Ten v prípade, že máme k hodnôt, ktoré triedime a sú zhora ohraničené nejakou hodnotou l, má časovú zložitosť $O(k+l)$. V našom prípade máme hodnoty ohraničené pomocou čísla m, keďže ak máme m letov, tak zo žiadneho mesta nemôže viesť viac ako m ciest. Preto s použitím counting sortu sa dala táto úloha vyriešiť dokonca v zložitosti $O(m+n)$.

Viktor

3. Lietajúce taniere

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

Zamyslime sa, ktorými letmi sa Bzučiakovi oplatí letieť. Z Národného Múzea do Hlavnej Stanice sa mu určite oplatí letieť prvým neblokovaným letom. Teda pre nás nemá zmysel blokovať tieto lety, pokiaľ je nejaký skorší aj tak nezablokovaný. Takže jednoducho budeme blokovať niekoľko prvých letov prvej linky.

V prípade druhej linky to je podobne: Bzučiak určite nepoletí z Hlavnej Stanice na Matričný Úrad žiadnym letom po prvom neblokovanom. Zároveň ale, na rozdiel od prvej situácie, platí, že určite nepoletí takým letom, ktorý odlieta pred tým, ako priletí Bzučiakov prvý let (nie preto, že by sa to neoplatilo, ale preto, že je to jednoducho nemožné).

Teda všetky možné blokovania letov, aké sa nám teoreticky môžu oplatiť (majú potenciál viesť ku hľadanému optimálnemu výsledku), vyzerajú nasledovne: Zablokujeme x prvých letov prvej linky, pričom $0 \leq x \leq k$. Bzučiak poletí prvým neblokovaným letom, teda do Hlavnej Stanice doletí v čase $a_x + d_a$, respektíve ak $n \leq x$, tak nepoletí vôbec, keďže sme zablokovali všetky lety na prvej linke. Ostáva nám $k - x$ letov, ktoré môžeme blokovať. Nemá zmysel blokovať lety druhej linky, ktoré odlietajú pred Bzučiakovym časom príchodu do hlavnej stanice. Zablokujeme teda prvých $k - x$ letov, ktoré z Hlavnej Stanice odlietajú v Bzučiakovom čase príchodu alebo neskôr. Opäť môže nastať možnosť, že $k - x$ je dosť na zablokovanie všetkých letov až po posledný. Ak nenastane, Bzučiak poletí letom číslo y, a do cieľa sa dostane v čase $b_y + d_b$.

Pre všetky možné hodnoty x musíme nájsť najvyššie dosiahnuteľné $b_y + d_b$, prípadne či pre nejaké x vieme dosiahnuť kompletné zablokovanie.

Pomalé riešenie

Najzjavnejšie riešenie, ktoré nám môže napadnúť je postupne vyskúšať všetky možné počty zablokovaných letov prvej linky x, a pre každý nájsť riešenie. Jednoducho pre každý čas príchodu $a_x + d_a$ nájdeme v časoch odchodu druhej linky prvý let, ktorý neodlieta pred ním, posunieme sa o $k - x$ letov neskôr na nejaký let b_y, a pamätáme si výsledok $b_y + d_b$. Na konci vypíšeme z týchto výsledkov ten najvyšší, respektíve ak niekedy dosiahneme kompletné zablokovanie, tak môžeme hľadanie ihneď prerušiť a vypísať -1.

Časová zložitosť tohto riešenia môže byť $O(n^2)$, pokiaľ prvý let druhej linky, ktorý Bzučiak stíha, hľadáme lineárne, teda postupne od začiatku. Môžeme si trošku prilepšiť použitím binárneho vyhľadávania, čím zlepšíme zložitosť na $O(n \cdot log(n))$. V oboch prípadoch pre každú možnú hodnotu x, ktorých určite nie je viac ako $2n$, hľadáme bod v usporiadanom poli, teda zložitosť je lineárna, vynásobená zložitosťou našeho hľadania. Posunúť sa po nájdení tohto letu o $k-x$ ďalej nám zaberie konštantný čas, keďže výstupom z hľadania je index nájdeného prvku, a k nemu jednoducho pripočítame $k - x$. Pamäťová zložitosť programu bude každopádne $O(n)$, keďže nám stačí pamätať si vstup a konštantné množstvo celočíselných premenných.

Vzorové riešenie

Hľadanie, ktoré sme v pomalších riešeniach vylepšovali, môžeme zlepšiť ešte viac. Uvedomme si, že keď x sa zvýši o 1, tak čas príletu do Hlavnej Stanice sa určite nezníži. Teda nemusíme zakaždým prehľadávať celé pole odletov, ale stačí nám pokračovať tam, kde sme prestali (pretože vieme, že letíme neskorším letom, ako predtým, teda lety, ktoré sme nestihli predtým, určite nestihneme ani teraz). Postačí nám teda hľadať lineárne, no pamätať si vždy bod, kde sme naposledy prestali, a nabudúce pokračovať v hľadaní odtiaľ.

Toto nové hľadanie prejde najviac všetkými n prvkami počas celého behu programu. Programu teda hľadanie celkovo zaberie najviac n času, teda časová zložitosť bude $O(n+n) = O(n)$. Pamäťová zložitosť je rovnaká, ako v pomalších riešeniach - $O(n)$.

Listing programu (C++)

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

int main()
{
        int n, da, db, k;
        std::cin >> n >> da >> db >> k;

        int atob [n] = {};
        for(int i=0; i<n; i++)
        {
            std::cin >> atob[i];
```

```
        }

        int btoc [n] = {};
        for(int i=0; i<n; i++)
        {
            std::cin >> btoc[i];
        }


        //najneskorsi let do ciela, ktorym sa nám Bzuciaka podarilo prinutit ist
        int best = 0;
        //prvy let do ciela, ktoreho odlet Bzuciak teoreticky stiha
        int b_dep_index = 0;

        //vyskusame zablokovat vsetky mozne pocty letov prvej linky
        for(int offset=0; offset<=k; offset++)
        {
                //pokial sa nam podari zablokovat vsetky, Bzuciak sa do ciela
                    ↪ nevie dostat a hladat dalej nie je potrebne
            if(offset >= n)
            {
                std::cout << -1 << std::endl;
                return 0;
            }

                //cas, v ktorom Bzuciak dorazi do prestupnej stanice
            int b_arrival = atob[offset] + da;

                //najdeme prvy let dalej, ktory Bzuciak teoreticky stiha
            while(btoc[b_dep_index] < b_arrival)
            {
                b_dep_index ++;
            }
                //Bzuciak neodide tymto letom, ale az o tolko letov neskor,
                    ↪ kolko ich este mozeme zablokovat
            int b_departure = b_dep_index + k - offset;

                //pokial zablokujeme vsetky zvysne lety, do ciela sa nedostane
            if(b_departure >= n)
            {
                std::cout << -1 << std::endl;
                return 0;
            }

                //pokial sme dosiahli zatial najlepsi vysledok, zapamatame si ho
            if((btoc[b_departure] + db) > best)
            {
                best = (btoc[b_departure] + db);
            }

        }
        //vypiseme najneskorsi prichod, aky sme dosiahli
        std::cout << best << std::endl;
        return 0;
}
```

Listing programu (Python)

```
n,da,db,k = [int(i) for i in input().split()]
a = input().split()
b = input().split()
for i in range(n):
    a[i] = int(a[i])
    b[i] = int(b[i])

#index letu prvej linky, ktorym Bzuciak poleti (rovny poctu zablokovanych letov prvej linky)
prvy = 0
#index letu druhej linky, ktory Bzuciak teoreticky stiha (neratame s blokovanim letov druhej linky)
druhy = 0
#odpoved - index letu druhej linky, ktorym Bzuciak pri vhodnom zablokovani poleti najneskor (rovne -1, pokial sa da Bzuciaka zablokovat uplne)
o = -1

#vyskusame vsetky mozne pocty zablokovanych letov prvej linky
while prvy <= k:
    #pokial na niektorej linke zablokujeme dost letov, aby Bzuciak musel ist indexom letu, ktory uz neexistuje, tak sme ho od ciela odrezali a dalej hladat nemusime
    if prvy >= n:
        o = -1
        break
    if druhy >= n:
        o = -1
        break

    #pokial Bzuciak nestiha let ulozeny v premennej druhy, tak ho posunieme, kym ho nebude stihat
    if b[druhy] < a[prvy]+da:
        druhy += 1
        continue

    #pocet letov, ktore este mozeme zablokovat na druhej linke
    ostava = k-prvy
    #index letu druhej linky, ktorym Bzuciak naozaj poleti
    x = druhy+ostava
    #pokial uz nestiha ziaden let, je od ciela odrezany
    if x >= n:
        o = -1
        break
    #pokial sme dosiahli zatial najlepsi vysledok, zapamatame si ho
    if x > o:
        o = x
    #pokial sme sa dostali az sem, ideme vyskusat dalsi pocet zablokovani na prvej linke
    prvy += 1

#vypiseme vysledok
if o == -1:
    print(-1)
else:
    print(b[o]+db)
```

4. Opačné čítanie

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

Pomalé riešenie

Najjednoduchšie riešenie tejto úlohy je vyrobiť si všetky možnosti premazaných substringov, ktoré zodpovedajú zadaniu. Akonáhle máme tieto substringy, tak cez každý prejdeme `for` cyklom, ktorý sa pozerá na znaky z obidvoch koncov substringu. Ak sa znaky spredu aj zozadu zhodujú, tak substring preskočíme. Ak sa nejaký/é znak/y nezhodoval/i, tak si započítame +1 do možností. Takto prejdeme cez všetky možné substringy a zistíme koľko z nich sa nečíta rovnako spredu aj zozadu.

Rýchle riešenie

Lepší prístup k tejto úlohe je zistiť si celkový možný počet substringov a od neho odčítať počet tých substringov, ktoré sa čítajú rovnako spredu aj zozadu. Celkový počet substringov je 2^k, pretože máme dokopy `k` znakov v stringu a každý z nich bude alebo nebude v substringu. Teraz ešte treba zistiť, čo od toho odčítať.

Existujú dva druhy stringov, ktoré nám vedia prísť na vstupe – párnej a nepárnej dĺžky. Najskôr sa budeme venovať stringom párnej dĺžky. Tie pozostávajú z dvojíc rovnakých znakov na opačných stranách stringu. Keď vymažeme nejaké znaky zo stringu, tiež máme dve možnosti novovzniknutého substringu – bude párnej alebo nepárnej dĺžky. Pozrime sa na všetky substringy párnej dĺžky. Predstavme si, že v strede stringu je zrkadlo. Keďže string sa dá čítať rovnako spredu aj zozadu, tak ľavá a pravá polovica vyzerajú rovnako, len zrkadlovo – `DaL0|0LaD`. Ak vymažeme znaky tak, že nám stále ostane párny počet znakov v substringu, tak vieme jednoducho zistiť, koľko je takých, ktoré sa čítajú rovnako spredu aj zozadu. Ak vymažeme znak na nultom indexe, tak musíme vymazať aj na poslednom – `aL00La`; ak na prvom, tak aj predposlednom – `DL00LD`; … . Takto sa dostaneme ku vzorcu $2^{k/2}$. Pretože ak vymažeme ktorýkoľvek znak z prvej polovice stringu, tak vymažeme aj jeho dvojicu v druhej polovici. Keďže mažeme dvojice, tak máme "polovičnú" dĺžku stringu – $k/2$; a každá dvojica bude alebo nebude v substringu. Berieme do úvahy aj to, že nevymažeme ani jeden znak alebo vymažeme všetky.

Ešte potrebujeme odčítať počet substringov nepárnej dĺžky, ktoré sa čítajú rovnako spredu aj zozadu. Je to jednoduchšie ako sa na prvý pohľad zdá. Majme všetky substringy párnej dĺžky z minulého odseku. Z dvojice znakov, ktorá je v strede vymažeme jeden znak a máme substring nepárnej dĺžky, ktorý sa číta rovnako spredu aj zozadu – napr. `aL0La`. Na čo však nesmieme zabudnúť je to, že môžeme vymazať pravý alebo ľavý znak, čo sa počíta ako dva rôzne substringy. Takže dokopy máme $2 \cdot 2^{k/2}$ možnoných substringov nepárnej dĺžky, ktoré sa čítajú rovnako spredu aj zozadu. Čo však nie je úplne pravda. Ako bolo aj predtým povedané, v $k/2$ sa ráta za možnosť aj prázdny string. Z prázdneho stringu však nevieme nič odčítať, takže vzorec bude vyzerať takto: $2 \cdot (2^{k/2} - 1)$. Celkový vzorec na vypočítanie všetkých možných párnych stringov, ktoré sa nedajú prečítať rovnako spredu aj zozadu je $2^k - 2^{k/2} - 2 \cdot (2^{k/2} - 1)$.

Pri stringu nepárnej dĺžky sa správame podobne ako pri párnej dĺžky. Tie tiež pozostávajú z dvojíc rovnakých znakov na opačných stranách stringu, ale majú jeden znak v strede navyše – `bysAsyb`. Predstavme si, že ten znak v strede navyše tam nie je – `byssyb`. V takom prípade sme opäť naspäť pri stringu párnej dĺžky a vieme použiť vzorec $2^k - 2^{k/2} - 2 \cdot (2^{k/2} - 1)$. Avšak, od celkového počtu ešte potrebujeme odčítať možnosti, kedy sme znak v strede nevymazali. To je v podstate to isté ako keď vymazávame znaky tak, aby sme mali substringy párnej dĺžky. Budeme vymazávať dvojice rovnakých znakov a vždy necháme stredný znak – `ysAsy`, `bsAsb`, `bAb`, … . Takže od celého vzorca ešte odčítame $2^{k/2}$, a teda výsledný vzorec pre string nepárnej dĺžky bude vyzerať takto: $2^k - 2^{k/2} - 2 \cdot (2^{k/2} - 1) - 2^{k/2}$.

Samozrejme, nakoniec nesmieme zabudnúť na modulo $10^9 + 7$ a implementovať to cez rýchle modulovanie.

Časová zložitosť je $O(n \cdot \log k)$, pretože počet možností vypočítame v čase $\log k$ a vypočítame ich n-krát, lebo máme n otázok. Pamäťová zložitosť je $O(k)$.

Listing programu (Python)

```
MOD = 1000000007

def moznosti(k):
    if k % 2 == 0:
        return (pow(2, k, MOD) - pow(2, (k//2), MOD) - 2*(pow(2, (k//2), MOD)
            ↪ -1)) % MOD
    else:
        return (pow(2, k, MOD) - pow(2, (k//2), MOD) - 2*(pow(2, (k//2), MOD)
            ↪ -1) - pow(2, (k//2), MOD)) % MOD
```

```
n = int(input())
for _ in range(n):
    slovo = input()
    k = len(slovo)
    print(moznosti(k))
```

fezjo

5. Bezkonkurenčná manufaktúra

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

Zadanie úlohy nám popisuje graf v tvare stromu a nariaďuje nájsť určitý kritický vrchol. Tento vrchol sa nachádza na ceste medzi najväčším počtom dvojíc. Otázkou je, koľko dvojíc to je (nazvime túto vlastnosť dôležitosť) a koľko dvojíc to bude, ak môžeme pridať jednu dodatočnú hranu (dôležitosť s dodatočnou hranou). Otázka na koľkých cestách sa nachádza vrchol je ekvivalentná otázke, koľko ciest sa preruší keď ho odstránime.

Triviálny bruteforce

Na začiatok navrhnime kľudne aj pomalé riešenie, ktoré nám však dá aspoň správny výsledok. Pre odpoveď na prvú otázku, môžeme skúsiť odstrániť každý možný vrchol a potom pre každú možnú dvojicu spustiť prehľadávací algoritmus (DFS, BFS, …), ktorý nám povie, či sú tieto dva vrcholy spojené. Keďže pred odstránením kritického vrcholu nutne spojené museli byť, počet dvojíc čo spojené nebudú bude dôležitosť tohoto vrcholu. Vrchol s najvyššou dôležitosťou je potom kritický vrchol. Toto vieme spraviť v čase $O(N^4)$, keďže pre každý vrchol a každú dvojicu spustíme prehľadávanie v $O(N)$.

Jednoduché vylepšenie bude spustiť prehľadávanie nie pre každú dvojicu, ale iba pre každý vrchol, keďže počas jedného vyhľadávania nájdeme všetky dosiahnuteľné vrcholy. Dostávame sa teda k časovej zložitosti $O(N^3)$ na zodpovedanie prvej otázky.

Pri zodpovedaní druhej otázky už poznáme kritický vrchol. Môžeme teda napríklad pre každú dvojicu vrcholov skúsiť pridať hranu a znova zistiť dôležitosť vrchola v čase $O(N^2)$, čím by sme získali celkovú časovú zložitosť $O(N^4)$.

Netriviálny bruteforce

Zamyslime sa, čo sa stane, ak odstránime tento kritický vrchol. Strom sa rozpadne na niekoľko súvislých komponentov, ich počet bude rovný počtu susedov kritického vrcholu. Samozrejme, z definície súvislého komponentu, všetky dvojice vrcholov v rámci jednotlivých komponentoch medzi sebou naďalej budú mať cestu (Mohli sme si všimnúť, že keď sme pri zodpovedaní prvej otázky robili prehľadávanie, veľa vrcholov malo rovnaký počet dosiahnuteľných vrcholov, keďže boli v tom istom komponente. Ďalšie vylepšenie je teda púštať prehľadávanie iba pre ešte nenavštívené vrcholy). Cesty, ktoré sa odstránením kritického vrcholu narušia budú teda nutne iba cesty medzi vrcholmi z dvoch rôznych komponentov.

To znamená, že nezáleží na konkrétnych koncových vrcholoch pridávanej hrany, ale iba na tom, ktoré komponenty daná hrana spája. Mohli by sme teda naše doterajšie riešenie zoptimalizovať tak, aby neskúšalo všetky dvojice vrcholov, ale iba všetky dvojice susedov kritického vrcholu. Keďže však môže mať vrchol až $N-1$ susedov, počet dvojíc susedov je stále $O(N^2)$, a teda si z pohľadu efektivity nepomôžeme. Predsa nám však toto uvedomenie pomôže.

Cesty, ktoré sa prerušia odstránením kritického vrcholu sú iba cesty medzi vrcholmi z dvoch rôznych komponentov a cesta bude prerušená všetkým takýmto dvojiciam vrcholov. Dôležitosť vrcholu sa teda dá vyjadriť iba na základe veľkosti komponentov jeho susedov. Nech množina susedov kritického vrcholu je S a K_x je veľkosť komponentu, v ktorom je sused x, potom dôležitosť daného vrcholu je [1]

$$\frac{1}{2}\sum_{i,j\in S, i\neq j} K_i * K_j$$

teda súčin počtu vrcholov v jednom komponente a počtu v vrcholov v druhom komponente pre každú dvojicu komponentov (predelený dvomi, keďže komponent A a B boli zarátané pre $i=A, j=B$ a $i=B, j=A$).

Zmysluplným pridaním hrany vieme spojiť práve dva komponenty. Keďže chceme minimalizovať dôležitosť vrcholu, chceme maximalizovať počet dvojíc vrcholov, ktoré ňou prepojíme. Zjavne chceme teda vybrať dva najväčšie komponenty.

[1] $\sum_{podmienka}\cdots$ označuje súčet nejakých čísel na základe danej podmienky. Teda ak napríklad $S=\{1,2,3\}$, potom $\sum_{i,j\in S, i\neq j} K_i * K_j$ vieme rozpísať ako $K_1*K_2+K_1*K_3+K_2*K_1+K_2*K_3+K_3*K_1+K_3*K_2$

Znovu vieme zoptimalizovať náš aktuálny algoritmus. Na zodpovedanie druhej otázky môžeme ľubovoľným prehľadávaním nájsť veľkosti jednotlivých komponentov pre už nájdený kritický vrchol, vybrať dva najväčšie z nich a znížiť odpoveď prvej otázky o ich súčin. Toto vieme spraviť v lineárnom čase. Celková časová zložitosť bude kvôli zodpovedaniu prvej otázky $O(N^3)$.

Zjavne je teraz hrdlom fľaše nájdenie kritického vrcholu. Môžeme takéto lineárne hľadanie veľkosti komponentov použiť pre každý vrchol, a teda v čase $O(N^2)$. Následne by sme pre každý vrchol v čase $O(s^2)$ kde s je počet jeho susedov vypočítali jeho dôležitosť. Dá sa ukázať, že aj keď pre N vrcholov robíme po $O(s^2)$ operácií, celková časová zložitosť je stále $O(N^2)$ (dôležité pri analýze časovej zložitosti je, že graf je strom a má teda iba málo hrán).

Optimálne riešenie

Aktuálne máme dva problémy. Hľadanie veľkosti komponentov nám trvá $O(N^2)$ a počítanie dôležitosti pre vrchol nám trvá $O(s^2)$. Ani jeden problém však našťastie nie až také ťažké vyriešiť.

Chceme veľkosti komponentov vypočítať na jeden prechod. Spustíme z koreňa DFS prehľadávanie, ktoré nám pre každý vrchol zistí, koľko vrcholov je v jeho podstrome. Robíme to rekurzívne. Pre vrchol, ktorý už nemá deti je odpoveď 1, pre vrchol čo má deti je odpoveď súčet výsledkov rekurzívnych volaní na jeho deti plus 1. Pre každý vrchol sa vieme pozrieť na veľkosti podstromov jeho detí, čo sú predsa veľkosti komponentov jeho susedov. Chýba nám iba veľkosť jedného komponentu a to komponentu suseda, ktorý je náš rodič. Jeho veľkosť však vieme samozrejme ľahko vypočítať ako počet vrcholov v strome mínus veľkosť podstromu aktuálneho vrcholu. Časová zložitosť tejto časti je teda $O(N)$.

Druhý problém si zase vyžaduje trochu matematiky. Pozrime sa na vzorec, podľa ktorého sme to počítali doteraz a skúsme ho upraviť.

$$\frac{1}{2}\sum_{i,j\in S, i\neq j} K_i * K_j$$
$$\frac{1}{2}\sum_{i\in S}\sum_{j\in S, i\neq j} K_i * K_j$$
$$\frac{1}{2}\sum_{i\in S}(K_i * \sum_{j\in S, i\neq j} K_j)$$
$$\frac{1}{2}\sum_{i\in S} K_i * (-K_i + \sum_{j\in S} K_j)$$
$$\frac{1}{2}\sum_{i\in S} K_i * (-K_i + (N-1))$$
$$\frac{1}{2}\sum_{i\in S}(K_i * (N-1) - K_i^2)$$
$$\frac{1}{2}(\sum_{i\in S} K_i * (N-1) - \sum_{i\in S} K_i^2)$$
$$\frac{1}{2}((N-1) * \sum_{i\in S} K_i - \sum_{i\in S} K_i^2)$$
$$\frac{1}{2}((N-1)^2 - \sum_{i\in S} K_i^2)$$

Pri úpravách sme použili dve netriviálne úpravy vo forme rovností:

1. $\sum_{j\in S, i\neq j} K_j = -K_i + \sum_{j\in S} K_j$, teda že súčet veľkostí všetkých susedných komponentov až na komponent suseda i je súčet všetkých mínus veľkosť toho jedného a
2. $\sum_{x\in S} K_x = N - 1$, teda že súčet veľkostí všetkých susedných komponentov je počet všetkých vrcholov okrem jedného vrcholu (toho kritického).

No a vidno, že takýto vzorec už zvládame vypočítať v lineárnom čase od počtu susedov pre každý vrchol, čo je dokopy iba dvojnásobok počtu hrán. Celková časová aj pamäťová zložitosť je teda $O(N)$.

Listing programu (Python)

```python
import sys
sys.setrecursionlimit(10 ** 6)

N = int(input())

graf = [[] for _ in range(N)]
for _ in range(N - 1):
    a, b = map(int, input().split())
    graf[a].append(b)
    graf[b].append(a)

odpovede = [0, 0]

def nechaj_najvacsie(najvacsie_dva, nove):
    return sorted(najvacsie_dva + [nove])[1:]

def dfs(aktualny_vrchol, otec):
    global odpovede

    velkost_podstromu, sucet_stvorcov_velkosti, najvacsie_dva = 1, 0, [0] * 2
    for sused in graf[aktualny_vrchol]:
        if sused == otec:
            continue
        velkost_synovho_podstromu = dfs(sused, aktualny_vrchol)
        velkost_podstromu += velkost_synovho_podstromu
        sucet_stvorcov_velkosti += velkost_synovho_podstromu ** 2
        najvacsie_dva = nechaj_najvacsie(najvacsie_dva,
            ↪ velkost_synovho_podstromu)
    if otec is not None:
        velkost_synovho_podstromu = N - velkost_podstromu
        sucet_stvorcov_velkosti += velkost_synovho_podstromu ** 2
        najvacsie_dva = nechaj_najvacsie(najvacsie_dva,
            ↪ velkost_synovho_podstromu)

    dolezitost = ((N - 1) ** 2 - sucet_stvorcov_velkosti) // 2
    if dolezitost > odpovede[0]:
        odpovede = dolezitost, dolezitost - najvacsie_dva[0] * najvacsie_dva[1]

    return velkost_podstromu

dfs(0, None)
print(*odpovede)
```

Listing programu (C++)

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

ll N;
vector<vector<ll>> graf;
pair<ll, ll> odpovede = {0, 0};

ll sqr(ll x) { return x * x; }

void nechaj_najvacsie(pair<ll, ll>& najvacsie_dva, ll nove) {
    if (nove >= najvacsie_dva.first)
```

```
        najvacsie_dva = {nove, najvacsie_dva.first};
    else if (nove > najvacsie_dva.second)
        najvacsie_dva = {najvacsie_dva.first, nove};
}

ll dfs(ll aktualny_vrchol, ll otec) {
    ll velkost_podstromu = 1;
    ll sucet_stvorcov_velkosti = 0;
    pair<ll, ll> najvacsie_dva = {0, 0};
    for (ll sused : graf[aktualny_vrchol]) {
        if (sused == otec)
            continue;
        ll velkost_synovho_podstromu = dfs(sused, aktualny_vrchol);
        velkost_podstromu += velkost_synovho_podstromu;
        sucet_stvorcov_velkosti += sqr(velkost_synovho_podstromu);
        nechaj_najvacsie(najvacsie_dva, velkost_synovho_podstromu);
    }
    if (otec != -1) {
        ll velkost_synovho_podstromu = N - velkost_podstromu;
        sucet_stvorcov_velkosti += sqr(velkost_synovho_podstromu);
        nechaj_najvacsie(najvacsie_dva, velkost_synovho_podstromu);
    }

    ll dolezitost = (sqr(N - 1) - sucet_stvorcov_velkosti) / 2;
    if (dolezitost > odpovede.first)
        odpovede = {dolezitost,
                    dolezitost - najvacsie_dva.first * najvacsie_dva.second};

    return velkost_podstromu;
}

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

    cin >> N;

    graf.resize(N);
    for (ll i = 0; i < N - 1; ++i) {
        ll a, b;
        cin >> a >> b;
        graf[a].push_back(b);
        graf[b].push_back(a);
    }

    dfs(0, -1);
    cout << odpovede.first << " " << odpovede.second << "\n";
}
```

Bubu

6. Yyha to je šuter

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

Kľúčom k riešeniu tejto úlohy je uvedomiť si dve veci. Za prvé: najkratšia trasa medzi dvomi bodmi je úsečka. Za druhé: smer, ktorým ideme sa nám neoplatí meniť nikde inde, ako na bodoch, ktoré definujú prekážky (konce v prípade úsečiek a rohy v prípade n-uholníkov).

Vďaka týmto dvom poznatkom vieme použiť abstrakciu, ktorou si celý problém vieme zjednodušiť na graf,

v ktorom potom už len stačí nájsť najkratšiu trasu. Vrcholmi tohoto grafu budú štart, koniec, a body, ktoré definujú prekážky. Hrany v tomto grafe budú existovať iba v prípade, že medzi danými dvoma vrcholmi existuje priama úsečka, ktorá neprechádza žiadnou prekážkou. Keďže hľadáme najkratšiu cestu v 2D priestore, musíme každej hrane priradiť aj jej dĺžku, ktorá bude rovnaká, ako dĺžka príslušnej úsečky.

(Táto abstrakcia funguje iba v prípade, že sa žiadne úsečky nedotýkajú, čo je špecifikované v zadaní)

Keďže má náš graf hrany s definovanou dĺžkou, jednoduché BFS nám nebude stačiť na nájdenie najkratšej trasy. Na to musíme použiť Dijkstrov algoritmus[2], ktorý túto trasu ľahko nájde.

Ku kompletnému teoretickému riešeniu nám teda chýba jediná vec, a to síce, ako si tento graf vyrobiť.

Generácia grafu

Ak chceme vygenerovať náš graf, musíme vedieť, ktoré hrany v ňom sú, a ktoré nie. Vec sa má tak, že teoreticky v ňom môže byť každá možná hrana medzi dvoma vrcholmi a bez toho, aby sme ich všetky skontrolovali nemôžeme žiadnu vylúčiť (až na výnimky, ktoré spomeniem nižšie). Náš prístup teda skontroluje každú dvojicu vrcholov, a zistí, či úsečka medzi nimi neprechádza cez žiadnu prekážku. Ak nie, môžeme ju pridať do grafu. Jej dĺžku ľahko vyrátame pomocou Pytagorovej vety.

V prípade, že všetky prekážky sú úsečkami (nepárne sady) je to celkom jednoduché: pre každú potenciálnu hranu skontrolujeme všetky prekážky, a ak ju niektorá pretína, vieme, že táto hrana nie je v grafe.

V prípade n-uholníkov je to trochu komplikovanejšie. Nemôžeme ich priamo zjednodušiť na úsečky, keďže ich vnútro je tiež nepriechodné. Iba, že by sme použili nejaký trik. Obvod n-uholníkov teda zjednodušíme na úsečky. Možností, ako vyriešiť vnútro je veľa.

Našim prístupom je, že vôbec nebudeme generovať hrany pre body, ktoré sa nachádzajú vnútri n-uholníkov (týmto automaticky vylúčime akékoľvek hrany, ktoré vchádzajú do a vychádzajú z n-uholníkov). Diagonály medzi rôznymi bodmi toho istého n-uholníka vieme veľmi ľahko skontrolovať a zakázať, keďže poznáme poradie bodov v n-uholníku.

Jediný problém, ktorý nám zostáva, sú hrany, ktoré prechádzajú krížom cez n-uholník, tak, že na nich ležia dva rohy n-uholníka. To vieme vyriešiť tak, že pri kontrole pretínania úsečiek vylúčime nielen prípad, kedy sa úsečky úplne pretnú, ale aj prípad, kedy sa dotknú. Zvyšok riešenia to neovplyvní, keďže v prípadoch, kde takéto úsečky potrebujeme nahradíme jednu úsečku AC dvoma (alebo viacerými) úsečkami AB a BC, v súčte rovnakej dĺžky. No v prípade hrany, ktorá ide krížom cez n-uholník, by táto hrana bola rozdelená na tri hrany - dve, ktoré končia v rohoch n-uholníka a jedna diagonála. Túto diagonálu však už vylúčime, a teda je problém vyriešený.

Riešenie a zložitosti

Naše riešenie načíta vstup, vygeneruje podľa popisu vyššie graf, a následne na ňom spustí Dijkstrov algoritmus.

Časová zložitosť nášho riešenia je $O(n^3)$, kde n je počet bodov definujúcich prekážky ($n = \sum m_i$ ak použijeme značenie zo zadania), keďže pri generácii grafu kontrolujeme každú potenciálnu hranu (ktorých je $O(n^2)$) na pretnutie s každou úsečkou definujúcou prekážku (ktorých je $O(n)$). Dijkstrov algoritmus vieme implementovať s časovou zložitosťou $O(n^2)$, čo celkovú časovú zložitosť neovplyvňuje.

Pamäťová zložitosť nášho riešenia je $O(n^2)$, keďže najväčšia vec, ktorú si musíme pamätať je graf, kde si musíme pre každú hranu uložiť jej dĺžku, prípadne existenciu (keďže dĺžku môžeme vždy znovu vyrátať v konštantnom čase).

Vzorové riešenie

Asymptotickú časovú zložitosť tohoto riešenia síce nezlepšíme, no vieme zlepšiť jeho pamäťovú zložitosť, a praktickú časovú zložitosť. To vieme spraviť tak, že namiesto generácie grafu pred spustením Dijkstrovho algoritmu budeme kontrolovať existenciu hrany v grafe počas jeho behu. Na kontrolu existencie hrán použijeme rovnaký systém, ako sme použili na generáciu grafu, akurát ich budeme kontrolovať len keď ich treba.

Takéto riešenie má stále v najhoršom prípade časovú zložitosť $O(n^3)$, keďže v každom kroku Dijkstrovho algoritmu musíme skontrolovať pre n potenciálnych ďalších bodov n potenciálnych prekážok na pretnutie (ak nerátame body, ktoré sme už použili tak v jednotlivých krokoch musíme v najhoršom prípade spraviť najviac $n^2, n(n-1), \cdots, 2n, n$ kontrol, čo sa sčíta na $O(n^3)$). Prakticky však bude náš program na vstupoch bežať násobne rýchlejšie.

Pamäťová zložitosť tohoto riešenia je $O(n)$, keďže si pamätáme len vstup a vzdialenosť každého bodu od začiatku, obe o veľkosti $O(n)$.

[2] https://www.ksp.sk/kucharka/dijkstra/

Podobným prístupom vieme dospieť aj k ďalšej optimalizácii. K rozhodnutiu negenerovať celý graf nás síce viedlo zlepšenie pamäťovej zložitosti, avšak tento prístup nám dovolí obmedziť vykonávanie časovo veľmi náročnej operácie (kontrola existencie hrany medzi dvoma vrcholmi) iba na užitočné prípady, teda iba na tie dvojice vrcholov, ktoré reálne zvažujeme pri prehľadávaní. Pokračovaním tejto myšlienky vieme dospieť k optimalizácii, kedy existenciu hrany nekontrolujeme pri pridávaní, ale pri vyberaní kandidáta nasledujúceho vrcholu pri prehľadávaní. Vykonávame Dijkstrovo prehľadávanie a tvárime sa, že každá hrana existuje až do posledného možného momentu. Takýmto spôsobom minimalizujeme počet overovaní existencie hrany, čo urýchli beh algoritmu. Ďalším praktickým vylepšením vie byť implementovanie prehľadávania A* namiesto Dijsktrovho algoritmu, čo je na grafoch na 2D ploche ľahký spôsob ako zrýchliť beh programu. Obe tieto optimalizácie stále nezlepšujú asymptotickú časovú zložitosť a nie sú potrebné na získanie plného počtu bodov za program. Prakticky však každé z nich zrýchľujú beh programu na naších vstupoch dvojnásobne.

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;
using ll = int;
using ld = long double;
#define len(x) ((ll)(x).size())
const ld INF = 1e100;

using point = complex<ld>;
using poly_t = vector<point>;
using edge_t = pair<ld, ll>;
using graph_t = vector<vector<edge_t>>;
using polypoints_t = vector<tuple<point, ll, ll>>;

inline ll previ(ll i, ll n) { return i - 1 + n * (i <= 0); }
inline ll nexti(ll i, ll n) { return i + 1 - n * (i >= n - 1); }
inline ld cross(point a, point b) { return (conj(a) * b).imag(); }
inline ld orient(point a, point b, point c) { return cross(b - a, c - a); }

inline bool inside_polygon(poly_t &poly, point p) {
    if (len(poly) < 3) return false;
    ld dir = orient(poly[0], poly[1], p);
    for (ll b = 0; b < len(poly); ++b)
        if (dir * orient(poly[previ(b, len(poly))], poly[b], p) <= 0)
            return false;
    return true;
}

inline bool lines_intersect(point a1, point a2, point b1, point b2) {
    return abs(cross(a2 - a1, b2 - b1)) > 1e-10 &&
        orient(b1, b2, a1) * orient(b1, b2, a2) <= 0 &&
        orient(a1, a2, b1) * orient(a1, a2, b2) <= 0 &&
        (a1 != b1 && a1 != b2 && a2 != b1 && a2 != b2);
}

inline bool line_intersects_polygon(point a1, point a2, poly_t &poly) {
    if (len(poly) < 2) return false;
    for (ll b = 0; b < len(poly); ++b)
        if (lines_intersect(a1, a2, poly[previ(b, len(poly))], poly[b]))
            return true;
    return false;
}

inline graph_t get_graph(vector<poly_t> &polys) {
    ll pn = accumulate(polys.begin(), polys.end(), 0,
```

```
        [](size_t a, const auto &c) { return a + len(c); }
    );

    vector<bool> inside;
    inside.reserve(pn);
    for (auto &poly: polys)
        for (auto pt: poly)
            inside.push_back(any_of(polys.begin(), polys.end(), [&](auto &pl) {
                return inside_polygon(pl, pt);
            }));

    graph_t res(pn);
    auto consider_edge = [&](ll v1, point p1, ll v2, point p2) {
        if (inside[v1] || inside[v2] ||
            any_of(polys.begin(), polys.end(), [&](auto pl) {
                return line_intersects_polygon(p1, p2, pl);
        }))
            return;
        ld dist = abs(p1 - p2);
        res[v1].push_back({dist, v2});
        res[v2].push_back({dist, v1});
    };

    ll v1 = 0;
    for (ll polyi1 = 0; polyi1 < len(polys); ++polyi1) {
        auto &poly1 = polys[polyi1];
        for (ll pi1 = 0; pi1 < len(poly1); ++pi1, ++v1) {
            if (inside[v1]) continue;
            point p1 = poly1[pi1];

            if (len(poly1) >= 2) {
                ll pi2 = previ(pi1, len(poly1));
                ll v2 = v1 - pi1 + pi2;
                consider_edge(v1, p1, v2, poly1[pi2]);
            }

            ll v2 = 0;
            for (ll polyi2 = 0; polyi2 < polyi1; ++polyi2) {
                auto &poly2 = polys[polyi2];
                for (ll pi2 = 0; pi2 < len(poly2); ++pi2, ++v2)
                    consider_edge(v1, p1, v2, poly2[pi2]);
            }
        }
    }

    return res;
}

inline point read_point() {
    ld x, y;
    cin >> x >> y;
    return point(x, y);
}

inline vector<poly_t> parse_input() {
    point pS = read_point();
    point pF = read_point();
```

```
    ll polyN;
    cin >> polyN;
    vector<poly_t> polys(2 + polyN);
    polys[0] = {pS};
    polys[1] = {pF};
    for (ll pli = 2; pli < len(polys); ++pli) {
        ll pn;
        cin >> pn;
        polys[pli].resize(pn);
        for (auto &p: polys[pli])
            p = read_point();
    }

    return polys;
}

inline polypoints_t get_polypoints(vector<poly_t> &polys) {
    polypoints_t points = {};
    for (ll pli = 0; pli < len(polys); ++pli)
        for (ll pti = 0; pti < len(polys[pli]); ++pti)
            points.push_back({polys[pli][pti], pli, pti});
    return points;
}

ld solve(vector<poly_t> &polys, polypoints_t &points) {
    ll pn = len(points);
    vector<bool> inside(pn);
    for (ll i = 0; i < pn; ++i)
        inside[i] = any_of(polys.begin(), polys.end(), [&](auto &pl) {
                return inside_polygon(pl, get<0>(points[i]));
        });

    auto consider_edge = [&](ll v1, point p1, ll v2, point p2) {
        return !(inside[v1] || inside[v2] ||
            any_of(polys.begin(), polys.end(), [&](auto pl) {
                return line_intersects_polygon(p1, p2, pl);
        }));
    };

    vector<bool> seen(pn, false);
    vector<ld> res(pn, INF);
    res[0] = 0;
    set<edge_t> PQ;
    PQ.insert({0, 0});
    while (!PQ.empty()) {
        auto [dist, pi1] = *PQ.begin();
        PQ.erase(PQ.begin());
        if (pi1 == 1) break;
        seen[pi1] = true;
        point p1; ll pli1, pl_pti1;
        tie(p1, pli1, pl_pti1) = points[pi1];
        ll len_poly1 = len(polys[pli1]);
        ll prev_pt2 = previ(pl_pti1, len_poly1);
        ll next_pt2 = nexti(pl_pti1, len_poly1);

        for (ll pi2 = 0; pi2 < pn; ++pi2) {
            if (seen[pi2])
                continue;
```

```
            auto [p2, pli2, pl_pti2] = points[pi2];
            ld dist2 = dist + abs(p1 - p2);
            if (res[pi2] <= dist2 ||
                (pli1 == pli2 && pl_pti2 != prev_pt2 && pl_pti2 != next_pt2) ||
                !consider_edge(pi1, p1, pi2, p2)
            )
                continue;
            PQ.erase({res[pi2], pi2});
            res[pi2] = dist2;
            PQ.insert({res[pi2], pi2});
        }
    }
    return res[1];
}

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

    auto polys = parse_input();
    auto points = get_polypoints(polys);
    ld res = solve(polys, points);
    cout << setprecision(18) << fixed << res << '\n';
}
```

Listing programu (Python)

```
import heapq

point = complex
poly_t = list[point]
graph_t = list[list[tuple[float, int]]]
polypoints_t = list[tuple[point, int, int]]

INF = 1e100


def cross(a: point, b: point) -> float:
    return (a.conjugate() * b).imag


def orient(a: point, b: point, c: point) -> float:
    return cross(b - a, c - a)


def inside_polygon(poly: poly_t, p: point) -> bool:
    if len(poly) < 3: return False
    dir = orient(poly[0], poly[1], p)
    return not any(dir * orient(poly[b - 1], poly[b], p) <= 0 for b in range(len
        ↪ (poly)))


# tu ignorujeme pripad kedy su usecky paralelne a zdielaju cast dlzky
def lines_intersect(a1: point, a2: point, b1: point, b2: point) -> bool:
    return (
        abs(cross(a2 - a1, b2 - b1)) > 1e-10 and
        orient(b1, b2, a1) * orient(b1, b2, a2) <= 0 and
```

```
        orient(a1, a2, b1) * orient(a1, a2, b2) <= 0 and
        (a1 != b1 and a1 != b2 and a2 != b1 and a2 != b2)
    )


def line_intersects_polygon(a1: point, a2: point, poly: poly_t) -> bool:
    return len(poly) >= 2 and \
        any(lines_intersect(a1, a2, poly[b - 1], poly[b]) for b in range(len(
            ↪ poly)))


def parse_input() -> list[poly_t]:
    l = tuple(map(float, input().split()))
    point_start, point_finish = point(*l[:2]), point(*l[2:])

    polyN = int(input())
    polys: list[poly_t] = [[point_start], [point_finish]] + [[] for _ in range(
        ↪ polyN)]
    for poly_idx in range(2, 2 + polyN):
        _ = int(input())
        l = tuple(map(float, input().split()))
        polys[poly_idx] = [point(l[i], l[i+1]) for i in range(0, len(l), 2)]
    return polys


def get_polypoints(polys: list[poly_t]) -> polypoints_t:
    points: polypoints_t = []
    for poly_idx in range(len(polys)):
        for pt_i, p in enumerate(polys[poly_idx]):
            points.append((p, poly_idx, pt_i))
    return points


def solve(polys: list[poly_t], points: polypoints_t) -> float:
    n_points = len(points)
    inside = [any(inside_polygon(pl, pt) for pl in polys) for poly in polys for
        ↪ pt in poly]

    def consider_edge(v1: int, p1: point, v2: int, p2: point) -> float:
        return not (inside[v1] or inside[v2] or  # trasa nemoze prechadzat bodom
            ↪  v strede n-uholnika
            any(line_intersects_polygon(p1, p2, pl) for pl in polys)  # ani
                ↪ krizom cez prekazky
        )

    res = [INF] * n_points
    point_queue: list[tuple[float, int, int]] = [(0.0, 0, 0)]
    while point_queue:
        dist, pt_i_1, pt_i_0 = heapq.heappop(point_queue)
        if res[pt_i_1] < INF or not consider_edge(pt_i_0, points[pt_i_0][0],
            ↪ pt_i_1, points[pt_i_1][0]):
            continue
        res[pt_i_1] = dist
        if pt_i_1 == 1:
            break
        p1, pli1, pl_pti1 = points[pt_i_1]
        len_poly1 = len(polys[pli1])
        prev_pt2 = (pl_pti1 if pl_pti1 else len_poly1) - 1
```

```
        next_pt2 = (pl_pti1 if pl_pti1 != len_poly1 - 1 else -1) + 1

        for pt_i_2 in range(n_points):
            if res[pt_i_2] < INF:
                continue
            p2, pli2, pl_pti2 = points[pt_i_2]
            if pli1 == pli2 and pl_pti2 != prev_pt2 and pl_pti2 != next_pt2:
                continue  # trasa medzi bodmi rovnakeho n-uholnika moze ist len
                    ↪ po obvode
            heapq.heappush(point_queue, (dist + abs(p1 - p2), pt_i_2, pt_i_1))
    return res[1]


if __name__ == '__main__':
    polys = parse_input()
    points = get_polypoints(polys)

    res = solve(polys, points)
    print(f"{res:.18f}")
```

7. Srny, Viki, Viki a iné divé veci

12 b za popis, 8 b za program

Túto úlohu budeme riešiť pomocou dynamického programovania. Budeme sa pozerať len na posledný krok. Zamyslime sa, čo by mohol byť jeden stav našej dynamiky. Potrebujeme ním nejak popísať stav oboch šúp kalerábu. Stav jednej šupy jednoznačne popisuje jedno číslo - počet zostávajúch kusov na danej šupe. Náš stav teda bude dvojica čísel, počty ostávajúcich kusov. Rôznych stavov teda bude $O(nm)$. Ako vieme zistiť riešenie pre jednu dvojicu?

Prvé riešenie

V jednom kroku Viki a Viki zvolia koľko kusov sa odstráni z ktorej šupy, a na základe toho pribudnú nejaké srny. Naše riešenie by teda mohlo prejsť cez všetky možnosti, a vybrať z nich najlepšiu - tú pri ktorej sa v koši objaví najmenej sŕn. Keďže máme potencionálne až n možností ako zvoliť x, a m možností ako zvoliť y, výpočet jedného stavu by nám zabral $O(nm)$ času. Celý program by teda bežal v časovej zložitosti $O(n^2m^2)$.

Optimalizácia

Všimnime si, že výraz $(S-x)$ a $(Z-y)$ má rovnaký efekt ako zníženie všetkých čísel o 1 a minimalizovanie výrazu SX. Tento krok nie je nutný na vyriešenie úlohy, ale výrazne zjednoduší zápis nasledujúcich vzorcov.

V optimálnom riešení bude v každom kroku buď x alebo y 1. Predpokladajme, že by v optimálnom riešení bol krok, pri ktorom zvolené x aj y bolo väčšie ako 1. Počet sŕn v koši by potom bol SZ. Nech s a z sú počty listov na posledných kusoch kalerábov tak, že $S = S' + s$ a $Z = Z' + z$. Počet sŕn v koši je teda $(S'+s)(Z'+z)$. Ak by sme ale najprv spravili krok, kde zvolíme hodnoty 1 a 1, a potom krok pre $x-1$ a $y-1$ v koši sa objaví $s*z+S'Z'$ sŕn. V optimálnom riešení sa však v koši objavilo $SZ = (s+S')*(z+Z') = sz+S'Z'+sZ'+zS'$ sŕn. Keďže ale $sZ'+zS' \geq 0$ tak aj v novom riešení sa v koši objavilo nanajvýš toľko sŕn, čo znamená, že môžeme (alebo dokonca musíme) spraviť krok dĺžky 1. Opakovaným aplikovaním tohoto postupu ukážeme, že určite existuje optimálne riešenie, v ktorom je vždy x alebo y rovné 1.

Tento poznatok môžeme využiť vo svojom riešení. Pri skúšaní už budeme skúšať iba tie dvojice x a y, kde $x = 1$ alebo $y = 1$. Nemáme už nm možností ale iba $n+m$ možností. Naše vylepšené riešenie teda má časovú zložitosť $O(nm(n+m))$, čiže rádovo kubickú.

Vzorové riešenie

Zatiaľ sme vždy pri výpočte počítali, že Viki a Viki spravia celý krok. Pozrime sa teraz na niečo ako medzikroky. Ak teraz rátame s možnosťou, že zvolíme $x = 1$, tak sa vieme vždy rozhodnúť, či zväčšíme y o 1, alebo tento krok "ukončíme". Vždy keď zväčšíme y o 1 sa v koši objaví sz sŕn, kde s je počet listov na poslednej šupy prvého kusu kalerábu, a z je počet listov na poslednej šupy druhého kusu kalerábu. Uvedomme si, že môžeme počet sŕn rátať takto postupne. Vyplýva to z distributívnosti násobenia a sčítavania ($sZ = s(z_1 + ... + z_{k-1} + z_k) = (sz_1 + ... + sz_{k-1}) + sz_k$).

Náš stav teda bude trojica počty zostávajúcich kusov jedného a druhého kalerábu a indikátor, z ktorého kusu berieme viac ako 1. Počet stavov sa teda zdvojnásobí, ale stále bude stavov $O(nm)$.

Hodnotu v jednom takomto stave vieme vypočítať v konštantnom čase. Možnosti, ktoré treba vyskúšať, sú zobrať ešte jeden alebo ukončiť. Ak ukončíme, tak máme ešte 2 možnosti, či v ďalšom kroku budeme brať viac z prvého alebo druhého kusu.

Časová zložitosť

Časová zložitosť nášho algoritmu teda bude $O(nm)$. Hodnotu pre každý z $O(nm)$ stavov vieme zistiť v konštantnom čase.

Pamäťová zložitosť

Ak si budeme pamätať hodnotu pre každý stav, priestorová zložitosť nášho algoritmu bude $O(nm)$. Ak by sme ale algoritmus implementovali iteratívne, mohli by sme si všimnúť, že dokážame zlepšiť pamäťovú zložitosť na O(n+m).

Program

Rekurzívne implementované riešenie:

Listing programu (C++)

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

#define ll long long
#define INF 123456789123456789
using namespace std;

vector<vector<vector<ll>>> memo;
vector<vector<ll>> v(2);

ll f(vector<ll> &i, ll k)
{
    if(i[0] == -1 && i[1] == -1) return 0;
    if(i[0] < 0 || i[1] < 0) return INF;
    if(memo[i[0]][i[1]][k] == -1)
    {
        auto ni = i;
        ni[k]--;
        memo[i[0]][i[1]][k] = f(ni, k) + v[0][i[0]]*v[1][i[1]];
        ni[!k]--;
        memo[i[0]][i[1]][k] = min(memo[i[0]][i[1]][k], v[0][i[0]]*v[1][i[1]] +
            ↪ min(f(ni, k), f(ni, !k)));
    }
    return memo[i[0]][i[1]][k];
}

int main()
{
    ios_base::sync_with_stdio(false);cin.tie(0);cout.tie(0);
    ll n, m;
    cin >> n >> m;

    v[0].assign(n, 0);
    v[1].assign(m, 0);

    for(ll i = 0; i < n; i++) cin >> v[0][i];
    for(ll i = 0; i < m; i++) cin >> v[1][i];

    for(ll i = 0; i < n; i++) v[0][i]--;
```

```
    for(ll i = 0; i < m; i++) v[1][i]--;

    memo.assign(n, vector<vector<ll>>(m, vector<ll>(2, -1)));
    n--;m--;
    vector<ll> x = {n, m};
    cout << min(f(x, 0), f(x, 1)) << '\n';

    return 0;
}
```

Riešenie s lineárnou pamäťovou zložitosťou:

Listing programu (C++)

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


#define ll long long
#define INF 123456789123456789

using namespace std;

int main()
{
    ios_base::sync_with_stdio(false);cin.tie(0);cout.tie(0);
    ll n, m;
    cin >> n >> m;

    vector<vector<ll>> v(2);
    v[0].assign(n, 0);
    v[1].assign(m, 0);

    for(int i = 0; i < n; i++) cin >> v[0][i];
    for(int i = 0; i < m; i++) cin >> v[1][i];

    for(int i = 0; i < n; i++) v[0][i]--;
    for(int i = 0; i < m; i++) v[1][i]--;

    vector<vector<vector<ll>>> memo(2, vector<vector<ll>>(m, vector<ll>(2, INF))
        ↪ );

    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < m; j++)
        {
            for(int k = 0; k < 2; k++)
            {
                memo[i&1][j][k] = INF;
                if(i == 0 && j == 0)
                {
                    memo[i&1][j][k] = v[0][0]*v[1][0];
                    continue;
                }

                if(i-1 >= 0 && j-1 >= 0)
                {
                    memo[i&1][j][k] = min(memo[i&1][j][k], v[0][i]*v[1][j] + min
                        ↪ (memo[(i-1)&1][j-1][k], memo[(i-1)&1][j-1][!k]));
                }
```

```
                if(k == 1)
                {
                    if(j-1 >= 0) memo[i&1][j][k] = min(memo[i&1][j][k], memo[i
                        ↪ &1][j-1][k] + v[0][i]*v[1][j]);
                }
                else{
                    if(i-1 >= 0) memo[i&1][j][k] = min(memo[i&1][j][k], memo[(i
                        ↪ -1)&1][j][k] + v[0][i]*v[1][j]);
                }
            }
        }
    }
    n--;m--;
    cout << min(memo[n&1][m][0], memo[n&1][m][1]) << '\n';

    return 0;
}
```

8. Abnormálne Veľký Hotel

12 b za popis, 8 b za program

Brute-force riešenie

Táto úloha má pomerne jednoduché bruteforce riešenie, ktoré stačí na polovicu bodov. Stačí simulovať proces prichádzania a odchádzania ľudí a udržiavať si pri tom dve informácie:

- Pole intervalov voľných izieb. Na začiatku behu obsahuje interval $[0, n)$.
- Pole intervalov priradených izieb, v ktorom si pre každú skupinu pamätáme priradené izby, nech ich vieme odubytovať, keď na to príde čas. Na začiatku je prázdne. Počas behu `pole_i` obsahuje interval, ktorý bol priradený i-tej skupine.

Keď príde check-in požiadavka, lineárne prejdeme pole obsahujúce úseky voľných izieb a nájdeme taký úsek, ktorý je dosť dlhý na ubytovanie celej skupiny a zároveň je najviac vľavo. Do nájdeného úseku ubytujeme hostí a vymažeme ho zo zoznamu voľných úsekov. Je však možné, že nájdený úsek bol dlhší než bolo nutné. V takom prípade pridáme naspäť do zoznamu nepoužitú časť úseku. Na záver vložíme na koniec poľa priradených intervalov izieb interval, ktorý sme priradili novej skupinke.

Keď príde check-out požiadavka, nazrieme do pola priradených izieb na korešpondujúci index. Tieto izby chceme uvoľniť, čo znamená, že ich chceme vložiť naspäť do poľa voľných úsekov izieb. Avšak, nemôžme tento úsek len tak vložiť naspäť do zoznamu voľných, lebo je možne, že existujú ďalšie izby v zozname voľných tesne pred/za naším úsekom. Napríklad, môže sa stať, že pole voľných úsekov aktuálne obsahuje úseky $[0, 3)$ a $[4, 7)$ a úsek, ktorý sa snažíme vložiť je $[3, 4)$. Za takých okolností najprv vymažeme úseky $[0, 3)$ a $[4, 7)$ zo zoznamu voľných úsekov, a potom vložíme dnu $[0, 7)$. Inými slovami, vymažeme voľné susediace úseky a potom vložíme naspäť úsek, ktorý je zlúčením úseku patriaceho skupinke z check-out požiadavky a okolitých voľných úsekov.

Pre každú požiadavku robíme viacero lineárnych prehľadávaní a mazaní na poli obsahujúcom intervaly voľných izieb. Týchto intervalov je $O(q)$, lebo po každej check-out požiadavke nám mohol pribudnúť interval do zoznamu. Po check-in požiadavke ostala dĺžka zoznamu v najhoršom prípade rovnaká. Celková časová zložitosť je teda $O(q^2)$, lebo je $O(q)$ požiadaviek a na každú odpovieme v $O(q)$ čase. Pamäťová zložitosť je $O(q)$, lebo ako sme spomínali, dĺžka pola voľných izieb je $O(q)$ a dĺžka pola priradených izieb je tiež $O(q)$, lebo aspoň polovica požiadaviek sú check-in požiadavky.

Vzorové riešenie

Vzorové riešenie sa od spomenutého brute-force riešenia líši len tým, ako si ukladáme voľné úseky. Brute-force riešenie je pomalé, lebo mu trvá lineárne dlho robiť operácie, ako napríklad hladať najľavejší dostatočne dlhý úsek izieb, mazať úseky izieb a hľadať susediace úseky izieb. Tieto operácie vieme všetky naimplementovať v logaritmickej zložitosti a to tak, že použijeme binárny vyhľadávací strom, napríklad treap[3].

[3] https://www.ksp.sk/kucharka/treap/

Majme teda treap voľných úsekov. Keď chceme mazať úseky izieb, je to jednoduché, lebo mazať z treapu sa dá v logaritmickom čase. Rovnako nájsť susediace úseky, v prípade, že kontrolujeme či je treba zjednocovať počas check-outu, je logaritmické, lebo treapy podporujú rýchle hľadanie najmenšieho väčšieho a najväčšieho menšieho prvku než je daný prvok.

Jediná operácia, ktorú treba domyslieť, je hľadanie najľavejšieho úseku dĺžky aspoň L. To dokážeme tak, náš treap upravíme, nech si každý vrchol pamätá okrem svojho intervalu aj dĺžku najdlhšieho intervalu vo svojom podstrome. Nájsť najľavejší úsek dĺžky aspoň L vieme nasledovnou rekurziou z koreňa treapu:

- Ak má môj ľavý podstrom dosť dlhý úsek, rekurzívne ho nájdem a vrátim ho – existuje dobrý úsek a dokonca existuje dobrý úsek vľavo, čo je to, čo chceme.
- Ak môj úsek je dosť dlhý, vrátim ho – lebo už viem, že vľavo nič nie je a lepšie vrátiť svoj úsek než úsek sprava.
- Rekurzívne nájdem úsek v pravom podstrome – musí tam byť, lebo viem že v mojom podstrome existuje dlhý úsek a viem že nie je ani vľavo, ani vo mne.

Posledná otázka, ktorú treba zodpovedať, je ako rátať pre každý podstrom dĺžku najdlhšieho úseku. Toto je relatívne triviálne, lebo stačí upraviť funckie treapu `split`, `merge`, `insert` a `erase` nech počas toho, ako manipulujú stromom prerátavajú hodnotu vo svojom vrchole z hodnôt vo svojom ľavom a pravom synovi a dĺžky svojho úseku.

Použitím treapu vieme všetky operácie potrebné na zodpovedanie jednej požiadavky spraviť v $O(\log q)$, a teda celková časová zložitosť je $O(q \cdot \log q)$. Pamäťová zložitosť je stále $O(q)$.

Riešenie intervaláčom

Existuje aj riešenie, ktoré stačí na 6 bodov, ktoré používa dynamický lazy[4] intervalový strom[5]. Takéto riešenie sa dá často použiť a je dobré ho poznať. V každom vrchole stromu si pamätáme najdlhší voľný interval v podstrome vrcholu, plus počet po sebe idúcich voľných izieb od ľavého konca (prefix) a od pravého konca (sufix). Keď máme tieto informácie pre ľavý a pravý podstrom, vieme ich zrátať aj pre ich rodiča:

- Najdlhší voľný interval môže byť buď najdlhší voľný interval z ľavého podstromu alebo z pravého, alebo to môže byť kombinácia sufixu ľavého vrcholu a prefixu pravého vrcholu.
- Prefix intervalu je rovnaký ako prefix ľavého intervalu, okrem prípadu, kedy je prefix ľavého intervalu rovnako veľký ako celý ľavý interval (čo znamená, že celý ľavý interval je voľný). V takom prípade je prefix celého intervalu zlúčením ľavého intervalu s prefixom pravého intervalu.
- Sufix intervalu sa ráta podobne ako prefix.

Ak použijeme dynamickú alokáciu vrcholov na to, aby sme vytvorili iba tie časti stromu, ktoré sú potrebné, tak vieme dosiahnuť časovú zložitosť $O(q \cdot \log n)$, čo je trochu horšie, než vzorové riešenie. Trochu vačším problémom je však pamäťová zložitosť, ktorá je tiež $O(q \cdot \log n)$, čo je príliš veľa na poslednú sadu vstupov.

Program

Riešenie treapom:

Listing programu (C++)

```
#include <algorithm>
#include <chrono>
#include <iostream>
#include <optional>
#include <random>

#define F first
#define S second

using namespace std;
using namespace chrono;
```

[4] https://www.ksp.sk/kucharka/lazy_intervalovy_strom/
[5] https://www.ksp.sk/kucharka/intervalovy_strom/

```
using ll = long long;

struct Vrchol {
    Vrchol* lavy;
    Vrchol* pravy;
    pair<ll, ll> usek;
    double priorita;
    ll maximum;
};

void update_max(Vrchol* koren) {
    koren->maximum = koren->usek.second - koren->usek.first;
    if (koren->lavy)
        koren->maximum = max(koren->maximum, koren->lavy->maximum);
    if (koren->pravy)
        koren->maximum = max(koren->maximum, koren->pravy->maximum);
}

pair<Vrchol*, Vrchol*> split(Vrchol* koren, pair<ll, ll> hranica) {
    if (!koren)
        return {nullptr, nullptr};

    if (koren->usek <= hranica) {
        auto par = split(koren->pravy, hranica);
        koren->pravy = par.first;
        update_max(koren);
        return {koren, par.second};
    }
    auto par = split(koren->lavy, hranica);
    koren->lavy = par.second;
    update_max(koren);
    return {par.first, koren};
}

Vrchol* merge(pair<Vrchol*, Vrchol*> par) {
    if (!par.first && !par.second)
        return nullptr;
    if (!par.first)
        return par.second;
    if (!par.second)
        return par.first;

    if (par.first->priorita >= par.second->priorita) {
        par.first->pravy = merge({par.first->pravy, par.second});
        update_max(par.first);
        return par.first;
    }
    par.second->lavy = merge({par.first, par.second->lavy});
    update_max(par.second);
    return par.second;
}

mt19937_64 mt(high_resolution_clock::now().time_since_epoch().count());
uniform_real_distribution<double> dist(0, 1);

Vrchol* insert(Vrchol* koren, pair<ll, ll> usek, double priorita = dist(mt)) {
    if (!koren || priorita > koren->priorita) {
```

```
        auto par = split(koren, usek);
        Vrchol* novy = new Vrchol{par.first, par.second, usek, priorita, 0};
        update_max(novy);
        return novy;
    }

    if (usek <= koren->usek) {
        koren->lavy = insert(koren->lavy, usek, priorita);
        update_max(koren);
        return koren;
    }

    koren->pravy = insert(koren->pravy, usek, priorita);
    update_max(koren);
    return koren;
}

Vrchol* erase(Vrchol* koren, pair<ll, ll> usek) {
    if (!koren)
        return nullptr;

    if (usek == koren->usek)
        return merge({koren->lavy, koren->pravy});

    if (usek < koren->usek) {
        koren->lavy = erase(koren->lavy, usek);
        update_max(koren);
        return koren;
    }

    koren->pravy = erase(koren->pravy, usek);
    update_max(koren);
    return koren;
}

optional<pair<ll, ll>> find_rooms(Vrchol* koren, ll rooms) {
    if (!koren || koren->maximum < rooms)
        return nullopt;

    auto res = find_rooms(koren->lavy, rooms);
    if (res)
        return res;
    if (koren->usek.second - koren->usek.first >= rooms)
        return koren->usek;
    return find_rooms(koren->pravy, rooms);
}

optional<pair<ll, ll>> find_before(Vrchol* koren, pair<ll, ll> usek) {
    if (!koren)
        return nullopt;

    if (usek.first < koren->usek.second)
        return find_before(koren->lavy, usek);

    if (usek.first == koren->usek.second)
        return koren->usek;

    return find_before(koren->pravy, usek);
```

```
}

optional<pair<ll, ll>> find_after(Vrchol* koren, pair<ll, ll> usek) {
    if (!koren)
        return nullopt;

    if (usek.second < koren->usek.first)
        return find_after(koren->lavy, usek);

    if (usek.second == koren->usek.first)
        return koren->usek;

    return find_after(koren->pravy, usek);
}

struct Solver {
    Vrchol* a{};
    vector<pair<ll, ll>> u{};

    Solver(ll n) : a(insert(nullptr, {0, n})) {}

    ll check_in(ll l) {
        auto opt = find_rooms(a, l);
        pair<ll, ll> p = *opt;
        a = erase(a, p);

        pair<ll, ll> o{p.F, p.F + l};
        if (o.S < p.S)
            a = insert(a, {o.S, p.S});

        u.push_back(o);

        return o.F;
    }

    void check_out(ll l) {
        pair<ll, ll> o = u[l];

        auto o_prev = find_before(a, o);
        if (o_prev) {
            o.F = o_prev->F;
            a = erase(a, *o_prev);
        }

        auto o_next = find_after(a, o);
        if (o_next) {
            o.S = o_next->S;
            a = erase(a, *o_next);
        }

        a = insert(a, o);
    }
};

int main(int argc, char** argv) {
    ios::sync_with_stdio(0);
    cin.tie(0);
```

```
    ll n, q;
    cin >> n >> q;

    Solver solver(n);
    while (q--) {
        char c;
        ll l;
        cin >> c >> l;

        if (c == 'I') {
            cout << solver.check_in(l) << '\n';
        } else {
            solver.check_out(l);
        }
    }
}
```

Riešenie intervaláčom:

Listing programu (C++)

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

#define F first
#define S second

using namespace std;

using ll = long long;
using pll = pair<ll, ll>;

bool cmp_size_order(const pll& a, const pll& b) {
    if (a.S - a.F != b.S - b.F)
        return a.S - a.F < b.S - b.F;
    return a > b;
}

class Node {
public:
    Node(ll begin, ll end, ll value)
        : begin_(begin)
        , end_(end)
        , prefix_(gen_prefix(value))
        , suffix_(gen_suffix(value))
        , max_(prefix_)
        , lazy_(value) {}

    pll gen_prefix(ll value) {
        if (value)
            return {begin_, end_};
        return {begin_, begin_};
    }

    pll gen_suffix(ll value) {
        if (value)
            return {begin_, end_};
        return {end_, end_};
```

```
    }

    void propagate() {
        if (lazy_) {
            left_->prefix_ = left_->gen_prefix(*lazy_);
            left_->suffix_ = left_->gen_suffix(*lazy_);
            left_->max_ = left_->prefix_;
            left_->lazy_ = lazy_;
            right_->prefix_ = right_->gen_prefix(*lazy_);
            right_->suffix_ = right_->gen_suffix(*lazy_);
            right_->max_ = right_->prefix_;
            right_->lazy_ = lazy_;
            lazy_ = nullopt;
        }
    }

    void set(ll begin, ll end, ll value) {
        if (begin >= end_ || end <= begin_)
            return;
        if (begin <= begin_ && end >= end_) {
            prefix_ = gen_prefix(value);
            suffix_ = gen_suffix(value);
            max_ = prefix_;
            lazy_ = value;
            return;
        }
        if (!left_) {
            ll mid = (begin_ + end_) / 2;
            left_ = new Node(begin_, mid, *lazy_);
            right_ = new Node(mid, end_, *lazy_);
        }
        propagate();
        left_->set(begin, end, value);
        right_->set(begin, end, value);

        prefix_ = left_->prefix_;
        if (prefix_.S == right_->prefix_.F)
            prefix_.S = right_->prefix_.S;

        suffix_ = right_->suffix_;
        if (suffix_.F == left_->suffix_.S)
            suffix_.F = left_->suffix_.F;

        max_ = {left_->suffix_.F, right_->prefix_.S};
        if (cmp_size_order(max_, left_->max_))
            max_ = left_->max_;
        if (cmp_size_order(max_, right_->max_))
            max_ = right_->max_;
    }

    optional<pll> find(ll len) {
        if (max_.S - max_.F < len)
            return nullopt;
        if (!left_) {
            ll mid = (begin_ + end_) / 2;
            left_ = new Node(begin_, mid, *lazy_);
            right_ = new Node(mid, end_, *lazy_);
        }
```

```
        propagate();

        auto res = left_->find(len);
        if (res)
            return res;

        if (right_->prefix_.S - left_->suffix_.F >= len)
            return {{left_->suffix_.F, right_->prefix_.S}};

        return right_->find(len);
    }

private:
    Node* left_ = nullptr;
    Node* right_ = nullptr;
    ll begin_;
    ll end_;
    pll prefix_;
    pll suffix_;
    pll max_;
    optional<ll> lazy_;
};

int main(int argc, char** argv) {
    ios::sync_with_stdio(0);
    cin.tie(0);

    ll n, q;
    cin >> n >> q;

    Node root(0, n, 1);
    vector<pll> u{};
    while (q--) {
        char c;
        ll l;
        cin >> c >> l;

        if (c == 'I') {
            pll p = *root.find(l);
            pll o = {p.F, p.F + l};
            root.set(o.F, o.S, 0);
            u.push_back(o);

            cout << o.F << '\n';
        } else {
            pll o = u[l];
            root.set(o.F, o.S, 1);
        }
    }
}
```

Brute-force riešenie:

Listing programu (C++)

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

```cpp
#define ALL(x) (x).begin(), (x).end()
#define F first
#define S second

using namespace std;

using ll = long long;
using pll = pair<ll, ll>;

int main(int argc, char** argv) {
    ios::sync_with_stdio(0);
    cin.tie(0);

    ll n, q;
    cin >> n >> q;

    vector<pll> a{{0, n}};
    vector<pll> u{};
    while (q--) {
        char c;
        ll l;
        cin >> c >> l;

        if (c == 'I') {
            vector<pll> f;
            copy_if(ALL(a), back_inserter(f), [&l](pll p) {
                return p.S - p.F >= l;
            });
            pll p = *min_element(ALL(f));
            a.erase(remove(ALL(a), p), a.end());

            pll o{p.F, p.F + l};
            if (o.S < p.S)
                a.push_back({o.S, p.S});

            u.push_back(o);

            cout << o.F << '\n';
        } else {
            pll o = u[l];

            auto o_prev = find_if(ALL(a), [&o](pll p) {
                return p.S == o.F;
            });
            if (o_prev != a.end()) {
                o.F = o_prev->F;
                a.erase(o_prev);
            }

            auto o_next = find_if(ALL(a), [&o](pll p) {
                return p.F == o.S;
            });
            if (o_next != a.end()) {
                o.S = o_next->S;
                a.erase(o_next);
            }

            a.push_back(o);
```

```
            }
        }
}
```