Korešpondenčný seminár z programovania
XXXVIII. ročník, 2020/21
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola zimnej časti

kubik

1. Hady. Prečo to musia byť zrovna hady!?

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

Jedno riešenie

Máme teda naše káble, v celej ich jednosmernej kráse. Podľa zadania sa v nich nachádzajú iba znaky < alebo > (kvačky), a reťazce <> (3D dlaždice). Vedeli by sme spraviť to, že načítame celý vstup, prečístíme ho od <> reťazcov a zbytok, čo nám ostane, budú kvačky indikujúce správny smer. Stačí sa teda len na jednu z nich pozrieť a prehlásiť, ktorým smerom mieri. Toto riešenie nie je ale úplne efektívne, a to hlavne preto, že si musíme zapamätať všetky znaky zo vstupu, čo je $O(n)$ – lineárna pamätová zložitosť. Časová zložitosť je ale taktiež lineárna – $O(n)$ – pretože iba vstup načítame, a zopár krát ho prejdeme. To je dobré, pretože aj tak potrebujeme lineárny čas už len na to, aby sme vstup načítali. Ak ste sa ešte neučili používať polia alebo reťazce (stringy), napísanie tohoto riešenia je skvelé cvičenie na oboznámenie sa ;)

Lepšie riešenie

Prirodzene prichádza otázka, dá sa to lepšie? Áno, dá. Skúsme sa pozrieť na počty kvačiek, konkrétne ľavých a pravých. Majme dve premenné, v ktorých si budeme ukladať počet ľavých a pravých kvačiek, ktoré sme videli. Môžeme si všimnúť, že pokiaľ sa v reťazci nachádza <>, pripočíta sa nám ľavá aj pravá kvačka. Pokiaľ sa ale v reťazci nachádza iba ľavá alebo pravá, pripočíta sa nám iba príslušná premenná. Čo to znamená? Predstavme si, že sa v reťazci nenachádzajú reťazce <>. Ako tento reťazec vyzerá? Chvíľka napätia! Vyzerá nejako takto <<<<<<<<<, takto >>>>>>>>>>, alebo taaaaktoooo <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<.

Pokiaľ sa vrátime k našemu počítaniu kvačiek, uvedomíme si, že po spočítaní nám ostane v jednej premennej viac započítaných kvačiek, ako v druhej. A to je naše riešenie. Spočítame ľavé/pravé kvačky, porovnáme, kde je ich viac a vyhlásime smer káblu. Toto riešenie má lepšiu pamäťovú zložitosť, konštantnú, teda $O(1)$. To znamená, že máme fixne pre akokoľvek veľký vstup iba dve premenné.[1]

Listing programu (C++)

```
#include <iostream>

using namespace std;

int main() {
    int pocet_lavych_kvaciek = 0,
        pocet_pravych_kvaciek = 0;
    char kvacka;
    while (cin >> kvacka) {
        if (kvacka == '<') {
            pocet_lavych_kvaciek++;
        } else {
            pocet_pravych_kvaciek++;
        }
    }
    if (pocet_pravych_kvaciek < pocet_lavych_kvaciek) {
        cout << "lavo-datovy" << endl;
    } else  {
        cout << "pravo-datovy" << endl;
```

[1] Aj napriek tomuto má vzorové riešenie v Pythone pamäťovú zložitosť $O(n)$, pretože načítava celý vstup naraz. Načítavanie znak po znaku je totiž v Pythone trochu nepraktické.

```
    }
    return 0;
}
```

Listing programu (Python)

```
#/usr/bin/env python3

kabel = input()

pocet_lavych_kvaciek = 0
pocet_pravych_kvaciek = 0
for sipka in kabel:
    if sipka == '<':
        pocet_lavych_kvaciek += 1
    elif sipka == '>':
        pocet_pravych_kvaciek += 1

if pocet_lavych_kvaciek < pocet_pravych_kvaciek:
    print("pravo-datovy")
elif pocet_lavych_kvaciek > pocet_pravych_kvaciek:
    print("lavo-datovy")
```

2. A nech sú moje hady v suchu

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

Hlavná myšlienka

Koľko vody (aký vysoký stĺpec vody) sa na jednom mieste[2] pozemku udrží? Predstavme si najnižšie políčko[3] nad zemou na nejakom mieste na pozemku. Ak sú niekde na obidve strany od neho políčka so zemou, tak voda, ktorá napŕší na toto políčko, neodtečie a toto políčko ostane zatopené. To isté platí aj pre každé ďalšie políčko nad ním. Čo v prípade, ak máme políčko, od ktorého je len na jednu stranu políčko so zemou? V takomto prípade sa tam voda neudrží, ale odtečie druhou stranou.

To znamená, že ak si nájdeme najvyššie miesto napravo a naľavo od nejakého miesta na pozemku, tak na tomto mieste pozemku sa udrží toľko vody, aký je rozdiel medzi výškou terénu nižšieho z týchto miest a miesta, na ktorom sme. Ak toto urobíme pre všetky miesta a sčítame množstvá vody, tak získame odpoveď – koľko vody sa udrží na celom pozemku.

Všetky nasledujúce riešenia počítajú odpoveď takýmto spôsobom, líšia sa len v tom, ako šikovne, a teda ako rýchlo, to robia.

Priamočiare riešenie

Najjednoduchšie riešenie je, že prejdeme všetky miesta na pozemku. Pre každé miesto najprv prejdeme všetky miesta na jednu a potom na druhú stranu. Takto pre obe strany nájdeme miesto s maximálnou výškou. Pre jedno miesto na pozemku má takéto prehľadanie zložitosť $O(n)$, a kedže to potrebujeme urobiť pre n miest, tak nám to celkovo zaberie $O(n^2)$ času. Pamäťová zložitosť je $O(n)$, lebo okrem vstupného poľa dĺžky n a niekoľko málo premenných si nič iné nemusíme pamätať. Za takéto riešenie ste mohli získať 2 body.

Listing programu (C++)

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

using namespace std;
```

[2] Pod slovom `miesto` v tomto vzoráku myslíme jedno z n (šírka pozemku) miest na pozemku.
[3] Pod slovom `políčko` v tomto vzoráku myslíme jedno políčko z dvojrozmernej mriežky výška × šírka pozemku.

```
int main()
{
    long long sum=0, n=0, max_zlava=0, max_sprava=0;
    cin>>n;
    long long vysky[n];
    for(long long i=0; i<n; i++)
    {
        cin>>vysky[i];
    }

    for(long long i=0; i<n; i++)
    {
        max_sprava=0, max_zlava=0;
        for(long long j=0; j<=i; j++)
        {
            max_zlava=max(max_zlava, vysky[j]);
        }
        for(long long j=i; j<n; j++)
        {
            max_sprava=max(max_sprava, vysky[j]);
        }

        sum+=(min(max_sprava, max_zlava)-vysky[i]);
    }

    cout<<sum<<endl;

    return 0;
}
```

Listing programu (Python)

```
n = int(input())
vysky = list(map(int, input().split()))

suma = 0

for i in range(len(vysky)):

    max_zlava = 0
    for j in range(i+1):
        if max_zlava < vysky[j]:
            max_zlava = vysky[j]

    max_sprava = 0
    for j in range(i, len(vysky)):
        if max_sprava < vysky[j]:
            max_sprava = vysky[j]

    suma += min(max_zlava, max_sprava) - vysky[i]

print(suma)
```

Lepšie riešenie

Lepšie riešenie, za 4 body, ste mohli získať využitím pozorovania, že v riešení hrubou silou robíme jednu veľmi podobnú vec veľakrát. Touto vecou je (ak prechádzame miesta na pozemku zľava doprava) zisťovanie najvyššieho miesta naľavo. Predstavme si, že sme na nejakom mieste i na pozemku. Najvyššie miesto naľavo

je buď také, aké bolo najvyššie miesto naľavo od miesta $i-1$, alebo je to samotné miesto $i-1$. Takto pri prechádzaní celého pozemku vieme pre každé miesto získať najvyššie miesto naľavo od neho v konštantom čase (porovnaním dvoch čísel). Najvyššie miesto napravo ale stále zisťujeme v zložitosti priemerne $O(n/2)$. Celkovo sme teda zložitosť zlepšili na $O(n(n/2))$, čo síce asymptotickú zložitosť nezlepší (Tá je stále $O(n^2)$), ale stačí to na získanie 4 bodov. Pamäťová zložitosť tohto riešenia je $O(n)$, lebo opäť si okrem vstupného poľa dĺžky n a niekoľko málo premenných nemusíme pamätať nič iné.

Listing programu (C++)

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

using namespace std;

int main()
{
    long long sum=0, n=0, max_zlava=0, max_sprava=0;
    cin>>n;
    long long vysky[n];

    for(long long i=0; i<n; i++)
    {
        cin>>vysky[i];
    }

    for(long long i=0; i<n; i++)
    {
        max_zlava=max(max_zlava, vysky[i]);

        max_sprava=0;
        for(long long j=i; j<n; j++)
        {
            max_sprava=max(max_sprava, vysky[j]);
        }

        sum+=(min(max_sprava, max_zlava)-vysky[i]);
    }

    cout<<sum<<endl;

    return 0;
}
```

Listing programu (Python)

```
n = int(input())
vysky = list(map(int, input().split()))

max_zlava = 0

suma = 0
for i in range(n):
    max_zlava = max(max_zlava, vysky[i])

    max_sprava = 0
    for j in range(i, len(vysky)):
        if max_sprava < vysky[j]:
            max_sprava = vysky[j]
```

```
    suma += (min(max_zlava, max_sprava) - vysky[i])

print(suma)
```

Vzorové riešenie

Vzorové riešenie využíva predchádzajúcu myšlienku. Ak prechádzame pozemok zľava doprava, ľahko (v konštantnom čase) vieme pre každé miesto zistiť najvyššie miesto naľavo od neho. Ak ale prechádzame pozemok sprava doľava, tak vieme pre každé miesto ľahko zistiť najvyššie miesto napravo od neho. Keď prejdeme celý pozemok, raz sprava a raz zľava, tak získame pre každé miesto informáciu o najvyššom mieste naľavo a napravo od neho. Obidva tieto prechody (raz zľava doprava a raz sprava doľava) budú v zložitosti $O(n)$. Následne musíme prejsť celý pozemok, pre každé miesto porovnať tieto dve hodnoty a vybrať menšiu z nich. To pre n miest trvá $O(n)$ času. Celkovo nám teda toto riešenie zaberie $O(n+n)$ času na predpočítanie najvyšších miest a potom $O(n)$ času na prejdenie pozemku a sčítanie hodnôt. Spolu je to teda asymptoticky $O(n)$. Pamäťová zložitosť tohto riešenia je stále $O(n)$, lebo okrem vstupného poľa dĺžky n si pamätáme len konštantne veľa (2) polí dĺžky n a niekoľko málo premenných.

Listing programu (C++)

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

using namespace std;

int main()
{
    long long sum=0, n=0, maxi=0;
    cin>>n;
    long long vysky[n+1], max_zlava[n+1], max_sprava[n+2];

    //nacitanie vstupu na indexy 1 az n
    for(long long i=1; i<=n; i++)
    {
        cin>>vysky[i];
    }

    //najvyssie miesto nalavo od miesta i
    max_zlava[0]=0;
    for(long long i=1; i<=n; i++)
    {
        max_zlava[i]=max(vysky[i], max_zlava[i-1]);
    }

    //najvyssie miesto napravo od miesta i
    max_sprava[n+1]=0;
    for(long long i=n; i>0; i--)
    {
        max_sprava[i]=max(vysky[i], max_sprava[i+1]);
    }

    for(long long i=1; i<=n; i++)
    {
        sum+=(min(max_sprava[i], max_zlava[i])-vysky[i]);
    }

    cout<<sum<<endl;
```

```
    return 0;
}
```

Listing programu (Python)

```
n = int(input())

# vstupne pole je na indexoch 1 az n
vysky = [0] + list(map(int, input().split()))
max_zlava = [0]*(n+1)
max_sprava = [0]*(n+2)

for i in range(1, n+1):
    max_zlava[i] = max(max_zlava[i-1], vysky[i])

for i in range(n, 0, -1):
    max_sprava[i] = max(max_sprava[i+1], vysky[i])

suma = 0
for i in range(1, n+1):
    suma += (min(max_zlava[i], max_sprava[i])-vysky[i])

print(suma)
```

Marek

3. Dobré hádanie

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

Poznáme recept (čísla a, b a c) špeciálnej množiny a potrebujeme vedieť povedať, aké číslo sa nachádza na n-tej pozícii v nej.

Nájdeme prvých n čísel

Prvým nápadom na riešenie môže byť, že podľa receptu vyrátame prvých n čísel. To vieme spraviť tak, že postupne pre každé číslo od 1 vyššie zistíme, či číslo patrí do špeciálnej množiny, a opakujeme, kým nenájdeme n takých čísel, pričom n-té z nich je odpoveďou na otázku.

Ako teda zistíme, či nejaké číslo patrí do Pythonovej množiny? No, priamočiaro implementujeme podmienky z receptu: a delí x, no b nedelí x. Alebo a, b aj c delia x. Nejaké a delí nejaké x, práve vtedy keď zvyšok po delení x/a je nula. Inak povedané, ak $x \bmod a = 0$.

Takéto riešenie by teda pre každú z m otázok prešlo všetky čísla od 1 po n-té číslo množiny. Všimnime si, že každé číslo množiny musí byť deliteľné a. Z tohto dôvodu si odhadnime, že čísel od 1 po n-té číslo množiny je $a \cdot n$. Tým pádom je časová zložitosť celého riešenia $O(man)$. Pamäťová zložitosť je konštantná, teda $O(1)$, keďže si nič špeciálne (žiadne polia, iba pár premenných) nepotrebujeme pamätať.

Za takéto riešenie bolo možné získať najviac 2 body.

Nebudeme sa opakovať

Hm… Keď pri každej otázke overujeme vždy čísla od 1, znamená to, že asi veľa čísel overujeme viacnásobne. Teda zbytočne strácame čas rátaním rovnakých vecí. Vieme vymyslieť také riešenie, ktoré každé číslo overí najviac jedenkrát?

Jasné. Vieme si na začiatku programu vyrátať prvých niekoľko veľa čísel Pythonovej množiny. Tie si budeme ďalej pamätať a pri otázke sa iba pozrieme do pamäte a vrátime číslo na n-tej pozícii. Prípadne si vieme najprv prečítať všetky otázky, nájsť najväčšie n a vyrátať si prvých $\max(n)$ čísel.

Samotné hľadanie odpovedí v predrátanej množine trvá konštantne dlho, no predrátanie nám podľa odhadu v predchádzajúcom prístupe zaberie $O(a \max(n))$. V pamäti si teraz potrebujeme držať predpočítanú Pythonovu množinu, čo znamená, že pamäťová zložitosť je $O(\max(n))$.

Za takéto riešenie bolo možné získať 4 body. Ak vylepšíme program tak, že nebudeme skúšať úplne všetky čísla po jednom ale iba po násobkoch a, dostaneme 6 bodov.

Koľké v poradí je toto číslo?

Vieme nejakým spôsobom zistiť odpoveď bez toho, aby sme si museli vyrátať všetky menšie čísla v množine? Áno, vieme. O ľubovoľnom čísle vieme povedať, na akej pozícii v Pythonovej množine sa nachádza (ak číslo nie je v množine, tak dostaneme pozíciu najbližšieho menšieho čísla z množiny).

Koľko existuje čísel menších alebo rovných x, ktoré delí a? Na túto jednoduchú otázku existuje jednoduchá odpoveď: x/a.

Koľko existuje takých čísel menších alebo rovných x, že ich delí a, b aj c súčasne? Je ich presne $x/\operatorname{lcm}(a,b,c)$. (LCM ako least common multiple alebo NSN – najmenší spoločný násobok)

Koľko existuje čísel menších alebo rovných x, ktoré delí a, ale nedelí ich b? Je to v podstate odčítanie množín. Čiže $x/a - x/\operatorname{lcm}(a,b)$.

Keďže dve pravidlá z receptu na zostrojenie Pythonovej množiny tvoria dve disjunktné množiny (v jednom b delí, v druhom nedelí), môžeme k nim pristupovať osobitne a iba ich sčítame: $(x/a - x/\operatorname{lcm}(a,b)) + (x/\operatorname{lcm}(a,b,c))$

Vidíme, že potrebujeme vedieť iba $\operatorname{lcm}(a,b)$ a $\operatorname{lcm}(a,b,c)$, ktoré sú pri všetkých otázkach rovnaké. Takže si ich môžeme vyrátať iba raz na začiatku a ďalej používať iba tieto dve zapamätané čísla, aby sme ich nerátali vždy odznova. Tým pádom zložitosť tohto výpočtu zanedbáme.

Binárne vyhľadávanie

Zistenie pozície nejakého čísla vieme šikovne využiť v riešení hlavne vtedy, keď budeme odpoveď hľadať binárne a nie zaradom od jednotky.

Na začiatku si stanovíme hranice intervalu, na ktorom hľadáme odpoveď, na najmenšie a najväčšie možné číslo v množine (10^{15} by malo stačiť). Pozrieme sa na číslo v strede tohto intervalu a zistíme, koľké je v poradí v Pythonovej množine. Ak je hľadané n väčšie, opakujeme postup s pravou polovicou tohto intervalu (stred, koniec). Ak je menšie, pokračujeme s ľavou polovicou (začiatok, stred).

Keď už dostaneme n, potrebujeme ešte nájsť najbližšie menšie alebo rovné číslo patriace do množiny, keďže číslo v strede ľubovoľného intervalu vôbec nemusí spĺňať recept. Toto zaberie rádovo zanedbateľne málo operácií. Maximálne ac, ak by sme postupne dekrementovali odpoveď po jednotkách. Ak uvažujeme rovno celé násobky a, bude to maximálne c operácií. Myšlienka dôkazu: Ak sú a a b rôzne, do množiny môže v najhoršom prípade patriť druhý najbližší menší násobok a, ak by ten prvý bol zároveň aj násobkom b. Ak sú a a b rovnaké, hľadáme najbližšie menšie alebo rovné číslo deliteľné a a c zároveň.

Pracovný interval pri hľadaní odpovede zmenšujeme binárnym vyhľadávaním vždy na polovicu. Z toho vyplýva, že pred nájdením n vykonáme $O(\log(n))$ operácií. Následné zarovanenie na najbližšie menšie alebo rovné číslo z množiny môžeme zanedbať. A teda celkovú zložitosť pri m otázkach dostaneme $O(m\log(n))$. Keďže si nepotrebujeme nič špeciálne pamätať (okrem konštantného počtu premenných), pamäťová zložitosť je konštatná, teda $O(1)$.

V tomto prípade nám toto riešenie stačilo na plný počet. Existuje však aj niečo ešte lepšie.

Počkať, to ešte nebol vzorák?

Pozrime sa ešte raz na tú matematiku. Už spomenutým vzorcom vieme v konštantnom čase povedať, na akej pozícii sa nachádza nejaké číslo (túto funkciu si nazvyme *get_pos*(x)). Vieme to ale aj opačne. Dôležité sú pre nás čísla $\operatorname{lcm}(a,b)$ a $\operatorname{lcm}(a,b,c)$. Medzi 0 a $\operatorname{lcm}(a,b,c)$ je rovnaký počet prvkov množiny ako medzi $\operatorname{lcm}(a,b,c)$ a $2\cdot\operatorname{lcm}(a,b,c)$. A rovnako to platí aj pre všetky ďalšie intervaly medzi bezprostredne nasledujúcimi násobkami $\operatorname{lcm}(a,b,c)$.

Vo vnútri týchto intervalov zase platí, že každých $\operatorname{lcm}(a,b)$ čísel sa opakuje rovnaký počet prvkov Pythonovej množiny (násobky a bez násobku $\operatorname{lcm}(a,b)$), keďže všeobecne platí, že $\operatorname{lcm}(a,b) \leq \operatorname{lcm}(a,b,c)$ pre ľubovoľné a,b,c. Práve pre zistenie presného počtu v každom takomto intervale použijeme už spomínanú funkciu *get_pos*(x). Teda, *get_pos*$(\operatorname{lcm}(a,b))$ nám povie, koľko prvkov množiny sa opakuje každých $\operatorname{lcm}(a,b)$ čísel, a rovnako to funguje aj pri $\operatorname{lcm}(a,b,c)$.

Ak teda dostaneme nejaké n, $n/$*get_pos*$(\operatorname{lcm}(a,b,c))$ nám povie, koľko krát sa v n-tom čísle množiny nachádza $\operatorname{lcm}(a,b,c)$, resp. na ktorom spomínanom intervale medzi násobkami $\operatorname{lcm}(a,b,c)$ sa nachádza n-té číslo. Podobne, zvyšok zase vydelíme *get_pos*$(\operatorname{lcm}(a,b))$ a zistíme, na ktorom miniintervale sa nachádza medzi násobkami $\operatorname{lcm}(a,b)$. Ak ešte máme nejaký zvyšok, je to počet násobkov a, ktoré ešte potrebujeme k číslu pridať. Tieto tri medzivýsledky vynásobíme príslučnými deliteľmi ($\operatorname{lcm}(a,b,c)$, $\operatorname{lcm}(a,b)$ a a), sčítame to a máme výsledok, čiže n-té číslo Pythonovej množiny. Samozrejme, delíme celočíselne so zvyškom.

Pozor ale na to, že pri delení prvého zvyšku x číslom $\operatorname{lcm}(a,b)$ chceme v skutočnosti deliť $x-1$ a k zvyšku tohto delenia potom prirátame jednotku. To z toho dôvodu, že v prípade, že by x bolo násobkom $\operatorname{lcm}(a,b)$, nezostal by nám žiadny zvyšok, čiže by sme k medzivýsledku už neprirátali žiadny ďalší násobok a v ďalšom

kroku a dostali by sme tým pádom zlý výsledok, keďže práve násobky $\text{lcm}(a, b)$ nepatria do množiny a patrí tam vždy až ďalší najbližší násobok a. Ak tam pridáme tú -1, tento hraničný prípad vyriešime.

Toto riešenie má časovú zložitosť konštantnú, teda $O(1)$, keďže si stačí iba raz vyrátať $\text{lcm}(a, b)$ a $\text{lcm}(a, b, c)$ a ďalej už vieme vyrátať výsledky iba vzorcom pozostávajúcim z konštantného počtu výpočtov. Pamäťová zložitosť je tiež $O(1)$, keďže si stačí pamätať iba konštantný počet premenných.

Nejakým nedopatrením sa stalo, že za toto riešenie bolo možné získať najviac rovnaký plný počet bodov (8), aj keď je lepšie než to predchádzajúce. Avšak, vo výnimočných prípadoch boli rozdané aj nejaké bonusové bodíky.

Listing programu (C++)

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

using namespace std;

int main () {
    long long a, b, c, lcm_ab, lcm_all, m, n;
    cin >> a >> b >> c >> m;

    // vyratame potrebne lcm
    lcm_ab = (a * b) / __gcd(a, b);
    lcm_all = (lcm_ab * c) / __gcd(lcm_ab, c);

    // vseobecne pozicia cisla n: n / a - n / lcm_ab + n / lcm_all
    // na kazdej takejto pozicii acko neratame
    long long pos_ab = lcm_ab / a - 1 + lcm_ab / lcm_all;
    // lcm_all je na kazdej takejto pozicii
    long long pos_all = lcm_all / a - lcm_all / lcm_ab + 1;

    for (long long i = 0; i < m; i++) {
        cin >> n;

        // zaratame vsetky lcm_all
        long long result = n / pos_all * lcm_all;
        // odratame si z n, ze uz sme zaratali tie lcm_all
        n -= n / pos_all * pos_all;

        // pozor na n-1
        if (n > 0) {
            // kolko krat vynechame ab
            result += (n - 1) / pos_ab * lcm_ab;
            // odratame uz tieto pozicie
            n -= (n - 1) / pos_ab * pos_ab;
        }

        // doplnime acka
        result += n * a;

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

Listing programu (Python)

```
from math import gcd
```

```
def main ():
    a, b, c, m = [int(i) for i in input().split()]

    # vyratame potrebne lcm
    lcm_ab = (a * b) // gcd(a, b)
    lcm_all = (lcm_ab * c) // gcd(lcm_ab, c)

    # vseobecne pozicia cisla n: n / a - n / lcm_ab + n / lcm_all
    # na kazdej takejto pozicii acko neratame
    pos_ab = lcm_ab // a - 1 + lcm_ab // lcm_all
    # lcm_all je na kazdej takejto pozicii
    pos_all = lcm_all // a - lcm_all // lcm_ab + 1

    for _ in range(m):
        n = int(input())

        # zaratame vsetky lcm_all
        result = n // pos_all * lcm_all
        # odratame si z n, ze uz sme zaratali tie lcm_all
        n -= n // pos_all * pos_all

        # pozor na n-1
        if n > 0:
            # kolko krat vynechame ab
            result += (n - 1) // pos_ab * lcm_ab
            # odratame uz tieto pozicie
            n -= (n - 1) // pos_ab * pos_ab

        # doplnime acka
        result += n * a

        print(result)


main()
```

Dávid

4. O žabách a hadoch

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

Táto úloha mala veľa rôznych spôsobov ako získať nejaký počet bodov. Ak napríklad skúšame všetky možnosti, hneď máme jeden, s trochou šťastia aj dva body. Ak však začneme využívať informácie zo vstupu, veľmi ľahko sa dostaneme na 4 body. Stačí totiž zistiť aké 4 farby potrebujeme, a postupne vyskúšať všetky ich variácie. Ak si však podobnú stratégiu skúsime ručne odohrať, zistíme, že od súpera dostávame viac informácii o riešení ako využívame a môžeme implementovať rôzne vylepšenia.

Listing programu (Python)

```
#!/usr/bin/python3
from itertools import permutations

c = []

for i in range(1, 7):
    print(i, i, i, i)
    a, b = map(int, input().split())
    for j in range(a):
        c.append(i)
```

```
for i in permutations(c):
    print(*i)
    a, b = map(int, input().split())
    if a == 4:
        break
```

Vzorové riešenie

Nebudeme to ďalej naťahovať, a povieme si, ako sa dopracovať ku riešeniu ktoré to zvláda na 5 tipov. Na začiatok si treba uvedomiť, že všetkých možností je len 6^4 teda 1296. To vôbec nie je veľa, teda ide nám najmä o hľadanie stratégie a nie ani tak o časovú zložitosť.

S každým tipom dostaneme od protihráča celkom veľa informácie. Každá odpoveď nám totiž rozdelí tých 1296 možností na tie, ktoré heslo môžu byť, a tie, ktoré heslo byť nemôžu. To, do ktorej skupiny patrí ktoré štvorčíslie, zistíme veľmi jednoducho, keď si spočítame, akú odpoveď by nám dal protihráč na náš tip, ak by dané štvorčíslie bolo heslo. Keď toto spravíme pre každú jednu možnosť, budeme mať nový zoznam prípustných možností, ktorý už bude výrazne kratší.

S takýmto pozorovaním sa už ľahko dostaneme na 8 až 10 bodov, podľa toho, ako si vyberieme ďalší tip. Ak si povieme, že vylosujeme náhodne z ostávajúcich možností, tak to niekedy vydá a niekedy nie. My by sme však chceli riešenie, ktoré funguje vždy na 5 tipov. Na to potrebujeme vybrať ten správny tip. Ak napríklad vieme, že `1122` má dve zhody, asi nebude najlepší tip `1123`, ale viac nám povie napríklad `3344`.

Najlepší je teda taký tip, ktorý vyradí najviac možností. Ako však taký tip nájsť, keď nepoznáme heslo? Povedzme, že po prvom tipe sme si vytvorili vyššie popísaný zoznam možných hesiel. Teraz chceme o nejakom nasledujúcom tipe zistiť, ako dobrý je. To budeme merať tak, že si pre každé možné heslo zistíme, akú odpoveď by sme dostali a pre každú možnú odpoveď (tých je 14) spočítame, koľko možných hesiel by nám ostalo, ak by sme danú odpoveď dostali. Nevieme však, akú odpoveď dostaneme, tak rátajme s najhoršou možnosťou. Teda takou, ktorá by vyradila najmenej hesiel. Keď si takéto číslo zrátame o každom možnom tipe, vieme si ľahko vybrať ten, ktorý vyradí najviac možností v najhoršom prípade. Treba si však uvedomiť, že niekedy môže byť najlepší aj tip, ktorý heslom určite nebude (ako v príklade v predošlom odseku). Aj bez tohoto pozorovania však už dostaneme 9 bodov.

Táto technika sa volá minmax, lebo najprv si pre každý tip zistíme minimum jeho prínosu a potom zoberieme maximum z týchto možností. Dá sa takmer univerzálne aplikovať na hry, kde nepoznáme ťah súpera, ale vieme nejakým spôsobom odsimulovať všetky jeho možnosti. Treba však myslieť na to, že výsledok nemusí byť optimálny, nakoľko napríklad v tejto úlohe sa vôbec nepozeráme na ťahy ktoré budú nasledovať[4].

Časová zložitosť

Časová zložitosť nás trochu potrápi, podľa toho, ako dôslední chceme byť. Označme f počet farieb a n dĺžku hesla. Povedzme pre jednoduchosť, že porovnanie dvoch tipov robíme v $O(n^2)$[5]. Prefiltrovať tipy podľa odpovede, ktorú dostaneme, nám potom trvá $O(f^n n^2)$. Ďalší krok je, že pre každý možný tip ($O(f^n)$) zisťujeme, pre každé možné heslo ($O(f^n)$), akú odpoveď by sme dostali ($O(n^2)$). Možných odpovedí je tiež n^2, teda najlepšiu z nich tiež nájdeme v $O(n^2)$. Výsledná časová zložitosť jedného tipu teda bude $O(f^{2n}n^2)$. Pamäťová zložitosť je $O(f^n)$.

Najhorší možný počet ťahov je takmer nemožné ručne odhadnúť na 5, preto sa jednoducho odvoláme na to, že skúsiť všetky skryté heslá nie je tak ťažké[6].

Listing programu (Python)

```
#!/usr/bin/python3
from collections import defaultdict
from itertools import product

odpovede = [(0,0),(0,1),(0,2),(0,3),(0,4),
            (1,0),(1,1),(1,2),(1,3),
            (2,0),(2,1),(2,2),
            (3,0),
```

[4] Môže sa totiž stať, že druhá najlepšia možnosť aktuálne nám zabezpečí lepšiu najhoršiu možnosť v ďalšom ťahu, ako tá aktuálne najlepšia.

[5] Dá sa to však aj v $O(n)$.

[6] http://www.cs.uni.edu/~wallingf/teaching/cs3530/resources/knuth-mastermind.pdf

```
            (4,0)]

def diff(a,b):
    a=list(a)
    b=list(b)
    right = 0
    notright = 0
    for i in range(len(a)):
        if a[i] == b[i]:
            right += 1
            a[i] = -1
            b[i] = -2
    for i in range(len(a)):
        for j in range(len(a)):
            if a[i] == b[j]:
                notright += 1
                b[j] = -2
                break
    return(right, notright)


n = 4
m = 6
s = set(product(range(1, m + 1), repeat=n))
all = s.copy()
t = (1, 1, 2, 2)
used = [t]
print(*t)
while True:
    a, b = map(int, input().split())
    if a == n:
        break
    s = {i for i in s if diff(used[-1], list(i)) == (a, b)}
    besttip = []
    minmax = 12345678
    for a in all:
        if a in used:
            continue
        m = defaultdict(int)
        for pos in s:
            ans = diff(a, pos)
            m[ans] += 1

        m = max(m.values())
        if m < minmax or (m == minmax and a in s):
            minmax = m
            besttip = a
    used.append(besttip)
    print(*besttip)
```

Jano

5. Nádoby

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

V tejto úlohe sme mali sústavu spojených nádob poprepájaných hadičkami, pričom do prvej nádoby sa napúšťala voda. Našou úlohou bolo zistiť, z ktorej nádoby sa voda vyleje.

Pozorovania

Môžme si všimnúť, že mnohé informácie, ktoré na vstupe dostaneme, pre nás vlastne nemajú žiaden význam. Voda začne tiecť z jednej nádoby do druhej, keď hladina dosiahne výšku najvyššieho bodu hadice, ktorou sú prepojené. Výšky koncových bodov hadice na to nemajú žiaden vplyv, môžme ich teda ignorovať. Výška dna nádoby tiež nič neovplyvňuje.

Pomalé riešenie

Budeme si pamätať, v ktorých nádobách už je voda, a v ktorých ešte nie.

Zakaždým preiterujeme všetky hadičky, aby sme našli tú, ktorá spája už zaplavenú nádobu s ešte nezaplavenou a spomedzi všetkých takýchto má najnižší vrchol. To nám povie, ktorá nádoba sa najbližšie zaplaví. Ak je však najnižšia výška vrchného okraja spomedzi už zaplavených nádob nižšia ako vrchol tejto hadičky, voda sa vyleje z najnižšej spomedzi už zaplavených nádob.

Časová zložitosť takéhoto riešenia je $O(n \cdot k)$.

Vzorové riešenie

Problém s predchádzajúcim riešením je, že zakaždým, keď sme chceli vedieť, do ktorej nádoby voda ďalej potečie, nám to zabralo veľmi veľa času. Tu nám pomôže halda[7].

Použijeme haldu, na ktorej budú udalosti dvoch typov, pretečenie cez okraj nádoby a pretečenie z jednej nádoby do druhej. Halda bude utriedená podľa výšky, ktorú musí voda dosiahnuť, aby udalosť nastala (teda pri udalostiach prvého typu výška nádoby, a pri udalostiach druhého typu výška najvyššieho bodu hadice). Udalosť s najnižšou výškou bude na vrchu haldy.

Na začiatku máme v halde iba udalosti, ktoré sa týkajú prvej nádoby.

Simulácia teda bude vyzerať takto. Zakaždým z haldy vyberieme udalosť s najnižšou výškou. Ak je to pretečenie cez hadičku do druhej nádoby, v ktorej ešte voda nie je, udalosti súvisiace s touto nádobou (pretečenie cez jej okraj, a cez hadičky do susedných nádob) pridáme do haldy a zapamätáme si, že tu je voda. Ak je to pretečenie cez okraj, máme odpoveď, ďalej už simulovať nemusíme.

Časová zložitosť

Jedna operácia vkladania alebo vyberania z haldy s s prvkami nám zaberie $O(\log s)$. Na halde môže byť najviac $2k + n$ udalostí (pre každú nádobu pretečenie cez okraj a pre každú hadičku pretečenie do jednej alebo druhej strany). Koľko operácii budeme robiť? Určite nie viac ako $4k+2n$, pretože každú udalosť môžme najviac raz vložiť a raz vybrať z haldy. Časová zložitosť teda bude $O((4k+2n) \cdot \log(4k+2n)) = O((n+k) \cdot \log(n+k))$.

Pamäťová zložitosť

Okrem vecí zo vstupu si potrebujeme držať v pamäti ešte našu haldu ($O(n+k)$) a jedno pole, na to, aby sme vedeli, v ktorých nádobách už je voda ($O(n)$), čo je dokopy $O(n+k)$.

Listing programu (C++)

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

using namespace std;

int main(){
    int n,k,_;
    cin >> n >> k;
    for(int i=0; i<n; i++){
        cin>>_;
    }
    vector<int> hor(n);
    for(int i=0; i<n; i++){
        cin >> hor[i];
    }
    vector<vector<pair<int,int>>> graf(n);
```

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

```
    for(int i=0; i<k; i++){
        int x,y,c;
        cin >> x >> y >> _ >> _ >> c;
        x--; y--;
        graf[x].push_back(pair<int,int>(y,c));
        graf[y].push_back(pair<int,int>(x,c));
    }

    vector<bool> voda(n);
    priority_queue<pair<int,pair<bool,int>>,std::vector<pair<int,pair<bool,int
        ↪ >>>,std::greater<pair<int,pair<bool,int>>>> halda;

    voda[0] = true;
    for(unsigned int i=0; i<graf[0].size(); i++){
        pair<int,int> h = graf[0][i];
        halda.push(pair<int,pair<bool,int>>(h.second, pair<bool,int>(false,h.
            ↪ first)));
    }
    halda.push(pair<int,pair<bool,int>>(hor[0], pair<bool,int>(true,0)));

    while(true){
        pair<bool,int> event = halda.top().second;
        halda.pop();
        if(event.first){
            cout<<event.second+1<<'\n';
            return 0;
        }else{
            int v = event.second;
            if (voda[v]){
                continue;
            }
            voda[v] = true;
            for(unsigned int i=0; i<graf[v].size(); i++){
                pair<int,int> h = graf[v][i];
                halda.push(pair<int,pair<bool,int>>(h.second, pair<bool,int>(
                    ↪ false,h.first)));
            }
            halda.push(pair<int,pair<bool,int>>(hor[v], pair<bool,int>(true,v)))
                ↪ ;
        }
    }
}
```

Listing programu (Python)

```
from heapq import heappush,heappop

n,k = map(int,input().split())
input()
hor = list(map(int,input().split()))

graf = [[] for _ in range(n)]
for _ in range(k):
    x,y,_,_,c = map(int,input().split())
    x -= 1
    y -= 1
    graf[x].append((y,c))
    graf[y].append((x,c))
```

```
voda = [False] * n
halda = []

voda[0] = True
for sused,c in graf[0]:
    heappush(halda, (c, False, sused))
heappush(halda, (hor[0], True, 0))

while True:
    _, pretieklo, v = heappop(halda)
    if pretieklo:
        print(v+1)
        exit()
    else:
        if voda[v]:
            continue
        voda[v] = True
        for sused,c in graf[v]:
            heappush(halda, (c, False, sused))
        heappush(halda, (hor[v], True, v))
```

fezjo

6. Opäť sa hady hadiť budú

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

Hlavná myšlienka

Ak by sme s úlohou nevedeli nijako pohnúť, mohli by sme naprogramovať nejakú rekurziu, čo by skúšala všetky permutácie ukladania hadov do mriežky a mala by otrasnú časovú zložitosť. Skúsenejšiemu riešiteľovi by sa však mala táto úloha rýchlo odhaliť ako úloha na dynamické programovanie.

Teda namiesto toho, aby sme navštívili každé možné rozloženie hadov a zväčšili počítadlo riešení o jedna, budeme veľa rozložení charakterizovať určitým stavom, navštevovať iba tieto menej početné stavy a počítadlo tak zväčšovať o veľké množstva naraz.

Teraz by sme boli radi, keby sme vedeli všetky rozloženia mriežky nejako rozumne reprezentovať do čo najmenej stavov, aby sme mali čo najlepšiu časovú zložitosť.

Poďme do mriežky umiestňovať hadov. Robiť to len tak náhodne asi nie je veľmi rozumné, a teda sa rozhodnime, že ich budeme umiestňovať napríklad po riadkoch zdola nahor – teda pre každý riadok sa rozhodneme koľko a kam chceme hadov položiť a potom prejdeme na ďalší riadok. Vďaka tomuto sa vždy nachádzame v stave, kde všetky riadky, na ktoré sme sa doteraz pozreli, už nemusíme uvažovať a všetky, na ktoré sme sa ešte nepozreli, sú prázdne. Namiesto toho aby sme si teda pamätali celú tabuľku s pozíciami hadov, nám bude stačiť si pamätať, koľko hadov sme už umiestnili, koľko riadkov sme už naplnili (počet neumiestnených hadov a prázdnych riadkov je triviálne dopočítateľný) a počet hadov pre každý stĺpec.

Čo sa týka stĺpcov, môžeme si povšimnúť, že ak máme nejaké rozloženie hadov, na poradí stĺpcov veľmi nezáleží a ľubovoľné ich premiešanie je tiež valídne rozloženie. Preto je lepšia reprezentácia, kde si namiesto pamätania počtu hadov pre každý stĺpec pamätáme iba počet stĺpcov čo majú 0, 1 a 2 hady.

V tomto momente vieme teda stav reprezentovať piatimi číslami (počet umiestnených hadov, naplnených riadkov a stĺpcov s 0, 1 a 2 hadmi), čím by sme získali približne $O(KNM^3)$ stavov. Toto je však stále omnoho viac, ako je potrebné na získanie 8 bodov. Našťastie by malo byť ľahké tento počet zredukovať. Totiž, počet stĺpcov s 0 a 1 hadmi sa dá ľahko vypočítať z ostatných hodnôt. Ak U je počet hadov, ktoré sme už umiestnili, a S_i je počet stĺpcov s i hadmi, tak $S_1 = U - 2S_2$ a $S_0 = M - (S_1 + S_2)$. Zostáva nám teda iba $O(KNM)$ stavov. Keďže K môže byť nanajvýš $2 \cdot \min(N, M)$ (inak neexistuje valídne rozloženie), tak počet stavov je $O(\min(N, M)^2 \cdot \max(N, M))$.

Samozrejme, existujú aj iné reprezentácie stavov, bolo by však zbytočné ich tu uvádzať.

Teraz treba nájsť spôsob, ako z aktuálneho stavu odvodiť ostatné, alebo naopak, ako z ostatných stavov odvodiť aktuálny. Ľahšie sa vysvetľuje tá druhá možnosť, takže spravme to. Povedzme, že chceme vyjadriť stav $D[n][u][s_2]$, teda stav, kde sme už naplnili n riadkov, umiestnili u hadov a máme s_2 stĺpcov s dvoma hadmi. Pripomínam, že z týchto troch informácií vieme dopočítať s_1 a s_0. Do tohoto stavu sme prišli z nejakého iného,

v ktorom sme mali o jeden menej naplnených riadkov, tak, že sme do jedného riadku pridali nula až dvoch hadov. Každú z týchto troch možností vyriešime samostatne.

Ak sme sa v predchádzajúcom stave rozhodli uložiť 0 hadov, počet uložených hadov ani počet s_2 sa nezmenil, teda sme do aktuálneho stavu iba pripočítali počet rozložení predchádzajúceho stavu. Teda $D[n][u][s_2]$ += $D[n-1][u][s_2]$.

Ak sme sa v predchádzajúcom stave rozhodli uložiť 1 hada, počet uložených hadov sa zvýšil o jedna a počet s_2 sa buď zmenil alebo nezmenil, podľa toho, do ktorého stĺpca sme daného hada vložili. Ak sme ho vložili do stĺpca s 0 hadmi (ktorých je s_0) tak sa s_2 nezmenilo. Ak sme ho vložili do stĺpca s jedným hadom (ktorých je s_1) tak sa s_2 zvýšilo o 1. Teda $D[n][u][s_2]$ += $s_0 \cdot D[n-1][u-1][s_2] + s1 \cdot D[n-1][u-1][s_2-1]$.

Ak sme sa v predchádzajúcom stave rozhodli uložiť 2 hadov, počet uložených hadov sa zvýšil o dva a počet s_2 sa znova mohol, ale nemusel, zmeniť. Tentoraz máme až 3 možnosti ako sme ich mohli uložiť:

1. oboch do nulového stĺpca – to môžeme spraviť $s_0(s_0-1)/2$ spôsobmi
2. jedného do nulového a jedného do jednotkového stĺpca – to môžeme spraviť $s_0 \cdot s_1$ spôsobmi
3. oboch do jednotkového stĺpca – to môžeme spraviť $s1(s1-1)/2$ spôsobmi

To, ako sa zmenilo s_2 je ľahké odvodiť. Dostávame teda $D[n][u][s_2]$ += $(s_0(s_0-1)/2) \cdot D[n-1][u-2][s_2] + (s_0 \cdot s_1) \cdot D[n-1][u-2][s_2-1] + (s_1(s_1-1)/2) \cdot D[n-1][u-2][s_2-2]$.

Zostáva nám iba pridať modulovanie a ošetriť, aby sme nepoužívali záporné indexy do polí. Začíname s informáciou $D[0][0][0] = 1$ a zvyšok tabuľky už vieme potom dopočítať. Vypočítanie ľubovoľného indexu trvá $O(1)$ a teda časová aj pamäťová zložitosť je $O(\min(N,M)^2 \cdot \max(N,M))$.

Posledná vec, ktorú si môžeme všimnúť, je, že výpočet $D[n]$ závisí iba od $D[n-1]$. Teda ak budeme tabuľku počítať v rozumnom poradí (najprv všetky hodnoty pre určité n, až potom pre $n+1$), môžeme na $D[n-2]$ vždy zabudnúť a znížiť tak pamäťovú zložitosť na $O(M \cdot \min(N,M))$. A keďže vieme, že počet riešení je rovnaký pre tabuľky rozmerov $N \times M$ a $M \times N$, tak môžeme N a M vymeniť tak, aby nám to vyhovovalo a získať pamäťovú zložitosť $O(min(N,M)^2)$.

Riešenie tejto úlohy obsahovalo veľa pozorovaní a trikov. Veľa z nich však nebolo nutných (napríklad zníženie pamäťovej zložitosti), alebo sa dali získať čiastkové body ak ste ich nenašli.

Listing programu (C++)

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

using ll = long long;
const ll MOD = 1e9 + 7;

int main() {
    ll N, M, K;
    cin >> N >> M >> K;

    if(N > M) // aj ked to nie je nutne, je fajn mat v zivote urcite istoty
        swap(N, M); // napriklad ze N<M
    if(K > N * 2)
        return 0;

    vector<vector<vector<ll>>> D(N+1, vector<vector<ll>>(K+1, vector<ll>(M+3, 0)
    ↪ ));
    D[0][0][0] = 1;
    for(ll n=0; n<N; ++n) { // ideme po N-1, v n=N uz budu vypocitane vysledky
        for(ll k=0; k<=K; ++k) {
            for(ll m2=0; m2<=M; ++m2) {
                // mi je pocet stlpcov co maju i policok obsadenych - potom
                ↪ plati:
                // m0+m1+m2 == M, 0*m0+1*m1+2*m2 == k
                ll &curr = D[n][k][m2];
                curr %= MOD; // vzdy sa snazme modulovat co najmenej
                // kedze modulo je pomale, teda namiesto toho aby sme nizsie
                // modulovali 6 krat ked pripocitavame do D, staci modulovat
```

```
                // iba raz ked chceme vyslednu hodnotu pouzit

                ll m1 = k - m2*2;
                ll m0 = M - m2 - m1;
                ll zostava = K - k;

                // na rozdiel od slovneho vozraku tu hodnoty tlacime namiesto
                // tahania. myslienkovo v tom nie je velky rozdiel, len sa mi
                // jedna vec lahsie vysvetlovala a druha lahsie programovala
                if(zostava >= 0) {
                    D[n+1][k][m2] += curr;
                }
                if(zostava >= 1) {
                    D[n+1][k+1][m2+1] += curr * m1;
                    D[n+1][k+1][m2] += curr * m0;
                }
                if(zostava >= 2) {
                    D[n+1][k+2][m2+2] += curr * m1 * (m1 - 1) / 2;
                    D[n+1][k+2][m2+1] += curr * m1 * m0;
                    D[n+1][k+2][m2] += curr * m0 * (m0 - 1) / 2;
                }
            }
        }
    }

    ll res=0;
    for(ll x: D.back().back())
        res = (res + x) % MOD;
    cout << res << endl;
}
```

Listing programu (Python)

```
# toto je vpodstate bezduchy prepis c++ kodu do pythonu
# ak viete c++ tak si skor precitajte to, su tam komentare navyse
# ale aj ked c++ neviete tak ho celkom iste bez vacsich problemov pochopite

MOD = 10**9+7

N, M, K = map(int, input().split())

if N > M:
    N, M = M, N
if K > N * 2:
    return 0

D = [[[0 for m in range(M+3)] for k in range(K+1)] for n in range(N+1)]
D[0][0][0] = 1
for n in range(N):
    for k in range(K+1):
        for m2 in range(M+1):
            # m0+m1+m2 == M, 0*m0+1*m1+2*m2 == k
            D[n][k][m2] %= MOD
            curr = D[n][k][m2]

            m1 = k - m2*2
            m0 = M - m2 - m1
            left = K - k
```

```
            if left >= 0:
                D[n+1][k][m2] += curr
            if left >= 1:
                D[n+1][k+1][m2+1] += curr * m1
                D[n+1][k+1][m2] += curr * m0
            if left >= 2:
                D[n+1][k+2][m2+2] += curr * m1 * (m1 - 1) // 2
                D[n+1][k+2][m2+1] += curr * m1 * m0
                D[n+1][k+2][m2] += curr * m0 * (m0 - 1) // 2
res=0
for x in D[-1][-1]:
    res = (res + x) % MOD

print(res)
```

Dano

7. Šibalské sny

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

Na vstupe sme mali strom popisujúci súhvezdie. Našou úlohou bolo spočítať, koľkými spôsobmi vieme preusporiadať hviezdy v súhvezdí, aby boli splnené isté podmienky. Konkrétnejšie, mali sme zrátať niečo ako koľko stromov je izomorfných so zadaným stromom tak, že koreňom je stále vrchol číslo 0. Presné podmienky si môžete pozrieť v zadaní.

Pozorovania

Zamyslime sa najskôr nad tým, kedy sa budú dať hviezdy preusporiadavať. Pozrime sa na koreň. Aby sme dostali iný strom spĺňajúci podmienky, môžeme zmeniť poradie tých podstromov koreňa, ktoré sú v nejakom zmysle rovnaké. Aby sme teda zachovali správne prepojenie vrcholov hranami, budeme môcť koreňu iba vymieňať rovnaké podstromy. Avšak, v rámci každého podstromu môžeme spraviť tiež nejaké preusporiadanie, ktoré štruktúru podstromu nezmení. Rekurzívne teda môžeme popísať vyhovujúce preusporiadania pre podstromy koreňa, a tak ďalej, až ku listom.

Ako ale povedať, či sú dva podstromy rovnaké?

Vzorové riešenie

Je množstvo možností, ktoré nám na riešenie tejto otázky napadnú. Či už je to porovnávanie počtu vrcholov, listov, alebo hĺbok, žiaden z týchto spôsobov nebude fungovať. Nájsť príklady dvoch rôznych stromov, pre ktoré tieto metriky budú tvrdiť, že sú rovnaké, nie je problém.

Riešenie je prekvapivo jednoduché. Dva podstromy sú zameniteľné práve vtedy, keď ich korene majú ako synov rovnaké podstromy. Nezáleží nám na poradí synov. Zameniteľné budú, pretože synov vieme vhodne preusporiadať a neporušíme pri tom žiadnu podmienku zo zadania.

Budeme teda podstromom priraďovať `id`. Dva podstomy budeme považovať za zameniteľné, ak majú rovnaké `id`. Tieto čísla budeme priraďovať rekurzívne od listov. Stačí nám na to jedno DFS[8]. Listy budú mať `id = 1`. Keď sa rekurzívne zavoláme do všetkých synov, budú už mať priradené `id`. Tieto ich `id` si zaradom dáme do poľa. Dostaneme tak napríklad pole `[1, 2, 4, 1, 1, 6]`. Z toho chceme určiť `id` aktuálneho vrchola. Zistili sme už ale, že na poradí synov nezáleží. Aby naše pole teda určovalo `id` vrchola, usporiadame si ho. Následne sa pozrieme, či sme už niekedy boli vo vrchole s týmto usporiadaným poľom `id`-čiek synov. Ak áno, pridelíme aktuálnemu vrcholu rovnaké `id` ako danému vrcholu. Ak sme takého pole ešte nevideli, pridelíme nášmu vrcholu napríklad najmenšie ešte nepridelené `id`.

Ostáva nám ešte vyriešiť kombinatorickú časť úlohy. Nech `ways[ID]` hovorí, koľkými spôsobmi vieme preusporiadať podstrom s `id = ID`. Už vieme, že môžeme vymieňať synov s rovnakým `id`. Každého syna zároveň vieme preusporiadať. Konkrétne, syna s `id = x` vieme usporiadať `ways[x]` spôsobmi. Jednotlivé typy synov sú na sebe nezávislé, preto tieto počty medzi sebou vynásobíme. Označme `cnt[ID]` počet synov aktuálneho vrchola s `id = ID`. Ďalej označme S množinu `id`-čiek synov aktuálneho vrchola (teda je bez duplikátov). Dostaneme nasledový vzťah:

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

$$\text{ways}[\text{ID}] = \prod_{x \in S} \text{cnt}[x]! \cdot \text{ways}[x]^{\text{cnt}[x]}$$

Samozrejme, všetko modulujeme. Celkovým výsledkom potom bude ways[$\text{id}_{\text{koreň}}$].
Časová zložitosť bude kvôli usporiadavaniu v DFS $O(n \log n)$ na test. Pamäť bude $O(n)$.

Listing programu (C++)

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

using namespace std;

typedef long long ll;

const ll MXN = 10111;
const ll MOD = 1000000007;
ll TC, n, a, b;
vector<vector<ll>> G;
map<vector<ll>, ll> ids_by_children; // Tymto namapujeme usporiadane pole id-
    ↪ ciek deti na id aktualneho vrchola
vector<ll> vertex_ids, ways, fact;

void calc_fact() {
    fact.clear();
    fact.push_back(1);
    for(ll i = 1; i < MXN; i++)
        fact.push_back((fact[i - 1] * i) % MOD);
}

// Umocnovanie q^x v O(log x) s modulovanim
ll expo(ll q, ll x) {
    if(x == 0)
        return 1LL;
    if(x == 1)
        return q;
    ll base = expo((q * q) % MOD, x / 2);
    if(x & 1LL)
        base = (base * q) % MOD;
    return base % MOD;
}

void dfs(ll u, ll p) {
    vector<ll> child_ids;
    map<ll, ll> cnt;
    for(int i = 0; i < G[u].size(); i++) {
        ll v = G[u][i];
        if(v != p) {
            dfs(v, u);
            child_ids.push_back(vertex_ids[v]);
            cnt[vertex_ids[v]]++;
        }
    }
    sort(child_ids.begin(), child_ids.end());
    bool done = true; // Ci sme uz pridelili id aktualnemu child_ids
    if(ids_by_children.find(child_ids) == ids_by_children.end()) {
        ids_by_children[child_ids] = ids_by_children.size(); // Pridelime
            ↪ sekvencne dalsie id
```

```
        done = false; // Ak sme uz davnejsie nepridelili id, tak este nemame
            ↪ zratane ways, tak si to poznacime
    }
    vertex_ids[u] = ids_by_children[child_ids];
    if(!done) {
        // Pouzijeme sucin zo vzoraku
        for (auto &x: cnt) {
            ways[vertex_ids[u]] *= (fact[x.second] * expo(ways[x.first], x.
                ↪ second)) % MOD;
            ways[vertex_ids[u]] %= MOD;
        }
    }
}

int main() {
    calc_fact(); // Predpocitame si faktorialy % MOD
    cin >> TC;
    while(TC--) {
        cin >> n;
        G.clear();
        G.resize(n);
        for(int i = 0; i < n - 1; i++) {
            cin >> a >> b;
            G[a].push_back(b);
            G[b].push_back(a);
        }
        ids_by_children.clear();
        ways.assign(n + 1, 1);
        vertex_ids.assign(n, -1);
        dfs(0, -1);
        cout << ways[vertex_ids[0]] << '\n'; // Koren je 0
    }

    return 0;
}
```

Listing programu (Python)

```
import sys

sys.setrecursionlimit(1000000)

MXN = 10111
MOD = 10**9 + 7
fact = [1]


# Logaritmicke umocnovanie q^x s modulovanim
# Da sa namiesto toho pouzit funkcia pow
def expo(q, x):
    if x == 0:
        return 1
    if x == 1:
        return q
    base = expo((q * q) % MOD, x // 2)
    if x % 2 == 1:
        base = (base * q) % MOD
    return base % MOD
```

```
def dfs(u, p, G, ids_by_children, vertex_ids, ways):
    child_ids = []
    cnt = {}
    for i in range(len(G[u])):
        v = G[u][i]
        if v != p:
            dfs(v, u, G, ids_by_children, vertex_ids, ways)
            child_ids.append(vertex_ids[v])
            if vertex_ids[v] not in cnt:
                cnt[vertex_ids[v]] = 0
            cnt[vertex_ids[v]] += 1
    child_ids = tuple(sorted(child_ids))  # Usporiadanie pole id-ciek deti
    done = True  # Ci sme uz aktualnemu child_ids pridelili id niekedy davnejsie
    if child_ids not in ids_by_children:
        ids_by_children[child_ids] = len(ids_by_children)  # Sekvencne pridelime
            ↪ dalsie id
        done = False  # Este sme nepridelili, musime to spravit v dalsich par
            ↪ riadkoch
    vertex_ids[u] = ids_by_children[child_ids]
    if not done:
        # Pouzijeme sucin zo vzoraku
        for k, v in cnt.items():
            ways[vertex_ids[u]] *= (fact[v] * expo(ways[k], v)) % MOD
            ways[vertex_ids[u]] %= MOD


# Predpocitame si faktorialy % MOD
for i in range(MXN):
    fact.append((fact[i] * (i + 1)) % MOD)

TC = int(input())

for tc in range(TC):
    ids_by_children = {}
    n = int(input())
    G = [[] for _ in range(n)]
    for i in range(n - 1):
        a, b = [int(x) for x in input().split()]
        G[a].append(b)
        G[b].append(a)
    ways = [1] * n
    vertex_ids = [-1] * n
    dfs(0, -1, G, ids_by_children, vertex_ids, ways)
    print(ways[vertex_ids[0]])  # Koren je 0
```

Paulinka

8. Inovatívny dážď

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

Možno ste si všimli, že táto úloha je nápadne podobná sedmičke z minulého kola. To veru nie je náhoda, vznikla pri písaní checkeru na tú sedmičku :) Tak sa pusťme do toho, veď napísať checker nemôže byť ťažšie ako originálne riešenie, či?[9]

Riešenie za tri body

Získať tri body nie je tak ťažké – každý obdĺžnik overíme v lineárnom čase: najskôr si nájdeme cestu

[9] a veru môže

v strome medzi vrcholmi, ktoré nás zaujímajú (na to postačí napríklad obyčajné DFSko), a potom pre každý vrchol skontrolujeme, či leží, alebo neleží v nejakom obdĺžniku. Ak vrchol leží na ceste, ale neleží v žiadnom obdĺžniku, vypíšeme `NIE`, a rovnako vypíšeme `NIE`, ak niektorý vrchol neležiaci na ceste zas leží v nejakom obdĺžniku.

Takéto riešenie má čas $O(n)$ na query, takže celková časová zložitosť je $O(nq)$ a pamäťová $O(n)$.

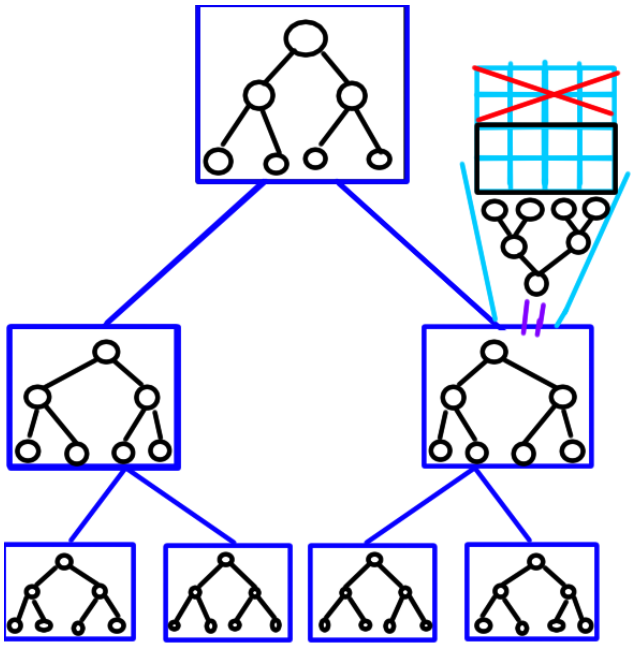
Koľko všetkých vrcholov leží v obdĺžniku?

Predstavme si najskôr zjednodušenú úlohu: daný je obdĺžnik, koľko vrcholov stromu v ňom leží? Keby sme mali miesto obldĺžnika iba 1D interval? Potom je úloha ľahko riešiteľná intervaláčom! A keď sa nad tým zamyslíme, tie obdĺžniky nie sú od intervalov až tak odlišné, takže tiež použijeme intervaláč, avšak 2D intervaláč!

2D intervalový strom

2D intervalový strom je intervalový strom, ktorý má v každom vrchole intervalový strom. Zdá sa to kus abstraktné?

Pozrime sa na obrázok:



Majme vrchol vo "veľkom" intervaláči, pokrývajúci interval $[a, b)$. Povedzme, že tieto intervaláče pokrývajú x-ovú súradnicu. V tomto vrchole je druhý intervaláč rovnakej veľkosti. Ten si v intervale $[c, d)$ pamätá hodnotu z políčok $[a, b) \times [c, d)$.

My chceme súčtový intervaláč, z ktorého dostaneme počet vrcholov v obdĺžniku, takže si tento "malý" intervaláč pamätá počet vrcholov v obdĺžniku s rohmi (a, c) a (b, d).

Update jedného políčka v intervaláči nám zaberie $O(\log^2 n)$ času, keďže najskôr potrebujeme updatnúť $O(\log n)$ "malých" intervaláčov po ceste. Podobne, query zaberie $O(\log^2 n)$ času.

Zdá sa však, že tento veľký 2D intervaláč zaberie veľa miesta: má $2n$ vrcholov, a ak je v každom jeden intervaláč zaberajúci $O(n)$ pamäte, potrebujeme až $O(n^2)$ pamäte! Už to začína znieť nerealizovateľne, avšak povšimnime si kontrast medzi pamäťovou a časovou zložitosťou: ak potrebujeme updatnúť n vrcholov, použijeme $O(n \log^2 n)$ času, ale až $O(n^2)$ pamäte! Teda veľa inicializovanej pamäte je vstutku zbytočnej. Čo s tým?

Môžeme použiť múdru, lenivú implementáciu intervaláča: každý vrchol (aj v "malom" intervaláči) má pointre (smerníky) na svoje dve deti (alebo NULL ak deti ešte neboli upravované). Ak príde query do vrchola a chcela by ísť do niektorého neexistujúceho dieťaťa, vrátime za daného potomka odpoveď nula. Ak príde update, ktorý by potreboval updatovať aj nejakého neexistujúceho potomka, potomka vytvoríme a pustíme update rekurzívne ďalej.

Takže vieme zaručiť, že nedostaneme asymptoticky horšiu pamäťovú zložitosť ako časovú zložitosť na update queries, teda $O(n \log^2 n)$.

Už sa blížime k riešeniu otázky: *koľko všetkých vrcholov leží v obdĺžniku?*

Najskôr si predpripravíme takýto 2D intervalový strom a updatneme $+1$ na políčka, kde ležia vrcholy.

Následne, pre každý obdĺžnik, povedzme s rohmi v (x_1, y_1) a (x_2, y_2), zavoláme query: ako v bežnom intervaláčovom hľadaní ideme po "veľkom" intervaláči, kým nie je rozsah x-súradníc vrcholu (povedzme že

$[a, b)$) podmnožina intervalu x-súradníc nášho obdĺžnika (teda $[x_1, x_2)$). V takomto vrchole pokračujeme naše hľadanie do "menšieho" intervaláča, kde následne nájdeme počet vrcholov so súradnicami $a \leq x < b$ a $y_1 < y_2$.

Takto vieme rovnako ako v normálnom 1D intervaláči vyskladať celý obdĺžnik, a teda získame počet vrcholov ležiacich v obdĺžniku.

Avšak, nejako stále ignorujeme otázku, ako zistiť, či vrcholy na nejakej ceste ležia v obdĺžniku?

Leží cesta v obdĺžniku?

Ďalší krok v ceste za riešením je zistenie, koľko vrcholov z nejakej cesty leží v obdĺžniku.

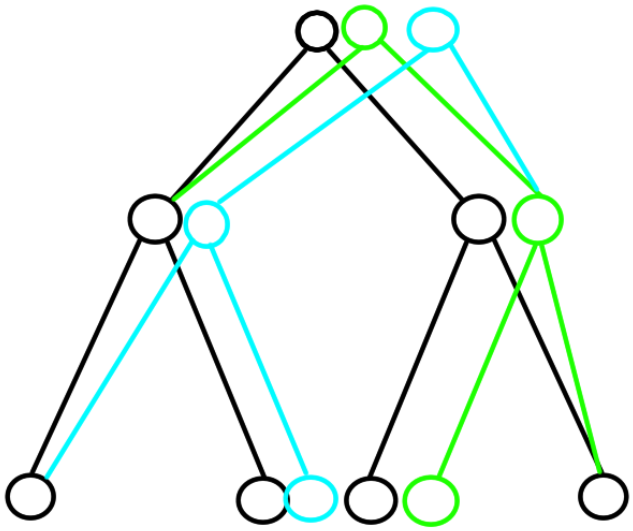
Nápad je nasledovný: Použijeme *perzistentný intervaláč*, a to konkrétne tak, že keď sa pozrieme na intervaláč v konkrétnom čase, bude to súčtový intervaláč vrcholov na nejakej ceste od koreňa do vrchola. Poďme sa na to pozrieť bližšie:

Perzistentný 2D intervaláč

Cieľom je mať intervalový strom, v ktorom sa vieme pýtať query nielen do priestoru, ale aj v čase – teda otázky typu: aký je súčet na tomto obdĺžniku pred u updatmi?

Ako sa to dá robiť? Používame 2D intervaláč naprogramovaný pomocou pointerov. Vždy, keď sa vrchol má updatnuť, neuložíme zmenu priamo – miesto toho vytvoríme nový, updatnutý vrchol.

Pre lepšiu ilustráciu použime obrázok:



Pri updatnutí tretieho listu (pozície 3) vytvoríme nový list (zelenou). Následne updatujeme jeho rodiča (keďže sa vytvorilo nové, updatnuté dieťa). Na staré dieťa nazabúdame, starý rodičovský vrchol ostáva ako je, ale vytvárame nový zelený vrchol, ktorému ako deti dáme aktuálne verzie synov (teda ľavý zelený list a pravý čierny list). Následne ideme zas hore, a vytvoríme nový koreň, ktorého pointer do pravého syna ukazuje práve na náš nový zelený vrchol.

Keďže my máme intervaláče v intervaláčoch, musíme ešte domyslieť technické detaily, čo ako, keď robíme tieto updaty. Napríklad "malé" intervaláče máme uložené vo vrcholoch väčšieho intervaláča ako pointre, a pri každom update dostaneme nový pointer, na nový koreň "malého" intervaláča, ktorý potom uložíme v novom vrchole.

A nie je toto nejaké pomalé? Veď predsa vytvárame veľa nových vecí!

Ale keď sa nad tým zamyslíme, nie je ich v skutočnosti až tak veľa. Pre každý update updatneme $O(\log n)$ vrcholov vo veľkom intervaláči a pre každý tento vrchol updatneme ešte $O(\log n)$ vrcholov v malom intervaláči. Takže dokopy dostaneme zložitosť $O(log^2 n)$ času aj pamäte na jednu update query, čo je pomerne akceptovateľné.

Použitie perzistentného intervaláča

Ako tento perzistentný intervaláč použiť?

Začneme s prázdnym 2D perzistentným intervaláčom. Zakoreňme si strom a prehľadávajme ho DFSkom. Kedykoľvek prídeme do nového vrcholu, pridajme $+1$ na jeho súradnicu v intervaláči. Kedykoľvek z vrcholu nadobro odídeme, pridajme -1 na jeho súradnicu v intervaláči.

Takto, keď sme v akomkoľvek vrchole, aktuálny stav intervaláča (ten po poslednom update) je, že sú +1 na všetkých predkoch vrchola (do tých sa už DFS dostalo, ale ešte ich neopustilo nadobro), ale zároveň, ak sme nejaký vrchol už navštívili, ale nie je predok, museli sme ho aj opustiť, takže na jeho súradnici je $+1-1=0$. Ak sme nejaký vrchol ešte nenavštívili, intervaláč si preň pamätá 0 (resp. si nepamätá nič, keďže sme sa k updatovaniu súradnice ešte nedostali, ale pre to, ako intervaláč funguje, to je rovnaké, ako keby tam bola nula).

Takže keď chceme dostať stav, že jediné +1 sú na súradniciach vrcholov na ceste z koreňa do nejakého vrchola v, stačí si zobrať koreň intervaláča z updatu hneď po navštívení v.

Už sa blížime ku koncu, ostávajú technické detaily.

A čo pre akúkoľvek cestu?

Akúkoľvek cestu v strome vieme rozdeliť na dve slížovité cesty od najbližšieho spoločného predka dole (pozri napríklad kuchárku[10]).

Najbližšieho spoločného predka vieme nájsť napríklad v čase $O(1)$ s predpočítaním $O(n \log n)$.

Keď máme takto rozdelené cesty, pre každú cestu získame počet vrcholov v obdĺžniku zvlášť, a keďže LCA leží na oboch, odrátame 1, ak leží v obdĺžniku.

Ako to teda nájsť pre peknú slížovitú cestu z predka p do potomka v? Vieme nájsť koľko vrcholov je v obdĺžniku pre cestu z koreňa do v. Chceli by sme odrátať tie vrcholy, ktoré ležia na ceste medzi koreňom a p. Ale to vieme, zistíme koľko je vrcholov v obdĺžniku na ceste medzi koreňom a rodičom p (aby sme neodrátali p), a to odčítame od počtu vrcholov v obdĺžniku na ceste medzi koreňom a v. A hurá, dostali sme, čo sme chceli.

A ako teda odpovedať na queries?

Poďme si to zrekapitulovať.

Dostaneme query vo forme dvoch vrcholov, a a b, a zopár (k) obdĺžnikov. Obdĺžniky sa zaručene neprekrývajú, takže máme o jeden kameň na srdci menej.

Zistime najskôr, koľko je všetkých vrcholov v strome ležiacich v obdĺžnikoch (to vieme, obyčajným 2D intervaláčom) v $O(k \log^2 n)$ čase.

Zistime, koľko vrcholov na ceste medzi a a b leží v obdĺžnikoch. Tiež to vieme zistiť v čase $O(k \log^2 n)$.

Pochopiteľne, tieto dva údaje sa musia rovnať, inak kričíme `NIE` (inak by nejaký vrchol mimo cesty ležal v obdĺžniku).

Ak sa rovnajú, to stále nie je automatické áno: môže sa stať, že v obdĺžnikoch leží menej vrcholov z cesty, ako má. Ako toto overiť? Jednoducho, stačí nám navyše zistiť počet vrcholov na ceste a porovnať ho s počtom vrcholov z cesty ležiacich v obdĺžnikoch.

Počet vrcholov na ceste sa dá zistiť jednoducho: zapamätáme si výšku každého vrchola, a potom dĺžka cesty je:

$$v_a + v_b - 2v_{\text{lca}}$$

Kde v_a je výška a, v_b je výška b a v_{lca} je výška najbližšieho spoločného predka. Tak, a toto nám dáva online odpoveď na query v $O(k \log^2 n)$ čase. Na predpočítanie sme použili čas $O(n \log^2 n)$ (na vybudovanie intervaláča a predpočítanie LCA, a teda celková časová zložitosť je $O((n + qk_{\max}) \log^2 n)$ a pamäťová $O(n \log^2 n)$.

Listing programu (C++)

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

struct node_1d {
    int xstart, xrange;
    node_1d *prvy, *druhy;
    int sum;

    node_1d(int N) {
        xstart = 0;
        xrange = N;
        sum = 0;
        prvy = NULL, druhy = NULL;
```

[10] https://www.ksp.sk/kucharka/lca/

```
    }

    node_1d(int start, int range, int update_to, int change) {
        xstart = start;
        xrange = range;
        sum = change;
        if (range != 1) {
            int child_range = range / 2;
            if (update_to < child_range + start) {
                prvy = new node_1d(start, child_range, update_to, change);
                druhy = NULL;
            }
            else {
                prvy = NULL;
                druhy = new node_1d(start + child_range, child_range, update_to,
                    ↪  change);
            }
        }
        else {
            prvy = NULL;
            druhy = NULL;
        }
    }

    node_1d(node_1d *previous, int update_to, int change) {
        if (previous -> xrange == 1) {
            xstart = previous -> xstart, xrange = previous -> xrange;
            prvy = NULL, druhy = NULL;
            sum = previous -> sum + change;
        }
        else {
            int child_range = previous -> xrange / 2;
            xrange = previous -> xrange, xstart = previous -> xstart;
            sum = previous -> sum + change;
            if (update_to < xstart + child_range) {
                if (previous -> prvy == NULL) prvy = new node_1d(xstart,
                    ↪ child_range, update_to, change);
                else prvy = new node_1d(previous -> prvy, update_to, change);
                druhy = previous -> druhy;
            }
            else {
                if (previous -> druhy == NULL) druhy = new node_1d(xstart +
                    ↪ child_range, child_range, update_to, change);
                else druhy = new node_1d(previous -> druhy, update_to, change);
                prvy = previous -> prvy;
            }
        }
    }

    int query(int z, int k) {
        if (xstart >= k || xstart + xrange <= z) return 0;
        if (z <= xstart && k >= xstart + xrange) return sum;
        return (prvy == NULL ? 0 : prvy->query(z, k)) + (druhy == NULL ? 0 :
            ↪ druhy -> query(z, k));
    }
};

struct node_2d {
```

```
int ystart, yrange;
node_1d *seg_tree;
node_2d *prvy, *druhy;

node_2d(int N) {
    ystart = 0;
    yrange = N;
    seg_tree = new node_1d(N);
    prvy = NULL, druhy = NULL;
}

node_2d(int start, int range, int to_x, int to_y, int change, int N) {
    ystart = start, yrange = range;
    seg_tree = new node_1d(0, N, to_x, change);
    if (range == 1) {
        prvy = NULL, druhy = NULL;
    }
    else {
        int child_range = range / 2;
        if (start + child_range > to_y) {
            druhy = NULL;
            prvy = new node_2d(start, child_range, to_x, to_y, change, N);
        }
        else {
            prvy = NULL;
            druhy = new node_2d(start + child_range, child_range, to_x, to_y
                ↪ , change, N);
        }
    }
}

node_2d(node_2d *prev, int to_x, int to_y, int change, int N) {
    yrange = prev -> yrange, ystart = prev -> ystart;
    seg_tree = new node_1d(prev -> seg_tree, to_x, change);
    if (yrange != 1) {
        int child_range = yrange / 2;
        if (ystart + child_range > to_y) {
            if (prev -> prvy == NULL) prvy = new node_2d(ystart, child_range
                ↪ , to_x, to_y, change, N);
            else prvy = new node_2d(prev -> prvy, to_x, to_y, change, N);
            druhy = prev -> druhy;
        }
        else {
            prvy = prev -> prvy;
            if (prev -> druhy == NULL) druhy = new node_2d(ystart +
                ↪ child_range, child_range, to_x, to_y, change, N);
            else druhy = new node_2d(prev -> druhy, to_x, to_y, change, N);
        }
    }
    else {
        prvy = NULL, druhy = NULL;
    }
}

int get_sum(int x1, int x2, int y1, int y2) {
    if (y1 >= ystart + yrange || y2 <= ystart) return 0;
    if (y1 <= ystart && y2 >= ystart + yrange) {
        int res = seg_tree -> query(x1, x2);
```

```
            return res;
        }
        return (prvy != NULL ? prvy -> get_sum(x1, x2, y1, y2) : 0) + (druhy !=
            ↪ NULL ? druhy -> get_sum(x1, x2, y1, y2) : 0);
    }

};

struct persistent_2dsegment_tree {
    int N;
    vector<node_2d *> roots;

    persistent_2dsegment_tree(int n) {
        N = pow(2, ceil(log2(n)));
        roots.push_back(new node_2d(N));
    }

    int update(int x, int y, int change) {
        int kam = roots.size();
        roots.push_back(new node_2d(roots.back(), x, y, change, N));
        return kam;
    }

    int query_sum(int time_start, int time_end, int x1, int y1, int x2, int y2)
        ↪ {
        int end = roots[time_end] -> get_sum(x1, x2, y1, y2);
        int start = roots[time_start] -> get_sum(x1, x2, y1, y2);
        return end - start;
    }
};

void dfs_to_segtree(int v, int f, persistent_2dsegment_tree *segtree, vector<int
    ↪ > &Z, vector<vector<int> > &hrany, vector<int> &X, vector<int> &Y) {
    int id = segtree -> update(X[v], Y[v], 1);
    Z[v] = id - 1;
    for (int w : hrany[v]) {
        if (w == f) continue;
        dfs_to_segtree(w, v, segtree, Z, hrany, X, Y);
    }
    segtree -> update(X[v], Y[v], -1);
}

void dfs_euler(int v, int f, int h, vector<vector<int> > &hrany, vector<pair<int
    ↪ , int> > &euler, vector<int> &ID, vector<int> &H) {
    ID[v] = (euler.size());
    H[v] = h;
    euler.push_back({h, v});
    for (int w : hrany[v]) {
        if (w == f) continue;
        dfs_euler(w, v, h + 1, hrany, euler, ID, H);
        euler.push_back({h, v});
    }
}

struct rmq {
    int N;
    vector<vector<pair<int, int> > > R;
```

```
    rmq() {}

    rmq(int n, vector<pair<int, int> > &A) {
        N = n;
        int id = 0;
        int jump = 1;
        R.push_back(A);

        while (jump < n) {
            R.push_back(vector<pair<int, int> >(n));
            for (int i = 0; i < n; i ++) {
                R[id + 1][i] = min(R[id][i], (i + jump < n ? R[id][i + jump] :
                    ↪ make_pair(N, N)));
            }
            id ++;
            jump *= 2;
        }
    }

    int query(int a, int b) {
        if (a > b) swap(a, b);
        int id = log2(b - a);
        return min(R[id][a], R[id][b - (1 << id)]).second;
    }
};

struct checker {
    int n;
    vector<vector<int> > hrany;
    vector<int> X, Y, Z, H, ID;
    persistent_2dsegment_tree *segtree;
    persistent_2dsegment_tree *all;
    int ind;
    rmq R;

    checker(int n, vector<vector<int> > &hrany, vector<int> &xcoor, vector<int>
        ↪ &ycoor): n(n), hrany(hrany), X(xcoor), Y(ycoor), segtree(new
        ↪ persistent_2dsegment_tree(n)), all(new persistent_2dsegment_tree(n)) {
        Z.resize(n);
        ID.resize(n);
        H.resize(n);
        vector<pair<int, int> > euler;
        dfs_to_segtree(0, -1, segtree, Z, hrany, X, Y);
        dfs_euler(0, -1, 0, hrany, euler, ID, H);
        R = rmq(2 * n - 1, euler);

        for (int i = 0; i < n; i ++) {
            ind = all -> update(X[i], Y[i], 1);
        }

    }

    bool check(int a, int b, int k, vector<int> x1, vector<int> y1, vector<int>
        ↪ x2, vector<int> y2) {
        for (int i = 0; i < k; i ++) {
            x1[i] --;
            y1[i] --;
        }
```

```
        int ida = ID[a], idb = ID[b];
        int lca = (a == b ? a : R.query(ida, idb));

        int count_in = 1 + (H[a] - H[lca]) + (H[b] - H[lca]);
        int count = 0;

        for (int i = 0; i < k; i ++) {
            int in_all = all -> query_sum(0, ind, x1[i], y1[i], x2[i], y2[i]);
            int path_in = segtree -> query_sum(Z[lca], Z[a] + 1, x1[i], y1[i],
                ↪ x2[i], y2[i]);

            path_in += segtree -> query_sum(Z[lca], Z[b] + 1, x1[i], y1[i], x2[i
                ↪ ], y2[i]);


            if (x1[i] <= X[lca] && x2[i] > X[lca] && y1[i] <= Y[lca] && y2[i] >
                ↪ Y[lca]) {
                path_in --;
            }
            if (in_all != path_in) return false;
            count += path_in;
        }
        return (count == count_in);
    }

};

int main() {
    int n;
    cin >> n;
    vector<vector<int> > hrany(n);
    vector<int> X, Y;

    for (int i = 0; i < n; i ++) {
        int x, y;
        cin >> x >> y;
        X.push_back(x - 1);
        Y.push_back(y - 1);
    }

    for (int i = 0; i < n - 1; i ++) {
        int a, b;
        cin >> a >> b;
        a --, b --;
        hrany[a].push_back(b);
        hrany[b].push_back(a);
    }
    checker C(n, hrany, X, Y);

    int q;
    cin >> q;

    for (int i = 0; i < q; i ++) {
        int a, b;
        cin >> a >> b;
        a --, b --;
        int k;
```

```
        cin >> k;
        vector<int> x1(k), x2(k), y1(k), y2(k);
        for (int j = 0; j < k; j ++) {
            cin >> x1[j] >> y1[j] >> x2[j] >> y2[j];
        }
        cout << (C.check(a, b, k, x1, y1, x2, y2) ? "OK" : "NIE") << endl;
    }
}
```