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

Vzorové riešenia 1. kola zimnej časti

Jitka

1. Hádam prežijeme zimu

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

Riešenie hrubou silou

Priamočiare riešenie je prejsť všetkými výmenami, ktoré existujú a overiť, či by k nim mohlo dôjsť. Takže skúsime vymeniť každú Gabikinu kôpku guliek postupne s každou Timkinou kôpkou. Pre každú výmenu musíme overiť, či sa môže uskutočniť, teda či majú obe párny súčet všetkých svojich guličiek. Toto by sme mohli overiť tak, že pre každú výmenu sčítame počty guliek na každej kôpke postupne pre Gabiku aj Timku a tento súčet zmodulujeme dvoma. Uvedomme si, že toto vôbec nemusíme robiť, stačí nám raz spočítať zvlášť súčet Gabikiných a Timkiných guliek ešte predtým ako budeme prechádzať výmenami. Pri skúšaní konkrétnej výmeny nám potom stačí Gabike odčítať počet guliek v kôpke, čo dáva Timke a pričítať počet guliek v kôpke, ktorú dostáva od Timky. Toto číslo následne vydelíme dvoma, aby sme zistili, či má Gabika párny počet guliek. To isté spravíme s Timkiným počtom guličiek, odrátame guličky, ktoré dáva Gabike a prirátame tie, ktoré od nej dostáva a vydelíme dvoma. Ak sú oba súčty deliteľné dvoma, zistili sme, že k výmene dôjde a môžeme ukončiť skúšanie. Ak je aspoň jeden nepárny, tak musíme skušať ďalšie výmeny.

Časová zložitosť tohto riešenia je $O(n^2)$, pretože každú z n Gabikiných kôpok guličiek skúsime vymeniť s n Timkinými kôpkami guličiek a vrámci jednej výmeny robíme len jednoduché aritmetické operácie. V dvoch poliach si pamätáme n Gabikiných a n Timkiných kôpok a okrem toho iba pomocné premenné, takže pamäťová zlozitisť je $O(n)$.

Vzorové riešenie

Na vylepšenie predchádzajúceho riešenia si musíme uvedomiť ako sa správajú párne a nepárne čísla pri súčte. Keď sčítame čísla rovnakej parity, sučet bude párny a keď sčítame čísla opačnej parity dostaneme číslo nepárne. Rovnako to platí aj pri rozdieli. Podľa týchto pravidiel vieme na základe parity súčtov všetkých guliek hovniválok povedať, kedy dôjde k výmene. Pripomeňme si ešte, čo sa stane keď príde k výmene medzi Timkou a Gabikou. Timka aj Gabika jednu kôpku darujú, teda odráta sa počet guliek, ktoré obsahovala a jednu dostanú, teda priráta sa jej počet guličiek. Poďme si teraz rozobrať možné prípady výmeny. Povedzme, že Gabika má súčet svojich guliek g a Timka t.

Ak g a t sú rôznej parity, napríklad g je párne a t nepárne (opačný prípad sa správa rovnako), tak k výmene nedôjde. Potrebovali by sme totiž zmeniť paritu t. Podľa toho, čo sme si povedali na začiatku vieme, že musíme k t prirátať párne číslo a odrátať nepárne alebo prirátať nepárne a odrátať párne. Ako sa pri týchto výmenách správalo g? Buď sme k nemu pripočítali nepárne číslo a odčítali od neho párne alebo opačne, každopádne sme zmenili jeho paritu na nepárnu.

Ak sú g aj t párne k výmene dôjde za istých podmienok. Oba súčty sú párne a teda im nechceme zmeniť paritu, to vieme, že sa stane ak ich zmeníme o párne číslo, teda buď pripočítame a odpočítame párne čísla alebo pripočítame a odpočítame nepárne čísla. Minimálne jeden z týchto prípadov vieme spraviť práve vtedy, keď Gabika má aspoň jednu kôpku rovnakej parity ako Timka. Inak sa výmena nemôže uskutočniť.

Ak sú g a t nepárne tiež môže nastať výmena. Keďže sú oba súčty nepárne potrebujeme im obom zmeniť paritu. Vieme, že nepárne číslo zmeníme na párne súčtom alebo rozdielom s nepárnym číslom. Od oboch čísel teda potrebujeme odpočítať číslo jednej parity a pripočítať k nemu číslo druhej parity. Toto v našej úlohe vieme spraviť ak má Gabika aspoň jednu kôpku s počtom guličiek rôznej parity ako aspoň jedna Timkina kôpka. Keď si vymenia túto dvojicu kôpok nastane presne tá situácia, ktorú potrebujeme. Inak k výmene nemôže dôjsť.

Celé riešenie vieme implementovať bez zapamätania si všetkých počtov guličiek na kôpkach v poli. Stačí nám si pri náčítávaní počtov guliek každej hovniválky zapamätať, či sme našli aspoň jeden párny počet, aspoň jeden nepárny počet a sčítavať celkový súčet všetkých guličiek. Pre ten následne určíme, či je párny alebo nepárny.

Už nekontrolujeme všetky výmeny, iba raz prejdeme vstupom pri jeho načítávaní, takže časová zložitosť tohto riešenia je lineárna $O(n)$ a keďže sme si ukázali, že nám netreba si pamätať vstup, iba pár premenných, tak pamäťová zložitosť je konštantná $O(1)$.

Listing programu (C++)

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

using namespace std;

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

    int gabika_parita = 0;
    int timka_parita = 0;
    bool gabika_parny = false;
    bool gabika_neparny = false;
    bool timka_parny = false;
    bool timka_neparny = false;

    for (int i = 0; i < n; i++)
    {
        int a;
        cin >> a;
        gabika_parita += a;
        if (a % 2 == 0)
        {
            gabika_parny = true;
        }
        else
        {
            gabika_neparny = true;
        }
    }

    for (int i = 0; i < n; i++)
    {
        int a;
        cin >> a;
        timka_parita += a;
        if (a % 2 == 0)
        {
            timka_parny = true;
        }
        else
        {
            timka_neparny = true;
        }
    }

    gabika_parita %= 2;
    timka_parita %= 2;

    if (gabika_parita == 0 and timka_parita == 0 and ((gabika_neparny and
        ↪ timka_neparny) or (gabika_parny and timka_parny)))
    {
        cout << "ano" << endl;
    }
```

```
    else if (gabika_parita == 1 and timka_parita == 1 and ((gabika_neparny and
        ↪ timka_parny) or (gabika_parny and timka_neparny)))
    {
        cout << "ano" << endl;
    }
    else
    {
        cout << "nie" << endl;
    }
}
```

Listing programu (Python)

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

gabika_parita = sum(gabika) % 2
timka_parita = sum(timka) % 2
gabika_parny = False
gabika_neparny = False
timka_parny = False
timka_neparny = False

for cis in gabika:
    if cis % 2 == 1:
        gabika_neparny = True
    else:
        gabika_parny = True

for cis in timka:
    if cis % 2 == 1:
        timka_neparny = True
    else:
        timka_parny = True


if gabika_parita == 0 and timka_parita == 0 and ((gabika_neparny and
    ↪ timka_neparny) or (gabika_parny and timka_parny)):
    print('ano')
elif gabika_parita == 1 and timka_parita == 1 and ((gabika_neparny and
    ↪ timka_parny) or (gabika_parny and timka_neparny)):
    print('ano')
else:
    print('nie')
```

2. Ováľané mince

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

Skúsme si najprv rozdeliť rovnomerne nejaké čísla $a, a+1, ..., b$. Veľmi rýchlo si môžeme všimnúť, že ak berieme najväčšie a najmenšie, druhé najväčšie a druhé najmenšie, tak tieto dvojice majú vždy rovnaký súčet.

Bohužiaľ, nám to ešte nepovie, ako rozdeliť čísla, aby bol medzi nimi čo najmenší rozdiel. Povedzme si najprv, že kedy vieme mince $1, 2, ..., n$ rozdeliť na rovnaké časti. Aby bolo možné rozdeliť mince rovnomerne, tak ich súčet musí byť párny, a teda v ňom musí byť párny počet nepárnych čísiel. To, či to v našej postupnosti platí, vieme šikovne zisťovať tak, že (počet čísel označme ako n), $n\%4 = 3$ alebo $n\%4 = 0$.

Ľahko si všimneme, že ak $n\%4 = 0$, tak mince vieme rovnomerne rozdeliť tak, že zoberieme dvojice 1 a n, 2 a $n-1$, … . Takýchto dvojíc je párny počet, a teda ich vieme rozdeliť na dve rovnaké polovice.

V prípade ak $n\%4 = 3$, tak je to už trochu zložitejšie. Ak ale vieme, že $n + 1$ mincí (teda $n\%4 = 0$) sa dá rozdeliť, tak toto sa líši len o $n + 1$ a teda ak jednej strane zoberieme mincu/mince v hodnote $(n + 1)/2$, tak budú obe polovice vyrovnané. (Dá sa to jednoduchšie predstaviť ak si to napíšete na papier.)

Čo ale v prípade, ak sa nedajú mince rozdeliť rovnomerne? Ak $n\%4 = 2$, tak máme vlastne oproti prípadu, ktorý vieme rovnomerne rozdeliť ($n\%4 = 0$), naviac čísla $n + 1$ a $n + 2$, a tieto čísla sa líšia len o 1, a to je minimálny rozdiel, čo vieme dosiahnuť. (Nech robíme čo robíme, nepárny súčet na dve rovnaké polovice nerozdelíme.)

A ostáva posledný prípad, keď $n\%4 = 1$. Tento prípad je zase veľmi podobný prípadu keď $n\%4 = 3$, a rovnako tu máme naviac dve čísla, a rovnako je minimálny rozdiel 1.

Keď už máme vypočítanú hodnotu, akú má dostať každý stavbár, teraz pokiaľ $p = 1$, tak musíme vypísať aj aké mince každý dostane.

Čísla vieme získavať buď tak, ako sme písali, teda po dvojiciach, alebo jednoduchšie, pažravo. Pažravo ich získame tak, že ideme od najväčších mincí, a počítame si súčet už zobraných mincí. Ak si danú mincu vieme zobrať (pripočítať k súčtu) bez toho, aby sme presiahli polovicu cekovej sumy, zoberieme ju.

Možno si kladiete otázku, či to takto pažravo môže fungovať. Odpoveď je, áno. Naše mince majú v súčte určite väčšiu hodnotu ako potrebujeme, a máme mince s každou hodnotou, ktorú by sme mohli potrebovať. Teda máme zaručené, že nech zmenšíme rozdiel medzi sumou, ktorú chceme získať a sumou, ktorú sme zatiaľ nazbierali, na ľubovoľnú hodnotu, tak určite bude existovať minca, ktorá sa nám do tohto rozdielu zmestí.

Čo sa týka časovej zložitosti, pokiaľ $p = 0$, tak časová zložitosť je $O(1)$, lebo odpoveď na jednu otázku vypočítame v konštantnom čase, vzorcom. V prípade, že $p = 1$, musíme dokopy vypísať n-prvkové pole, ktoré ale vieme získať v čase priamo úmernom od n, takže zložitosť je $O(n)$. Pamätať si nemusíme nič, pretože hodnoty poľa môžeme vypočítať vo for-cykle bez toho, aby sme si okrem niekoľko málo premenných niečo naviac pamätali, a preto je pamäťová zložitosť konštantná $= O(1)$.

Listing programu (C++)

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

using namespace std;

int main()
{
    long long t=0, n=0, i=0, j=0, suma=0, odpoved=0, suma_doteraz=0;
    vector<long long> prva, druha;
    cin>>t;
    for(i=0;i<t;i++)
    {
        cin>>n>>odpoved;
        suma=n*(1+n)/2;
        cout<<(suma%2)<<endl;
        if(odpoved==1)
        {
            suma_doteraz=0;
            prva.clear();
            druha.clear();
            for(j=n;j>0;j--)
            {
                if(suma_doteraz+j <= suma/2)
                {
                    suma_doteraz+=j;
                    prva.push_back(j);
                }
                else
                    druha.push_back(j);
            }
            cout<<prva.size();
            for(j=0;j<prva.size();j++)
```

```
                cout<<" "<<prva[j];
            cout<<endl;

            cout<<druha.size();
            for(j=0;j<druha.size();j++)
                cout<<" "<<druha[j];
            cout<<endl;

        }
    }
}
```

Listing programu (Python)

```
def vysledok(k):
    return int((k*(k+1)//2) % 2)


def vypis(k):
    polovica = int((k*(k+1)//2) / 2)
    p1 = []
    p2 = []
    sum1 = 0
    sum2 = 0
    for i in range(k, 0, -1):
        if sum1+i > polovica:
            sum2 += i
            p2.append(i)
            continue
        sum1 += i
        p1.append(i)
    print(len(p1), *p1)
    print(len(p2), *p2)


n = int(input())
for i in range(n):
    k, mam_vypisat = map(int, input().split())
    print(vysledok(k))
    if mam_vypisat:
        vypis(k)
```

3. Vystupovanie

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

Pomalé riešenie

Ako prvé si predstavíme priamočiare riešenie bez žiadnych trikov. Budeme jednoducho vstupné pole lineárne prehľadávať počnúc od indexu l a počítať, koľkokrát sme už narazili na zastávku v správnej vzdialenosti v od Jozefovho domu. Akonáhle narazíme na k-tu takú, vypíšeme jej index. Ak ale prekonáme index r pred tým, ako nájdeme odpoveď, odpovieme -1. Toto riešenie by bolo optimálne, v prípade, keby sme dostali len $q = 1$ otázok. To, že otázok môže byť veľa, nám ale napovedá, že sa asi dá najprv so vstupnými údajmi vykonať nejaká transformácia, ktorá nám potom umožní odpovede zisťovať rýchlejšie ako v lineárnom čase.

Vzorové riešenie

Kľúčový trik k riešeniu úlohy je nasledujúci: Pre každú vzialenosť v od domu si vytvoríme pole, v ktorom sa

nachádzajú čísla (indexy) zastávok, ktoré majú túto vzdialenosť. Napríklad ak je na vstupe pole: $[3, 4, 5, 4, 5]$ tak by sme si vytvorili v pamäti štruktúru podobnú nasledujúcej:

```
X[3]: 1
X[4]: 2 4
X[5]: 3 5
```

Na jej uskladnenie môžeme použiť buď hash mapu (dict) alebo jednoduché pole veľkosti 10^6, pretože to je maximálna hodnota v.

Všimnime si, že každé pole $X[v]$ je usporiadané. Teraz vieme na otázky odpovedať nasledujúcim postupom: Pozrieme sa do poľa $X[v]$ a nájdeme v ňom najmenšiu hodnotu väčšiu alebo rovnú l. Ako na to? Postupné prechádzanie po jednom funguje, ale má zložitosť $O(n)$, takže si moc nepomôžeme. Zachráni nás ale binárne vyhľadávanie, ktoré to spraví v $O(\log n)$. Zastávka, ktorú takto nájdeme, je prvá taká, cez ktorú prejde autobus s Jozefom a je v správnej vzdialenosti od jeho domu. My chceme ale k-tu, nie prvú, takže sa v tomto poli $X[v]$ pozrieme o $k-1$ pozícii ďalej a nájdeme presne to, čo hľadáme. Ešte musíme ale vyriešiť dve prekážky: Ak by sme pri tomto procese vybehli z poľa, tak vieme, že taká zastávka neexistuje a odpovieme -1. Druhý problém je r z otázky, no jednoducho sa stačí pozrieť na číslo zastávky, ktorú sme našli a ak je väčšie ako r, tak opäť vypíšeme -1, pretože zastávka už je mimo Jozefovej platnosti lístka.

Celková časová zložitosť riešenia (s použitím hash mapy) je $O(n + q \log n)$. n pochádza z predpočítania štruktúry X a $q \log n$ sú jednotlivé odpovede na otázky. Pamäťová zložitosť je $O(n)$, pretože jediné čo si pamätáme je X, pričom dokopy sa v ňom nachádza presne n prvkov. Keby sme pre X nepoužili hash mapu ale pole, zložitosti by boli trochu iné. Museli by sme na začiatku inicializovať pole veľkosti 10^6, čo je určite viac ako n, pretože n dosahuje len 10^5. Taktiež si ho musíme pamätať. Ak by sme pomenovali $w = 10^6$ maximálnu hodnotu v, tak by časová zložitosť bola $O(w + n + q \log n)$ a pamäťová $O(w + n)$.

Listing programu (C++)

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

using namespace std;

int main() {
    int n, q;
    cin >> n >> q;
    vector<vector<int>> V(1000001);
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        V[x].push_back(i+1);
    }
    for (int i = 0; i < q; i++) {
        int l, r, v, k;
        cin >> l >> r >> v >> k;
        k--;
        int f = upper_bound(V[v].begin(), V[v].end(), l-1) - V[v].begin();
        if (f + k >= V[v].size() || V[v][f+k] > r)
            cout << -1 << '\n';
        else
            cout << V[v][f+k] << '\n';
    }
}
```

Listing programu (Python)

```python
import bisect
from collections import defaultdict
```

```
def readints():
    return map(int, input().split())

n, q = readints()
X = list(readints())
# dict, ale ked pristupime k neexistujucemu prvku,
# tak automaticky vyrobi prazdny list
V = defaultdict(list)
for i, x in enumerate(X, 1):
    V[x].append(i)

for _ in range(q):
    l, r, v, k = readints()
    k-=1
    # Binarne vyhladavanie
    f = bisect.bisect(V[v], l-1)
    if f+k >= len(V[v]) or V[v][f+k] > r:
        print(-1)
    else:
        print(V[v][f+k])
```

Medved

4. Nové PINy pre trezor

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

Pomalé riešenie

Jeden spôsob je cez brute-force. Držíme si v pamäti PINy ako idú zaradom. Iterujeme cez pole PINov od začiatku. Pre každý PIN si budeme v pamäti držať najlepší výsledok pre ten PIN. Čiže pre prvý PIN v poli by to bola 0. Pre každý ďalší PIN porovnáme najlepšie výsledky s PINmi ktoré sme už prešli, ktoré sa zároveň od terajšieho PINu líšia len v jednej cifre. Pridáme najlepší výsledok PINu, ktorý porovnávame, a pridáme k nemu rozdiel cifier, v ktorých sa dva PINy líšia. Na konci iterovania sa pozrieme, aké najväčšie číslo sme týmto dosiahli, a to bude naša odpoveď. Takýto algoritmus by mal časovú zložitosť $O(p^2)$, pretože by sme pre každý PIN vykonali minimálne toľko operácii, koľko PINov je už za nami, a máme dokopy p pinov.

Vzorové riešenie

Vieme sa dostať k lepšiemu riešeniu pomocou dynamického programovania. Vieme, že $n \leq 6$, a tým pádom počet rôznych PINov určite nepresiahne 10^6. Inicializujeme pole L o veľkosti 10^n, kde každý index reprezentuje jeden možný PIN. Napr. PIN 00002 by sa nachádzal na indexe 2 v poli L, a na tomto indexe by sme našli najlepšiu odpoveď pre tento PIN.

Budeme si pamätať najlepšiu možnú odpoveď pre každý možný PIN, čo na začiatku bude -1. -1 preto, lebo chceme vedieť, či sa daný PIN vôbec nachádza v našom poli PINov, a bolo by dobré túto skutočnosť nejak označiť. Pre každý PIN, ktorý pri iterovaní stretneme, nastavíme v poli L na jeho indexe najlepšiu odpoveď na aspoň 0. Týmto označíme, že sa tento PIN v poli PINov nachádza.

Trik v tomto riešení je, že keď iterujeme PINmi, tak namiesto toho, aby sme kontrolovali všetky predošlé PINy, tak skontrolujeme všetky PINy, ktoré sú tomuto PINu "susedné". Dva piny sú susedné vtedy, keď sa líšia v jednej cifre. Každý PIN má teda $10n - 1$ susedných PINov, pretože každý PIN má n cifier, kde cifra môže byť ľubovoľné číslo od 0 po 9. Pre každý takýto susedný PIN sa pozrieme na pole L, a ak sme tento PIN stretli (čiže hodnota na tom indexe nie je -1), tak zoberieme rozdiel cifier týchto dvoch PINov, a sčítame ho s hodnotou v poli L. Takto dostaneme potenciálnu najlepšiu odpoveď pre tento PIN. Ak v ďalšom susednom PINe vypočítame vyššiu hodnotu, tak v poli L prepíšeme na tú vyššiu. Takúto kontrolu všetkých susedných PINov spravíme pre každý PIN vo vstupe a na konci algoritmu vypíšeme najvyššiu možnú odpoveď, ktorú sme kedy vypočítali. Časová zložitosť tohto algoritmu je $O(10np) = O(np)$, čo je v našom prípade lepšie ako $O(n^2)$, pretože u nás platí $1 \leq n \leq 6$ a $1 \leq p \leq 100,000$. Pamäťová zložitosť $O(10^n)$, kvôli dĺžke poľa L.

Listing programu (C++)

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

using namespace std;

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

    vector<int> pwrs(len+1);
    pwrs[0] = 1;
    for(int i=1;i<=len;++i) pwrs[i] = pwrs[i-1] * 10;

    vector<int> best(pwrs[len],-1);

    for(int x : a)
    {
        best[x] = max(best[x],0);
        for(int p=0;p<len;++p)
        {
            int dig = (x/pwrs[p])%10;
            int raw = x - dig*pwrs[p];
            for(int d=0;d<=9;++d)
            {
                int src = raw + d*pwrs[p];
                if(best[src] < 0) continue;
                best[x] = max(best[x],best[src] + abs(d-dig));
            }
        }
    }

    cout << *max_element(best.begin(),best.end()) << "\n";

}
```

Listing programu (Python)

```python
n = int(input())
p = int(input())
l = [-1 for i in range(10**n)]
result = 0
pows = [1]
for i in range(n + 1):
    pows.append(pows[-1] * 10)
for i in range(p):
    pin = input()
    bestanswer = 0
    for j in range(n):
        otherpin = list(pin)
        for x in range(10):
            otherpin[j] = str(x)
            newpin = int(''.join(otherpin))
            if l[newpin] > -1:
```

```
                answer = (abs(newpin - int(pin)))//pows[n-j-1] + l[newpin]
                if answer > bestanswer:
                    bestanswer = answer

    l[int(pin)] = bestanswer
    if bestanswer > result:
        result = bestanswer

print(result)
```

5. Iba aby hladný nebol

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

Na vstupe sme mali strom. Našou úlohou bolo spočítať, koľko existuje postupností vrcholov dĺžky k takých, že počas prechádzania postupnosti prejdeme po aspoň jednej označenej hrane. Takéto postupnosti budeme nazývať dobrými.

Hrubá sila

Riešenie hrubou silou je priamočiare, nie však jednoduché. Nezľaknite sa preto, ak ho budete čítať. Vzorové riešenie sa ukáže ako omnoho jednoduchšie.

Môžeme si postupne vygenerovať všetky možné postupnosti dĺžky k a pre každú z nich overiť, či je dobrá, alebo nie.

Na implementáciu takéhoto riešenia vieme použiť napríklad rekurzívny backtracking. Keď máme postupnosť vygenerovanú, musíme overiť, či je dobrá. To vieme napríklad tak, že pre každý vrchol v postupnosti pustíme BFS[1], či DFS[2]. V tomto prehľadávaní overíme, či na najkratšej ceste z aktuálneho vrchola postupnosti do nasledujúceho prejdeme po označenej hrane.

Inou, o niečo krajšou možnosťou by bolo predpočítať si pre každú dvojicu vrcholov, či sa na najkratšej ceste medzi nimi nachádza označená hrana. Potom by sme vedeli už počas rekurzie zisťovať, či sme cez nejakú označenú hranu prešli, alebo nie. Vyhli by sme sa tak overovaniu na konci rekurzie. Toto predpočítanie vieme spraviť drobnou úpravou algoritmu Floyda a Warshalla.

Možných postupností n vrcholov dĺžky k je n^k. Každú z nich samostane vygenerujeme a overíme, či je dobrá. V závislosti od implementácie nám to bude trvať niečo medzi $O(kn^{k+1})$ a $O(n^{k+3})$. Pamäť bude tiež v závislosti od implementácie $O(n)$ až $O(n^2)$. Závisí to od toho, či si pamätáme iba strom, alebo si niečo predpočítame pre každú dvojicu vrcholov.

Vzorové riešenie

Stačí použiť bežný trik. Ak nevieme jednoducho spočítať počet dobrých ciest, možno vieme ľahko zistiť počet zlých. Dobré potom zrátame tak, že od počtu všetkých odpočítame počet zlých.

Už vieme, že počet všetkých postupností je n^k.

Ako vyzerá zlá postupnosť? Neprejdeme počas nej po žiadnej označenej hrane. Skúsme teda zo stromu odstrániť všetky označené hrany. Týmto sa nám strom rozpadne na niekoľko komponentov. Každá zlá postupnosť potom musí byť **celá** v jednom komponente. Totiž, keby zahŕňala zlá postupnosť vrcholy z rôznych komponentov, museli by sme niekedy prejsť z jedného komponentu do druhého. Jedinou možnosťou, ako by sme to vedeli, je označená hrana. To je ale spor s predpokladom, že táto postupnosť je zlá.

Keď teda chceme zistiť počet zlých postupností, stačí nám ich vypočítať pre každý komponent zvlášť. Hodnoty pre jednotlivé komponenty potom iba sčítame.

Koľko zlých postupností dĺžky k je v komponente veľkosti x? To už predsa vieme, je to niečo podobné ako počet postupností v celom strome, akurát teraz nemá veľkosť n, ale x. Bude to teda x^k.

Nech sa nám odobratím označených hrán strom rozpadol na c komponentov. Ich veľkosti sú $x_1, \ldots, x_c$. Potom počet **dobrých** postupností je $n^k - \sum_{i=1}^{c} x_i^k$.

Na zrátanie veľkostí komponentov vieme použiť jedno DFS[3].

Takéto riešenie by malo získať 6 bodov. Na plný počet musíme naviac použiť umocňovanie v logaritmickom čase. O tomto jednoduchom triku si môžete viac prečítať tu[4].

[1] https://www.ksp.sk/kucharka/bfs/
[2] https://www.ksp.sk/kucharka/dfs/
[3] https://www.ksp.sk/kucharka/dfs/
[4] https://www.ksp.sk/kucharka/modularna_aritmetika/#wiki-toc-rychle-umocnovanie

DFS potrebuje $O(n)$ času. Následne veľkosť každého komponentu umocníme na k-tu. Komponentov je najviac n. Takéto riešenie teda bude mať časovú zložitosť $O(n \log k)$.

Pamätať si potrebujeme celý strom. Pamäťová zložitosť je preto $O(n)$.

Listing programu (C++)

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

using namespace std;

typedef long long ll;

ll n, k, a, b, c;
vector<vector<ll>> G;
vector<bool> vis;

const ll MOD = 1000000007;

// Na umocnovanie si spravime vlastnu funkciu, lebo to potrebujeme modulovat
ll expo(ll a, ll x) {
    if(x == 0LL)
        return 1LL;
    if(x == 1LL)
        return a;
    ll res = expo(a, x / 2);
    res = (res * res) % MOD;
    if(x & 1LL)
        res = (res * a) % MOD;
    return res;
}

ll dfs(ll u) {
    int comp_sz = 1;
    vis[u] = true;
    for(int i = 0; i < G[u].size(); i++) {
        ll v = G[u][i];
        if(!vis[v])
            comp_sz += dfs(v);
    }
    return comp_sz;
}

int main() {
    cin >> n >> k;
    G.resize(n);
    vis.assign(n, false);
    for(int i = 0; i < n - 1; i++) {
        cin >> a >> b >> c; a--; b--;
        // Hrany s exkrementom rovno odignorujeme
        if(c == 0) {
            G[a].push_back(b);
            G[b].push_back(a);
        }
    }
    ll bad = 0;
    ll all = expo(n, k);
    for(int i = 0; i < n; i++)
        if(!vis[i])
```

```
            bad += expo(dfs(i), k);
    // Ak nechapete, preco takto modulujeme, pozrite si, ako funguje % v C++.
    cout << (((all - bad) % MOD) + MOD) % MOD << '\n';


    return 0;
}
```

Listing programu (Python)

```
import sys


sys.setrecursionlimit(200000)

MOD = 1000000007


def expo(a, x):
    if x == 0:
        return 1
    if x == 1:
        return a
    res = expo(a, x // 2)
    res = (res * res) % MOD
    if x & 1:
        res = (res * a) % MOD
    return res


def dfs(u, vis, G):
    comp_sz = 1
    vis[u] = True
    for v in G[u]:
        if not vis[v]:
            comp_sz += dfs(v, vis, G)
    return comp_sz


n, k = [int(x) for x in input().split()]
vis = [False] * n
G = [[] for _ in range(n)]
for i in range(n - 1):
    a, b, c = [int(x) for x in input().split()]
    a -= 1
    b -= 1
    if c == 0:
        G[a].append(b)
        G[b].append(a)

bad = 0
all_sequences = expo(n, k)
for i in range(n):
    if not vis[i]:
        bad += expo(dfs(i, vis, G), k)

print(((all_sequences - bad) % MOD + MOD) % MOD)
```

fezjo

6. Váľanie hypergúľ

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

Samozrejme pre nepárne časy nikdy neexistuje žiadna cesta (trojitý zápor) a teda otázky s nepárnymi časmi už ďalej nebudeme rozoberať.

Prvé dve sady

Na prejdenie prvej sady nám stačí pre každú otázku simulovať pohyb larvy backtrackingom. Nech D je počet dimenzií a C je počet sekúnd pohybu larvy, teda počet krokov čo má spraviť. Na začiatku je pozícia larvy zoznam D núl. V každom kroku rekurzie sa skúsime pohnúť pre každú dimenziu do každého z dvoch smerov. V rekurzii hĺbky C sa pozrieme, či je pozícia znova D núl a ak áno, výsledok zvýšime o jedna. Pri backtrackingu sa veľmi oplatí prerušiť rekurziu ak vieme, že aktuálny stav určite nevedie k výsledku. V tejto úlohe je to napríklad ak je vzdialenosť od začiatku väčšia ako čas čo nám zostáva. Časová zložitosť pre riešenie bez rušenia rekurzie je $O(Q \cdot (2D)^C)$.

Riešenie druhej sady je podobné, ale využije šikovnejšiu reprezentáciu pozície larvy. Vo výsledku nás nezaujíma, či sa larva najprv pohla v smere prvej či druhej dimenzie. Pozíciu vieme reprezentovať iba ako pole P, kde na i-tom indexe bude počet dimenzií, pre ktoré sme vo vzdialenosti i. Toto nám samo o sebe veľmi nepomáha – musíme ešte zmeniť ako sa vnárame do rekurzie. Namiesto toho aby sme si vybrali jednu dimenziu v ktorej sa pohneme, vyberieme si skupinu dimenzií s rovnakou vzdialenosťou (teda jeden index v P). Môžeme si všimnúť, že nech sa pohneme v ľubovoľnej z nich, následná reprezentácia pozície larvy bude rovnaká. Teda vieme vykonať vnáranie sa do rekurzie iba raz, a výsledok vynásobiť počtom dimenzií v tejto skupine. Horný odhad časovej zložitosti pre riešenie bez rušenia rekurzie je $O(Q \cdot C^C)$, keďže teraz v rekurzií skúšame 2 smery pre $C/2$ skupín (vzdialenosť v ľubovoľnej dimenzií nemôže byť viac, inak by sme sa nestihli vrátiť). Môžeme si všimnúť, že ide naozaj iba o horný odhad, keďže $100 \cdot 15^{15}$ by určite neprešlo v časovom limite.

Obe riešenia predpokladali, že si pri rekurzií budeme stavové pole posielať ako referenciu, lebo to by sme vždy mali robiť. V tom prípade budú pamäťové zložitosti iba lineárne závislé od veľkosti týchto poli, teda $O(D)$ pre prvú a $O(C)$ pre druhú sadu.

Listing programu (Python)

```
def rekurzia(pocet, zostava):
    # ak je nasa vzdialenost od zaciatku vacsia ako cas
    # co nam zostava tak mozeme ukoncit rekurziu
    if sum(map(abs, pocet)) > zostava:
        return 0

    if zostava == 0:
        return 0 if any(pocet) else 1

    res = 0
    for dimenzia in range(len(pocet)):
        for smer in (-1, 1):
            pocet[dimenzia] += smer
            res += rekurzia(pocet, zostava-1)
            pocet[dimenzia] -= smer

    return res

G = int(input())
for i in range(G):
    d, c = map(int, input().split())
    if c % 2:
        print(0)
        continue
    pocet = [0] * d
    print(rekurzia(pocet, c))
```

Posledná sada

Pozrime sa na limity vstupov. G je veľké. Nie je dobré počítať odpoveď pre každú otázku nanovo. Nerobme to. Niečo si predpočítajme. Čo ak máme odpoveď pre každý čas menší ako C pre každú dimenziu menšiu ako D? Vieme z týchto informácií niečo vypočítať? Možno $C+2$ pre každé D, možno $D+1$ pre každé C? Ukážeme si to druhé z nich.

Najprv si ale predstavme ako vyzerá nejaká valídna trasa: Ak sa larva pohne v určitej dimenzií niektorým smerom, bude sa niekedy neskôr musieť pohnúť opačným. Teda trasu dĺžky c v d dimenziách si vieme predstaviť ako postupnosť c Jahodových a Karamelových čísel od 1 po d (kde Jahodové čísla reprezentujú opačný pohyb voči Karamelovým), ktoré vieme rozdeliť do Jahodovo-Karamelových dvojíc, kde čísla v rámci dvojice sú si rovné.

Naspäť k počítaniu. Majme výsledok $V(c, d)$ vypočítaný pre každú dvojicu (c, d), kde $c \leq C$ je dĺžka trasy a $d \leq D$ je počet dimenzií ktoré môžeme na danej trase použiť. Potom vieme vypočítať aj $V(c, D+1)$, pre ľubovoľné $c \leq C$. Výsledok $V(c, D+1)$ je počet postupnosti Jahodových a Karamelových čísel od 1 po $D+1$ s hore uvedenými vlastnosťami. Každú takúto postupnosť vieme vytvoriť z postupnosti Jahodových a Kararamelových čísel od 1 po D, do ktorej pridáme niekoľko Jahodových a Karamelových čísel s hodnotou $D+1$. To koľko čísel $D+1$ pridáme nám určuje, akú dĺžku mala postupnosť pred tým ako sme ich pridali. Teda ak pridáme x čísel, pridávame ich do postupnosti s parametrami $(c-x, D)$, ktorých vieme že je $V(c-x, D)$.

Potrebujeme ešte zistiť, koľkými spôsobmi vieme týchto x čísel do postupnosti pridať. Máme x čísel, všetky sú od seba na nerozoznanie. Potrebujeme ich ochutiť tak, aby polovica z nich bola Jahodová a polovica Karamelová (aby sme vo výsledku skončili v dimenzií $D+1$ na pozícii 0). Teda potrebujeme vybrať $x/2$ prvkov z x, čo vieme spraviť $\binom{x}{x/2}$ spôsobmi. Teraz máme postupnosť x ochutených čísel s hodnotou $D+1$ a postupnosť $c-x$ ochutených čísel s hodnotami od 1 po D a chceme zistiť, koľkými spôsobmi ich vieme skombinovať do jednej. Výsledná postupnosť bude mať c prvkov a ak vyberieme na ktorých pozíciách budú čísla s hodnotou $D+1$, jednoznačne to určí pozíciu zvyšných $c-x$ čísel. Vidíme teda, že to vieme spraviť $\binom{c}{x}$ spôsobmi.

Keďže počet postupností s parametrami $(c-x, D)$, počet ochutení a počet premiešaní daných dvoch postupností sú nezávislé operácie, *počet postupností s parametrami* $(c, D+1)$ *v ktorých má* x *čísel hodnotu* $D+1$ bude súčinom týchto troch čísel. Nakoniec počet postupností s parametrami $(c, D+1)$ bez obmedzenia na x bude súčet výsledkov pre každé x od 0 po c vrátane. Výsledok teda vieme zapísať ako: $V(c, D+1) = \sum_{x=0}^{c} V(c-x, D) \cdot \binom{x}{x/2} \cdot \binom{c}{x}$

Hodnoty $V(c, 0)$ sú triviálne (pre $c = 0$ je to 1, pre ostatné 0). Postupne budeme zvyšovať d pre ktoré vždy pre všetky c vypočítame $V(c, d)$. Toto spravíme pre všetky c a d potrebné na zodpovedanie všetkých otázok na vstupe.

Všetko čo sme tu vymysleli by však bolo celkom premárnené ak by sme taktiež nevedeli rýchlo počítať $\binom{y}{x}$. To vieme robiť veľa rôznymi spôsobmi – najjednoduchším je predpočítanie si Pascalovho trojuholníku, ale dá sa aj predpočítanie faktoriálov a ich inverzov pre modulo alebo vypočítanie podielu ako rozdiel množín faktorizácie faktoriálov.

Časová zložitosť tohoto riešenia je $O(Q+C^2 \cdot D)$, keďže predpočítavame tabuľku s rozmermi $C \times D$ s časovou zložitosťou $O(C)$ pre každý prvok. Pamäťová zložitosť bude $O(C \cdot D)$, alebo $O(Q + C \cdot D)$ ak sa rozhodneme najprv nájsť maximálne hodnoty C a D na vstupe a nemuseli tak počítať prebytočné masívnu tabuľku.

Listing programu (C++)

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

using ll = long long;

int main() {
    const ll MOD = 1e9 + 7;

    ll G;
    cin >> G;

    vector<pair<ll, ll>> Gulky(G);
    ll maxD=0, maxC=0;
    for(int i=0; i<G; ++i) {
        cin >> Gulky[i].first >> Gulky[i].second;
        maxD = max(maxD, Gulky[i].first);
```

```
        maxC = max(maxC, Gulky[i].second);

    }
    ++maxD, ++maxC;

    // predpocitanie kombinacnych cisel pomocou Pascalovho trojuholnika
    vector<vector<ll>> nCr(maxC, vector<ll>(maxC, 0));
    nCr[0][0] = 1;
    for(int n=1; n<maxC; ++n) {
        nCr[n][0] = nCr[n][n] = 1;
        for(int r=1; r<n; ++r) {
            nCr[n][r] = (nCr[n-1][r-1] + nCr[n-1][r]) % MOD;
        }
    }

    vector<vector<ll>> RES(maxD, vector<ll>(maxC, 0));
    for(int d=0; d<maxD; ++d) // vieme ze odpoved pre c=0 je vzdy 1
        RES[d][0] = 1;
    for(int d=1; d<maxD; ++d) {
        for(int c=2; c<maxC; c+=2) {
            for(int pridame=0; pridame<=c; pridame+=2) {
                ll pocet_v_menej_dimenziach = RES[d-1][c-pridame];
                ll pozicie_pridanych = nCr[c][pridame];
                ll ochutenie_pridanych = nCr[pridame][pridame/2];
                ll sucin = ((pocet_v_menej_dimenziach * pozicie_pridanych) % MOD
                            * ochutenie_pridanych) % MOD;
                RES[d][c] = (RES[d][c] + sucin) % MOD;
            }
        }
    }

    for(pair<ll, ll> p: Gulky)
        cout << RES[p.first][p.second] << '\n';
}
```

Listing programu (Python)

```
MOD = 10**9 + 7

G = int(input())
Gulky = [list(map(int, input().split())) for i in range(G)]
maxD, maxC = [max(map(lambda x: x[i], Gulky))+1 for i in range(2)]

nCr = [[0 for r in range(maxC)] for n in range(maxC)]
nCr[0][0] = 1
for n in range(1, maxC):
    nCr[n][0] = nCr[n][n] = 1
    for r in range(1, n):
        nCr[n][r] = (nCr[n-1][r-1] + nCr[n-1][r]) % MOD

RES = [[0 for i in range(maxC)] for j in range(maxD)]
for d in range(maxD):
    RES[d][0] = 1
for d in range(1, maxD):
    for c in range(2, maxC, 2):
        for pridame in range(0, c+1, 2):
            pocet_v_menej_dimenziach = RES[d-1][c-pridame]
            pozicie_pridanych = nCr[c][pridame]
```

```
            ochutenie_pridanych = nCr[pridame][pridame//2]
            RES[d][c] += pocet_v_menej_dimenziach * pozicie_pridanych *
                ↪ ochutenie_pridanych
        RES[d][c] %= MOD

for g in Gulky:
    print(RES[g[0]][g[1]])
```

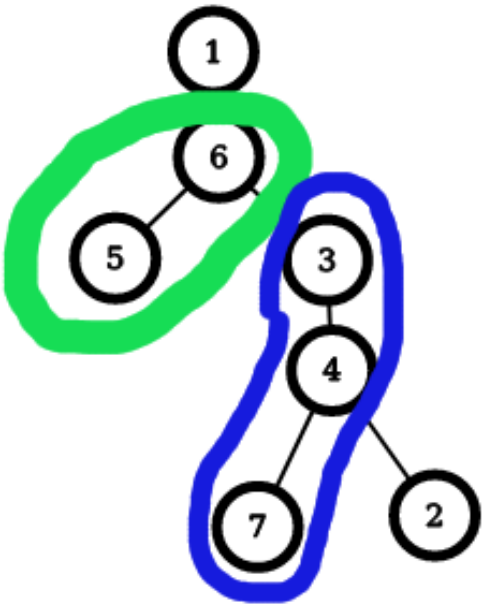
paulinia

7. Áááá, záplavy

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

Táto úloha má dve časti. V prvej potrebujeme položiť strom (sieť hrobiek posvätných hovniválov) do roviny ($n \times n$ mriežky), tak aby sme následne v druhej časti vedeli pre ľubovoľný pár vrcholov vybrať dva neprekrývajúce sa obdĺžniky, ktoré pokrývajú presne vrcholy na ceste medzi nimi.

Predstavme si zakorenený strom. Vtedy si každú cestu vieme rozdeliť na dve časti, v ktorých ideme len hore:



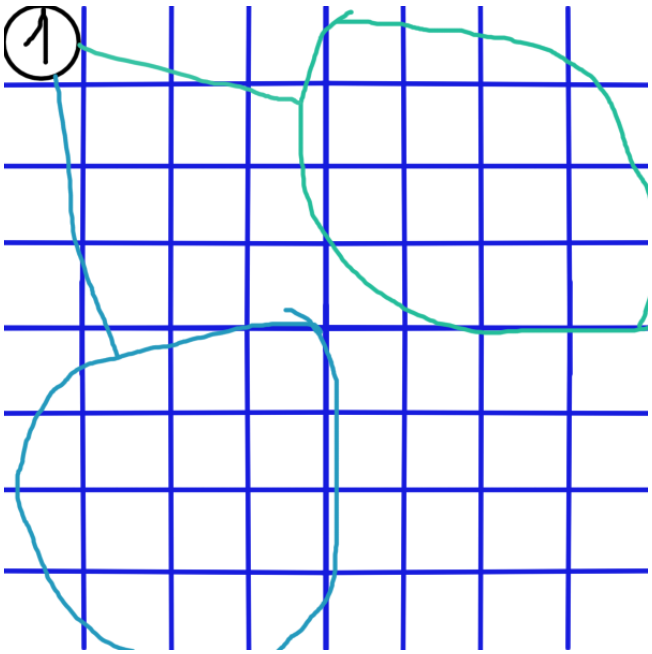
Nápad, na ktorom vzorák postavíme, je nasledovný: zakoreníme strom a posadíme ho do roviny tak, aby sme každú cestu dohora vedeli položiť do obdĺžnika.

Následne, aby sme pre pár vrcholov našli (najviac) dva neprekrývajúce sa obdĺžniky, najskôr nájdeme ich najbližšieho spoločného predka. Takto cestu rozdelíme na dve cesty dohora a tie vieme pokryť.

Položenie stromu do roviny

Binárny strom

Najskôr si zobme prípad binárneho stromu. Položme koreň stromu do ľavého horného rohu.



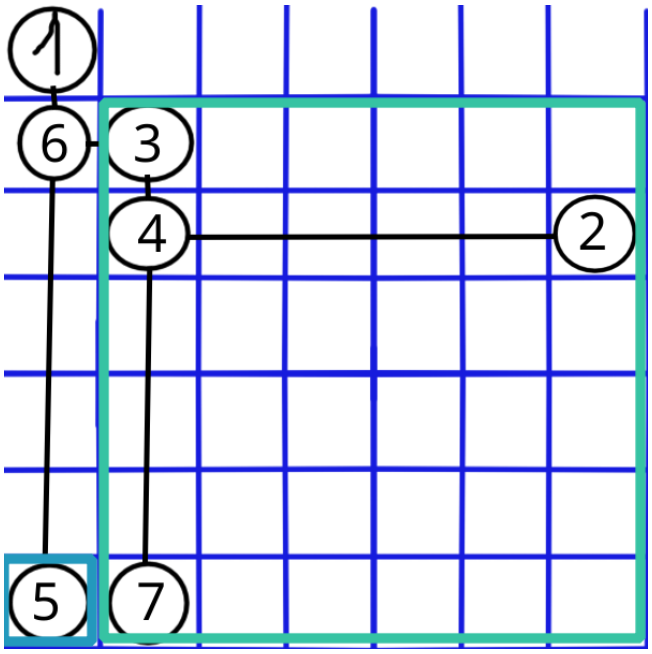
Koreň má najviac dvoch synov. Zjednodušene, prvý podstrom dáme do ľavého dolného, a druhý do pravého horného rohu, ako na obrázku.

Následne všetky obdĺžniky z prvého podstromu nepretínajú druhý podstrom a naopak.

Myšlienka je, položiť podstromy do ich podobdĺžnikov rekurzívne.

Začnime technickými detailami: chceme aby odbdĺžniky na podstromy nezdieľali žiadne x- any y- súradnice. To vieme dosiahnuť napríklad tak, že zistíme počet vrcholov v podstrome číslo jedna. Určíme stromu štvorec veľkosti tohto podstromu, končiaci v ľavom dolnom rohu (ako na obrázku). Potom druhému podstromu tiež určíme štvorec zodpovedajúci jeho veľkosti - "ukotvený" v pravom hornom rohu. Keďže súčet veľkostí podstromov je $n-1$, použité súradnice sa nebudú prekrývať. V prípade, že vrchol má len jedného syna, podstrom položíme do štvorca so stranou $n-1$.

Následne, zavoláme rekurzívnu funkciu na menšie štvorce a položíme do nich podstrom. Pre ilustráciu, pre strom v prvom obrázku dostaneme nasledovný obrázok:



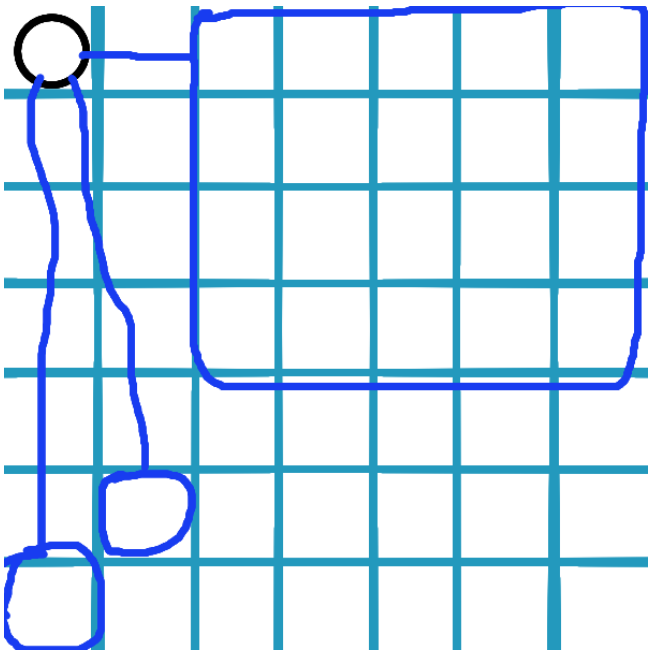
Pri takomto uložení stromu do roviny platí nasledovné. Pre každý vrchol v, všetky vrcholy od neho dohora a doľava sú jeho predkovia (rozmyslite si to). Každý syn je buď viac doprava alebo nižšie ako jeho rodič, takže pre cestu z v do predka p nám vždy stačí vziať obdĺžnik s pravým dolným rohom vo v a ľavým horným rohom v p.

Takéto uloženie stromu do roviny vieme jednoducho implementovať pomocou dvoch DFS (jedno na zistenie

veľkostí podstromov a druhé na samotné uloženie stromu) v $O(n)$ čase aj pamäti.

Všeobecný prípad

Čo v prípade, že se strom nie je binárny? Použijeme podobnú techniku ako pri binárnych stromoch. Avšak, namiesto rozdelenia štvorca na dva podštvorce ho rozdelíme na toľko podštvorcov, koľko má koreň synov, podľa diagonály (ak by sme strom z príkladu hore zakorenili v 6, potom dostaneme rozdelenie ako na ďalšom obrázku).



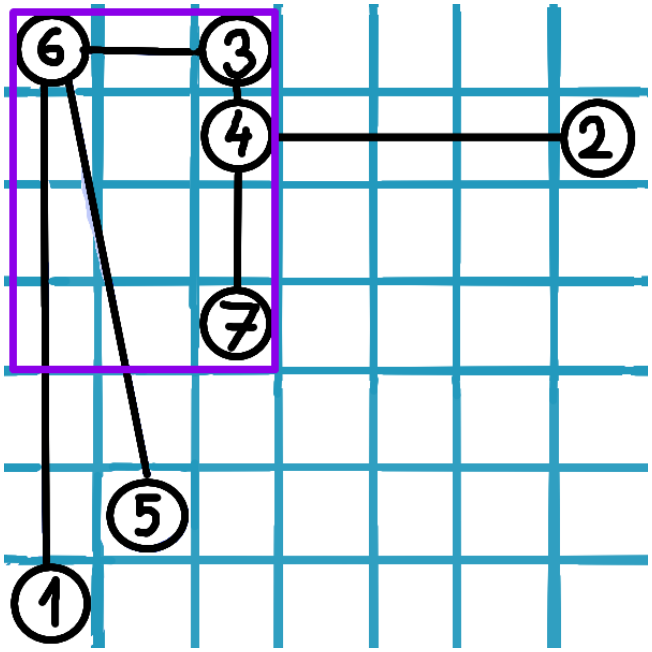
Podštvorce sa nám do štvorca zmenstia, keďže suma veľkostí podstromov je $n - 1$. Argument, prečo toto položenie do roviny funguje, je veľmi podobný ako pri binárnom strome: zoberme si ľubovoľný vrchol v: všetky vrcholy naľavo hore od v musia byť jeho predkovia (súrodenci v a iná rodina je vždy buď napravo alebo dole). Keďže sú položení vo vhodnom poradí (rodič je vždy naľavo hore od detí), každý obdĺžnik s pravým dolným rohom vo vrchole v obsahuje práve nejakú cestu hore. A naopak, každá cesta hore je položená v obdĺžniku s pravým dolným rohom vo v a ľavým horným rohom v predkovi.

Toto uloženie stromu do roviny je tiež v lineárnom čase a pamäti.

Ako zodpovedať otázky

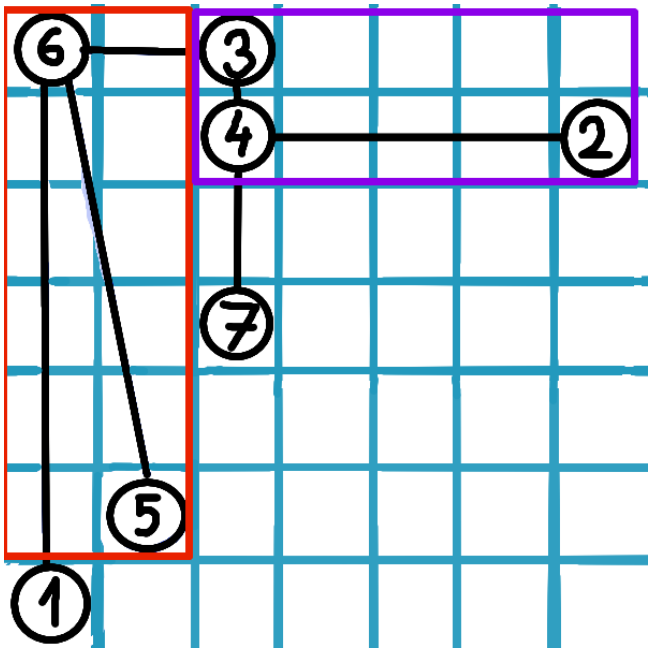
Keď už sme strom položili do roviny, prídeme k druhej časti problému. Majme daný pár vrcholov, v a w a chceme nájsť dva neprekrývajúce sa obdĺžniky, tak aby pokrývali práve cestu medzi v a w. Začneme tým, že nájdeme najbližšieho spoločného predka (pozri nižšie). Máme dve možnosti: buď je najbližší spoločný predok jeden z vrcholov v, w (bez ujmy na všeobecnosti, nech je to w). Potom vieme položiť celú cestu do jedného obdĺžnika, s pravým dolným rohom vo v a ľavým horným rohom v w.

Na obrázku je príklad pre $w = 6$, $v = 7$:



Druhý prípad nastáva, ak najbližší spoločný predok nie je ani v, ani w. Povedzme, že je to nejaký vrchol p. V tom prípade cestu rozdelíme na dve cesty hore: (napríklad) od v do p a od w do jedného zo synov v p. Nemôžeme položiť do obdĺžnikov cesty z v do p aj w do p, pretože potom by mali prekryv. Keď nájdeme vrchol, do ktorého chceme viesť cestu z w, potom už jednoducho získame dva obdĺžniky: jeden má rohy vo v a p a druhý v w a správnom synovi w.

Na obrázku je príklad pre $v = 5$ a $w = 2$:



Najbližší spoločný predok

Posledný problém je nájdenie najbližšieho spoločného predka. Najbližší spoločný predok v a w je najhlbší taký vrchol (teda najbližší k v a w), že jeho podstrom obsahuje oba v a w.

Naivné hľadanie najbližšieho spoločného predka vyzerá asi takto: (bez ujmy na všeobecnosti, nech v je hlbší vrchol ako w) z vrchola v ideme hore (po rodičoch) kým sa nedostaneme na rovnakú hĺbku, v akej je w. Ak sme takto vošli do vrchola w, potom w je najbližší spoločný predok. Ak nie, pokračujeme hore, tentoraz posunieme vždy v a w naraz, až kým sa nestretnú. Vrchol, v ktorom sa stretnú je najbližší spoločný predok. Toto má časovú zložitosť lineárnu v hĺbke stromu, čo môže byť lineárne v n, takže dostaneme čas $O(n)$ na query. Toto riešenie nám dá polovicu bodov.

Hľadať to vieme aj rýchlejšie, a to napríklad v čase $O(\log n)$ na query (pozri kuchárku[5]). V tomto čase vieme nájsť predka o k-generácií vyššie. Ak vieme hĺbku najbližšieho spoločného predka (označme ju ako h_p, a hĺbky v ktorej sú v a w postupne ako h_v a h_w), potom do jedného obĺžnika dáme cestu z v do predka o $h_v - h_p$ generácií vyššie od v (najbližšieho spoločného predka) a do druhého cestu z w do predka w o $h_w - h_p - 1$ vyššie (v prípade že najbližší spoločný predok nie je jeden z koncových vrcholov). Rohové body obdĺžnikov teda vieme nájsť v čase $O(\log n)$ na query, čo nám stačí na plný počet bodov.

Listing programu (C++)

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

using namespace std;

const int infinity = 2000000999 / 2;

vector<pair<int, int> > locs;

void set_tomb_location(int ID, int X, int Y) {
    locs[ID] = make_pair(X, Y);
}

vector<vector<int> > hrany;
vector<int> euler, parents, Z;

int dfs_sizes(int v, int f, int h, vector<int> &sizes) {
    Z[v] = euler.size();
    euler.push_back(h);
    parents[v] = f;
    sizes[v] = 1;
    if (hrany.size() != 0) for(int w : hrany[v]) {
        if(w != f) {
            sizes[v] += dfs_sizes(w, v, h + 1, sizes);
            euler.push_back(h);
        }
    }
    return sizes[v];
}

int dfs_build(int v, int f, int x, int y, vector<int> &X, vector<int> &Y, vector
    ↪ <int> &sizes) {
    X[v] = x, Y[v] = y;
    int fur = x;
    int sz = sizes[v] - 1;
    if (hrany.size() != 0) for(int w : hrany[v]) {
        if(w == f) continue;
        sz -= sizes[w];
        fur = dfs_build(w, v, fur + 1, y + sz, X, Y, sizes);
    }
    return fur;
}

struct rmq {
    int n;
    vector<vector<int> > R;

    rmq(int m, vector<int> &A) {
        n = m;
```

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

```
        R.push_back(A);
        int jump = 1, id = 0;
        while(jump < n) {
            R.push_back(vector<int>(n));
            for (int i = 1; i < n; i ++) {
                R[id + 1][i] = min(R[id][i], R[id][min(n - 1, i + jump)]);
            }
            jump *= 2;
            id ++;
        }
    }
    rmq() {}

    int query(int z, int k) {
        int id = log2(k - z);
        int jump = (1 << id);
        return min(R[id][z], R[id][k - jump]);
    }
};

rmq minis;
vector<vector<int> > minx, miny, maxx, maxy, lca;

void build_tombs(int n){
    vector<int> sizes(n, 0), X(n), Y(n);
    parents.resize(n);
    Z.resize(n);
    dfs_sizes(0, 0, 0, sizes);
    dfs_build(0, 0, 1, 1, X, Y, sizes);
    for(int i = 0; i < n; i ++) set_tomb_location(i, X[i], Y[i]);
    minis = rmq(euler.size(), euler);
    maxy.push_back(Y);
    minx.push_back(X);
    miny.push_back(Y);
    maxx.push_back(X);
    lca.push_back(parents);

    int jump = 1, id = 0;
    while(jump < n) {
        maxy.push_back(Y);
        minx.push_back(X);
        miny.push_back(Y);
        maxx.push_back(X);
        lca.push_back(parents);

        for(int i = 0; i < n; i ++) {
            lca[id + 1][i] = lca[id][lca[id][i]];
            maxy[id + 1][i] = max(maxy[id][i], maxy[id][lca[id][i]]);
            maxx[id + 1][i] = max(maxx[id][i], maxx[id][lca[id][i]]);
            miny[id + 1][i] = min(miny[id][i], miny[id][lca[id][i]]);
            minx[id + 1][i] = min(minx[id][i], minx[id][lca[id][i]]);
        }
        jump *= 2;
        id ++;
    }
}

void jump_to(int v, int len) {
```

```
    int mnx = infinity, mxx = 0, mny = infinity, mxy = 0;
    for(int i = 0; i < minx.size(); i ++) {
        if(len & (1 << i)) {
            mnx = min(minx[i][v], mnx);
            mny = min(miny[i][v], mny);
            mxx = max(maxx[i][v], mxx);
            mxy = max(maxy[i][v], mxy);
            v = lca[i][v];
        }
    }
    cout << mnx << " " << mny << " " << mxx << " " << mxy << "\n";
}

void query(int a, int b){
        int h = minis.query(min(Z[a], Z[b]), max(Z[a], Z[b]) + 1);
    int ha = euler[Z[a]], hb = euler[Z[b]];
    cout << (hb != h ? 2 : 1) << "\n";
    jump_to(a, ha - h + 1);
    if(hb != h) jump_to(b, hb - h);
    fflush(stdout);
}


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

    int N;
    cin >> N;
    locs.resize(N);
    hrany.resize(N);

    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);
    }

    build_tombs(N);
    for(int i = 0; i < N; i ++) {
        cout << locs[i].first << " " << locs[i].second << "\n";
    }
    fflush(stdout);

    while(true) {
        int a;
        cin >> a;
        if (a < 1) return 0;
        a --;
        int b;
        cin >> b;
        b --;
        query(a, b);
    }
```

```
}
```

Dano

8. Lajno nazmar nevyjde

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

Snáď ste sa len nenechali zmiasť číslom tejto úlohy. Riešenie je možno jednoduchšie, ako by ste čakali.

Dôležité pozorovanie

Pozrime sa, čo sa stane, ak sme na i-te poschodie pridali hovniválа. Nech na tom poschodí bolo doteraz a_i hovniválov. Stavba tohto poschodia by im zabrala $\frac{c_i}{a_i}$ času. Po pridaní jedného hovnivála im stavba zaberie $\frac{c_i}{a_i+1}$ času. Týmto pridaním sme teda ušetrili $\frac{c_i}{a_i} - \frac{c_i}{a_i+1} = \frac{c_i}{a_i(a_i+1)}$.

Môžeme teda rozmiestňovať chrobáky pažravo. Najskôr dáme po jednom na každé poschodie. Následne vždy dáme hovniválа na to poschodie, kde ušetrí najviac času. Toto vieme spraviť v čase $O(H \log n)$ napríklad pomocou maximovej haldy. V halde budú všetky poschodia a bude utriedená podľa času, ktorý ušetríme pridaním chrobáka na dané poschodie.

Zrýchlenie

Keďže chrobákov je naozaj veľa, potrebujeme niečo zrýchliť. Zaveďme si parameter x. Tento parameter budeme binárne vyhľadávať. Pre x budeme vedieť zrátať nasledovné. Koľko chrobákov vieme pridávať na poschodia, kým sa náš časový zisk (ten zlomok) stane menší ako x. Budeme chcieť nájsť také x, pre ktoré použijeme H hovniválov.

Pre poschodie i vieme zrátať počet hovniválov a_i na ňom ako $x = \frac{c_i}{a_i(a_i+1)}$. Toto vieme vypočítať kvadratickou rovnicou. Keď teda v binárnom vyhľadávaní kontrolujeme dané x, tak prejdeme všetky poschodia. Na každé dáme nanajvýš a_i hovniválov (tých z kvadratickej rovnice). Ostane nám teda rozmiestniť posledných nanajvýš n. Tých ale vieme doriešiť skôr spomínaným pažravým spôsobom pomocou haldy.

Časová zložitosť tohto riešenia je v závislosti od implementácie zhruba $O(n(\log C + \log H + \log n))$, kde C je maximum z c_i. Pamäťová zložitosť je $O(n)$.

Listing programu (C++)

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

const int MAXN = (int)1e5+5;

ll n, H, c[MAXN];

ll count(double x) {
    ll sum = 0;
    // Pouzijeme kvadraticku rovnicu pre zratanie c[i] / (a_i * (a_i + 1)) >= x
    for (int i = 0; i < n; i++)
        sum += ll((sqrt(1 + 4 * c[i] / x) - 1) / 2);
    return sum;
}

int main() {
    cin >> n >> H;
    H -= n;

    for (int i = 0; i < n; ++i)
        cin >> c[i];

    double lo = 0, hi = 1e18;
    for (int i = 0; i < 200; ++i) {
        double mid = (lo + hi) / 2;
```

```
        if (count(mid) >= H)
            lo = mid;
        else
            hi = mid;
    }

    double ans = 0;
    ll sum = 0;

    // Uz vieme zrat x z lo
    // Teraz by sme si mohli zratat pre kazde poschodie, kolko hovnivalov nan
       ↪ mozeme dat pre nase x
    // Ostalo by nam nutne nieco medzi 0 a n hovnivalmi
    // Tych by sme pazravo rozmiestnili na poschodia pomocou haldy.
    // Mame ale aj inu moznost.
    // Ked ak sme pouzili viac, alebo menej chrobakov a mame spravne x, tak
       ↪ staci vhodne upravit zratanu odpoved
    for (int i = 0; i < n; i++) {
        ll x = ll((sqrt(1 + 4 * c[i] / lo) - 1) / 2);
        ans += 1.0 * c[i] / (x + 1);
        sum += x;
    }

    // Od aktualnej odpovede odcitame / pricitame tolko x, kolkych chrobakov
       ↪ mame naviac / menej
    cout << setprecision(9) << fixed << (ans - (H - sum) * lo) << endl;
}
```