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

Vzorové riešenia 2. kola zimnej časti

pepe

1. Turisticky na oslavu

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

Myšlienka riešenia

Aby sme zistili, koľko vedúcich sa dostane na chatu, potrebujeme najskôr zistiť počet vedúcich, ktorí budú pri každej skale tvoriť ľudský rebrík. To docielime tak, že od výšky skaly odčítame výšku skoku vedúceho, čo nam dá počet, koľko vedúcich je potrebné obetovať na ľudský rebrík. Ak je počet záporné číslo, znamená to, že vedúci vie preskočit danú skalu, a teda nebolo treba obetovať žiadneho vedúceho.

Po zistení počtu vedúcich, ktorí tvoria ľudský rebrík, odčítame dané číslo od celkového počtu vedúcich. (Vedúci, ktorí tvoria ľudský rebrík už ďalej ísť nemôžu)

Túto úvahu aplikujeme pre každú skalu na vstupe.

Ukážeme si myšlienku na 2. príklade zo zadania:

vstup

```
3 5 5
4 7 3
```

výstup

```
3
```

Počet skál 3, výška skoku 5, počet vedúcich 5.

Skala výšky $4 \to 4 - 5 = -1$, zoberieme `max(-1, 0)`, teda $5 - 0 = 5$, všetci vedúci prejdu.
Skala výšky $7 \to 7 - 5 = 2$, zoberieme `max(2, 0)`, teda $5 - 2 = 3$, dvaja vedúci tvoria rebrík, zvyšní traja prejdu.
Skala výšky $3 \to 3 - 5 = -2$, zoberieme `max(-2, 0)`, teda $3 - 0 = 3$, traja vedúci sa dostali na chatu.

Optimálne riešenie

Optimálne riešenie číta výšky skál a hneď rozhoduje, či bude treba obetovať vedúcich. Ak áno, odčíta počet obetovaných vedúcich od celkového počtu vedúcich.

Stačí nám prejsť vstupom iba raz, a teda časová zložitosť riešenia je $O(n)$. Pri čítaní nám stačí si pamätať iba aktuálnu výšku skaly, a teda pamäťová zložitosť riešenia je $O(1)$.

Listing programu (Python)

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

for v in vysky:
    if skok < v:
        poc_v -= v-skok

print(max(0, poc_v))
```

Listing programu (C++)

```
#include <iostream>

auto main() -> int
{
    long long n = 0, v = 0, k = 0;
    std::cin >> n >> v >> k;
```

```
    for (auto i = 0, x = 0; i < n && v > 0; ++i) {
        std::cin >> x;
        v -= std::max(0ll, x - k);
    }

    std::cout << std::max(0ll, v) << '\n';
}
```

Adri a Paulinka

2. Organizácia Kapustnice

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

Ľubovoľné správne riešenie

Pozrime sa najskôr na prípad, že nám netreba minimalizovať počet pridaných tort a vedúcich, ale stačí nám ľubovoľné správne riešenie. Všimnime si, že nezáleží, ako ďaleko sa vedúci pohne, aby získal tortu, ale stačí, aby nejakú získal. Taktiež, všetky torty sú rovnaké, takže nezáleží na tom, ktorú tortu zje ktorý vedúci.

Jednoduché riešenie je napríklad nasledovné: ku každému vedúcemu dáme tortu a ku každej torte dáme vedúceho (bez ohľadu na to, či by tá torta mohla byť zjedená v originálnom rozostavení, resp. či by ten vedúci nejakú tortu mohol v originálnom rozostavení zjesť). Toto riešenie vieme naprogramovať v čase $O(n)$ s konštantnou pamäťou (stačí si nám pamätať, či je práve na rade vedúci alebo torta).

Všetky takéto riešenia budú mať tvar `VTVTVT...` – môžeme vidieť, že každý vedúci zje presne jednu tortu a každá torta bude zjedená.

Vzorové riešenie

Predchádzajúce riešenie je jednoduché, ale vždy pridáva n nových tort a vedúcich, a to aj vtedy, keď to netreba. Ako to zlepšiť?

Základná myšlienka je dať pred tortu vedúceho len vtedy, ak by táto torta ostala nezjedená, a dať vedúcemu tortu len vtedy, ak by žiadnu v originálnom rozostavení nedostal.

Skúsme nasimulovať, čo by sa stalo pri originálnom rozostavení vedúcich a tort. Predstavme si, že postupne prechádzame stôl zľava doprava a udržiavame si premenné dv – "doterajší počet vedúcich" a dt – "doterajší počet tort".

Ako prvé si všimnime, že ak $dv >= dt$, potom každá torta je zjedená. Hodnota $dv - dt$ udáva počet ešte nenajedených vedúcich (do danej pozície na stole) (vždy, keď príde torta, počet nenajedených vedúcich sa zníži, vždy, keď príde vedúci, počet nenajedených vedúcich sa zvýši).

Podobne, ak máme také rozostavenie tort a vedúcich, že každá torta bude zjedená, tak vždy bude platiť $dv >= dt$ (ak máme pred nejakou pozíciou viac tort ako vedúcich, všetky torty pochopiteľne nemôžu byť zjedené).

Poďme najskôr pridať vedúcich. Všimnime si, že ak pridáme v vedúcich na začiatok, potom sa počet nenajedených vedúcich (na každej pozícii) zvýši o v.

Počet nezjedených tort na nejakej pozícii vieme vyrátať ako $dt - dv$. Nech najväčšia hodnota, ktorú $dt - dv$ v nejakom bode nadobudne, je nt. To znamená, že musíme pridať určite aspoň nt vedúcich, aby nám nikdy neostali nezjedené torty. Vieme ich všetkých napríklad pridať na začiatok. Potom bude počet nezjedených tort všade zmenšený o nt a teda nikdy nám neostanú nezjedené torty. Ak by sme pridali menej než nt vedúcich, v nejakom bode by nám ostali nezjedené torty, preto toto je najlepšie, čo vieme spraviť.

Čo s tortami? Všimnime si, že ak nám nikdy neostanú nezjedené torty, hodnota $dv - dt$ po prejdení celého vstupu je presne počet vedúcich, ktorí nedostali tortu. Nazvime túto hodnotu nv. Tá nám vlastne hovorí, o koľko je pri stole viac vedúcich ako tort. Očividne musíme pridať aspoň nv tort – ak by sme pridali menej, potom by pri stole bolo menej tort ako vedúcich, takže určite nebudú všetci vedúci najedení. Taktiež si všimnime, že nám stačí pridať všetkých nv tort na koniec stola. Takto sa k nim všetci zostávajúci nenajedení vedúci určite dostanú.

Takže si to zhrňme: prejdeme postupne stôl a počítame si, koľko vedúcich a tort sme zatiaľ videli. Zapamätáme si, akú najväčší počet nezjedených toriet ($dt - dv$) sme videli. Ak je tento počet kladný, pridáme taký počet vedúcich na začiatok. Následne sa pozrieme, koľko nenajedených vedúcich ($dv - dt$) nám na konci ostalo, a pridáme k tomu počet pridaných vedúcich (nt). Toto číslo nám povie, koľko tort treba pridať na koniec. Časová aj pamäťová zložitosť je $O(n)$, keďže raz prejdeme vstupom, a potrebujeme si ho pamätať, aby sme mohli vypísať výstup.

Listing programu (Python)

```
#!/usr/bin/env python3
n=int(input())
s=input()
z=0
o=0
for x in s:
    if x=='V':
        o+=1
    elif o:
        o-=1
    else:
        z+=1
print('V'*z+s+'T'*o)
```

Listing programu (C++)

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

using namespace std;

#define FOR(i,n)        for(int i=0;i<(int)n;i++)
#define FOB(i,n)        for(int i=n;i>=1;i--)
#define MP(x,y) make_pair((x),(y))
#define ii pair<int, int>
#define lli long long int
#define ld long double
#define ulli unsigned long long int
#define lili pair<lli, lli>
#ifdef EBUG
#define DBG     if(1)
#else
#define DBG     if(0)
#endif
#define SIZE(x) int(x.size())
const int infinity = 2000000999 / 2;
const long long int inff = 4000000000000000999;

typedef complex<long double> point;

template<class T>
T get() {
    T a;
    cin >> a;
    return a;
}

template <class T, class U>
ostream& operator<<(ostream& out, const pair<T, U> &par) {
    out << "[" << par.first << ";" << par.second << "]";
    return out;
}

template <class T>
ostream& operator<<(ostream& out, const set<T> &cont) {
    out << "{";
    for (const auto &x:cont) out << x << ", ";
```

```
    out << "}";
    return out;
}

template <class T, class U>
ostream& operator<<(ostream& out, const map<T,U> &cont) {
    out << "{";
    for (const auto &x:cont) out << x << ", ";

    out << "}"; return out;
}

template <class T>
ostream& operator<<(ostream& out, const vector<T>& v) {
  FOR(i, v.size()){
    if(i) out << " ";
    out << v[i];
  }
  out << endl;
  return out;
}

bool ccw(point p, point a, point b) {
  if((conj(a - p) * (b - p)).imag() <= 0) return false;
  else return true;
}

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

    int n = get<int>();
    string s = get<string>();

    int pridatL = 0;
    int act = 0;

    FOR(i, n) {
        if (s[i] == 'T') {
            if (act) act --;
            else pridatL ++;
        }
        else act ++;
    }

    FOR(i, pridatL) cout << "V";
    cout << s;
    FOR(i, act) cout << "T";
    cout << endl;
}
```

3. Raz počuť (ne)stačí

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

Pomalé riešenie

Každá informácia mala práve tri možnosti, čo sa s ňou mohlo stať (byť zabudnutá, zapamätaná zľava,

zapamätaná sprava). Túto znalosť môžeme využiť na to, aby sme vygenerovali (napríklad rekurzívne) všetky možné postupnosti patriace novým vedúcim. Potom všetky prejdeme a zistíme, či sa nejaká zhoduje so vstupnou. Dokonca si ich ani nemusíme pamätať, stačí každú postupnosť porovnať rovno potom, čo ju vytvoríme. Tento prístup určite funguje, ale jeho časová zložitosť je bolestivá: potrebujeme vygenerovať 3^n postupností dĺžky n a každú porovnať so vstupnou. Takéto riešenie teda nebude fungovať pre n rádovo väčšie, než napríklad 10.

Vzorové riešenie

Alebo sa najskôr zamyslime nad tým, ako by mohla postupnosť zabúdaných informácií vyzerať v nejakom všeobecnom prípade. Ak je náš človek naozaj nový vedúci, potom niekoľko informácií zabudol instantne (I), niekoľko si zapamätal zľava (L), a niekoľko sprava (P), pričom niekoľko môže byť aj 0. Vieme si to predstaviť aj tak, že existujú tri nezávislé priehradky, do ktorých informácie postupne ukladal, a na konci ich jednoducho prilepil za seba, v poradí I, prevrátené L (pretože informácie z L vyberáme v opačnom poradí, ako sme ich tam) dávali, a R.

Každá priehradka má prvky v stúpajúcom poradí - keďže informácie v stúpajúcom poradí prichádzali, nemôže to ani byť inak. Takže keď boli prilepené za seba, vznikla postupnosť čísel, ktorá najskôr niekoľkokrát stúpala (I), potom klesala (prevrátené L) a nakoniec znova stúpala (R). Medzi stúpajúcou a klesajúcou postupnosťou je buď nárast alebo pokles, čo vieme interpretovať ako súčasť ľubovoľnej z nich.

Pre každú zadanú postupnosť nám teda stačí overiť, či má takýto formát. Na to potrebujeme zistiť počet zmien jej smeru. Postupnosť budeme prechádzať postupne, a každý prvok porovnáme s predošlým. Začneme so stúpajúcim smerom postupnosti, (prvá priehradka je I) a vždy, keď sa smer zmení, prirátame si k počtu zmien 1. Nakoniec overíme, či je počet zmien menší alebo rovný dvom - to zodpovedá prechodom medzi I a L a medzi L a R. Ak nastalo viac zmien, určite nešlo o nového vedúceho.

Časová a pamäťová zložitosť

Každú postupnosť prejdeme práve raz, a pre každý jej prvok vykonáme konštantný počet operácií, takže časová zložitosť pre jedného človeka je $O(n)$. Pre t ľudí potom $O(\sum_t n)$. Pamäťová zložitosť je dokonca konštantná, pretože nám stačí pamätať si počet zmien a aktuálny smer postupnosti.

Listing programu (Python)

```
def clovek():
    n = int(input())
    post = list(map(int, input().split()))

    if n < 4:
        return True

    zmen = 0
    stup = True
    for i in range(1, n):
        akt = post[i] > post[i - 1]
        if akt != stup:
            stup = not stup
            zmen += 1
            if zmen > 2:
                return False
    return True

for i in range(int(input())):
    print("Novy veduci" if clovek() else "Neveduci")
```

Listing programu (C++)

```
#include <iostream>

using namespace::std;
```

```
int main(){
    int t, n, x, lastx, zmen;
    bool stup;

    cin >> t;
    while (t--){
        stup = true;
        zmen = 0;
        cin >> n;
        cin >> lastx;
        while (--n){
            cin >> x;
            if (stup != (x > lastx)){
                stup = !stup;
                zmen++;
                if (zmen == 3){
                    cout << "Neveduci" << endl;
                    zmen++;
                }
            }
            lastx = x;
        }
        if (zmen <= 2) cout << "Novy veduci" << endl;
    }
}
```

Viki

4. Týrajú ma hladom

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

Úlohu zo zadania môžeme preformulovať tak, že chceme nájsť najdlhšiu súvislú podpostupnosť vstupného slova skladajúcu sa najviac z $k+1$ rôznych písmen.

Totiž ak k písmen prepíšeme na to $k+1$vé, budeme mať úsek písmen, ktoré vyzerajú rovnako.

Priamočiare riešenie

Ak by sme vedeli kde hladaný interval začína, vieme pomerne jednoducho zistiť aký je dlhý. Skúsime preto od každého písmena na vstupe spočítať aký dlhý interval by na ňom mohol začínať a riešením bude ich maximum.

Skúsime to teda naprogramovať.

Na pamätanie si, či sme dané písmenko už zarátali nám pomôže `set`, do ktorého budeme pridávať každé písmenko. Set je štruktúra ktorá ukladá iba rôzne prvky, preto sa stačí po každom vložení písmena pozrieť na veľkosť setu. Ak veľkosť setu presiahne $k+1$, vieme, že už v ňom máme viac ako $k+1$ písmeniek, teda posledné písmenko už do aktuálnej postupnosti patriť nemôže.

Toto riešenie má časovú zložitosť $O(n^2)$[1] a pamäťovu $O(n)$. Mohli ste zaň získať 4 body.

Ide to ale aj lepšie

Keď sme narazili na $k+2$ písmeno v predošlom riešení, museli sme celý interval zahodiť a začať od začiatku. Po krátkom zamyslení však zistíme, že ten nasledujúci interval sa nebude veľmi líšiť. Môžu nastať dve situácie. Buď sa prvé písmenko v intervale ešte niekde nachádza a teda nasledujúci interval bude rovnaký, alebo odstránením prvého písmena nám klesne aj počet rôznych písmen v sete a môžeme interval predĺžiť.

Aby sme mohli takto postupovať, musíme si namiesto setu ale pamätať aj počet výskytov v intervale pre každé písmeno. Rôznych znakov môže byť najviac 62 (čísla, veľká a malá anglická abeceda), takže najlepšie bude použiť statické pole. Ak ho spravíme o niečo dlhšie, môžeme do neho indexovať priamo ASCII hodnotou znaku. Namiesto veľkosti setu si potom potrebujeme v nejakej premennej pamätať počet rôznych písmen v intervale.

Vstupné slovo budeme teda pechádzať z ľava do prava a to nasledovne:

[1] V závislosti od programovacieho jazyka, ak použijeme implementáciu založenú na hashovaní je to $O(n^2)$ ale ak máme set implementovaný pomocou binárneho vyhľadávacieho stromu je to $O(n^2 \log p)$ kde p je počet rôznych písmen.

Kým je počet menší ako $k+2$, budeme sa koncom hýbať doprava. Keď sa pohneme, prečítame písmenko a započítame jeho výskyt. Ak je prvý, počet rôznych písmen zväčšíme o jeden.

Ak sa nám ale stalo, že počet je väčší ako $k+1$, máme maximálny interval a môžeme posunúť začiatok. Písmeno ktoré na začiatku intervalu strácame odrátame z výskytov a skontrolujeme či bolo posledné. Ak sme zistili, že to bolo posledné písmeno toho druhu v intervale, znížime počet rôznych písmen o jeden môžeme zase posunúť koniec. Inak sa počet rôznych písmen nezmenil a koniec nevieme posunúť, takže pokračueme v posúvaní začiatku. Medzičasom si budeme pamätať maximálnu dĺžku intervalu. Takémuto postupu sa niekedy hovorí aj *dvaja bežci*.

Časová a pamäťová zložitosť

V tomto riešení máme dva indexy do pola ktoré postupne posúvame od začiatku po koniec. Každý teda nezávysle na druhom spraví $O(n)$ krokov. Okrem toho používame statické pole na počet výskytov písmen, do ktorého však indexujeme iba keď posúvame jeden z indexov, teda časovú zložitosť to nemení a ostáva $O(n)$.

Skúseného riešiteľa neprekvapí, že pamäťová zložitosť je tiež $O(n)$[2] nakoľko viac pamäte v čase $O(n)$ nestihneme ani naplniť.

Listing programu (Python)

```
n, k = map(int, input().split())
w = input()
u = 0
s = 0
e = 0
ans = 1
ans_start = 0
new_ans = 0
c = [0]*255
k+=1

for e in range(len(w)):
        if c[ord(w[e])] == 0:
                u+=1
        c[ord(w[e])]+=1

        while u > k:
                c[ord(w[s])]-=1
                if c[ord(w[s])] == 0:
                        u-=1
                s+=1

        new_ans = e-s+1;

        if new_ans > ans:
                ans = new_ans;

print(ans)
```

Listing programu (C++)

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

string w;
int k, n, u = 0, s = 0, e = 0, ans = 1, ans_start = 0, new_ans = 0;
```

[2] Aby sme boli presní, je to $O(n+p)$ kde p je počet možných písmen, no ak zadanie hovorí že pre dosť veľké n je $p < n$ stačí písať $O(n)$.

```
vector<int> c(255);

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

        cin >> n >> k >> w;
        k++;

        for (unsigned int e = 0; e < n; e++) {
                if (!c[w[e]]) u++;
                c[w[e]]++;

                while (u > k) {
                        c[w[s]]--;
                        if (!c[w[s]]) u--;
                        s++;
                }

                new_ans = e-s+1;

                if (new_ans > ans) {
                        ans = new_ans;
                        //ans_start = s;
                }
        }

        cout << ans << '\n';
}
```

danza

5. Idem, padám, balancujem

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

V tejto úlohe sme mali zadaný graf na n vrcholoch. Zaujímalo nás nájdenie takej cesty z vrchola 1 do vrchola n, ktorá má párnu dĺžku menšiu ako $2n$.

Pomalé riešenie

Zadanie úlohy vyzerá takmer ako bežné hľadanie najkratšej cesty. Jediným problémom je, že naša cesta musí mať párnu dĺžku.

Môžeme teda spraviť nasledovný trik. Vyrobíme si nový graf na vrcholoch 1 až n. V tomto novom grafe bude hrana medzi každou dvojicou vrcholov a, b takou, že v pôvodnom grafe boli a, b vo vzdialenosti 2. Inak povedané, každá hrana v novom grafe bude predstavovať dve hrany pôvodného grafu.

Keď teraz nájdene najkratšiu cestu mezdi vrcholmi 1 a n v novom grafe, bude mať nejakú dĺžku d. Táto cesta bude zodpovedať ceste dĺžky $2d$ v pôvodnom grafe. Ak je $2d < 2n$, našli sme vyhovujúce riešenie. Na hľadanie najkratšej cesty môžeme použiť vhodné prehľadávanie grafu.

Toto riešenie bude mať časovú zložitosť $O(n^3)$ kvôli vytváraniu nového grafu.

Listing programu (Python)

```
import sys
sys.setrecursionlimit(200000)

def dfs(a):
    # print('a n:',a,n)
    if visited[a] == True:
        return []
```

```
    if a == v-1:
        return [a]
    visited[a] = True
    for b,c in G2[a]:
        res = dfs(b)
        if len(res) > 0:
            res.append(c)
            res.append(a)
            return res
    return []

v, e = map(int, input().split())
G = [[] for i in range(v)]
for i in range(e):
    a,b = map(int, input().split())
    a -= 1
    b -= 1
    G[a].append(b)
    G[b].append(a)

G2 = [[] for i in range(v)]
for a in range(v):
    for b in G[a]:
        for c in G[a]:
            if b < c:
                G2[b].append((c,a))
                G2[c].append((b,a))


visited = [False for i in range(v)]
res = dfs(0)
if len(res) == 0:
    print(-1)
else:
    print(*map(lambda x: x+1, reversed(res)))
```

Listing programu (C++)

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

using namespace std;

int v,e;
vector<vector<pair<int, int>>> G2;
vector<bool> visited;

vector<int> dfs(int a){
    //# print('a n:',a,n)
    vector<int> res;
    if (visited[a] == true)
        return res;
    if (a == v-1){
        res.push_back(a);
        return res;
    }
    visited[a] = true;
    for (auto bc : G2[a]){
```

```
        int b = bc.first;
        int c = bc.second;
        res = dfs(b);
        if (res.size() > 0){
            res.push_back(c);
            res.push_back(a);
            return res;
        }
    }
    return res;
}

int main(){

    cin >> v >> e;
    vector<vector<int>> G(v);
    for (int i=0; i<e; i++){
        int a,b;
        cin >> a >> b;
        a -= 1;
        b -= 1;
        G[a].push_back(b);
        G[b].push_back(a);
    }

    G2.resize(v);
    for (int a=0; a<v; a++)
        for (auto b : G[a])
            for (auto c : G[a])
                if (b < c){
                    G2[b].push_back({c,a});
                    G2[c].push_back({b,a});
                }


    visited.resize(v,false);
    vector<int> res = dfs(0);
    if (res.size() == 0)
        cout<<"-1"<<endl;
    else{
       cout<<res[res.size()-1]+1;
        for (int i=res.size()-2; i>=0; i--)
            cout<<' '<<res[i]+1;
        cout<<endl;
    }
}
```

Štandardná úloha

Táto úloha je celkom štandardná. Pri bežnom hľadaní najkratšej cesty si v každom vrchole pamätáme najmenšiu vzdialenosť, na ktorú sa do daného vrchola vieme dostať zo zdroja. My si ale v každom vrchole budeme pamätať dve informácie. Jedna z nich bude predstavovať najmenšiu párnu vzdialenosť, na ktorú sa do daného vrchola vieme dostať. Druhá bude analogicky predstavovať najmenšiu nepárnu vzdialenosť do daného vrchola zo zdroja.

Obe tieto informácie vieme počas prehľadávania grafu jednoducho aktualizovať. Princíp je rovnaký, ako pri obyčajnom BFS.

Netreba najkratšiu cestu

Zadanie od nás vyžaduje iba nájsť nejakú nie príliš dlhú párnu cestu. Namiesto BFS teda môžeme použiť aj DFS. Opäť budeme mať 2 kópie pôvodného grafu. Jednu nazveme nepárna, druhú párna. V každom kroku DFS sa presunieme do susedného vrchola, avšak v opačnej kópii. Prehľadávanie začneme v jednej z kópií vo vrchole 1. Ak sa nám podarí dostať do vrchola n v tej istej kópií, našli sme párnu cestu.

Ak niekedy počas prehľadávania prejdeme už viac, ako $2n$ vrcholov, nevnoríme sa ďalej, pretože by takáto cesta bola príliš dlhá.

Takéto riešenie má časovú aj pamäťovú zložitosť $O(n + m)$.

Listing programu (Python)

```
import sys
sys.setrecursionlimit(200000)

def dfs(a, n):
    # print('a n:',a,n)
    if visited[n][a] == True:
        return []
    if n == 0 and a == v-1:
        return [a]
    visited[n][a] = True
    for b in G[a]:
        res = dfs(b, n^1)
        if len(res) > 0:
            res.append(a)
            return res
    return []

v, e = map(int, input().split())
G = [[] for i in range(v)]
for i in range(e):
    a,b = map(int, input().split())
    a -= 1
    b -= 1
    G[a].append(b)
    G[b].append(a)

visited = [[False for i in range(v)], [False for i in range(v)]]
res = dfs(0,0)
if len(res) == 0:
    print(-1)
else:
    print(*map(lambda x: x+1, reversed(res)))
```

Listing programu (C++)

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

using namespace std;

vector<vector<bool>> visited(2);
int v, e;
vector<vector<int>> G;


vector<int> dfs(int a, int n){
    //# print('a n:',a,n)
```

```
    vector<int> res;
    if (visited[n][a] == true)
        return res;
    if (n == 0 and a == v-1){
        res.push_back(a);
        return res;
    }
    visited[n][a] = true;
    for(auto b : G[a]){
        res = dfs(b, n^1);
        if (res.size() > 0){
            res.push_back(a);
            return res;
        }
    }
    return res;
}

int main(){
    cin >> v >> e;
    G.resize(v);

    for (int i=0; i<e; i++){
        int a, b;
        cin >> a >> b;
        a -= 1;
        b -= 1;
        G[a].push_back(b);
        G[b].push_back(a);
    }
    visited[0].resize(v,false);
    visited[1].resize(v,false);
    vector<int> res = dfs(0,0);
    if (res.size() == 0)
        cout<<"-1"<<endl;
    else {
        cout<<res[res.size()-1]+1;
        for (int i=res.size()-2; i>=0; i--)
            cout<<' '<<res[i]+1;
        cout<<endl;
    }
}
```

6. Čarovný lexikón kúziel

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

Úlohou bolo hľadať počet výskytov podreťazca v reťazci, ktorý sa mení. Po každej zmene bolo treba znova zistiť počet výskytov podreťazca.

Pomalé riešenie

Prvou možnosťou je jednoduché prehľadanie reťazca, pričom pre každú pozíciu overíme, či sa na nej začína podreťazec. Overenie prebehne porovnávaním znakov podreťazca s aktuálne overovanou časťou reťazca. Pre každý znak, ktorých je n, skontrolujeme prinajhoršom m znakov, teda celkovo urobíme $O(nm)$ operácii. Časová zložitosť je teda $O(nm)$ Po každej zmene reťazca, ktorých je q, kontrolujeme znovu celý reťazec, teda časová zložitosť riešenia bude $O(qnm)$. Pamäťová zložitosť je $O(n+m)$, lebo okrem reťazca a podreťazca si nemusíme nič pamätať. Toto riešenie prejde prvou sadou testov.

Listing programu (C++)

```
#include <iostream>

std::string text, pattern;

int substrSearchNaive() {
    int patternsFound = 0;
    for (size_t i = 0; i < text.length(); i++) {
        bool found = true;
        for (size_t j = 0; j < pattern.length(); j++) {
            if (text[i + j] != pattern[j]) {
                found = false;
                break;
            }
        }
        patternsFound += found;
    }
    return patternsFound;
}

int replaceNaive(const size_t where, const std::string &what) {
    text.replace(where, what.length(), what);
    return substrSearchNaive();
}

int main() {
    int q;
    std::cin >> text >> pattern >> q;
    int patternCount = substrSearchNaive();
    std::cout << patternCount << "\n";
    for (int i = 0; i < q; i++) {
        int where;
        std::string what;
        std::cin >> where >> what;
        patternCount = replaceNaive(where, what);
        std::cout << patternCount << "\n";
    }
}
```

Trochu lepšie riešenie

Pomalé riešenie môžeme vylepšiť tak, že celý text prehľadáme len raz a po každej zmene už prehľadáme len zaujímavú časť reťazca. Zaujímavá je samotná zmenená časť a aj rovnako dlhá časť pred ňou a po nej. To je potrebné aby sme zistili, či sa úpravou textu neporušil alebo nevytvoril nový výskyt, ktorý sa nenachádal úplne celý v zmenenej časti textu. Keďže sme neprehľadali celý reťazec, musíme zistiť rozdiel počtu výskytov podreťazca v zaujímavej časti pred a po zmene, čiže zaujímavú časť musíme prehľadať dvakrát. Po úvodnom prehľadaní si zapamätáme počet výskytov v celom reťazci a ten potom zvyšujeme alebo znižujeme pomocou rozdielu získaného po každej zmene. Časová zložitosť sa tým zlepší na $O(nm + qm^2)$. Pamäťová zložitosť sa oproti predošlému riešeniu nezmení. Toto riešenie prejde prvé dve sady testov.

Listing programu (C++)

```
#include <iostream>

std::string text, pattern;

int substrSearchNaive(size_t start=0, size_t end=-1u) {
    int patternsFound = 0;
```

```
    for (size_t i = start; i < std::min(end, text.length()); i++) {
        bool found = true;
        for (size_t j = 0; j < pattern.length(); j++) {
            if (text[i + j] != pattern[j]) {
                found = false;
                break;
            }
        }
        patternsFound += found;
    }
    return patternsFound;
}

int substrSearchNear(const int where) {
    int patternsFound = 0;
    size_t start = std::max(0, where - (int) pattern.length());
    size_t end = std::min(text.length(), where + pattern.length() * 2);
    return substrSearchNaive(start, end);
}

int replaceNaive(const size_t where, const std::string &what) {
    int previousCount = substrSearchNear(where);
    text.replace(where, what.length(), what);
    int currentCount = substrSearchNear(where);
    return currentCount - previousCount;
}

int main() {
    int q;
    std::cin >> text >> pattern >> q;
    int patternCount = substrSearchNaive();
    std::cout << patternCount << "\n";
    for (int i = 0; i < q; i++) {
        int where;
        std::string what;
        std::cin >> where >> what;
        patternCount = replaceNaive(where, what);
        std::cout << patternCount << "\n";
    }
}
```

Rabinov-Karpov algoritmus

Algoritmus spočíva na rolling hash funkcii. To je taká hash funkcia, ktorá vie vypočítať nový hash pomocou starého hashu v konštantnom čase. Na začiatku si vypočítame hash podreťazca a hash prvých m znakov reťazca. Tak isto ako v pomalom riešení prehľadávame reťacec po znakoch, avšak výskyt podreťazca neoverujeme pre každú pozíciu, ale len ak sa zhoduje hash práve prehľadávanej časti textu s hashom hľadaného podreťazca. Po každom znaku vieme zo starého hashu vypočítať nový v konštantnom čase. Časová zložitosť tohto riešenia je v najhoršom prípade $O(mn + qm^2)$ čo je zložitosť predchádzajúceho riešenia, ale v očakávanom a veľmi pravdepodobnom prípade $O(n + qm)$ čo je zložitosť optimálneho riešenia. Prvý prípad nastane, ak sa text skladá iba z podreťazca, čiže podreťazec sa vyskytuje na každej pozícii. Tomuto sa môžeme vyhnúť a algorimus urýchliť vynechaním overenia v prípade zhody hashu. Vtedy hrozí, že v prípade kolízie hashu nesprávne určíme, že sa vyskytol podreťazec. Dá sa ukázať, že kolizii je veľmi málo, navyše sa vyskytnú v iných prípadoch závisiacich od zvolenej hashovacej funkcie resp. prvočísla, ktorým modulujeme. Pre zníženie pravdepodobnosti kolízie môžeme použiť viac ako jednu hashovaciu funkciu. Pamäťová zložitosť bude $O(n+m)$, keďže si pamätáme reťazec dĺžky n a podreťazec dĺžky m. Toto riešenie prejde prvé tri sady testov a v prípade vynechania overenia prvé štyri sady testov.

Listing programu (C++)

```
#include <iostream>
#include <cmath>

// ALPHABET_SIZE is the number of characters in the input alphabet
#define ALPHABET_SIZE 256
#define PRIME_MODULUS 15285151248481

int64_t patternHash = 0;
int64_t hashSlider = 1;

int search(const std::string &pattern, const std::string &text) {
    int result = 0;
    int64_t slidingHash = 0;

    // Calculate the hash value of pattern and first
    // window of text
    for (size_t i = 0; i < pattern.length(); i++) {
        slidingHash = (ALPHABET_SIZE * slidingHash + text[i]) % PRIME_MODULUS;
    }

    // Slide the pattern over text one by one
    for (size_t i = 0; i <= text.length() - pattern.length(); i++) {

        // Check the hash values of current window of text
        // and pattern. If the hash values match then only
        // check for characters on by one
        if (patternHash == slidingHash) {
            bool found = true;
            /* Check for characters one by one */
            for (size_t j = 0; j < pattern.length(); j++) {
                if (text[i + j] != pattern[j]) {
                    found = false;
                    break;
                }
            }
            result += found;
        }

        // Calculate hash value for next window of text: Remove
        // leading digit, add trailing digit
        if (i < text.length() - pattern.length()) {
            slidingHash = (ALPHABET_SIZE * (slidingHash - text[i] * hashSlider)
                ↪ + text[i + pattern.length()]) % PRIME_MODULUS;

            // We might get negative value of slidingHash, converting it
            // to positive
            if (slidingHash < 0)
                slidingHash += PRIME_MODULUS;
        }
    }
    return result;
}

void calculatePatternHash(const std::string &pattern) {
    for (size_t i = 0;  i < pattern.length(); i++) {
        patternHash = (ALPHABET_SIZE * patternHash + pattern[i]) % PRIME_MODULUS
```

```
            ↪ ;
        if (i != pattern.length() - 1) {
            hashSlider = (hashSlider * ALPHABET_SIZE) % PRIME_MODULUS;
        }
    }
}

int main() {
    std::string pattern, text;
    int numChanges;
    std::cin >> text >> pattern >> numChanges;
    calculatePatternHash(pattern);
    int patternCount = search(pattern, text);
    std::cout << patternCount << "\n";
    for (int i = 0; i < numChanges; i++) {
        int where;
        std::string what;
        std::cin >> where >> what;
        const int textOffset = std::max(0, (int) (where - what.length()));
        std::string subText = text.substr(textOffset, pattern.length() * 3);
        const int prevCount = search(pattern, subText);
        text.replace(where, what.length(), what);
        subText.replace(what.length(), what.length(), what);
        subText = text.substr(std::max(0, (int) (where - what.length())),
            ↪ pattern.length() * 3);
        const int currCount = search(pattern, subText);
        patternCount += currCount - prevCount;
        std::cout << patternCount << "\n";
    }
}
```

Vzorové riešenie

Vzorové riešenie využíva KMP algoritmus[3]. Časová zložitosť predspracovania, čiže konštrukcie funkcie `next` bude $O(m)$ a samotné prehľadanie reťazca bude trvať $O(n)$. Po prvom prehľadaní celého reťazca už kontroluje len zaujímavé časti reťazca. Zaujímavá je samotná zmenená časť a aj rovnako dlhá časť pred ňou a po nej. To je potrebné aby sme zistili, či sa úpravou textu neporušil alebo nevytvoril nový výskyt, ktorý sa nenachádal úplne celý v zmenenej časti textu. Časová zložitosť riešenia bude $O(n + qm)$. Pamäťová zložitosť bude $O(n + m)$, keďže si pamätáme reťazec dĺžky n a podreťazec a automat dĺžky m.

Listing programu (C++)

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

std::vector<size_t> next;
std::string text, pattern;

void constructPrefix() {
    next = std::vector<size_t>(pattern.length(), 0);
    size_t state = 0; // dlzka najdlhsieho prefixosuffixu
    size_t i = 1; // prave spracovavane pole funkcie, musi zacinat od 1, lebo
        ↪ prvy prvok bude vzdy 0
    while (i < pattern.length()) {
        // pokym sedi znak z podretazca
        if (pattern[i] == pattern[state]) {
```

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

```
            next[i] = ++state;
            i++;
        }
        // vykonava sa, ak doslo k nezhode az pokym je stav > 0
        else if (state > 0) {
            state = next[state - 1];
        } else {
            next[i] = 0;
            i++;
        }
    }
}

/*
 * prehlada cast textu ohranicenu cez parametre start a end
 * vrati pocet vyskytov podretazca v prehladanej casti
 */
int kmpSearch(const size_t start, const size_t end) {
    int patternOccurences = 0;
    size_t state = 0;
    for (size_t i = start; i <= end; i++) {
        char textChar = text[i];
        // ak doslo k nezhode, vratit stav automatu na najblizsi mozny zaciatok
           ↪ podretazca
        while (state > 0 && textChar != pattern[state]) {
            state = next[state - 1];
        }
        // ak sedi znak s podretazcom, zvysit stav automatu
        if (textChar == pattern[state]) {
            state++;
        }
        // ak je stav rovny dlzke podretazca, nasiel sa jeho vyskyt
        if (state == pattern.length()) {
            patternOccurences++;
            state = next[state - 1];
        }
    }
    return patternOccurences;
}

/*
 * upravi retazec a vrati rozdiel poctov vyskytov podretazca v retazci pred a po
    ↪  uprave
 */
int updateText(const int where, const std::string &what) {
    // oznacuje poziciu prveho zaujimaveho znaku v retazci
    // prvy vyskyt, ktory ovplyvnime je ten, ktoreho posledny znak je na prvej
       ↪ zmenenej pozicii v texte
    size_t start = std::max(0, where - (int) pattern.length() + 1);
     // oznacuje poziciu posledneho zaujimaveho znaku v retazci
     // posledny vyskyt, ktory ovplyvnime je ten, ktoreho prvy znak je na
        ↪ posledej zmenenej pozicii v texte
    size_t end = std::min(text.length() - 1, where + pattern.length() * 2 - 2);
    int occurencesBeforeReplace = kmpSearch(start, end);
    text.replace(where, what.length(), what);
    int occurencesAfterReplace = kmpSearch(start, end);
    return occurencesAfterReplace - occurencesBeforeReplace;
}
```

```
int main() {
    int q;
    std::cin >> text >> pattern >> q;
    constructPrefix(); // skonstruovanie automatu pre nas podretazec, bude
        ↪ rovnaky pre cely priebeh
    int patternCount = kmpSearch(0, text.length()); // uvodne prehladanie celeho
        ↪  retazca
    std::cout << patternCount << "\n";
    for (int i = 0; i < q; i++) {
        int where;
        std::string what;
        std::cin >> where >> what;
        patternCount += updateText(where, what);
        std::cout << patternCount << "\n";
    }
}
```

Paulinka

7. Kto Spasí Pochúťku

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

Ako prvé, poďme si úlohu trochu učesať. Na prvé prečítanie úloha môže vyzerať trochu desivo, ale na druhé prečítanie už skúsenejší riešiteľ všimne, že úloha je o hľadaní najkratšej cesty v grafe.

Vrcholy v tomto prípade sú vedúci a T2, a medzi vedúcimi `A` a `B` je hrana, ak si vedia medzi sebou premeniť tortu a hrana medzi vedúcim a T2 je ak vedúci vie doniesť tortu do T2. V tomto grafe je následne úloha nájsť najkratšiu cestu medzi Timkou a T2, alebo rozhodnúť, že žiadna cesta neexistuje.

Ako na to?

Máme celý graf

Riešenie, ktoré nám prirodzene napadne je vygenerovať si celý graf, a následne na ňom zbehnúť náš obľúbený rýchly algoritmus na hľadanie najkratších ciest v grafoch – BFS.

Ako vygenerujeme graf? Pre každú dvojicu vedúcich stačí skontrolovať, či minimum ich dosahov je aspoň vzdialenosť do polcesty medzi nimi. Takto vieme postaviť graf v čase $O(N^2)$ a následne v lineárnom čase vo veľkosti grafu spustiť BFS. Graf však môže mať tiež veľkosť najviac $O(N^2)$ – v prípade, že sú všetci vedúci ochotný ísť ďaleko.

Ak niečo ďalej nezlepšíme, dostaneme kvadratické riešenie, ktoré nám dá 4 body. Toto riešenie vieme implementovať aj v lineárnej pamäti (napríklad si graf konštruujeme postupne, a nepamätáme si všetky hrany naraz).

Malé grafy

BFS vieme robiť v čase lineárnom od veľkosti grafu. Ak je graf reprezentovaný v zadaní *malý*, keby sme ho vedeli rýchlo zkonštruovať, vieme ho aj rýchlo prehľadať.

Predstavme si, že máme vedúcich v zadaní utriedených podľa pozície. i-ty vedúci, stojaci na pozícii x_i, ochotný prejsť d_i si vie tortu potenciálne vymeniť iba s vedúcimi na pozíciách medzi $x_i - 2d_i$ a $x_i + 2d_i$.

Mohli by sme si preto napríklad spraviť nasledovné: utriedme si vedúcich podľa pozície. Keď sa v BFS dostaneme k vedúcemu číslo i, začneme si pole prechádzať (do oboch strán) kým neprídeme k vedúcemu nachádzajúcemu sa mimo intervalu, kde i-ty vedúci vie potenciálne tortu vymeniť. Pre každého vedúceho v intervale si už vieme overiť existenciu hrany v konštantnom čase.

Takto i-teho vedúceho v BFS spracujeme v čase $O(d_i)$, takže dokopy dostaneme časovú zložitosť $O(n \max_i d_i + n \log n)$ ($O(n \log n)$ je preto, lebo musíme triediť). Pamäťovú zložitosť vieme stále mať lineárnu, keďže zostrojený graf nedržíme v pamäti. Takéto riešenie nám dá 6 bodov.

Kde je to pomalé?

Problém je, že vo všeobecnosti môžu byť jednotlivé d_i veľké, takže aj počet hrán v grafe príliš veľký, aby sme sa na ne pozreli. Všimnime si, že keď už v BFS pridáme vrchol do fronty (alebo už spracujeme), tak keď sa pri spracovávaní iných vrcholov pozeráme na hrany idúce do tohto vrcholu, je to zbytočné, pretože tento vrchol už druhý krát nebudeme pridávať do fronty (a teda druhý krát spracovávať).

Mohli by sme si vrchol z grafu po jeho prvotnom pridaní do grafu vymazať aby sme sa tomuto vyhli? Na prvý pohľad to neznie sľubne, keďže nevieme rozumne efektívne vymazávať elementy zo stredu poľa.

Problém je ale aj to, že keby sme aj vedeli, stále sa nám môže stať, že väčšina vedúcich ku ktorým je i-ty vedúci nablízku sú príliš leniví aby si ním vymenili tortu. Takže by sme potrebovali vymyslieť spôsob akým sa nepozerať na príliš veľa neexistujúcich hrán.

Mohli by sme to vyriešiť použitím vhodnej dátovej štruktúry?

Meníme pole za intervaláč

Najskôr sa pozrime na druhý problém. Kedy si dvaja vedúci i a j vedia vymeniť tortu? Keď je rozdiel ich vzdialeností najviac $2\min(d_i, d_j)$, inak povedané (BUNV $x_i < x_j$) ak $x_j \leq x_i + 2d_i$ a zároveň $x_i \geq x_j - 2d_j$.

Predstavme si, že by sme vedeli spraviť nasledovné: pre všetkých vedúcich stojacich medzi x_i a $x_i + 2d_i$, pozrime sa na takého vedúceho j pre ktorého je $x_j - 2d_j$ čo *najmenšie*. Ak $x_j - 2d_j > x_i$, potom ani žiadny ostatní vedúci v napravo od i-tého vedúceho si s ním môžu vymeniť tortu (ak sú príliš ďaleko od i-teho vedúceho, ten im nie je ochotný predať tortu; všetci vedúci v dosahu i-teho vedúceho ale majú $x_{j'} - 2d_{j'} > x_j - 2d_j > x_j$, teda nie sú ochotní chodiť až ku vedúcemu i). Podobne vieme pre vedúcich naľavo od vedúceho i hľadať zase blízkeho vedúceho j s najvyšším $x_j + 2d_j$.

Hľadanie maxím a miním v intervale znie ako práca pre intervalový strom[4].

Ten má aj výhodu, že v ňom vieme rýchlo (v $O(\log n)$) updatovať dáta.

Mohli by sme preto spraviť nasledovné: majme dva intervalové stromy. Minimový s hodnotami $x_j - 2d_j$, a maximový s hodnotami $x_j + 2d_j$.

Vždy keď spracovávame vedúceho i v BFS, pozrieme sa na vedúcich v dosiahnuteľnom intervale naľavo v maximovom intervaláči. Nech j je iný vedúci s najvyššou hodnotou $x_j + 2d_j$ nachádzajúci sa na pozíciach medzi $x_i - 2d_i$ a x_i. Potom, ak si s ním nevieme predať tortu, skončili sme – žiadny iný dosiahnuteľný vedúci napravo od i-teho vedúceho si s ním tortu nevymení. Ak sme ale naďabili na hranu, potom môžme pokračovať.

Aby sme sa ale vyhli spracovaniu nájdeného vedúceho neskôr v BFS (ako sme si všimli hore, to robí problémy), *vymažeme ho zo stromov*. Vymazať pozíciu v strede intervalového stromu na prvý pohľad znie ako pomalé, avšak namiesto vymazania môžeme nastaviť pozíciu vedúcemu v minimovom strome na $-\infty$ a v maximovom strome na $+\infty$ (kde nekonečno je nejaké dostatočne veľké číslo), takže keby sme ho našli ako minimum/maximum tak hľadania vždy ukončíme (keďže to znamená, že žiadny nespracovaný/nevidený vedúci intervale nie je).

Keď j-teho vedúceho týmto spôsobom vyhodíme zo stromu, môžme pokračovať hľadanie vedúcich ktorým možno odovzdať tortu, až kým nám buď dôjdu vedúci, alebo všetci vedúci v intervale sú príliš lenivý vymeniť si tortu s i-tym vedúcim.

Následne toto isté opakujeme pre dosiahnuteľných vedúcich napravo od i-teho, pomocou minimového intervaláča.

Každého vedúceho spracujeme najviac raz, a dokopy sa cez intervaláče pozrieme na najviac $O(n)$ možných hrán (najviac n hrán ktorými sa v BFS vyberieme, a pre každý vrchol sa pozrieme na najviac dve počet neexistujúce hrany resp. hrany do už navštíveného vrchola). Každá otázka, alebo update do intervaláča má časovú zložitosť $O(\log n)$, takže časová zložitosť tohto algoritmu je $O(n \log n)$. Intervaláč vieme postaviť v pamäti $O(n)$, takže aj celý algoritmus vieme implementovať v lineárnej pamäti.

Technické detaily

Zabudli sme spomenúť dve veci: T2, a ako sa v intervaláči pozrieť na interval vedúcich na nejakom intervale pozícií.

Začnime tým jednoduchším: T2. Potom, ako sme spočítali vzdialenosti od Timky pre všetkých vedúcich vieme spočítať minimálnu vzdialenosť do T2 jednoducho: v konštantnom čae sa pre každého vedúceho spýtame, či vie dostiahnuť T2, a to tak, že nájdeme vedúceho s minimálnou vzdialenosťou od Timky ktorý vie tortu dostať do T2.

Čo s druhým problémom? Jedno z riešení je použiť kompresiu súradníc: dať v intervalom strome vedúcich v utriedenom poradí podľa ich pozície, a pomocou utriedenej množiny (kde si uskladníme dvojice `(pozícia, relatívna pozícia)`) zistiť ktoré relatívne pozície pripadajú ku danému intervalu pozícií.

Druhé riešenie je mať súradnice priamo v intervaláčoch: majme vedúcich vložených do intervaláča podľa ich utriedených pozícií, ale priamo v intervaláči si zapamätajme aj súradnice. Potom v každom vrchole si môžeme navyše pamätať minimálnu a maximálnu pozíciu v listoch patriacich pod vrchol. Takto sa môžeme intervaláč pýtať priamo na interval pozícií, bez toho aby sme museli riešiť navyše kompresiu súradníc.

Žiadne z týchto dvoch riešení nezhoršuje časovú zložitosť celkového riešenia.

[4] pozri článok o intervalovom strome v kuchárke KSP[5]

Listing programu (C++)

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

using namespace std;

#define FOR(i,n)        for(int i=0;i<(int)n;i++)
#define FOB(i,n)        for(int i=n;i>=1;i--)
#define MP(x,y) make_pair((x),(y))
#define ii pair<int, int>
#define lli long long int
#define ld long double
#define ulli unsigned long long int
#define lili pair<lli, lli>
#ifdef EBUG
#define DBG     if(1)
#else
#define DBG     if(0)
#endif
#define SIZE(x) int(x.size())
const int infinity = 2000000999;
const long long int inff = 4000000000000000999;

typedef complex<long double> point;

template<class T>
T get() {
    T a;
    cin >> a;
    return a;
}

template <class T, class U>
ostream& operator<<(ostream& out, const pair<T, U> &par) {
    out << "[" << par.first << ";" << par.second << "]";
    return out;
}

template <class T>
ostream& operator<<(ostream& out, const set<T> &cont) {
    out << "{";
    for (const auto &x:cont) out << x << ", ";
    out << "}";
    return out;
}

template <class T, class U>
ostream& operator<<(ostream& out, const map<T,U> &cont) {
    out << "{";
    for (const auto &x:cont) out << x << ", ";

    out << "}"; return out;
}

template <class T>
ostream& operator<<(ostream& out, const vector<T>& v) {
  FOR(i, v.size()){
    if(i) out << " ";
```

```
    out << v[i];
  }
  out << endl;
  return out;
}

bool ccw(point p, point a, point b) {
    if((conj(a - p) * (b - p)).imag() <= 0) return false;
    else return true;
}

struct intervalac {
    int N;
    vector<ii> miniL, maxiR;
    vector<int> minD, maxD;
    vector<int> start, range;

    intervalac(int n, vector<int> &d, vector<int> &pos) {
        N = pow(2, (ceil(log2(n))));

        miniL.resize(2 * N, {infinity, -1});
        maxiR.resize(2 * N, {-infinity, -1});
        minD.resize(2 * N, infinity);
        maxD.resize(2 * N, infinity);

        range.resize(2 * N, N);
        start.resize(2 * N, 0);

        for (int i = 2; i < 2 * N; i ++) {
            range[i] = range[i / 2] / 2;
            start[i] = start[i / 2] + (i % 2 ? range[i] : 0);
        }

        for (int i = N + n - 1; i > 0; i --) {
            if (i >= N) {
                miniL[i] = {pos[i - N] - 2 * d[i - N], i - N};
                maxiR[i] = {2 * d[i - N] + pos[i - N], i - N};
                minD[i] = pos[i - N];
                maxD[i] = pos[i - N];
            }
            else {
                miniL[i] = (miniL[i * 2].first > miniL[i * 2 + 1].first ? miniL[
                    ↪ i * 2 + 1] : miniL[i * 2]);
                maxiR[i] = (maxiR[i * 2].first > maxiR[i * 2 + 1].first ? maxiR[
                    ↪ i * 2] : maxiR[i * 2 + 1]);
                minD[i] = min(minD[i * 2], minD[i * 2 + 1]);
                maxD[i] = max(maxD[i * 2], maxD[i * 2 + 1]);
            }
        }

//          DBG cout << "minD:" << minD << "maxD: " << maxD;
    }

    ii query_upd_mini(int l, int r, int id) {
//          DBG cout << "In query_upd_mini(l = " << l << ", r = " << r << ", id =
    ↪   " << id << "), interval [" << minD[id] << ", " << maxD[id] << "] maxR = "
    ↪   << maxiR[id] << endl;
        if (l > maxD[id] || r < minD[id]) return {infinity, -1};
```

```
        if (l <= minD[id] && r >= maxD[id]) return miniL[id];

        ii resF = query_upd_mini(l, r, id * 2), resS = query_upd_mini(l, r, id *
           ↪  2 + 1);
        ii res = min(resF, resS);
        return res;
    }

    ii query_upd_maxi(int l, int r, int id) {
        if (l > maxD[id] || r < minD[id]) return {-infinity, -1};
        if (l <= minD[id] && r >= maxD[id]) return maxiR[id];
        ii resF = query_upd_maxi(l, r, id * 2), resS = query_upd_maxi(l, r, id *
           ↪  2 + 1);
        ii res = max(resF, resS);
        return res;
    }

    void update_to_none(int pos, int id) {
//          DBG cout << pos << " <- infinity " << endl;
        if (pos < start[id] || pos >= start[id] + range[id]) return;
        if (range[id] == 1) {
            maxiR[id] = {-infinity, -1};
            miniL[id] = {infinity, -1};
            return;
        }
        update_to_none(pos, id * 2 + 1);
        update_to_none(pos, id * 2);
        miniL[id] = min(miniL[id * 2], miniL[id * 2 + 1]);
        maxiR[id] = max(maxiR[id * 2], maxiR[id * 2 + 1]);
    }
};



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

    int n = get<int>();
    int D = get<int>();

    vector<ii> pos;
    vector<int> d;
    FOR(i, n) pos.push_back({get<int>(), i});
    FOR(i, n) d.push_back(get<int>());

    sort(pos.begin(), pos.end());

    vector<int> sorted_pos, sorted_d;
    int timka = -1;
    FOR(i, n) {
        if (pos[i].second == 0) timka = i;
        sorted_pos.push_back(pos[i].first);
        sorted_d.push_back(d[pos[i].second]);
    }

    DBG cout << pos;
```

```
    DBG cout << "Going to build segment tree,\n" << sorted_pos << sorted_d;

    intervalac I(n, sorted_d, sorted_pos);
    vector<int> dist(n, infinity);

    DBG cout << "timka = " << timka << " n = " << n << endl;

    dist[timka] = 0;
    queue<int> Q;
    Q.push(timka);
    I.update_to_none(timka, 1);

    // BFS

    while (Q.size()) {
        int v = Q.front();
        DBG cout << "v = " << v << " queue size: " << Q.size() << endl;
        Q.pop();

        int l = sorted_pos[v] - 2 * sorted_d[v];
        int r = sorted_pos[v] + 2 * sorted_d[v];

        while (true) {
            ii zlava = I.query_upd_maxi(l, sorted_pos[v], 1);
            DBG cout << "zlava: " << zlava << endl;

            if (zlava.second >= 0 && 2 * min(sorted_d[v], sorted_d[zlava.second
                ↪ ]) >= abs(sorted_pos[v] - sorted_pos[zlava.second])) {
                I.update_to_none(zlava.second, 1);
                if (dist[zlava.second] != infinity) {
                    DBG cout << "ALERT FOR " << zlava << endl;
                }
                dist[zlava.second] = dist[v] + 1;
                Q.push(zlava.second);
                continue;
            }

            ii zprava = I.query_upd_mini(sorted_pos[v], r, 1);

            DBG cout << "zprava: " << zprava << " pos is " << sorted_pos[zprava.
                ↪ second] << " di = " << sorted_d[zprava.second] << " dv = " <<
                ↪ sorted_d[v] << " and pos " << sorted_pos[v] << endl;

            if (zprava.second >= 0 && 2 * min(sorted_d[v], sorted_d[zprava.
                ↪ second]) >= abs(sorted_pos[v] - sorted_pos[zprava.second])) {
                I.update_to_none(zprava.second, 1);
                if (dist[zprava.second] != infinity) DBG cout << "ALERT! Zprava
                    ↪ " << zprava << endl;
                dist[zprava.second] = dist[v] + 1;
                Q.push(zprava.second);
                continue;
            }

            break;
        }
    }

    int mindistance = infinity;
```

```
    DBG cout << "   dists: " << dist;

    FOR(i, n) {
        if (2 * min(D, sorted_d[i]) >= abs(sorted_pos[i])) {
            mindistance = min(mindistance, 1 + dist[i]);
        }
    }

    if (mindistance < infinity) cout << mindistance << endl;
    else cout << "Torta nebude" << endl;

}
```

Hodobox

8. Ako priečinky robia radosť

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

Prevedieme si najprv úlohu do informatickej terminológie. Vedúci sú vrcholy, a odovzdanie priečinka je orientovaná hrana – ak vedúci x vie odovzdať vedúcemu y priečinok s radosťou odovzdávania $a_y - b_x$, tak z vrcholu x vedie hrana do vrcholu y s váhou $a_y - b_x$. Životná púť priečinka je teda cesta v tomto grafe, a hodnota je súčet váh hrán na nej.

Môžeme si o tomto grafe navyše všimnúť, že je acyklický, keďže každý vedúci vie odovzdať priečinok v ostro neskoršom roku, ako ho dostal. Je to teda DAG (directed acyclic graph).

N < 20 znamená...

...že použijeme starú dobrú hrubú silu, a prejdeme si všetky možné púte priečinka. Skúsime začať v každom vrchole, pozrieme sa na všetky ostatné, a rekurzívne pokračujeme do každého, do ktorého vedie hrana.

Hodnotu každej púte, ktorú spracujeme si odložíme, a po prehľadávaní ich usporiadame a sčítame k najhodnotejších z nich.

Listing programu (C++)

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

using namespace std;

struct veduci {
    long long pride, odide, v_pride, v_odide;
};

vector<veduci> v;
vector<long long> sols;

// existuje put s hodnotou p prechadzajuca veducim i
void bf(int i,long long p)
{
    sols.push_back(p);
    for(int k=0;k<v.size();++k)
    {
        if(v[k].pride == v[i].odide && v[k].v_pride > v[i].v_odide)
            bf(k,p+v[k].v_pride-v[i].v_odide);
    }
}

int main()
{
    int n,k;
```

```
    cin >> n >> k;

    v.resize(n);
    for(int i=0;i<n;++i)
    {
        cin >> v[i].pride >> v[i].odide >> v[i].v_pride >> v[i].v_odide;
    }

    for(int i=0;i<n;++i)
        bf(i,0);

    sort(sols.begin(),sols.end(),greater<long long>());

    long long ans = 0;
    for(int i=0;i<min(k,(int)sols.size());++i)
        ans += sols[i];

    cout << ans%1000000007 << "\n";

}
```

Aby sme určili zložitosti, musíme zistiť, koľko ciest vie existovať v orientovanom acyklickom grafe. Môžeme využiť fakt, že v takomto grafe je cesta jednoznačne určená množinou vrcholov v nej, keďže cez vrcholy na ceste vieme ísť len v jednom poradí, a bez cyklov sa nemôžu ani opakovať. Samozrejme, nie každá podmnožina vrcholov tvorí platnú cestu, ale každá platná cesta je určená podmnožinou, dáva nám teda horný odhad. Počet podmnožín n vrcholov je $O(2^n)$, toľko ciest teda v najhoršiom prípade nájdeme. Každá cesta má najviac n vrcholov, a v každom vrchole prejdeme zvyšné vrcholy aby sme skúsili cestu predĺžiť, náš odhad časovej zložitosti je teda $O(n^2 2^n)$, a pamäťovej $O(2^n)$.

Najdrahšia cesta v DAGu

Poďme teda vymyslieť niečo prefíkanejšie ako skúšanie všetkých možností. Počet ciest v našom grafe je exponenciálny, ale nás z nich zaujíma len málo z nich – v ďalších dvoch sadách dokonca len jedna, tá najdrahšia.

Skúsme teda nájsť najdrahšiu cestu tak, že si pre každý vrchol zistíme najdrahšiu cestu, ktorá končí v ňom – naša celková najdrahšia cesta musí končiť v niektorom vrchole, bude teda jednou z nich.

Môžeme si napríklad všimnúť, že ak máme cestu prechádzajúcu vrcholmi $v_1, v_2, \cdots, v_{d-1}, v_d$ kde d je jej dĺžka, tak ak je to najdrahšia cesta končiaca vo v_d, tak cesta $v_1, v_2, \cdots, v_{d-1}$ je najdrahšia cesta končiaca vo v_{d-1}. Ak by totiž bola iná cesta ktorá je drahšia a končí vo v_{d-1}, tak by sme ju mohli napojiť na v_d a získať drahšiu cestu ako tú s ktorou sme začali, čo je v rozpore s tým že sme začali s najdrahšou cestou končiacou vo v_d.

Z tohto pozorovania vyplýva, že na nájdenie najdrahšej cesty končiacej vo vrchole v stačí nájsť najdrahšie cesty končiace vo vrcholoch, z ktorých sa vieme priamo dostať do v, a najdrahšia cesta bude jedna z nich plus cena hrany, ktorou sa na ňu napojíme.

Stačí nám teda prejsť vrcholy vo vhodnom poradí, aby sme vždy zrátali cenu najdrahšej cesty pre nejaký vrchol v skôr, ako ju budeme rátať pre vrcholy do ktorých sa vieme dostať z v. Keďže vieme, že vedúci môže odovzdať priečinok len takému vedúcemu, ktorý príde do KSP až po ňom, môžeme si vedúcich zoradiť podľa roku príchodu, a prejdeme ich v tomto poradí.

Nakoniec vypíšeme cenu najdrahšej cesty, ktorú sme takto našli.

Listing programu (C++)

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

using namespace std;

struct veduci {
    long long pride, odide, v_pride, v_odide;
    long long radost = 0;
};
```

```
bool podla_prichodu(veduci a,veduci b)
{
    return a.pride < b.pride;
}

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

    // nejdeme riesit k>1
    if(k>1) return 0;

    vector<veduci> v(n);
    for(int i=0;i<n;++i)
        cin >> v[i].pride >> v[i].odide >> v[i].v_pride >> v[i].v_odide;

    sort(v.begin(),v.end(),podla_prichodu);

    for(int o=0;o<v.size();++o) {
        for(int p=o+1;p<v.size();++p) {
            if(v[o].odide == v[p].pride && v[p].v_pride > v[o].v_odide)
                v[p].radost = max(v[p].radost, v[o].radost + (v[p].v_pride - v[o
                    ↪ ].v_odide) );
        }
    }

    long long best = 0;
    for(int i=0;i<n;++i)
        best = max(best,v[i].radost);

    cout << best%1000000007 << "\n";

}
```

Keďže pre každého vedúceho prejdeme každého, ktorý prišiel po ňom, budeme mať časovú zložitosť $O(n^2)$. Pre každého si okrem vstupných hodnôt pamätáme len jednu premennú navyše – najdrahšiu cestu končiacu v ňom – pamäť bude $O(n)$.

Najdrahšia cesta v priečinkovom DAGu

V predošlom riešení skúšame veľa nezmyselných dvojíc vedúcich, aj takých, ktorí si nevedia odovzdať priečinok (lebo prvý neodchádza v roku, v ktorom druhý prichádza).

Rozdeľme si teda vedúcich na kôpky – nech P_r je zoznam vedúcich ktorí prichádzajú v nejaký rok r, a O_r je zoznam vedúcich ktorí odchádzajú v rok r. Každý vedúci sa vyskytne raz na zozname prichádzajúcich, a raz na zozname odchádzajúcich.

Ak chceme prechádzať vedúcich podľa roku príchodu, prejdeme kôpky vedúcich v P pre postupne stúpajúce roky r (len tie, ktoré boli na vstupe). Potom ich skúšame napojiť na najdrahšie cesty končiace vo vedúcich v O_r, keďže priečinky vedia dostať len od nich.

Stále však môžeme mať až kvadraticky veľa dvojíc – napríklad ak polovica vedúcich odchádza v jednom roku, a druhá polovica vedúcich v tomto roku prichádza.

Skúsme teda namiesto toho, aby sme pre každého prichádzajúceho vedúceho v skúšali všetkých vedúcich, ktorí odchádzajú v rovnaký rok (a majú menšiu hodnotu priečinka) a napojiť sa na ich cestu, priamo určiť, ktorá z nich bude najdrahšia po napojení na v.

Zoberme si teda príkladového vedúceho v, ktorý príde a bude mať hodnotu priečinka p, a majme dve cesty na ktoré sa vieme napojiť – jednu s hodnotou h, končiacu vo vedúcom ktorý ešte využíva svoj priečinok na o, a druhú s hodnotami H, O.

Akú hodnotu budú mať tieto cesty po napojení na v? Prvá cesta bude mať $h+(p-o)$, a druhá $H+(p-O)$.

Aby bola prvá cesta drahšia ako druhá, $h + (p - o) > H + (p - O)$, tak $h - o > H - O$. Stačí teda zobrať cestu, ktorá má túto hodnotu (určenú jej hodnotou a vedúcim, v ktorom končí) najväčšiu, a tú napojiť na vedúceho v. Všimnime si, že na ktorú cestu je najlepšie sa napojiť nezávisí od využitia p, ktorý pre priečinok má prichádzajúci vedúci – jediné čo musíme dodržať je, aby cesty ktoré uvažujeme končili vo vedúcich, ktorí sú mu ochotní priečinok odovzdať.

Usporiadajme si teda vedúcich ktorí prichádzajú a odchádzajú v daný rok, podľa ich využitia priečinka, a prejdime ich v tomto poradí. Udržiavame si pritom vyššie popísanú hodnotu najlepšej doposiaľ nájdenej cesty. Keď prídeme na odchádzajúceho vedúceho, pozrieme sa či cesta, ktorá v ňom končí nie je lepšia ako tá čo sme mali doteraz, a ak áno, prepíšeme si ju. Keď prídeme na prichádzajúceho vedúceho, priradíme mu hodnotu najlepšej cesty, plus jeho využitie priečinka v prváku, tiež ako popísané vyššie.

Takto vieme nájsť najdrahšiu cestu končiacu v každom vedúcom, pričom však nepozeráme na všetky možné odovzdávanie medzi nimi, pozrieme sa len na každého vedúceho práve raz.

Listing programu (C++)

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

using namespace std;

struct veduci {
    long long pride, odide, v_pride, v_odide;
    long long radost = 0;
};

bool podla_prichodu(veduci a,veduci b)
{
    return a.pride < b.pride;
}

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

    if(k>1) return 0;

    vector<veduci> v(n);
    for(int i=0;i<n;++i)
        cin >> v[i].pride >> v[i].odide >> v[i].v_pride >> v[i].v_odide;

    map<int,vector<int> > pridu,odidu;
    for(int i=0;i<n;++i)
    {
        pridu[ v[i].pride ].push_back(i);
        odidu[ v[i].odide ].push_back(i);
    }

    for(auto it=odidu.begin();it!=odidu.end();it++)
    {
        int kedy = it -> first;
        vector<int> kto_odide = it -> second;

        if(!pridu.count(kedy)) continue;
        vector<int> kto_pride = pridu[kedy];


        int P = 0, O = 0, ps=kto_pride.size(), os=kto_odide.size();
```

```
        // usporiadame si odchadzajucich podla hodnoty priecinka pri ich odchode
        vector<pair<int,int> > odchadzajuci;
        for(int i=0;i<os;++i)
            odchadzajuci.push_back({v[ kto_odide[i] ].v_odide,kto_odide[i]});
        sort(odchadzajuci.begin(),odchadzajuci.end());

        // aj prichadzajucich
        vector<pair<int,int> > prichadzajuci;
        for(int i=0;i<ps;++i)
            prichadzajuci.push_back({v[ kto_pride[i]].v_pride,kto_pride[i]});
        sort(prichadzajuci.begin(),prichadzajuci.end());

        // nase 'H-O'
        long long najlepsi_priecinok = -(1e10);
        // spracujeme vsetkych prichadzajucich veducich
        // dvoma bezcami, aby sme spracovali veducich podla hodnoty priecinka
        // pri prichode/odchode
        while(P<ps)
        {

            if(O == os || odchadzajuci[O].first >= prichadzajuci[P].first)
            {
                // veduci prichadza, zratame prenho najlepsiu put
                int kto = prichadzajuci[P].second;
                v[kto].radost = max(0LL, najlepsi_priecinok + v[kto].v_pride);

                ++P;

            }
            else {
                // veduci odchadza, preratame si najlepsi priecinok
                int kto = odchadzajuci[O].second;
                najlepsi_priecinok = max(najlepsi_priecinok, v[kto].radost - v[
                    ↪ kto].v_odide);

                ++O;
            }
        }

    }

    long long best = 0;
    for(int i=0;i<n;++i)
        best = max(best,v[i].radost);

    cout << best%1000000007 << "\n";

}
```

V prípade, že sa všetci vedúci vyskytnú v jeden deň, nám ich usporiadanie zaberie $O(n \log n)$. Pamäť ostane na $O(n)$.

A teraz už len trochu všeobecnejšie...

Čo potrebujeme, aby sme toto riešenie rozšírili ak nás nezaujíma len najdrahšia cesta, ale k najdrahších ciest? Pôjdeme na to rovnako – pre každý vrchol si zapamätáme k najdrahších ciest, ktoré v ňom končia. Tu využijeme podmienku zo zadania, že $n \cdot k \leq 10^6$, čiže síce vedia byť n aj k veľké, nemôžu byť veľké naraz, a vieme si tieto cesty zapamätať – bude ich najviac $n \cdot k$.

Analogicky, aby sme našli k najdrahších ciest končiacich v danom vrchole, musíme skúsiť k najlepších

ciest ktoré sa na neho môžu napojiť (ktorých hodnota mínus využitie priečinka vedúceho, v ktorom končia, je najvyššia).

Namiesto jednej hodnoty najlepšej cesty, si teda budeme pamatať usporiadanú množinu najlepších ciest. Budeme prechádzať vedúcich rovnako ako v predošlom riešení. Keď spracujeme prichádzajúceho vedúceho, vypočítame jeho najlepších (najviac) k ciest ktoré v ňom končia. Keď spracujeme odchádzajúceho vedúceho, hodnoty jeho ciest pridáme do našej množiny, a ak ich v nej je viac ako k, zahodíme tie najhoršie.

Počas spracovania prichádzajúcich vedúcich, ktorých je dokopy n, vypočítame najviac k najdrahších ciest ktoré v nich končia, spravíme teda $O(nk)$ operácií. Počas spracovania odchádzajúcich vedúcich, ktorých je tiež dokopy n, vypočítame hodnoty ciest ktoré v nich končia, a pridáme ich do našej usporiadanej množiny. Ak použijeme rozumnú dátovú štruktúru (napr. v C++ `multiset`), pridávanie hodnoty do, a odstraňovanie najmenšej hodnoty z nej nás bude stáť $O(\log k)$ operácií, ak jej veľkosť je $O(k)$. Za každého odchádzajúceho vedúceho teda spravíme v najhoršom prípade $O(k \log k)$ práce, dokopy teda $O(nk \log k)$. Spolu s usporiadaním vedúcich aby sme ich prešli v správnom poradí teda spravíme $O(n(\log n + k \log k))$ operácií.

Pamäťová zložitosť bude $O(nk)$, keďže si pre každého vedúceho pamätáme najviac k hodnôt ciest.

Listing programu (C++)

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

struct veduci
{
    long long pride, odide, v_pride, v_odide;
    int id;
    vector<long long> pute = {0};
};

struct udalost{

    bool prichadza;
    int id;
    long long v;
};

bool cmp_udalost(udalost a,udalost b)
{
    if(a.v!=b.v)
        return a.v < b.v;
    if(a.prichadza != b.prichadza)
        return a.prichadza > b.prichadza;

    return a.id < b.id;
}

long long solve()
{

    int n,k;
    cin >> n >> k;
    vector<veduci> v(n);

    for(int i=0;i<n;++i)
    {
         v[i].id = i;
        cin >> v[i].pride >> v[i].odide >> v[i].v_pride >> v[i].v_odide;
    }

    unordered_map<int, vector<int> > pridu, odidu;
```

```
    set<int> roky;

    for(int i=0;i<n;++i)
    {
        roky.insert(v[i].pride);
        pridu[ v[i].pride ].push_back(i);
        odidu[ v[i].odide ].push_back(i);
    }

    // prejdeme veducich, ktori pridu/odidu v dany rok
    for(int day : roky)
    {
        if(odidu.count(day)==0) continue;

        vector<int> prichadzajuci = pridu[day];
        vector<int> odchadzajuci = odidu[day];

        // usporiadame si udalosti (relevantny veduci prichadza/odchadza)
        // do poradia podla hodnoty priecinka daneho veduceho pri
        // prichode/odchode
        vector<udalost> U;

        for(int x : prichadzajuci)
        {
            udalost e;
            e.id = x;
            e.v = v[e.id].v_pride;
            e.prichadza = true;

            U.push_back(e);
        }

        for(int x : odchadzajuci)
        {
            udalost e;
            e.id = x;
            e.v = v[e.id].v_odide;
            e.prichadza = false;

            U.push_back(e);
        }

        sort(U.begin(),U.end(),cmp_udalost);
        multiset<long long> najlepsie;

        // prejdeme udalosti
        for(udalost u : U)
        {
            // ak veduci prichadza, zratame mu (najviac) k najlepsich puti
            if(u.prichadza)
            {
                for(auto it: najlepsie)
                {
                    v[u.id].pute.push_back(it+u.v);
                }
            }
            else
```

```
            {
                // preratame moznosti na najlepsie priecinky
                for(long long p: v[u.id].pute)
                {
                    najlepsie.insert(p-u.v);
                }

                // ak ich mame viac ako k, zahodime najhorsie
                while(najlepsie.size() > k)
                {
                    najlepsie.erase(najlepsie.begin());
                }
            }
        }
    }

    long long ans = 0;

    vector<long long> pute;

    for(int i=0;i<n;++i)
    {
        for(auto x: v[i].pute)
            pute.push_back(x);
    }

    sort(pute.begin(),pute.end(),greater<long long>());

    for(int i=0;i<k && i<pute.size();++i)
        ans += pute[i];

    return ans%1000000007;
}

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

    cout << solve() << "\n";
    return 0;
}
```