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

Vzorové riešenia 1. kola letnej časti

kristina

1. Kristínine ponožtičky

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

Ako už vidno zo zadania, sú iba dve možnosti ako môže Kristína prejsť po schodoch – buď začne ľavou nohou, alebo pravou. Stačí nám teda vypočítať špinavosti oboch možností a vybrať tú menšiu.

Špinavosť pre pravú nohu spočítame tak, že v cykle budeme striedavo pripočítavať ku výslednej špinavosti pravú a ľavú časť schodu. Špinavosť pre ľavú nohu vypočítame obdobne, len začneme ľavou časťou schodu.

Všimnite si, že k vyriešeniu tejto úlohy nepotrebujeme pole – jednotlivé špinavosti vieme spočítavať už pri načítavaní vstupu.

Časová zložitosť je $O(n)$ - pri sčítavaní špinavostí v cykle prejdeme všetkých n schodov. Ak sme nepoužili polia, stačili nám dve premenné so špinavosťami a teda pamäťová zložitosť je $O(1)$. S použitím polí je pamäťová zložitosť $O(n)$.

Listing programu (Python)

```
n = int(input())

sum_lava, sum_prava = 0, 0

for i in range(n):
    l, p = [int(x) for x in input().split()]
    sum_lava += l if i % 2 == 0 else p
    sum_prava += p if i % 2 == 0 else l

if sum_prava > sum_lava:
    print("lava")
elif sum_lava > sum_prava:
    print("prava")
else:
    print("je to jedno")
```

Listing programu (C++)

```
#include <iostream>

using namespace std;

int main() {
    int n;

    int sum_lava = 0;
    int sum_prava = 0;

    cin >> n;

    int l, p;
    for (int i = 0; i < n; i++) {
        cin >> l >> p;
        sum_lava += i % 2 == 0 ? l : p;
```

```
        sum_prava += i % 2 == 0 ? p : 1;
    }

    if (sum_lava > sum_prava) {
        cout << "prava" << endl;
    } else if (sum_prava > sum_lava) {
        cout << "lava" << endl;
    } else {
        cout << "je to jedno" << endl;
    }

    return 0;
}
```

emmika

2. Romantické výhľady

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

Vyhliadky, náučný chodník a okruh, po ktorom može Kika s Aďom chodiť, až do vysilenia. Aj o tom bola táto úloha a my sa teraz pozrieme na nejaké jej riešenia.

Bruteforce

V prvom rade si však ujasnime jednu vec. Náučný chodník je síce okruh, ale aj tak si ho budeme reprezentovať ako obyčajné pole. Akurát, vždy, keď sa budeme pozerať na prvú (resp. poslednú) vyhliadku, tak za jej prvú ľavú (resp. pravú) susednú vyhliadku budeme považovať poslednú (resp. prvú) vyhliadku.

Aby sme sa však vyhli problémom s prvou a poslednou vyhliadkou, ich susedmi a ich indexami v našom poli, tak si jednoducho celé pole zapamätáme dvakrát za sebou (skopírujeme si danú postupnosť vyhliadok a vložíme ju za už zapamätanú postupnosť). Takže vo výsledku budeme mať hneď za posledným prvkom prvý.

Prvá myšlienka, ktorá nám môže napadnúť je vyskúšať všetky možnosti trojíc vyhliadok a zistiť, či pre ne platia Kikine podmienky.

Teda postupne si zvolíme jednotlivé výhliadky tak, že za prvú dosadíme postupne všetky možnosti, za tretiu tak isto a za prostrednú zvolime niektorú z vyhliadok medzi prvou a druhou. Pre túto trojicu musí platiť to, že dve krajné vyhliadky z tejto trojice majú súčet ich vzdialeností k prostrednej najmenší možný.

Môžeme teda ísť cez všetky takéto trojice a pamätať si, akú najlepšiu trojicu sme zatiaľ videli. Keďže sme si pole skopírovali, stačí nám prechádzať cez trojice indexov (i, j, k) pre ktoré platí $0 \leq i \leq j \leq k < 2n$. Súčet vzdialeností k prostrednej je potom $k - i$. Na konci vypíšeme skutočné indexy najlepšej nájdenej trojice. Všimnite si, že na to, aby sme z upravených pozícií vyhliadok dostali ich skutočné pozície, stačí nám zobrať zvyšok po vydelení n.

Takže, čo sa týka zložitostí: jednorázovo kopírujeme vstupné n-prvkové pole a skúšame overovať všetky trojice, ktorých je rádovo n^3. Z tohoto nám vyjde časová zložitosť $O(n^3)$. Pamäťová bude $O(n)$ pretože si pamätáme len jedno pole, ktoré má $2n$ prvkov (ale 2 je konštanta, takže tú zanedbáme).

Listing programu (Python)

```
n = int(input())
s = [int(i) for i in input().split()]

spots = []
for i in range(2*n):
    spots.append(s[i%n])

highest = -1
index = -1

start = -1
stop = -1

done = False
```

```
for i in range(2 * n):
    for j in range(2 * n):
        for k in range(i+1, j):
            if j - i == 2 and spots[i] < spots[k] > spots[j]:
                highest = spots[k]
                index = k % n
                start = i % n
                stop = j % n

                done = True
                break
        if done:
            break
    if done:
        break

print(start, index, stop)
```

Niečo lepšie

Skúšanie všetkých trojíc nie je úplne optimálne. Ak by sme však vedeli povedať niečo viac o tých troch vyhliadkach, tak by nám to pomohlo. Vieme že prostredná z nich musí byť vyššia ako jej susedné. To je ale len jej vzťah k jej susedným vyhliadkam a nehovorí to nič o vzťahu k ostatným vyhliadkam. Čiže môže mať kľudne tretiu najmenšiu nadmorskú výšku a s jej susednými spĺňať Kikine podmienky, ale môže to byť kľudne aj najvyššie položená vyhliadka s jej susedmi. A od tohoto sa odrazíme.

Dôležité pozorovanie je, že výšky vyhliadok sú navzájom *rôzne*. A tým pádom, ak zoberieme najvyššiu vyhliadku, jej susedné vyhliadky (z každej strany) musia byť nutne nižšie. Takáto trojica má súčet rozdielu vzdialeností 2, a lepší súčet dosiahnuť nemôžme (zamyslite sa).

Takže nám stačí nájsť vyhliadku s najväčšou nadmorskou výškou a vziať jej susedov. Ako ju nájsť?

Ak by sme mali vyhliadky zoradené podľa výšky vzostupne, tak by to bola tá posledná. Ale ako potom nájdem jej najbližšie susedné vyhliadky, keď zmením ich poradie, tým že ich zoradím? Budem si jednoducho pamätať dve polia – jedno bude kópia druhého, ale bude zoradené podľa nadmorských výšok. Zoradiť ho môžeme pomocou nejakého triediaceho algoritmu[1] alebo jednoducho využijeme funkciu `sort`, ktorá je už vo väčšine jazykov implementovaná.

Teraz najvyššiu výhliadku nájdeme v pôvodnom, nezoradenom poli a vezmeme jej susedov. Tu nemusíme nejak špeciálne riešiť prípad, že prostredná je prvá alebo posledná v poli a teda jej najbližší sused je na opačnom konci pola. Skrátka jednoducho pridáme podmienky, ktoré nám vrátia presných susedov na základe pozície najvyššej vyhliadky.

Časová zložitosť tohoto riešenia bude $O(n \log n)$ – najzložitejšia časť v tomto riešení je triedenie, ktorého zložitosť je $O(n \log n)$. Potom už len hľadáme pozíciu najvyššej vyhliadky v pôvodnom poli, čo bude v najhoršom prípade $O(n)$, ale to je oproti $O(n \log n)$ zanedbateľné. Pamäťová zložitosť
bude $O(n)$, lebo si pamätáme iba 2 polia dĺžky n.

Listing programu (Python)

```
n = int(input())
spots = [int(i) for i in input().split()]

spotsCopy = spots.copy()
spotsCopy.sort()

highestIdx = spots.index(spotsCopy[-1])

if highestIdx + 1 >= len(spots):
    print(highestIdx-1, highestIdx, 0)
elif highestIdx - 1 < 0:
    print(len(spots)-1, 0, 1)
```

[1] https://www.ksp.sk/kucharka/triedenie/

```
else:
    print(highestIdx-1, highestIdx, highestIdx+1)
```

Optimálne riešenie

Keď sa lepšie pozrieme na predošlé riešenie, tak zistíme, že pri prechode poľa s nadmorskými výškami nemusíme iba hľadať pozíciu najvyššej vyhliadky, ale aj zisťovať, ktorá výhliadka je najvyššia. Vďaka tomu sa môžeme úplne zbaviť zoraďovania vyhliadok a zrýchliť naše riešenie.

Takže stačí nám raz prejsť pole zo vstupu a postupne hľadať najvyššiu vyhliadku. Budeme mať nejaké dve premenné, v ktorých si budeme v každom momente pamätať doteraz najvyššiu vyhliadku a jej pozíciu. Čiže pri prechode poľa, vždy keď nájdeme vyhliadku, ktorá je vyššie ako tá, ktorú si pamätáme ako doteraz najvyššiu, tak tie dve premenné prepíšeme. No a na konci budeme mať v premenných najvššiu vyhliadku aj s jej pozíciou.

Už nám stačí iba nájsť najbližšie susedné vyhliadky k najvyššej a vypísať ich.

Keďže algoritmus iba raz prejde n-prvkové pole, časová zložitosť bude $O(n)$ a pamätáme si tiež len toto pole a konštatný počet premenných, takže pamäťová zložitosť bude $O(n)$.

Listing programu (Python)

```
n = int(input())
spots = [int(i) for i in input().split()]

highest = max(spots)
index = spots.index(highest)

if index + 1 >= len(spots):
    print(index-1, index, 0)
elif index - 1 < 0:
    print(len(spots)-1, 0, 1)
else:
    print(index-1, index, index+1)
```

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

3. Inotaj

Veľmi pomalé riešenie

Na začiatok je dobré si uvedomiť, že inotaj je množina slov, čiže na poradí slov v inotaji *nezáleží*. Prvý spôsob riešenia, ktorý mnohým z vás mohol napadnúť je vyskúšať všetky možnosti. Postupovali by sme postupne zhora dole, čiže najprv by sme prešli riešenie zahŕňajúce všetkých n slov, potom všetky $n-1$ slovné riešenia, $n-2$ slovné riešenia atď. Prvé riešenie, ktoré by v aspoň jednom znaku prešlo kontrolou (t. j. súčet výskytov jedného zo znakov je aspoň polovica všetkých znakov) by bol náš inotaj a vypísali by sme počet slov ktoré obsahuje.

Toto riešenie má ale veľmi pomalú časovú zložitosť: $O(nk2^n)$. To je zapríčinené tým, že počet všetkých podmnožín n-prvkovej množiny je 2^n, a pre každú takúto množinu musíme prejsť najviac n slov, každé dĺžky najviac k. Pamäťová zložitosť tohto programu je $O(nk)$.

Na čo sme prišli

Zaujímavé pozorovanie, ktoré nám s úlohou pomôže je nasledovné: niektoré slová nám "kazia" kontroly a niektoré nie. Vlastne, pre konkrétny znak x, nám kontrolu, či znak x tvorí aspoň polovicu znakov inotaja, kazia slová v ktorých x tvorí menej než polovicu znakov. Mohli by sme sa teda pozrieť na jednotlivé písmená samostatne, a roztriediť si slová, podľa toho či nám kontrolu kazia, alebo nie (pre každé písmeno osobitne). Najskôr si nejako kvantifikujme "kazenie."

Ako správne vyriešiť

Naše úvahy nás doniesli do momentu, kde sme si uvedomili, že je veľmi dôležitý pomer súčtu výskytu písmena v slove ku súčtu všetkých znakov slove. Pomer nám však dáva len zlomok, percentuálne vyjadrenie. Keby že porovnávame slová podľa toho ako veľmi sa nám oplatí ich zobrať, dlhšie slovo s pomerom pod 50% nám pokazí kontrolu viac ako kratšie slovo s rovnakým, niekedy aj horším pomerom.

Radšej si preto vypočítame, koľko výskytov písmena x je v slove nad polovicou celkového súčtu jeho znakov. Ak je pod polovicou, tento koeficient (c) bude záporný. Potom vieme písmeno po písmene pažravo brať slová s najlepšími hodnotami c čo sme si vypočítali. To sa dá implementovať pomocou funkcie `sort`; pre každé písmeno si zoradíme slová podľa $c_1, ...c_k$ od najlepšieho po najhorší. Následne postupne berieme slová, kým súčet $c_1, ...c_i$ nie je menší ako 0. Toto aplikujeme pre každé písmeno. Na výstup vypíšeme maximum slov, ktoré sme zobrali, čiže hodnotu i.

Prečo to funguje

Ukážme si teraz, že ak existuje inotaj dĺžky L kde má písmeno x nadpolovičnú väčšinu výskytov, tak náš algoritmus určite nájde aspoň taký dlhý inotaj, kde bude mať písmeno x tatiež nadpolovičnú väčšinu výskytov. Predstavme si, že náš algoritmus nájde najdlhší inotaj s nadpolovičným výskytom x dĺžky L' pozostávajúci zo slov $a_1, a_2, ..., a_{L'}$, pričom slová sme si zoradili podľa koeficientu, tak že a_L má najnižší koeficient (vzhľadom na písmeno x).

Predstavme si, že by existoval nejaký inotaj dĺžky $L > L'$ pre toto písmeno, a povedzme že $b_1, b_2, ..., b_{L'}, ..., b_L$ je taký inotaj, ktorý má navyše najväčší prienik[2] ako inotaj náš algoritmus našiel (slová sme zoradili podľa koeficientu tak ako predtým).

Najskôr uvažujme, čo by sa stalo ak existuje slovo a_i v nájdemom inotaji, ktoré zároveň nie je v dlhšom inotaji. Potom, keďže algoritmus berie slová od najväčšieho koeficientu kým môže, musí existovať medzi slovami dlhšieho inotaju slovo s menších koeficientom, b_j. Všimnite si, že keď vymeníme slovo b_j za a_j, dostaneme stále inotaj! Ale to je v rozpore s tým, ako sme dlhší inotaj vybrali. Tento prípad teda nemôže nastať.

Teda musí platiť, že všetky slová v nájdenom inotaji boli aj tomto dlhšom inotaji. Potom zoberme si prvé slovo ktoré už algoritmus nezobral do inotaju. Kvôli tomu, že sme si slová vzali pažravo, toto slovo má aspoň taký koeficient ako b_L, takže jedno pridaním by sme stále získali inotaj. Ale toto je v rozpore s tým, že si už algoritmus toto slovo nezobral.

Takže ani jeden s týchto prípadov nemôže nastať, a náš algoritmus vždy nájde nejaký najdlhší inotaj.

Zložitosti

Vypočítanie koeficientov pre každé slovo pre každé písmeno trvá $O(5nk) = O(nk)$. Utriedenie koeficientov pre každé písmeno trvá $O(5n\log(n)) = O(n\log(n))$. Pažravé branie v usporiadaných poliach trvá $O(5n) = O(n)$. Toto riešenie má teda celkovú časovú zložitosť $O(nk) + O(n\log(n)) + O(n) = O(nk + n\log n)$. Ak na triedenie použijeme counting sort[3], časová zložitosť sa zníži na $O(nk)$.

Pamäťová zložitosť tohto programu je $O(nk)$.

Listing programu (Python)

```
n, k = [int(x) for x in input().split()]
v = [input() for _ in range(n)]

letters = "fkmsp"
ans = 0
for c in letters:
    benefits = [0] * n
    for i in range(n):
        benefits[i] = 2 * v[i].count(c) - len(v[i])
    benefits.sort(reverse=True)
    iter_ans, iter_sm = 0, 0
    for i in range(n):
        iter_sm += benefits[i]
        if iter_sm >= 0:
            iter_ans += 1
    ans = max(ans, iter_ans)
print(ans)
```

Listing programu (C++)

[2] počet rovnakých slov
[3] https://en.wikipedia.org/wiki/Counting_sort

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

int main() {
    int n,k;
    cin >> n >> k;
    char pismena[5] = {'f', 'k', 's', 'm', 'p'};
    int out = 0;

    vector<vector<int>> pocet
    {
        {},
        {},
        {},
        {},
        {},
    };

    string u;

    for (int w = 0; w < n; w++)
    {
        cin >> u;
        for (int i = 0; i < 5; i++)
        {
            pocet[i].push_back(0);
            for (int j = 0; j < u.size(); j++)
            {
                if (u[j]==pismena[i]) pocet[i][w]++;
                else pocet[i][w]--;
            }
        }
    }

    for (int i = 0; i < 5; i++)
    {
        sort(pocet[i].begin(),pocet[i].end());
        reverse(pocet[i].begin(),pocet[i].end());
        int h = 0, g = 0, f = 0;
        for (int j = 0; j < n; j++)
        {
            h += pocet[i][j];
            if(h<0) {
                g=1;
                f=j;
                break;
            }
        }
        if (g==0) f=n;
        out = max(out, f);
    }

    cout << out << endl;
```

```
    return 0;
}
```

mirko

4. Småland

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

Táto úloha bola grafový problém. Ak ste o grafoch[4] ešte nepočuli, v Kuchárke KSP[5] je k nim pekný úvod.

V našej úlohe sú teda ostrovy *vrcholmi* grafu a mosty medzi nimi *hranami* grafu. Potom skupinky ostrovov budú *komponenty súvislostí* a špeciálne ostrovy sú vrcholy stupňa 1, ktoré tiež niekedy nazývame *listy* grafu.

Len pre doplnenie, graf si môžete predstaviť ako “sieť”, kde vrcholy sú body spájané čiarami, čo sú hrany. Komponent súvislostí je potom taká skupinka vrcholov, v ktorej vieme po hranách prejsť z každého vrcholu do každého. Nakoniec, stupeň vrchola je počet hrán, ktoré sa na neho napájajú (odborne povieme, že tieto hrany sú s ním *incidentné*).

V úlohe sme začínali s n izolovanými vrcholmi, pričom každý z nich je samostatný komponent súvislostí s jedným vrcholom.

Ak sme niekedy dostali query na stavbu mostu `! a b`, vytvorili sme v podstate hranu medzi nejakými vrcholmi a a b, čím sme spojili ich dva komponenty súvislostí do jedného spoločného.

Ak sme dostali otázku typu `? x`, pýtali sme sa vlastne na počet listov v komponente, ktorý obsahuje vrchol x.

Jednoduché riešenie

Najjednoduchší spôsob, ako sa dala táto úloha riešiť bolo simulovať graf presne tak, ako vzniká. To znamená, že sme si pamätali všetky hrany grafu, a keď nám prišla query na vytvorenie hrany, tak sme ju pridali do grafu. Keď nám prišla otázka na počet listov, tak sme spustili z tohoto vrcholu nejaké prehľadávanie, napríklad DFS[6], ktorým sme zistili počet listov.

Pridávanie hrany do grafu vieme robiť v zložitosti $O(1)$. Keď zisťujeme počet listov, tak musíme v najhoršom prípade prehľadať celý graf, čo vieme urobiť v zložitosti $O(n + m)$ (kde n je počet vrcholov a m počet hrán). Pamäťová zložitosť je $O(n + m)$, keďže si potrebujeme pamätať celý graf.

Pomalšie riešenie

Iný spôsob, ako túto úlohu riešiť, ktorý je aj základ pre vzorové riešenie je tak, že sme rovno simulovali komponenty.

Komponenty sme si uchovávali v nejakom poli. Každý komponent si pamätá, aké vrcholy obsahuje a koľko z týchto vrchol je listov. Tiež si niekde uchovávame informáciu o tom, aký stupeň má každý vrchol.

Ak potom dostaneme inštrukciu na stavbu mostu medzi vrcholmi a a b, stačí nám vyhľadať, v ktorých komponentoch sa tieto vrcholy nachádzajú.

Ak sú v rovnakom komponente, nič sa nedeje, len im upravíme stupeň, pretože sme medzi nimi vytvorili hranu. Tiež, ak bol nejaký z nich list, tak už nie je, a teda komponentu upravíme informáciu o tomto.

Ak sú ale komponenty obsahujúce dané vrcholy odlišné, musíme ich zlúčiť, čiže chceme záznamy o vrcholoch z jedného komponentu premiestniť do druhého. Ostatné informácie o stupňoch vrcholov a listoch upravíme rovnako, ako v predchádzajúcom prípade.

Ak sa niekedy spýtame na počet listov v komponente nejakého vrcholu, stačí nám jednoducho vyhľadať, v ktorom komponente je daný vrchol a pozrieť sa do záznamov, koľko listov má daný komponent.

Ako ste si mohli všimnúť, najpomalšia operácia, ktorú tu vykonávame, je vyhľadávanie vrcholu v komponente. V tomto prípade však časová zložitosť tejto operácie záleží od konkrétnej implementácie a voľby dátovej štruktúry reprezentujúcej komponenty alebo vrcholy.

Vyššie navrhnuté riešenie s použitím polí pre komponenty by malo časovú zložitosť vyhľadávania vrcholu $O(n)$.

Keďže si uchovávame v poli zoznam komponentov a pre každý komponent zoznam ich vrcholov, pamäťová zložitosť bude $O(n)$, pretože pokiaľ premiestňujeme vrcholy medzi poľami, uchovávať budeme stále n vrcholov.

Za takéto a podobne rýchle riešenia ste mohli získať 2 body.

V prípade, že ste robili to, že ste vždy spájali menší komponent do väčšieho, dá sa ukázať, že takéto spájanie komponentov má amortizovanú časovú zložitosť $O(\log n)$.

[4] Grafmi myslíme dátové štruktúry, nie grafy funkcií, ktoré ste mali na matematike

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

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

Niečo trochu lepšie

V predchádzajúcom riešení sme mali problém s pomalým prehľadávaním a spájaním komponentov.

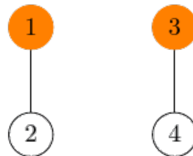
O komponentoch súvislostí však vieme, že sú navzájom disjunktné[7], a že spojenie dvoch komponentov urobíme prepojením dvoch vrcholov, ktoré sú v nich obsiahnuté.

Toto vieme využiť na to, že komponenty už nebudeme reprezentovať ako polia vrcholov, ale ako stromy. Opäť, ak ste sa ešte so stromami nestretli, krátky úvod nájdete znova v Kuchárke KSP[8].

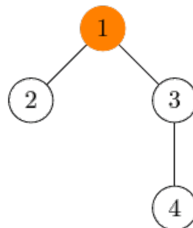
Na začiatku máme izolované vrcholy, čiže jednovrcholové komponenty súvislostí zodpovedajúce stromom s jedným vrcholom:

Ak spojíme vrcholy 1 a 2 (vykonáme `! 1 2`), spojíme aj ich stromy, čiže len jeden zavesíme na druhý:

Spravme to isté pre vrcholy 3 a 4:



Teraz ak chceme spojiť napríklad vrcholy 2 a 4 do jedného komponentu, mohli by sme ich na seba zavesiť ako chceme, ale najlepšie to bude urobiť takto:



Ako asi vidíte, dva vrcholy sú v rovnakom komponente, ak majú spoločný koreň^ [Najvyšší vrchol v strome, nemá rodičovský vrchol] (označený oranžovo na obrázkoch vyššie). Tiež ak chceme zistiť, v akom komponente sa vrchol nachádza, nepotrebujeme už prehľadávať všetky komponenty, stačí nám len sledovať, akého rodiča má daný vrchol a sledovať jeho rodiča rodiča a tak ďalej, až kým neprídeme ku koreňu, ktorý *reprezentuje* náš komponent.

Pre každý vrchol nám potom stačí uchovávať si len jeho rodiča, pričom korene, teda vrcholy, ktoré nemajú rodičov, budú rodičmi samým sebe.

Ak teraz budeme chcieť spojiť nejaké dva komponenty hranou medzi vrcholmi a a b (`! a b`), jednoducho budeme sledovať rodičov každého vrcholu, až kým sa nedopracujeme ku koreňu (volajme ho *reprezentant*).

Ak je reprezentant rovnaký, oba vrcholy sú v rovnakom komponente. Ak sú reprezentanti odlišní (nazvime si ich r_a a r_b), môžeme napríklad koreňu r_a priradiť ako rodiča r_b, čiže nový veľký komponent bude mať r_b ako reprezentanta.

Informácie o stupňoch vrcholov si uchovávame rovnako, ako v predchádzajúcom riešení. Aktuálny počet listov daného komponentu bude uložený pre daného reprezentanta.

Ak sa potom spýtame na počet listov pre komponent nejakého vrcholu, stačí nám nájsť jeho reprezentanta a pozrieť sa v nejakom poli na zodpovedajúce miesto.

To, čo sme tu objavili, sa volá *Union-find* alebo *Disjoint-set data structure*, resp. jeho *naivná implementácia* (v ďalšej sekcii si ukážeme, ako môže vyzerať sofistikovanejšia implementácia).

Keďže vyhľadávame v stromoch, časová zložitosť vyhľadávania bude prinajhoršom $O(n)$, keď máme len jeden komponent a všetky vrcholy v ňom majú presne jedného potomka. Potom ten strom vyzerá viac ako stožiar, a teda musíme prejsť všetkých n vrcholov, aby sme sa dostali až k reprezentantovi.

Tu sme si teda rýchlosťou veľmi nepomohli oproti predchádzajúcemu riešeniu, avšak voľba union-find dátovej štruktúry nám dáva možnosť pridať rôzne optimalizácie, ktoré to zrýchlia.

[7] Sú nezávislé, nemajú spoločný vrchol

[8] `https://www.ksp.sk/kucharka/grafy_uvod/#wiki-toc-zakorenene-stromy`

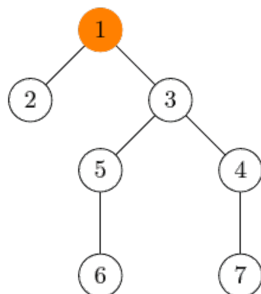
Časová zložitosť spájania komponentov, teda stavania mostov, bude tiež $O(n)$, pretože stále musíme hľadať reprezentantov. Časová zložitosť na *query* je teda lineárna.

Pamäťová zložitosť je stále $O(n)$ rovnako, ako v predchádzajúcom riešení, pretože stále nám na toto všetko stačia len polia.

Optimálne riešenie

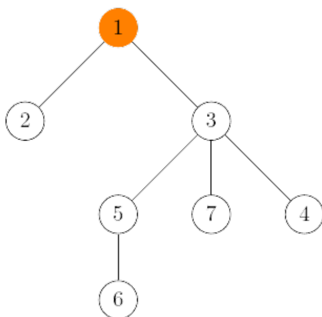
Dá sa to ešte lepšie? Áno, dá. Vďaka tomu, že si komponenty reprezentujeme ako union-find dátovú štruktúru, dokážeme urobiť nejaké optimalizácie.

Ak sa zamyslíte, zistíte, že komponenty s veľkým počtom vrcholov budú eventuálne reprezentované dosť hlbokými stromami. Napríklad, takto by mohol vyzerať strom reprezentujúci 7-vrcholový komponent:



Ak by sme teraz chceli nájsť reprezentanta tohto komponentu z vrcholu 7, trvalo by nám trochu dlhšie dopracovať sa až do oranžového koreňa. Viete si potom asi predstaviť, aké neefektívne by bolo prechádzať hlboký strom pre komponenty s niekoľkými tisícmi vrcholov.

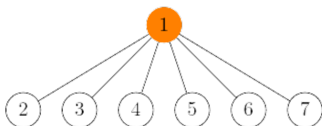
Tento problém ale vieme jednoducho vyriešiť. Ak napríklad budeme prechádzať strom od vrcholu 7, zároveň s hľadaním reprezentanta môžeme tiež prelinkovať prechádzaný vrchol na prarodiča[9] nášho vrcholu. V našom prípade by to mohlo vyzerať takto:



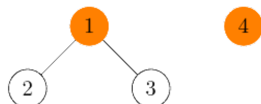
Odteraz ak niekedy budeme znovu potrebovať prechádzať strom od vrcholu 7, ušetríme si jeden krok a nič sme nepokazili, pretože nás zaujíma len príslušnosť vrcholu k reprezentantu, nie štruktúra stromu.

Tento proces sa nazýva *lazy path halving*. *Lazy* preto, lebo toto robíme len keď máme potrebu prechádzať strom z nejakého vrcholu. *Path halving* odkazuje zas na to, že vrcholy prelinkovávame na prarodičov, čiže "preskočíme" polovicu úrovní na ceste ku koreňu, a teda dĺžka cesty sa skráti na polovicu.

Vo všeobecnosti tento mechanizmus ale nazývame *path compression* a *path halving* je len jeho konzervatívnejšia forma. V podobnom duchu ale môžeme robiť aj extrémnejšie úpravy stromu:

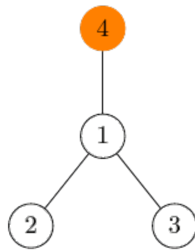


Ďalší problém, ktorý nám v praxi môže spomaliť naše riešenie je ten, že pri naivnej implementácii spájame komponenty len tak, že ku koreňu stromu jedného komponentu pripojíme koreň stromu druhého komponentu. Potom vymyslime si takéto dva stromy reprezentujúce komponenty:

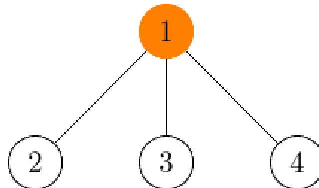


Ich zjednotenie naivnou implementáciou union-findu z predchádzajúceho riešenia bude vyzerať takto:

[9] Rodičovský vrchol rodičovského vrcholu



Tento nový strom má hĺbku 2, čiže na nájdenie reprezentanta by sme potrebovali spraviť dva kroky hore stromom. Pritom kľudne by sme vedeli postaviť aj efektívnejší strom s hĺbkou len 1:



Čo sa stalo? V naivnej implementácii napájame stromy na seba nemenne len jedným spôsobom a neberieme ohľad na to, ako bude vyzerať výsledok. V praxi potom naše stromy budú veľakrát vyzerať ako palice s veľkou hĺbkou, čiže opäť naša najdôležitejšia úloha hľadania reprezentanta bude prebiehať pomalšie.

Ideálne by sme ale chceli mať viac "rozkošatené" stromy s čo najviac vrcholmi na jednej úrovni, aby sme mali plytké stromy, v ktorých sa vieme rýchlo dostať do koreňa.

Jedna z možností, ako takúto vlastnosť zabezpečiť, je pripájať koreň menšieho stromu (s menším počtom vrcholov) ku koreňu väčšieho stromu. To vieme jednoducho implementovať napríklad tak, že si pre každý koreň uložíme, koľko má jeho strom vrcholov. Keď potom spájame dva stromy, len si porovnáme ich veľkosti a podľa toho sa rozhodneme, ako ich chceme spojiť.

Tejto technike sa hovorí *weighted union* alebo konkrétnejšie *union by size*.

Najhoršia časová zložitosť hľadania reprezentanta s *weighted union* bez *path compression* bude $O(\log n)$, pretože budeme hľadať vo vyváženom strome, v ktorom budeme musieť spraviť najviac toľko krokov hore, aká je jeho hĺbka.

Toto sa ale zmení, keď pridáme *path compression*. Kombinácia týchto dvoch mechanizmov potom bude mať amortizovanú časovú zložitosť $O(\log^* n)$, kde $\log^*$ je iterovaný logaritmus, ktorý vráti počet logaritmovaní, ktoré musíme aplikovať na pôvodnú hodnotu, kým nedostaneme výsledok menší alebo rovný 1. Môžete si to predstaviť tak, že nám vráti počet, koľkokrát musíme na kalkulačke stlačiť tlačidlo logaritmu, aby sme dostali výsledok menší alebo rovný 1.

Časová zložitosť tejto verzie union-findu sa zvykne ešte označovať aj ako $O(\alpha(n))$, kde $\alpha(n)$ je takzvaná *inverzná Ackermannova funkcia*[10], ktorá rastie podobne extrémne pomaly ako $\log^* n$. Keďže $\alpha(n) \leq 5$ pre $n < 2^{65536}$[11], tak prakticky pre akékoľvek vstupy dostávame konštantnú časovú zložitosť $O(1)$ na *query*.

Presná analýza časovej zložitosti sa v tomto prípade dosť komplikovane odvádza, ale pokiaľ by Vás to zaujímalo, môžete sa pozrieť napríklad sem[12].

Pamäťová zložitosť bude opäť rovnaká, ako v predchádzajúcich riešeniach, teda $O(n)$ pretože si stále vystačíme s niekoľkými lineárnymi poľami.

Za takéto a podobné riešenia využívajúce dostatočne rýchlu implementáciu union-findu ste mohli získať plný počet bodov.

Listing programu (Python)

```
"""
Union-find solution with path halving and weighted union

Should have O(alpha(n)) - near-constant amortized time complexity for union and
    ↪ find
O(n) space complexity
"""
```

[10] https://en.wikipedia.org/wiki/Ackermann_function#Inverse
[11] Len pre porovnanie, počet atómov v pozorovateľnom vesmíre je približne 2^{265}
[12] https://people.ksp.sk/~kuko/gnarley-trees/UnionFind.html#analysis

```python
def find(x: int) -> int:
    """
    Find the root vertex of a component the vertex `x` belongs to.
    Perform path halving along it.
    """

    # Traverse the union find tree
    while parents[x] != x:
        parents[x] = parents[parents[x]]
        x = parents[x]

    return x


# Number of vertices and queries
n, q = map(int, input().split())

# Union find root list and sizes component sets for weighted union
parents = list(range(n))
sizes = [1] * n

# Number of leaves in component a vertex belongs to
# and degrees of vertices just to check whether a vertex is a leaf
n_leaves = [1] * n
degrees = [0] * n

for _ in range(q):
    q_type, *info = input().split()

    # Create an edge
    if q_type == '!':
        u, v = map(int, info)

        # Find parent vertices of components vertices u, v belongs to
        parent_v = find(v)
        parent_u = find(u)

        # If u, v are not in the same component
        if parent_u != parent_v:
            # Merge the components by performing weighted union of sets
            # Try to balance both sets' sizes controlling the tree height
            if sizes[parent_u] < sizes[parent_v]:
                parents[parent_u] = parent_v
                sizes[parent_v] += sizes[parent_u]

                # Modify number of leaves in the created component
                n_leaves[parent_v] += n_leaves[parent_u] - ((degrees[u] == 1) +
                    ↪ (degrees[v] == 1))

            else:
                parents[parent_v] = parent_u
                sizes[parent_u] += sizes[parent_v]

                n_leaves[parent_u] += n_leaves[parent_v] - ((degrees[u] == 1) +
                    ↪ (degrees[v] == 1))

        else:
```

```
            # Change the number of leaves in the component if we are creating an
              ↪ edge between leaves
            n_leaves[parent_v] -= (degrees[u] == 1) + (degrees[v] == 1)

        degrees[u] += 1
        degrees[v] += 1

    # Get the number of leaves in a component
    elif q_type == '?':
        v = int(info[0])
        parent = find(v)

        print(
            0 if degrees[v] == 0
            else n_leaves[parent]
        )
```

Listing programu (C++)

```
// C++ version of sol.py

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

int findRoot(int x, std::vector<int> &parents) {
    int modX = x;

    while (parents[modX] != modX) {
        parents[modX] = parents[parents[modX]];
        modX = parents[modX];
    }

    return modX;
}

int main() {
    int n, q;
    std::cin >> n >> q;

    std::vector<int> parents(n), sizes(n, 1);
    std::iota(parents.begin(), parents.end(), 0);

    std::vector<int> nLeaves(n, 1), degrees(n, 0);

    for (int i = 0; i < q; i++) {
        char qType;
        std::cin >> qType;

        if (qType == '!') {
            int u, v;
            std::cin >> u >> v;

            int parentV = findRoot(v, parents);
            int parentU = findRoot(u, parents);

            if (parentU != parentV) {
                if (sizes[parentU] < sizes[parentV]) {
```

```
                    parents[parentU] = parentV;
                    sizes[parentV] += sizes[parentU];

                    nLeaves[parentV] += nLeaves[parentU] - ((degrees[u] == 1) +
                        ↪ (degrees[v] == 1));
                }
                else {
                    parents[parentV] = parentU;
                    sizes[parentU] += sizes[parentV];

                    nLeaves[parentU] += nLeaves[parentV] - ((degrees[u] == 1) +
                        ↪ (degrees[v] == 1));
                }
            }
            else {
                nLeaves[parentV] -= (degrees[u] == 1) + (degrees[v] == 1);
            }

            degrees[u] += 1;
            degrees[v] += 1;
        }
        else if (qType == '?') {
            int v;
            std::cin >> v;

            int parent = findRoot(v, parents);

            std::cout << ((degrees[v] == 0) ? 0 : nLeaves[parent]) << std::endl;
        }
    }

    return 0;
}
```

Dávid

5. Takmer na to vidím

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

Zadanie tejto úlohy hovorí, že máme pre orientovaný graf nájsť nejaké ohodnotenie hrán, tak aby spĺňalo niekoľko podmienok. Tento orientovaný graf má navyše vyznačený štartovací vrchol a niekoľko cieľových vrcholov. Nemenej dôležitá je informácia, že v cieľových vrcholoch končia cesty ktoré popisujú *sufixy* nejakého slova (alebo postupnosti čísel v našom prípade).

Keďže sú to sufixy, môžeme si všinúť, že každá trasa končí rovnakým číslom, teda napríklad 1. Z toho nám hneď vyplýva, že hrany ktoré vedú do cieľových vrcholov musia niesť práve číslo 1. Získali sme jeden znak riešenia. Aby sme mohli určiť nasledujúci, stačí sa posunúť po hranách ktoré sme prave označili. My totiž vieme, že všetky cesty sú sufixy, takže aj predposledné písmeno musia mať rovnaké. Pozrieme sa teda na všetky hrany ktoré vedú ku tým ktoré sme práve označili a vieme že musia mať rovnaké číslo. Ak o niektorej hrane vieme, aké má číslo, tak ho len priradíme všetkým ostatným a máme druhý znak riešenia. Ak žiadna z hrán ešte číslo nemá, musíme im priradiť nejaké nové číslo (napríklad 2).

Tento postup následne stačí opakovať, až kým sa dostaneme do takého stavu, že do daných vrcholov žiadne hrany nevedú. Vtedy sme sa totiž všetkými cestami dostali do štartovného vrcholu a ak si medzičasom budeme aj zapisovať čísla ktoré hranám priraďujeme, máme aj finálne slovo, skonštruované od konca.

Implementácia

Postup popísaný vyššie je v podstate BFS, teda prehladávanie do šírky[13]. Zásadný rozdiel je v tom, že graf neprehľadávame z jedného počiatočného vrchola, ale zo všetkých cieľových vrcholov ako počiatočných naraz a posúvame sa po hranách v opačnom smere. Aby sa to viac podobalo na BFS, môžeme si predstaviť že máme

[13] https://www.ksp.sk/kucharka/bfs/

ešte jeden cieľ, z ktorého vedie hrana práve do tých vrcholov, ktoré sú cieľové. Potom sa to už podobá na klasické BFS z tohoto vrchola.

Druhý rozdiel od klasickej implementácie BFS je v tom, že sa potrebujeme vždy pozerať na celú "vrstvu" vrcholov, ktoré sú v rovnakej vzdialenosti od cieľa. Toto dosiahneme jednoducho tak, že namiesto toho aby sme z fronty vyberali jeden vrchol, vyberieme ich naraz všetky. To budú totiž vrcholy v rovnakej vzdialenosti. Následne ich spracujeme, teda pozrieme sa na všetky hrany ktoré do nich vedú, nájdeme medzi nimi ohodnotenú, alebo vyberieme nové číslo a všetky tieto hrany ohodnotíme. Až po tomto kroku pridáme do fronty nové vrcholy, teda začiatky novoohodnotených hrán, ktoré sú opäť o jedna ďalej od cieľa. Inak povedané, vo fronte bude jedno pole vrcholov ktoré sú v rovnakej vzdialenosti od cieľa, a teda to vlastne ani nemusí byť fronta.

Časová a pamäťová zložitosť

Čo si treba uvedomiť je, že síce hovoríme že je to "nejaké BFS", ale v skutočnosti to nie je jedno prehľadanie grafu. Aby sme mohli povedať že je to lineárne, mselo by platiť že každú hranu prejdeme konštantný počet krát – čo nie je pravda.

Náš algoritmus dá na začiatku do fronty všetky cieľové vrcholy, tých môže byť najviac n. V ďalšom kroku pridávame do fronty všetky vrcholy pred nimi. Tých môže byť najviac $n-1$, pretože jedna z ciest patrila suffixu dĺžky 1 a tou sme sa už dostali do cieľa. Rovnako v každom ďalšom kroku nám určite jedna cesta vypadne a teda v najhoršom prípade budeme do fronty pridávať postupne $n, n-1, n-2, \cdots, 1$ vrcholov čo je spolu $O(n^2)$. Čo sa pamäte týka, ukladáme si vstup, a niekoľko pomocných štruktúr, koré nepresahujú veľkosť vstupu, teda pamäťová zložitosť je $O(m+n)$

Listing programu (Python)

```
n, m, t = [int(x) for x in input().split()]
terminal = [int(x)-1 for x in input().split()]
hrany = [] # zoznm hran
susedia = [[] for x in range(n)] # zoznam susedov

for i in range((m)):
    a, b = [int(x)-1 for x in input().split()]
    susedia[b].append(a) # ukladame opacne hrany
    hrany.append((a,b))

oznacenia = {}
fronta = set(terminal)
dalsie_cislo = 1
slovo = []

while len(fronta)!=0:
    oznacenie = dalsie_cislo
    nova_fronta = set()
    for v in fronta:
        for s in susedia[v]:
            nova_fronta.add(s)
            if oznacenia.get((v,s)) != None:
                oznacenie = oznacenia.get((v,s))

    for v in fronta:
        for s in susedia[v]:
            oznacenia[(v,s)] = oznacenie

    fronta = nova_fronta
    if oznacenie == dalsie_cislo:
        dalsie_cislo += 1
    slovo.append(oznacenie)

slovo.pop()
slovo.reverse()
```

```
print(len(slovo), dalsie_cislo-2)
print(*slovo)

print(*[oznacenia[(b,a)] for (a,b) in hrany])
```

Tomi

6. Idiomatický slovník

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

Potrebujeme zistiť, či sa reťazec dá rozdeliť na niekoľko párne dlhých palindrómov. V tomto vzoráku budeme pod "palindróm" myslieť párne dlhý palindróm.

Bruteforce

Spočítajme dvojrozmernú tabuľku, kde pre každú začiatočnú a konečnú pozíciu zistíme, či je daný podreťazec vstupného reťazca palindróm. Pre dvojznakové podreťazce je to ľahké, stačí tie dva znaky porovnať. Dlhší podreťazec je palindróm vtedy, keď sa jeho prvý a posledný znak rovnajú, a všetko medzi nimi je tiež palindróm (zistíme z tabuľky). Tabuľku môžeme vypĺňať napríklad v poradí vzostupne podľa dĺžky podreťazca, alebo zostupne podľa začiatočnej pozície, aby sme vždy čítali len z buniek čo už sme naplnili.

Potom už len treba zistiť, či v reťazci vieme cez palindrómy preskákať zo začiatku na koniec. Keď prídeme na nové miesto, pozrieme sa aké palindrómy tam začínajú a na aké ďalšie miesta vďaka tomu dokážeme skočiť. Zo stringologického pohľadu sa pýtame o každom prefixe vstupného reťazca, či je dobrý (nakrájateľný na palindrómy). Z grafového pohľadu hľadáme či existuje cesta v topologicky zoradenom grafe, ktorého vrcholy sú pozície v reťazci (0 až n) a hrany sú palindrómy.

Toto riešenie má časovú aj pamäťovú zložitosť $O(n^2)$.

Trochu lepší bruteforce

Ak máme šťastie a vo vstupe je len málo palindrómových podreťazcov, počítať a pamätať si celú $(n+1) \times (n+1)$ tabuľku je trochu mrhanie. Radšej si pamätajme pre každú pozíciu iba zoznam palindrómov, čo na nej začínajú. Grafovo povedané, namiesto matice susednosti si pamätajme zoznamy susednosti každého vrcholu. Spočítame ich tak, že pre každú možnú stredovú pozíciu rozširujeme palindróm doľava aj doprava, až kým nenarazíme na dva rôzne znaky alebo na kraj vstupu.

Dalo by sa povedať, že toto riešenie má časovú aj pamäťovú zložitosť $O(n+p)$, kde p je počet palindrómových podreťazcov. Ale to samozrejme stále môže byť priveľa. Napríklad pre vstup zložený iba z n áčiek je $p = O(n^2)$.

Ide to pažravo

V skutočnosti platí, že môžeme vždy pažravo (greedy) spredu reťazca odkrojiť najkratší možný palindrómový prefix. Grafovo povedané, môžeme kľudne skákať po najkratšej hrane a nič si tým nepokazíme, nedostaneme sa do žiadnej slepej uličky. Stringologicky povedané, ak z dobrého reťazca (takého čo sa dá nakrájať na palindrómy) odrežeme najkratší palindrómový prefix, aj ten zvyšok bude dobrý reťazec.

Dokážeme to takto. Ukážeme, že ak existuje akékoľvek riešenie, ktoré by náš greedy algoritmus nevyplodil, tak existuje aj krajšie riešenie, ktoré sa od výstupu nášho algoritmu líši trošku menej. Opakovaním tohto procesu dôjdeme k riešeniu, čo sa od výstupu nášho algoritmu nelíši vôbec.

Poďme na vec. Majme nejaké riešenie (nejaké korektné rozdelenie celého reťazca na palindrómy). Možno už začína k najkratšími palindrómami (t.j. takými čo by na svojej pozícii vybral aj greedy algoritmus), ale tie nás nezaujímajú. Pozrime sa na prvé miesto, kde sa nezhodnú: najkratší vybrateľný palindróm je xx^{-1}, ale zvolené riešenie si namiesto toho vybralo odrezať nejaký dlhší palindróm yy^{-1}. (Reťazec x je prvá polovica a x^{-1} znamená opak x.)

Rozoberme dve možnosti:

- y je relatívne dlhý ($|y| \geq 2|x|$). Keďže yy^{-1} sa na vstupe prekrýva s xx^{-1}, xx^{-1} musí byť prefixom y. Zapíšme ho ako $y = xx^{-1}z$ s nejakým (možno prázdnym) zvyškom z. Tým pádom $yy^{-1} = xx^{-1}zz^{-1}xx^{-1}$. Super, to sú tri palindrómy. Môžeme z obidvoch koncov yy^{-1} odrezať xx^{-1} a text medzi nimi bude tiež palindróm. Riešenie, čo takto dostaneme, už začína na nie k ale aspoň $k+1$ najkratších palindrómov. O krok sme sa priblížili ku greedy algoritmu.

- y je relatívne krátky ($|x| < |y| < 2|x|$). Rozdeľme x na dve neprázdne časti p, q na tom mieste, kde končí y. Nech $x = pq$ kde q má $|y| - |x|$ znakov. Potom $xx^{-1} = pqq^{-1}p^{-1}$, $y = pqq^{-1}$, $yy^{-1} =$

$pqq^{-1}qq^{-1}p^{-1}$. Z polohy na vstupe vieme, že xx^{-1} je prefixom yy^{-1}, preto (po dosadení) $pqq^{-1}p^{-1}$ je prefixom $pqq^{-1}qq^{-1}p^{-1}$, preto (po škrtnutí pqq^{-1}) p^{-1} je prefixom $qq^{-1}p^{-1}$. Tak ho napíšme ako $qq^{-1}p^{-1} = p^{-1}z$ pre nejaký neprázdny zvyšok z. Lenže tým pádom môžeme dosadiť $xx^{-1} = pp^{-1}z$, čo je spor s pôvodným predpokladom, že xx^{-1} je na tomto mieste najkratší vybrateľný palindróm. Takže tento prípad nemôže nastať.

Pre úplnosť, podobne sa dá dokázať, že môžeme odkrojiť úplne ktorýkoľvek palindróm. Je to úplne jedno – žiaden rez nevyrobí z dobrého reťazca zlý. Ale toto v našom riešení nebudeme potrebovať, nám sa hodia krátke palindrómy.

Hashovanie

Plán je jasný: náš program prejde celým vstupom a vždy keď si všimne, že od pozície predošlého rezania po aktuálnu pozíciu je to palindróm, odkrojí ho.

Ako môžeme rýchlo zistiť o ľubovoľnom podreťazci, či je to palindróm? Použijeme hashovanie. Hashovanie sa bežne používa, keď chceme rýchlo testovať rovnosť dvoch podreťazcov, ale test palindromicity je vlastne iba test rovnosti vhodného podreťazca pôvodného vstupu a vhodného podreťazca obráteného vstupu.

Hashovacie funkcie prevádzajú reťazce na čísla a správajú sa pomerne chaoticky. Ak sa dva reťazce rovnajú, samozrejme budú mať aj rovnaký hash. Ak sa nerovnajú, pomerne pravdepodobne budú mať rôzny hash. Takže pri porovnávaní dvoch reťazcov sa oplatí najprv porovnať ich hashe. Ak sú rôzne, ušetrili sme si kopu práce. Ak sú rovnaké, môžeme si dať tú námahu porovnať ich znak po znaku (so šťastím ani netreba).

Vyberme si nejaké pekné prvočíslo p (napríklad $10^9 + 9$) a nejakú peknú konštantu a (napríklad 47; niektorí machri používajú v každom vstupe iné náhodné číslo). Definujme *polynomiálny rolling hash* reťazca $S = c_1c_2\dots c_n$ ako súčet $H(S) = (c_1a^1 + c_2a^2 + \dots + c_na^n) \bmod p$.

Táto definícia má všelijaké pekné vlastnosti. Môžeme si zapamätať nielen finálny výsledok, ale aj medzisúčty pre každý prefix nášho reťazca. Vďaka nim budeme vedieť počítať aj hash ľubovoľného podreťazca: $H(S[x..y]) = (H(S[0..y]) - H(S[0..x]))/a^x$. Pozor, že odčítanie a delenie tiež robíme modulo p.

Delenie modulo p sa dá robiť tak, že nájdeme inverzné prvky pomocou modulárneho umocňovania a malej Fermatovej vety. Ale to v tejto úlohe vôbec netreba. Stačí si všimnúť, že nás vždy zaujímajú iba porovnania tvaru $e/a^f \equiv g/a^h \pmod p$, čo sa dá upraviť na $e \cdot a^h \equiv g \cdot a^f \pmod p$, čím sa delenia úplne zbavíme.

Greedy algoritmus s hashovaním má priemernú časovú aj pamäťovú zložitosť $O(n)$.

Listing programu (C++)

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

const long long P = 1000000009;

bool run() {
  string S;
  cin >> S;

  int N = S.size();

  vector<long long> expo(N+1);
  vector<long long> hashforw(N+1);
  vector<long long> hashback(N+1);

  expo[0] = 1;
  for (int i = 0; i < N; i++) {
    expo[i + 1] = (expo[i] * 47) % P;
    hashforw[i + 1] = (hashforw[i] + expo[i + 1] * S[i]) % P;
    hashback[i + 1] = (hashback[i] + expo[i + 1] * S[N - 1 - i]) % P;
  }

  int start = 0;
```

```
  for (int end = 1; end <= N; end++) {
    if ((end - start) % 2 == 0) {
      int mid = (start + end) / 2;

      // S[start..mid] ==? S[mid..end].reverse()
      // (hashforw[mid] - hashforw[start]) / expo[start] ==? (hashback[N - mid]
          ↪ - hashback[N - end]) / expo[N - end]
      // (hashforw[mid] - hashforw[start]) * expo[N - end] ==? (hashback[N - mid
          ↪ ] - hashback[N - end]) * expo[start]

      long long l = ((hashforw[mid] - hashforw[start] + P) % P) * expo[N - end]
          ↪ % P;
      long long r = ((hashback[N - mid] - hashback[N - end] + P) % P) * expo[
          ↪ start] % P;
      bool are_probably_equal = l == r;

      if (are_probably_equal) {
        bool are_really_equal = true;
        for (int i = 0; i < mid - start; i++) {
          if (S[start + i] != S[end - 1 - i]) {
            are_really_equal = false;
            break;
          }
        }
        if (are_really_equal) {
          start = end;
        }
      }
    }
  }

  return start == N;
}

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

  while (T--) {
    cout << (run() ? "1" : "0") << endl;
  }
}
```

7. Nádlab

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

Načítavanie

Prvý krok nutný na vyriešenie úlohy je načítanie vstupu. Keďže je to 7. úloha, aj trošku neštandardný formát vstupu by nám nemal robiť problém. Najťažšie je načítať parametre kartičiek – chceme načítať dve čísla na jednom riadku, ale odignorovať posledný znak.

V Pythone vieme tento riadok načítať napríklad nasledovne:

```
gram, perc = input().split()
gram, perc = int(gram[:-1]), int(perc[:-1])
# alebo kompaktnejšie
gram, perc = map(lambda s: int(s[:-1]), input().split())
```

V C++ zase takto:

```
int gram, perc;
// pomocou scanf
scanf("%dg %d%% ", &gram, &perc);
// alebo, keďže `cin`:
// - číselné typy načítava pokým nenarazí na nečíselný znak
// - ignoruje vedúce whitespace
// stačí načítať do čísla a následne načítať koncový znak do typu `char`
char koncovy;
cin >> gram >> koncovy >> perc >> koncovy;
```

Aby sa nám uľahčil život, ďalej vo vzorovom riešení, pod hodnotou 'x%'-kartičky myslíme zlomok granadíru ktorý ešte ostane po použití tejto kartičky, teda $1 - x/100$.

Bruteforce

Najprv si uvedomme, že ak vo výsledku máme dve po sebe idúce kartičky použité rovnakým spôsobom (teda obe `g` alebo obe `%`), nezáleží na ich poradí.

Ďalej si uvedomme, že sa nám nikdy neoplatí použiť `%`-kartičku po `g`-kartičke. Bez ujmy na všeobecnosti nech ide o posledné dve kartičky a nech množstvo granadíru zostávajúceho po použití všetkých zvyšných kartičiek je R. Potom výsledok (koľko granadíru zostane) po použití kartičiek v poradí `g%` bude $(R - m) \cdot k = R \cdot k - m \cdot k$ zatiaľčo v poradí `%g` bude $R \cdot k - m$, kde $0 \leq b \leq 1$. Vidíme teda, že použitím kartičiek v poradí `g%` iba zbytočne odčítavame menej. Výsledné kartičky teda obsahujú najprv postupnosť `%`-kariet a potom už iba `g`-kariet.

Spojením týchto dvoch uvedomení zistíme, že našou úlohou je iba pre každú kartu rozhodnúť o jej type, čo už potom jednoznačne určuje výsledok. Naskytá sa nám teda priamočiare riešenie vyskúšať všetkých 2^N možností, každú vyhodnotiť a vypísať najmenšiu.

Najťažšou časťou takéhoto programu je iterovanie cez všetky kombinácie. To sa štandardne robí prostredníctvom iterovania cez čísla od 0 po $2^N - 1$. Binárny zápis každého tohoto čísla reprezentuje jednu podmnožinu N-prvkovej množiny.

```
for binarna_mnozina in range(1 << N):
    mnozina = set(i for i in range(N) if binarna_mnozina & (1 << i))
```

Pamäťová zložitosť je $O(N)$ a časová zložitosť je $O(N \times 2^N)$. Efektívnou implementáciou tohoto prístupu sa dali prejsť až dve sady.

Stredová stretávka

Pri riešení úloh vieme použiť obmedzenia vstupov ako pomôcku na nájdenie riešenia. V tomto prípade sme si mohli všimnúť, že N je podozrivo malé, a teda by sa úloha mohla dať riešiť *Meet-In-The-Middle* prístupom.

Princíp MITM riešení je využiť symetriu problému a rozdeliť ho na dve časti, ktoré sa dajú riešiť nezávisle. Snažíme sa teda nájsť také rozdelenie, aby každá časť bola dostatočne malá na to, aby sme ju vedeli vyriešiť nejakým iným prístupom (napríklad bruteforce) a výsledky z oboch častí dakázali spojiť do výsledného riešenia.

Ako by to vyzeralo? Rozdelíme si množinu kartičiek na dve rovnako veľké disjunktné časti. Pre každú z nich zbehneme bruteforce a zapamätáme si všetky podvýsledky – podvýsledok je optimálne riešenie pre zvolené určenie typov pre danú polovicu kartičiek. Pre každú časť ich bude $2^{N/2}$.

My už vieme, že určením typov kartičiek je jednoznačne určený výsledok čo nimi vieme dosiahnuť a efekt ich uplatnenia vieme zjednodušene reprezentovať ako dve čísla – súčet všetkých `g`-kartičiek a súčin všetkých `%`-kartičiek, teda (M, K).

Skombinovanie podvýsledkov z týchto dvoch podmnožín je potom jednoduché. Ak chceme skombinovať podvýsledok $(M_{1,i}, K_{1,i})$ s podvýsledkom $(M_{2,j}, K_{2,j})$, kombinácia bude $(M_{1,i} + M_{2,j}, K_{1,i} \cdot K_{2,j})$, čo po konzumácii spôsobí zostatok granadíru $(H \cdot K_{1,i} \cdot K_{2,j}) - (M_{1,i} + M_{2,j})$.

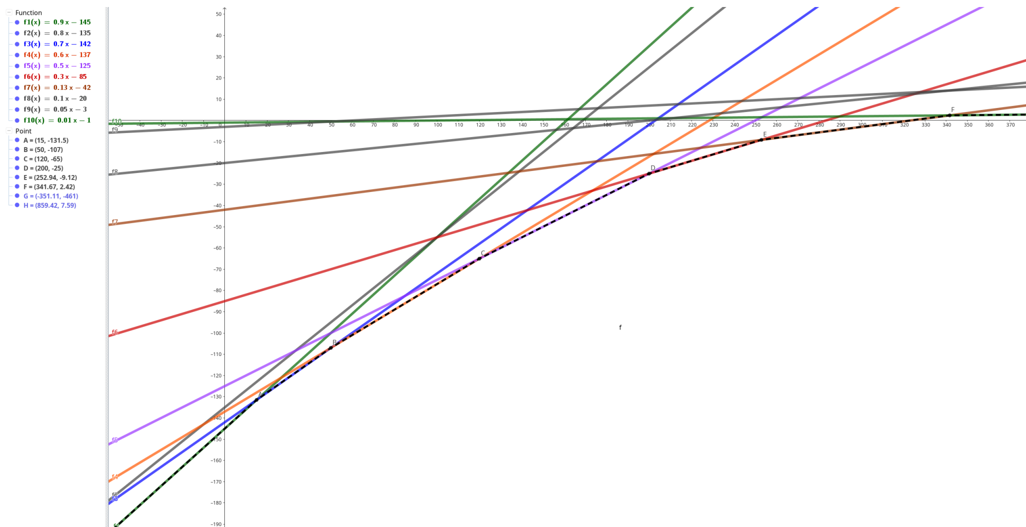
Nás by teraz zaujímalo, aký je najlepší možný výsledok. To zistíme tak, že pre každý podvýsledok prvej časti nájdeme preňho najlepší podvýsledok v druhej časti – teda taký, ktorý minimalizuje daný výraz.

Naivným skúšaním všetkých možností by sme znova získali bruteforce ($O(2^{N/2} \times 2^{N/2})$). Naším cieľom je pre fixné $(M_{1,i}, K_{1,i})$ minimalizovať $(H \cdot K_{1,i} \cdot K_{2,j}) - (M_{1,i} + M_{2,j})$ cez všetky $(M_{2,j}, K_{2,j})$. Všimnime si, že to je to isté ako minimalizovať $H_i \cdot K_{2,j} - M_{2,j}$, pre $H_i := H \cdot K_{1,i}$, keďže $M_{1,i}$ a $K_{1,i}$ sú v tomto prípade konštanty. Problém je, že H_i je konštanta iba pre jedno konkrétne $K_{1,i}$, pričom rôznych $K_{1,i}$ je až $2^{N/2}$.

Konvexný obal

Sťažme si úlohu. Ak by sme vedeli nájsť optimálne $(M_{2,j}, K_{2,j})$ pre každé reálne H', tak by sme ho predsa vedeli nájsť aj pre ľubovoľné konkrétne H_i. Z H_i sa teda stáva reálna premenná x a $x \cdot K_{2,j} - M_{2,j}$ nám určuje lineárnu funkciu – priamku. Každé jedno $(M_{2,j}, K_{2,j})$ určuje jednu priamku a nás pre každé x zaujíma najnižší bod ležiaci na niektorej z týchto priamok. Inak povedané, hľadáme konvexný obal týchto priamok.

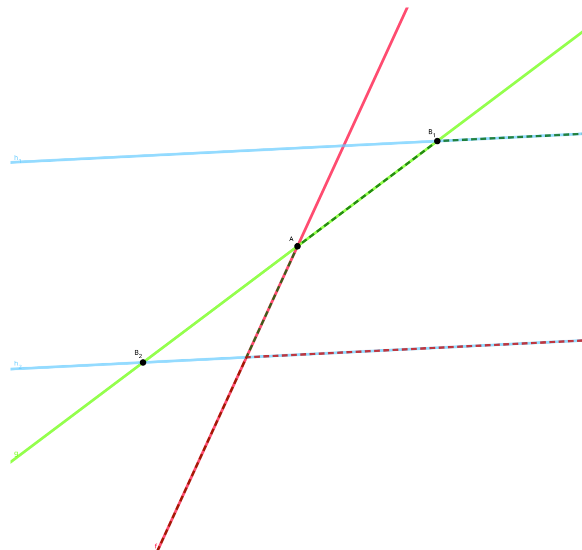
Na obrázku vidíme 10 lineárnych funkcií. Ich spodný konvexný obal je označený prerušovanou čiernou čiarou. Niektoré (šedé) priamky nie sú súčasťou konvexného obalu. Každá priamka ktorá je súčasťou je ňou iba na jednom súvislom intervale.



Hľadanie konvexného obalu priamok je podobné hľadaniu konvexného obalu bodov. Priamky si utriedime podľa ich smernice a potom ich prechádzame od najväčšej po najmenšiu (teda budeme tvoriť *KO* zľava doprava), pričom si udržiavame doteraz nájdený *KO*. Každú priamku sa pokúsime pridať do *KO*. Môžu nastať dve situácie – buď nová priamka nepatrí do *KO* alebo patrí.

V prípade, že nová priamka $f_{\text{nová}}$ patrí do *KO* tak existuje nejaký najľavejší bod, ktorý patrí aj tejto priamke aj *KO* – teda to bude priesečník $f_{\text{nová}}$ a niektorej doterajšej priamky $f_{\text{hľadaná}}$ z *KO*. Otestujme, či posledná priamka *KO* $f_{\text{posledná}}$ je $f_{\text{hľadaná}}$. Ak je priesečník priamok $f_{\text{nová}}$ a $f_{\text{posledná}}$ ľavší než priesečník priamok $f_{\text{posledná}}$ a $f_{\text{predposledná}}$, tak $f_{\text{posledná}}$ nie je $f_{\text{hľadaná}}$ a dokonca $f_{\text{posledná}}$ nie je ani súčasťou *KO*. V takom prípade ju odstránime a pokračujeme v hľadaní najľavejšieho bodu.

Nech červená priamka je $f_{predposledn}$, zelená je $f_{posledn}$ a modrá je f_{nov}. Potom podľa toho, či je B naľavo alebo napravo od A vieme povedať či $f_{posledn}$ patrí do *KO*.



Inak povedané: Tým, že sa na priamky pozeráme v poradí s klesajúcou smernicou platí, že najnovšia priamka je vždy lepšia ako nejaký (potenciálne nulový) počet priamok na konci *KO*. Teda porovnávame novú priamku s poslednou priamkou *KO* a odstraňujeme túto poslednú priamku pokým je horšia ako nová. Ošetrenie rovnobežných priamok ponechávame na čitateľa.

Časová zložitosť je $O(N \log(N))$ a pamäťová zložitosť je $O(N)$, kde N je počet priamok. Viac si o konvexnom obale môžete prečítať v kuchárke[14].

Dôsledok

S nájdeným konvexným obalom potom pre ľubovoľné H_i vieme nájsť najlepšie $(M_{2,j}, M_{2,j})$ binárnym vyhľadávaním. V prípade, že sa na jednotlivé H_i budeme pýtať v utriedenom poradí ani nemusíme binárne vyhľadávať, ale len prechádzať postupne priamky *KO*. Tento prístup sa volá *Convex Hull Optimization/Trick*.

Pamäťová zložitosť je $O(2^{N/2})$ a časová zložitosť je $O(2^{N/2} \times \log(2^{N/2})) = O(N \times 2^{N/2})$ (logaritmus pochádza z triedenia v hľadaní *KO* a triedenia H_i alebo binárneho vyhľadávania). Oproti bruteforce teda úspešne zvládame približne dvakrát väčšie vstupy. Pre aktuálne obmedzenia toto však nie je dostatočne rýchle riešenie. Na iných vstupoch (ktoré by mali napríklad väčšie hodnoty gramov) by toto bol dobrý prístup. Tento prístup zvláda prejsť až tri sady, ale v hodnotení popisov vie takéto riešenie získať plný počet.

Vzorové riešenie

Čo ďalšie by sme si mohli všimnúť zo zadania? Gramáže kartičiek sú nanajvýš 10^4 a spolu s malým N teda aj súčet gramáží bude malé číslo.

Ako sme si už povedali, pre zvolené určenie typov kartičiek je výsledok závislý iba od súčinu %-kartičiek ($= k$) a súčtu g-kartičiek ($= m$). Preto ak by sme mali pevne určený nejaký súčin k, našou snahou je maximalizovať súčet m. Naopak, ak by sme mali pevne určený nejaký súčet m, našou snahou je dosiahnuť minimálny súčin k.

A my vieme, že rôznych súčtov m je málo. Ak pre každé možné m nájdeme minimálne k, tak sme našli riešenie. Naskytá sa nám celkom štandardné riešenie dynamickým programovaním – stav je určený počtom kartičiek ktorým sme už určili typ a súčtom m týchto kartičiek. Pre každý stav si pamätáme minimálny súčin k pre zvyšné kartičky. V každom stave sa do ďalšieho stavu dostaneme určením typu ďalšej kartičky. Teda:

```
dp[i+1][m]  = min(dp[i+1][m], dp[i][m] * k_i) # ak použijeme % kartičku
dp[i+1][m+m_i] = min(dp[i+1][m+m_i], dp[i][m]) # ak použijeme g kartičku
```

Poznámky k implementácii

Nakoniec, aby sme vedeli vypísať riešenie, musíme vedieť z tabuľky nejako vyčítať ako sme určovali kartičky. Toto si vieme v tabuľke buď priamo pamätať, alebo si to vieme v prípade tejto úlohy jednoducho spätne zistiť (keďže existujú iba dve možnosti), čo nám ušetrí pamäť a kúsok urýchli program (nie asymptoticky, ale iba konštantne).

Vieme ešte riešenie urýchliť? Mohli by sme veľkosť tabuľky ďalej zmenšiť tým, že jednotlivé riadky budú mať veľkosť iba doterajšieho súčtu gramáží kartičiek. Tomuto ďalej pomôžeme ak si kartičky na začiatku utriedime podľa gramáže, keďže v tomto prípade bude veľkosť tabuľky minimálna. Znova, nejde o asymptotické zlepšenie, ale vieme takto reálne program zrýchliť dvojnásobne.

Dynamiku neodporúčame kódiť v Pythone, keďže priamy prepis C++ do Pythonu je často krát 100x pomalší. Dodatočnými optimalizáciami vieme získať násobne zrýchlenia, ale môžeme byť šťastný keď sa dostaneme na 30x spomalenie. Ako malý tip povieme, že je oveľa rýchlejšie napísať `if x < y: y = x` než `y = min(y, x)`.

V úlohách kde násobíme veľa desatinných čísel a záleží nám na presnosti sa väčšinou oplatí počítať si súčet logaritmov namiesto priameho súčinu. V tomto prípade to však nie je nutné, keďže násobíme maximálne 40 čísel, čo zvládneme presne uložiť do dostatočne veľkého dátového typu.

Zložitosť

Časová a pamäťová zložitosť je $O(N \times \sum A_i) = O(N^2 \times MaxA)$. Vzorové riešenie v C++ zvláda vstupy aj pre $N = 100$.

Často upozorňujeme, že pamäťová zložitosť sa pri úlohách s dynamickým programovaním dá znížiť na veľkosť menšiu ako celkový počet stavov. Väčšinou to dosiahneme tak, že si pamätáme iba posledné dva riadky tabuľky. V tomto prípade si však v tabuľke okrem najlepšieho výsledku implicitne pamätáme aj cestu a teda ich nemôžeme zabudnúť! Alebo? V skutočnosti to je možné a dokonca bez toho aby sa nám zhoršila časová zložitosť. Viac informácií nájdete v tomto codeforces tutoriáli[15] v poslednom odseku sekcie *Store results only for two layers of DP state domain*.

Uvedomme si, že všetky tri uvedené riešenie prehľadajú všetkých 2^N možností. Prvý prístup naivne, druhý šikovne a tretí iba tak, že nevykonáva žiadnu robotu dva krát – dynamické programovanie pracuje iba na stavoch, ktoré sú relevantné a pre každý takýto stav si pamätá iba jeden výsledok. Toto je dôvod, prečo tieto riešenia fungujú, zatiaľ čo heuristiky a greedy riešenia nie.

[14] https://www.ksp.sk/kucharka/konvexny_obal
[15] https://codeforces.com/blog/entry/43256

Listing programu (Python)

```
INF = 1e18
div = lambda x: 1 - x / 100

def solve():
    N, X = map(int, input().split())
    C = [tuple(map(lambda s: int(s[:-1]), input().split())) for _ in range(N)]

    dp = [[1.0]]
    for ni in range(N):
        xm, xd = C[ni]
        dp.append([INF] * (len(dp[ni]) + xm))
        d = div(xd)
        for mi in range(len(dp[ni])):
            if dp[ni][mi] == INF:
                continue
            dp[ni + 1][mi] = min(dp[ni + 1][mi], dp[ni][mi] * d)
            dp[ni + 1][mi + xm] = min(dp[ni + 1][mi + xm], dp[ni][mi])

    finish = (X, -1)
    for mi, d in enumerate(dp[-1]):
        finish = min(finish, (X * d - mi, mi))

    mi = finish[1]
    ans = []
    for ni in range(N - 1, -1, -1):
        xm = C[ni][0]
        card = mi >= xm and dp[ni + 1][mi] == dp[ni][mi - xm]
        ans.append((card, ni))
        mi -= card * xm
    ans.sort()
    for card, ni in ans:
        print(ni + 1, "%g"[card])

TC = int(input())
for _ in range(TC):
    solve()
```

Listing programu (C++)

```
#include "fezjo.h"

void solve() {
    int N, X;
    cin >> N >> X;

    vector<pair<int, int>> C(N);
    char trash;
    for (int i = 0; i < N; i++)
        cin >> C[i].first >> trash >> C[i].second >> trash;

    auto [final, ans] = solve(N, X, C);
    for (auto [card, ni] : ans)
        cout << (ni + 1) << " " << "%g"[card] << '\n';
    // cerr << fixed << setprecision(10) << final << endl;
}
```

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

    int TC;
    cin >> TC;
    while (TC--)
        solve();
}
```

8. A čo tak viac Torty?

12 b za popis, 8 b za program

Poďme sa pozrieť, ako vieme postupne riešiť jednotlivé operácie, a tak prísť ku vzorovému riešeniu. Pre jednoduchosť si predstavme, že každé `V` nahradíme 1, a každé `T` nahradíme −1, a takto z reťazca dostaneme postupnosť čísel, $x_1, \ldots, x_l$, kde l je aktuálna dĺžka postupnosti.

Výsledok

Zo vzorového riešenia Organizácie Kapustnice[16], vieme, že výsledok pre postupnosť $x_1, \ldots, x_l$ je

$$l + V + \max\{\sum_{i=1}^{l} x_i + V, 0\}$$

kde $V = -\min_{0 \leq i \leq l}\{\sum_{j=1}^{i} x_j\}$ je najmenší súčet prefixu postupnosti, teda počet vedúcich, ktorý treba pridať na koniec stola.

Podobne, ak chceme vyriešiť úlohu pre podreťazec $x_{i_z}, x_{i_z+1}, \ldots, x_{i_k}$, vtedy sa nám vzorec jednoducho zmení na

$$l + V + \max\{\sum_{i=i_z}^{i_k} x_i + V, 0\}$$

kde

$$V = -\min_{i_z-1 \leq i \leq i_k}\{\sum_{j=i_z}^{i_k} x_j\}$$

Prvá sada – malé vstupy

V prvej sade si vieme dovoliť spraviť každú operáciu v linárnom čase v závislosti od dĺžky reťazca. Každú query, ktorá nie je *vysledok* vieme vykonať v lineárnom čase, a rovnako tak vieme aj spočítať horeuvedený vzorec. Takto dostaneme riešenie so zložitosťou $O(q(q+n))$ (keďže dĺžka reťazca nikdy nepresiahne $q+n$).

Druhá sada – pridávame na koniec

Ako si môžeme všimnúť, ak pridáme na koniec postupnosti novú hodnotu $x_{l+1} \in \{1, -1\}$, vieme výsledok vypočítať v konštantnom čase – za predpokladu„ že si pamätáme pár údajov pre predchádzajúci reťazec. Menovite, ak si pamätáme

$$V_l = -\min_{i_z-1 \leq i \leq i_k}\{\sum_{j=i_z}^{i_k} x_j\}$$

a

$$S_l = \sum_{i=1}^{l} x_i$$

Následne vieme tieto hodnoty updatnúť v konštantnom čase na

$$S_{l+1} = S_l + x_{l+1}$$

[16] https://www.ksp.sk/ulohy/zadania/2387/

a

$$V_{l+1} = -\min\{-V_l, S_{l+1}\}$$

Z týchto hodnôt už vieme v konštatnom čase zistiť výsledok pre $x_1 x_2 \dots x_l x_{l+1}$. Teda dostaneme riešenie v čase $O(n+q)$, a pamäti $O(n)$ riešiace druhú sadu.

Tretia sada – ľubovoľný podinterval

Pozrime sa teraz na tretiu sadu: máme nemenný reťazec, ale chceme vedieť vyriešiť úlohu na ľubovoľnom podintervale. Skúsenejší riešiteľ už v tomto vidí interval… čo znie ako intervaláč! A naozaj, napriek tomu, že namiesto priamočiareho minimového, alebo súčtového intervaláča, musíme vymyslieť niečo prefíkanejšie.

Chceli by sme si vedieť v intervaláči pamätať také hodnoty aby sme z nich vedeli zrekonštruovať

- súčet na danej podpostupnosti
- minimá prefixových súčtov v danej podpostupnosti

Prvú hodnotu vieme vypočítať jednoduchým súčtovým intervaláčom (pozri https://www.ksp.sk/kucharka/intervalovy_strom/). Čo s druhou hodnotou? Základná myšlienka je, že ak vieme súčet na nejakej podpostupnosti $x_{i_1} \dots x_{i_2}$ (označme ho s), a taktiež navyše vieme minimá prefixových súčtov na podpostupnosti $x_{i_1} \dots x_{i_2}$ a $x_{i_2} + 1, \dots, x_{i_3}$ (označme ich postupne v_1 a v_2), potom minimum prefixových súčtov na podpostupnosti $x_{i_1}, \dots, x_{i_2}, \dots, x_{i_3}$ je minimum z v_1 a $v_2 + s$.

Na základe tejto myšlienky vieme následne skonštruovať intervalový strom, ktorý si okrem prefixových súčtov navyše pamätá v každom vrchole minimum prefixových súčtov na jeho podintervale – vypočítať tieto hodnoty si vieme rekurzívne, na základe horeuvedeného vzorca.

Výsledok na zadanom intervale vieme získať skombinovaním údajov z $O(\log n)$ vrcholov stromu, tak ako v klasickom intervaláči. Takto teda získame riešenie pre tretiu sadu, ktoré beží v čase $O(n + q \log n)$ a $O(n)$ pamäti.

Pridávame *vymeň*

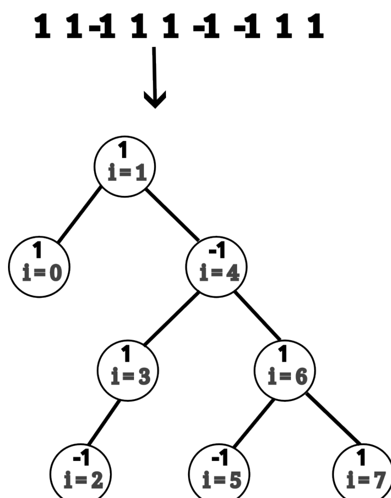
Toto intervaláčové riešenie vieme jednoducho vylepšiť, aby vedelo vyriešiť aj sady 4 a 5. V nich navyše pribúda operácia *vymen*. Všimnime si, že jej pridaním sa nám stále nezmení dĺžka postupnosti. Preto by sme mohli použiť horeuvedené riešenie, ak by sme do intervaláča vedeli implementovať aj query na menenie hodnôt v postupnosti. To je štandardná intervaláčová operácia, ktorú vieme dosiahnuť aj s naším intervaláčom: súčtovú časť intervaláča vieme updatnuť ako v klasickom súčtovom intervaláči, a hodnotu prefixových miním tiež meníme len na $O(logn)$ vrcholoch ktoré idú z listu (kde je menená hodnota) do koreňa. Spočítame ju rovnakým rekurzívnym vzorčekom, ako keď sa počítali počiatočné hodnoty intervaláča.

Dostaneme teda riešenie riešiace sady 3, 4 a 5 bežiace v čase $O(n + q \log n)$ a pamäti $O(n)$.

Ďalšie sady – nastupuje treap

Problém s operáciami *pridaj* a *zmaz* je, že intervaláč má fixné poradie prvkov, a hoci by sme teoreticky vedeli doimplementovať odoberanie a pridávanie prvkov na konci poľa (zamyslite sa :)), problém je pridávanie/vymazanie prvku na/z ľubovoľného indexu. Mohli by sme nejako náš intervaláč zovšeobecniť?

Odpoveď je áno. Predstavme si, že by sme mali binárny strom, pre ktorý platí, že každý vrchol reprezentuje pozíciu v postupnosti (nazvime ju i), všetky vrcholy v jeho ľavom podstrome (ak existuje) sú pozície *pred* i a všetky pozície v pravom podstrome (ak existuje) sú *za* i. Napríklad, pre vstup zo zadania by mohol strom vyzerať nasledovne.



Všimnime si, že pre takýto strom vieme rovnako ako pre intervaláč vypočítať v každom vrchole hodnoty súčtu jeho intervalu a minimum prefixových súčtov na jeho intervale, a následne vieme zistiť hodnotu na nejakom v intervale v čase lineárnom od *hĺbky stromu* (podobne ako v intervaláči, rozmyslite si to).

Problém je práve s hĺbkou tohto stromu (a pridávaním a zmazaním). Potrebovali by sme taký strom, kde je hĺbka približne logaritmická a navyše vieme horeuvedené operácie vykonávať dostatočne rýchlo. Ako odpoveď nám príde *treap*.

Treap

Treap je, ako názov napovedá, strom (tree) aj https://www.ksp.sk/kucharka/halda/ (heap) zároveň. Predstavme si, že pre každú pozíciu si náhodne vygenerujeme jej *prioritu*, a v strome budeme udržiavať pravidlo, že v každom podstrome má najväčšiu prioritu jeho vrchol. Dá sa ukázať, že pre dané poradie vrcholov (ako v strome opísanom vyššie) a prioritu vrcholov, existuje práve jeden tvar, ktorý strom musí mať, a zároveň, ak priority vyberáme náhodne, bude hĺbka stromu $O(\log V)$ kde V je počet vrcholov.

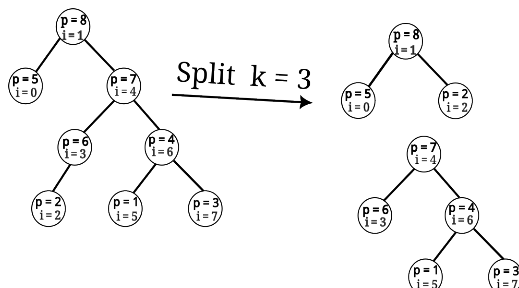
Mohli by sme sa teraz zastaviť a spýtať sa "počkať, ale prečo práve treap? Je to jediné riešenie?", a odpoveď je: nie, ale je pomerne príjemný na implementáciu, ako môžete vidieť v https://www.ksp.sk/kucharka/treap/.

Môžete si všimnúť, že síce väčšina implementácií treapov ktoré nájdete (napríklad v kuchárke) používa treap ako binárny vyhľadávací strom, kde sú prvky zoradené od najmenšieho po najväčší. V našom prípade však prvky chceme mať zoradené podľa poradia v postupnosti. Na to však netreba robiť veľké zmeny. Stačí keď pri vyhľadávaní konkrétneho prvku sa algoritmus nebude rozhodovať na základe toho či je väčší ako x ale či je pred ním aspoň k prvkov.

Hlavné dve funkcie treapu (pomocou ktorých vieme už v princípe všetko ostatné potrebné implementovať), ako v kuchárke môžte vidieť sú *split* a *merge*. V https://www.ksp.sk/kucharka/treap/ nájdete ako tieto funkcie implementovať, ale my si tu v skratke opíšeme čo robia a ako ich použiť na implementáciu *pridaj* a *zmaz*.

Split

Prvou z nich je *split*, ktorá zoberie treap a číslo $0 \leq k$ a vráti dva treapy: prvý obsahujúci prvých k pozícií vstupného treapu, a druhý obsahujúci ostatné pozície. (Na obrázku p označuje prioritu vrcholu.)



Split vieme naprogramovať jednoduchou rekurzívnou funkciou (ako tiež možno vidieť v kuchárke) zobrazenou na nasledovnom pseudokóde.

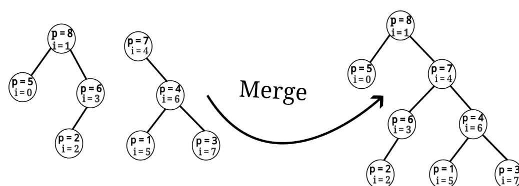
```
Split(T, k):
    Ak k = 0, vráť (prázdny treap, T)
```

```
    Ak veľkosť(T) je najviac $k$, vráť (T, prázdny treap)
    Ak veľkosť(T -> prvý syn) je najviac k
        T1, T2 je výsledok z Split(T -> prvý syn, k)
        Vymeň prvého syna stromu T za T2 a vráť (T1, modifikovaný T)
    Ak veľkosť(T -> prvý syn) je viac ako k,
        T1, T2 je výsledok z Split(T -> druhý syn, k - 1 - (veľkosť(T -> prvý syn))
        Vymeň druhého syna stromu T za T1 a vráť (modifikovaný T, T2)
```

Všimnime si, že si chceme okrem informácií o súčtoch a miním prefixových súčtov pre jednotlivé podstromy navyše udržovať informáciu o veľkostiach jednotlivých podstromov. Keďže pri funkcii split sa mení najviac "hĺbka stromu"[17] podstromov, a preto nám stačí tieto informácie (nápodobne ako pri intervaláči) zmeniť len raz v každej vrstve rekurzie funkcie.

Merge

Druhá funkcia na ktorej treap stojí je *merge*. Tá zoberie na vstupe dva treapy (reprezentujúce dve postupnosti) a vráti jeden treap reprezentujúci zreťazenie ich postupností. Príklad tejto operácie vidíme na nasledovnom obrázku.



Vieme ju, podobne ako *split* naprogramovať pomocou jednoduchej rekurzívnej funkcie, ako vidíme na nasledovnom pseudokóde.

```
Merge(T1, T2)
    Ak T1 je prázdny treap, vráť T2. Ak je T2 prázdny treap, vráť T1.
    Ak priorita(T1) > priorita(T2)
        Vymeň druhého syna T1 za výsledok Merge(T1 -> druhý syn, T2) a vráť modifikovaný T1
    Inak, vymeň prvého syna T2 za výsledok Merge(T1, T2 -> prvý syn) a vráť modifikovaný T2
```

Všimnite si, že podobne ako pri *split*, aj počet rekurzívnych volaní pri *merge* nepresiahne hĺbky $T1$ a $T2$, a teda najviac toľkým podstromom treba aktualizovať hodnoty.

Ako to teda vyriešiť?

Poďme si to celé dať dokopy. Pre každý vrchol v treape si chceme pamätať nasledovné informácie:

- Priorita (náhodne vygenerovaná)
- Prvok postupnosti na pozícii odpovedajúcej vrcholu
- Veľkosť podstromu
- Súčet pozícií na intervale odpovedajúcom podstromu
- Minimum z prefixových súčtov na intervale odpovedajúcom podstromu

Vždy keď sa vrchol, alebo jeho synovia zmenia, updatneme posledné tri informácie (súčet a minimum vypočítame tak, ako v intervaláči).

Pozrime sa najskôr na to, ako do existujúceho treapu, označme to T, pridať prvok p na pozíciu i. Stačí nám použiť *split* a *merge*, tak že si rozdelíme treap na prvky "pred" a "za" novopridaným prvkom, a následne pomocou mergu, už treap "zlepíme" dokopy spolu s novým prvkom. Môžeme to vidieť na nasledovnom pseudokóde:

```
T1, T2 = split(T, i)
T = merge(T1, merge(treap(p), T2))
```

kde `treap(p)` je jednoprvkový treap reprezentujúci postupnosť p.

Zmazanie intervalu od z po k (vrátane) vieme implementovať podobne, tak že si najskôr rozbijeme treap na tri treapy – reprezentujúce pozície pred intervalom, pozície v intervale, a pozície za intervalom, a prvý a tretí zlepíme *merge*-om späť dokopy.

[17] o ktorej vieme očakávať že je $O(\log V)$ ak priority volíme náhodne

```
T1, T2 = split(T, z)
T2, T3 = split(T2, k - z + 1)
T = merge(T1, T3)
```

Aby sme nezabudli na query *zmen*, tú vieme implementovať buď cez skombinovanie *zmaz* a *pridaj*, alebo priamo rekurzívnou funkciou – podobne ako update v intervaláči, ktorú nechávame ako úlohu na zamyslenie.

No, a na koniec, keď dostaneme počiatočný reťazec, vieme z neho vytvoriť treap napríklad pomocou n operácií *pridaj* na koniec.

Časová zložitosť je $O((n+q)\log(n+q))$[18], a pamäť nám stačí $O(n)$.

Plný počet bodov – nebojte sa lenivého treapu

V poslednej sade nám prišla posledná z operácií *reverzni*. Táto operácia vyzerá na prvý pohľad zákerne, ale našťastie, ukáže sa, že na ňu stačí treap a nemusíme vymýšľať divokejšie dátové štruktúry.

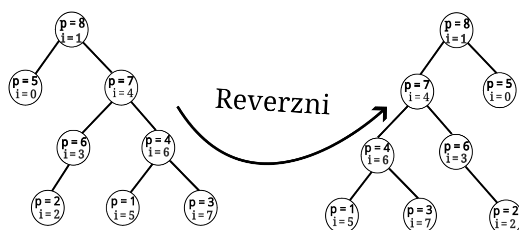
S predpokladom, že ideme upravovať nejaký treap, poďme sa najskôr zamyslieť, ako sa nám zmení výsledok pre nejaký interval, ak ho celý reverzneme. Ako sme si už predtým všimli, ak nejaký interval rozdelíme na podintervaly, ak vieme pre ne hodnoty súčtov a miním prefixových súčtov, vieme z toho poskladať výsledok na celom intervale. Konkrétne sa pozreme na jeho nový súčet a minimum prefixových súčtov.

Súčet, prirodzene, ostáva ten istý. Zato nové minimum prefixových súčtov bude minimum *sufixových súčtov*[19]. Síce ich nevieme spočítať priamo z minima prefixových súčtov, všimnite si, že podobne ako minimá prefixových súčtov, ich vieme "poskladať" zo súčtov, a miním sufixových súčtov pre postupnosť.

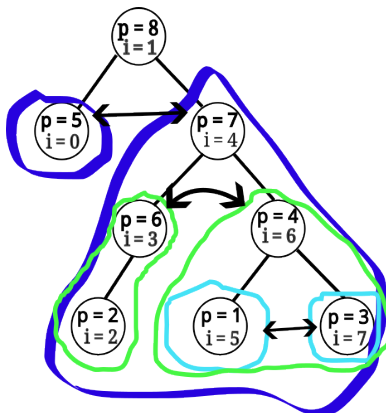
Konkrétne, ak vieme že postupnost $x_{i_1},\ldots,x_{i_2}$ má súčet s_1 a minimum sufixových súčtov S_1, a postupnosť $x_{i_2+1},\ldots,x_{i_3}$ má súčet s_2 a minimum sufixových súčtov S_2, potom je minimum sufixových súčtov postupnost $x_{i_1},\ldots x_{i_2},x_{i_2+1},\ldots x_{i_3}$ rovné $\min(S_2, s_2+S_1)$.

Mohli by sme si teda v každom vrchole aj zapamätať *minimum sufixových súčtov na príslušnom intervale.*

Teraz vieme, ako sa nám mení výsledok pre obrátený interval, tak poďme sa zamyslieť nad tým, ako by sme mohli obrátiť (pre začiatok) *celý* treap. Pozrime sa na príklad reverzu. Pre prehľadnosť sme aj reverznutému stromu ponechali pôvodné indexy v postupnosti



Všimnite si, že reverznutý treap je vlastne zrkadlový obraz pôvodného. Vieme ho tak dosiahnuť tak, že v každom podstrome vymeníme prvého a druhého syna, ako ilustruje nasledujúci obrázok.



Vedeli by sme teda, rekurzívne otočiť celý strom (a vždy tiež vymeniť hodnotu miním sufixových a prefixových súčtov) v lineárnom čase od jeho veľkosti. To je však príliš pomalé.

Odpoveďou je lenivosť – nemeňme veci, kým nemusíme. Namiesto toho aby sme otáčali celý strom naraz, zaznačme si, v jeho koreni, že ho chceme otočiť. Keď sa do koreňa nabudúce pozrieme, otočíme jeho synov a vymeníme hodnoty prefixových a sufixových súčtov, a zaznačíme si do synov, že keď do nich nabudúce prídeme,

[18] s dostatočne veľkou pravdepodobnosťou, ak priority naozaj generujete dostatočne náhodne

[19] skoro prefixové súčty, ale odzadu

majú byť otočení. Nápodobne, vždy keď dôjdeme do vrcholu v ktorom máme zaznačené, že ho treba otočiť[20], mu vymeníme deti a príslušné hodnoty miním a deti (ak nejaké má) označíme lazy flagom, že ich treba tiež otočiť. Všimnite si, že takto vždy odpovieme správnu hodnotu (keďže nepotrebujeme otáčať celý strom, aby sme získali hodnotu pre otočený podstrom), a zároveň nám to zaberie najviac konštatne-krát viac operácií (otočíme hodnoty v jednom vrchole len vtedy keby sme s ním aj tak interagovali).[21].

A čo s tým, keď nemáme celý treap, ale chceme iba otočiť jeho časť? Vieme použiť naše nápomocné funkcie *split* a *merge* – interval, ktorý chceme otočiť "vysekneme" z treapu pomocou *splitu*, označíme jeho koreň, že ho bude treba otočiť, a pomocou *merge* zlepíme strom nanovo, ako je zobrazené na nasledujúcom pseudokóde na otočenie intervalu medzi z a k (vrátane).

```
T1, T2 = split(T, z)
T2, T3 = split(T2, k - z + 1)
T2 -> otočiť = true
T = merge(T1, merge(T2, T3))
```

Pozor si treba dávať na to, že do funkcií *split* a *merge* treba implementovať, aby otočili podstrom(y) do ktorého/ých sa funkcia pozerá, ak sú označné na otočenie.

Takto dostaneme vzorové riešenie v rovnakom čase aj pamäti ako aj riešenie bez reverzu.

Záverečná otázka na zamyslenie: vedeli by sme funkciu *reverzni* implementovať v intervalovom strome?

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;
```

[20] zvyčajne sa to označuje ako "lazy flag"

[21] Celý prístup je podobný ako v lazy-propagácii intervalového stromu, viď https://www.ksp.sk/kucharka/lazy_intervalovy_strom/

```
}

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 node {
    int value, key;
    int mini, minsuf, sum, size;
    bool lazy;
    node *prvy, *druhy;

    node (int vl, int ke, node *p, node *s, bool flag): value(vl), key(ke), lazy
      ↪ (flag), prvy(p), druhy(s) {
        sum = (prvy == NULL ? 0 : prvy -> sum) + (druhy == NULL ? 0 : druhy ->
          ↪ sum) + value;
        if (prvy == NULL) {
            mini = min(0, value);
            if (druhy != NULL) {
                mini = min(mini, value + get_min(druhy));
            }
        }
        else {
            mini = min(0, min(get_min(prvy), prvy -> sum + value));
            if (druhy != NULL) {
                mini = min(mini, prvy -> sum + value + get_min(druhy));
            }
        }

        if (druhy == NULL) {
            minsuf = min(0, value);
```

```
            if (prvy != NULL) minsuf = min(minsuf, value + get_sufmin(prvy));
        }
        else {
            minsuf = min(get_sufmin(druhy), druhy -> sum + value);
            if (prvy != NULL) minsuf = min(minsuf, druhy -> sum + value +
                ↪ get_sufmin(prvy));
            minsuf = min(0, minsuf);
        }
        size = 1 + (prvy == NULL ? 0 : prvy -> size) + (druhy == NULL ? 0 :
            ↪ druhy -> size);
    }

    node *reverse() {
        return new node(value, key, prvy, druhy, !lazy);
    }

    node *propagate_lazy() {
        if (!lazy) return this;
        return new node(value, key, (druhy != NULL ? druhy -> reverse() : druhy)
            ↪ , (prvy != NULL ? prvy -> reverse() : prvy), 0);
    }

    void print(int id, string k = "") {
        cout << k << "Node #" << id + (prvy == NULL ? 0 : prvy -> size) << ": ["
            ↪  << value << ", " << key << "] lazy? " << lazy << " size = " <<
            ↪ size << " (min, sufmin) = (" << mini << ", " << minsuf  << ") sum
            ↪ = " << sum << endl;
        if (prvy != NULL) {
            cout << k << "First {" << endl;
            prvy -> print(id, k + "\t");
            cout << k << "}" << endl;
        }
        if (druhy != NULL) {
            cout << k << "Second {" << endl;
            druhy -> print(id + (prvy == NULL ? 0 : prvy -> size) + 1, k + "\t")
                ↪ ;
            cout << k << "}" << endl;
        }
    }

    int get_min(node *T) {
        if (T == NULL) return 0;
        return (T -> lazy ? T -> minsuf : T -> mini);
    }

    int get_sufmin(node *T) {
        if (T == NULL) return 0;
        return (T -> lazy ? T -> mini : T -> minsuf);
    }
};

node *merge (node *S, node *T) {
    if (S == NULL) return T;
    if (T == NULL) return S;

    S = S -> propagate_lazy();
    T = T -> propagate_lazy();
```

```
    if (S -> key > T -> key) {
        node *newsec = merge(S -> druhy, T);
        return new node(S -> value, S -> key, S -> prvy, newsec, S -> lazy);
    }
    node *newfir = merge(S, T -> prvy);
    return new node(T -> value, T -> key, newfir, T -> druhy, T -> lazy);
}

pair<node *, node *>split(node *T, int pos) {
    if (T == NULL) return {NULL, NULL};
    if (pos == 0) return {NULL, T};
    if (T -> size < pos) return {T, NULL};
    T = T -> propagate_lazy();

    if ((T -> prvy ==  NULL ? 0 : T -> prvy -> size) >= pos) {
        auto res = split(T -> prvy, pos);
        return {res.first, new node(T -> value, T -> key, res.second, T -> druhy
          ↪ , T -> lazy)};
    }

    auto res = split(T -> druhy, pos - (T -> prvy ==  NULL ? 0 : T -> prvy ->
        ↪ size) - 1);
    return {new node(T -> value, T -> key, T -> prvy, res.first, T -> lazy), res
        ↪ .second};
}

ii solve_interval(node *T, int z, int k, int sumpred) { // returns {minval, sum}
    if (T == NULL) return {sumpred, sumpred};
    T = T -> propagate_lazy();

    DBG cout << "In " << T -> key << " interval [" << z << ", " << k << ")
        ↪ sumpred = " << sumpred << endl;
    if (z >= k) return {sumpred, sumpred};
    if (z >= T -> size || k <= 0) return {sumpred, sumpred};

    if (z <= 0 && T -> size <= k) return {sumpred + T -> get_min(T), sumpred + T
        ↪  -> sum};

    auto from_first = solve_interval(T -> prvy, z, k, sumpred);
    DBG cout << "[In " << T -> key << "] From first son returns " << from_first
        ↪ << endl;

    if (T -> prvy != NULL && k <= T -> prvy -> size) return from_first;
    if ((T -> prvy != NULL ? T -> prvy -> size : 0) < z) return solve_interval(T
        ↪  -> druhy, z - 1 - (T -> prvy == NULL ? 0 : T -> prvy -> size), k - 1
        ↪ - (T -> prvy == NULL ? 0 : T -> prvy -> size), sumpred);

    int minsofar = min(from_first.first, from_first.second + T -> value);
    int sumsofar = from_first.second + T -> value;

    auto from_second = solve_interval(T -> druhy, z - 1 - (T -> prvy == NULL ? 0
        ↪  : T -> prvy -> size), k - 1 - (T -> prvy == NULL ? 0 : T -> prvy ->
        ↪ size), sumsofar);
    DBG cout << "[In " << T -> key << "] From second vracia " << from_second <<
        ↪ endl;
    return {min(minsofar, from_second.first), from_second.second};
}
```

```
node *swap_val(node *T, int p, int nv) {
    if (T == NULL) return T;
    T = T -> propagate_lazy();
    if (p < (T -> prvy == NULL ? 0 : T -> prvy -> size)) return new node(T ->
        ↪ value, T -> key, swap_val(T -> prvy, p, nv), T -> druhy, T -> lazy);
    if (p == (T -> prvy == NULL ? 0 : T -> prvy -> size)) return new node(nv, T
        ↪ -> key, T -> prvy, T -> druhy, T -> lazy);
    return new node(T -> value, T -> key, T -> prvy, swap_val(T -> druhy, p - 1
        ↪ - (T -> prvy == NULL ? 0 : T -> prvy -> size), nv), T -> lazy);
}

node* insert_node(node *T, int p, int nv, int key) {
    if (T == NULL) return new node(nv, key, NULL, NULL, 0);
    T = T -> propagate_lazy();
    if (key > T -> key) {
        auto prel = split(T, p);
        return new node(nv, key, prel.first, prel.second, 0);
    }
    if (p <= (T -> prvy == NULL ? 0 : T -> prvy -> size)) {
        return new node(T -> value, T -> key, insert_node(T -> prvy, p, nv, key)
            ↪ , T -> druhy, T -> lazy);
    }
    return new node(T -> value, T -> key, T -> prvy, insert_node(T -> druhy, p -
        ↪  (T -> prvy == NULL ? 0 : T -> prvy -> size) - 1, nv, key), T -> lazy)
        ↪ ;
}

node *erase_interval(node *T, int a, int b) {
    auto split1 = split(T, a);
    auto split2 = split(split1.second, b - a);
    return merge(split1.first, split2.second);
}

node *reverse_interval(node *T, int a, int b) {
    auto split1 = split(T, a);
    auto split2 = split(split1.second, b - a);
    return merge(split1 .first, merge((split2.first != NULL ? split2.first ->
        ↪ reverse() : NULL), split2.second));
}

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

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

    string r = get<string>();
    node *root = NULL;
    FOR(i, n) {
        root = insert_node(root, i, (r[i] == 'V' ? 1 : -1), rand());
        DBG root -> print(0);
    }

    DBG cout << "Queries: " << endl;

    FOR(i, q) {
        string query = get<string>();
```

```
        if (query == "vysledok") {
            int a = get<int>();
            int b = get<int>() + 1;
            ii res = solve_interval(root, a, b, 0);
            DBG cout << "res = " << res << endl;
            cout << (- res.first) + (res.second - res.first) + (b - a) << endl;
        }
        else if (query == "vymen") {
            int a = get<int>();
            int b = (get<char>() == 'V' ? 1 : -1);
            root = swap_val(root, a, b);
        }
        else if (query == "pridaj") {
            int a = get<int>();
            int b = (get<char>() == 'V' ? 1 : -1);
            root = insert_node(root, a, b, rand());
        }
        else if (query == "zmaz") {
            int a = get<int>();
            int b = get<int>() + 1;
            root = erase_interval(root, a, b);
        }
        else if (query == "reverzni") {
            int a = get<int>();
            int b = get<int>() + 1;
            root = reverse_interval(root, a, b);
        }
        DBG if (root != NULL) root -> print(0);
        DBG cout << "End of query " << i << ": " << query << endl;
    }
}
```