Korešpondenčný seminár z programovania
XLII. ročník, 2041/42
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola zimnej časti

Michal Staník

1. Základné zručnosti

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

Môžeme si všimnúť, že nezáleží na tom, ktoré ceruzky použije MisQo na stupienky, pretože o každej vieme, že má dĺžku aspoň 1 (garantuje nám to zadanie) a žiadne iné obmedzenia na ne nemáme.

Bočné ceruzky chceme mať čo najdlhšie, aby sme mohli mať potenciálne čo najdlhší rebrík. Keďže je nám jedno, ktoré ceruzky budú stupienky, je najlepšie použiť dve najdlhšie ceruzky zo vstupu ako bočné, nikde inde nám tieto dlhé ceruzky chýbať nebudú.

Najväčšia možná dĺžka rebríka je potom daná dĺžkou kratšej z dvoch najdlhších ceruziek – ak má druhá najdlhšia ceruzka zo vstupu dĺžku l, tak najdlhší možný rebrík má $l - 1$ stupienkov.

Druhá vec, ktorá nás obmedzuje, je celkový počet ceruziek. MisQo nemôže postaviť rebrík s viac ako $n - 2$ stupienkami (ktoré mu ostali po odložení dvoch bočných ceruziek).

Obe obmedzenia platia súčasne, teda vezmeme minimum z ich hodnôt. Ak je $n = 1$, druhá najdlhšia ceruzka neexistuje – tento prípad vyriešime osobitne a vypíšeme preň 0 – žiaden rebrík sa postaviť nedá.

Celé riešenie vieme implementovať v čase lineárnom od počtu ceruziek: $O(n)$. Stačí nám dvakrát prejsť celý zoznam dĺžok a vypočítať zopár porovnaní.

Listing programu (Python)

```
n = int(input())
if n <= 1:
    print(0)
    exit()
pencil_lengths = [int(input()) for i in range(n)]
longest = max(pencil_lengths)
pencil_lengths.remove(longest)
second_longest = max(pencil_lengths)
print(min(n - 2, second_longest - 1))
```

Pre pohodlnejšiu implementáciu najskôr nájdeme maximum dĺžok ceruziek, odstránime ho zo zoznamu pomocou funkcie `remove` a nájdeme maximum zostávajúcich dĺžok, ktoré je druhou najväčšou dĺžkou ceruzky. Je dôležité mať na pamäti, že funkcia `remove` má lineárnu zložitosť voči dĺžke zoznamu, pretože všetky prvky za odstraňovaným prvkom sa musia posunúť o jednu pozíciu doľava v zozname a týchto prvkov môže byť až $O(n)$. Tu na tom nezáleží, pretože aj tak robíme lineárne hľadanie maxima. Avšak neopatrné volanie funkcie `remove` (napr. opakovane v cykle) môže často zhoršiť zložitosť algoritmu, a preto na to treba myslieť.

Úloha sa dá implementovať v čase $O(n)$ aj s konštantnou pamäťovou zložitosťou, nepotrebujeme si totiž pamätať celý zoznam ceruziek, ale stačia nám iba aktuálne dve najdlhšie a celkový počet ceruziek.

Listing programu (Python)

```
n = int(input())

naj, dvanaj = -1, -1
for i in range(n):
    k = int(input())

    if k > naj:
```

```
        dvanaj = naj
        naj = k
    elif k > dvanaj:
        dvanaj = k

print(max(0, min(dvanaj-1, n-2)))
```

2. A sme doma

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

Prechádzanie poschodiami od z do k (Počet bodov: 4/8)

Predpokladajme, že Vašino ide postupne, po jednom, od najnižšieho poschodia z po najvyššie k. V tomto prípade, vždy keď príde na nejaké poschodie (dokopy n-krát), rozhoduje sa, či cez neho môže prejsť. Rozhodovanie znamená, že sa pozrie do poľa veľkosti p, v ktorom sú uložené čísla mysterióznych poschodí, či sa tam dané poschodie nachádza. Toto overenie mu vždy trvá $O(p)$ operácií. Ak zistí, že sa tam aktuálne poschodie nenachádza, vie, že dovtedajší súvislý úsek nemysterióznych poschodí môže zvýšiť o 1. Inak musí dĺžku znulovať. Z takto napočítaných úsekov si vyberie najdlhší. Časová zložitosť bude $O(n) + n \cdot O(p)$, čo je $O(n \cdot p)$. Pamäťová zložitosť je $O(p)$, nakoľko si pamätáme pole mysterióznych poschodí.

Mysteriózne poschodia v množine (Počet bodov: 6/8)

Postupujeme tak isto ako v predchádzajúcom riešení, len namiesto ukladania mysterióznych poschodí do poľa si ich ukladáme do dátovej štruktúry množina. Totiž overenie, či sa v množine nachádza nejaký prvok je v programovacích jazykoch zvyčajne implementované ako veľmi rýchla operácia. Dátová štruktúra `set` v Pythone a `std::unordered_set` v C++ túto operáciu podporujú v $O(1)$. Časová zložitosť teda bude len $O(n \cdot 1) = O(n)$. Pamäťová zložitosť sa nezmenila.

Listing programu (Python)

```
z, k = map(int, input().split())
p = int(input())
p = set(map(int, input().split()))


t = 0
b = 0
for i in range(z, k+1):
    if i in p:
        t = 0
    else: t += 1
    b = max(b, t)

print(b)
```

Optimálne riešenie (Počet bodov: 8/8)

Poďme teraz vyriešiť druhý problém pôvodného riešenia, a to, že prechádzame cez všetky poschodia. Všimnime si, že počet mysterióznych poschodí je oveľa menší než celkový počet poschodí. Využime tento fakt v náš prospech.

Opäť môžeme ísť zdola nahor. Keďže nemysteriózne poschodia nám cestu nekazia, nemusíme sa o ne zaujímať a môžeme ich preskočiť. Aby sme prechádzali mysetriózne poschodia naozaj v poradí zdola nahor, vopred si ich usporiadame od najmenšieho po najväčšie. Dva prvky vedľa seba potom znamenajú možnú súvislú dĺžku jazdy výťahom.

Na zjednodušenie implementácie na úvod tohoto zoznamu pridáme $z - 1$ a na záver $k + 1$. Tieto čísla budú slúžiť ako zarážky, aby aj prípadné koncové úseky poschodí boli z oboch strán ohraničené mysterióznymi poschodiami. Uvedomme si, že aj keby jedno z týchto pridaných zarážkových poschodí bolo uvedené ako mysteriózne na vstupe, fungovanie programu nám to nepokazí.

Každý súvislý bezproblémový úsek je teraz ohraničený dvoma, v našom poli susednými, mysterióznymi poschodiami. Keďže chcem nájsť najdlhší úsek, porovnávame každé dve susedné mysterózne poschodia a zapisujeme si najväčší rozdiel, ktorý kedy nameriame. Odpoveď vypíšeme.

Rýchle triediace algoritmy ako napríklad quicksort alebo mergesort majú pre n prvkov časovú zložitosť $O(n \log n)$. My samozrejme použijeme vstavané triediace algoritmy našich jazykov, ktoré majú rovnakú časovú zložitosť. Jedno prejdenie a porovnanie susedov trvá $O(p)$, čo je zanedbateľné v porovnaní s náročnejšiou operáciou triedenia. Celková časová zložitosť je teda $O(p \log p)$ – n sa nám zmenilo na p, avšak pribudol nám logaritmus (ale nezúfajte, vieme sa ho zbaviť!).

Pamäťová zložitosť stále zostáva $O(p)$, lebo rovnako ako v prvom riešení si udržiavame mysteriózne poschodia v poli dĺžky p.

Kód

Listing programu (Python)

```python
def solve(z,k,poschodia):
    poschodia = [z - 1] + sorted(poschodia) + [k + 1]

    maximalny_rozdiel = 0
    for i in range(1, len(poschodia)):
        maximalny_rozdiel = max(poschodia[i] - poschodia[i - 1] - 1, maximalny_rozdiel)

    return maximalny_rozdiel


z,k = input().split()
x = input()
nums = input().split()
poschodia = [int(i) for i in nums]

print(solve(int(z), int(k), poschodia))
```

Riešenie v ideálnom svete (Počet bodov: 8/8)

Pre isté špeciálne prípady vstupov existujú aj rýchlejšie spôsoby usporiadavania prvkov ako v $O(p \log p)$. Napríklad ak máme usporiadavať celé čísla v určenom rozsahu, môžeme použiť algoritmus bucket sort, ktorý to dokáže v čase $O(p)$. Viac o bucket sorte si môžeš prečítať napríklad tu, na wiki[1]. S jeho použitím vyriešime problém logaritmu a dostávame optimálnu časovú aj pamäťovú zložitosť $O(p)$ (lepšia nemôže byť kvôli veľkosti vstupu).

Ak si použil bucket sort, veľmi pravdepodobne si ale zistil, že kód ti bežal pomalšie než s použitím vstavaného sortu. To je normálne (štandardná knižnica sa poráža ťažko) a stále od nás dostávaš extra pochvalu.

3. Stankova dilema

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

Bruteforce

Bruteforce sa väčšinou robí tak, že vyskúšame všetky možnosti. Keďže máme zistiť, koľko dvojíc predmetov je podobnejších ako rozdielnejších, stačí vyskúšať každú dvojicu a spraviť pre ne XOR a AND. Keď bude ich XOR väčší ako ich AND, zapamätáme si to. Všetkých dvojíc je $\binom{n}{2}$. To znamená, že toto riešenie má časovú zložitosť $O(n^2)$. Pamäťovú zložitosť má $O(n)$, pretože si potrebujeme zapamätať všetky čísla zo vstupu, aby sme ich mohli navzájom porovnať.

[1] https://cs.wikipedia.org/wiki/P%C5%99ihr%C3%A1dkov%C3%A9_%C5%99azen%C3%AD

Zaujímavá myšlienka

Dve čísla sú podobnejšie ako rozdielnejšie vtedy, keď ich najľavejší jednotkový bit je na "rovnakom mieste" (nasleduje za ním v oboch číslach rovnako veľa bitov). Majme dve čísla v binárnom tvare $1\square_{k-2}\square_{k-3}...\square_1\square_0$ a $1\square_{k-2}\square_{k-3}...\square_1\square_0$, kde k je počet bitov čísla a štvorček značí bit. Ich AND je $1\square_{k-2}\square_{k-3}...\square_1\square_0$ a ich XOR je $0\square_{k-2}\square_{k-3}...\square_1\square_0$. Očividne platí, $1\square_{k-2}\square_{k-3}...\square_1\square_0 > 0\square_{k-2}\square_{k-3}...\square_1\square_0$. To znamená, že sú podobnejšie ako rozdielnejšie. Naopak, ak máme dve čísla - $1\square_{k-2}\square_{k-3}...\square_1\square_0$ a $0\square_{k-2}\square_{k-3}...\square_1\square_0$, ktoré nemajú najľavejší jednotkový bit na rovnakom mieste, tak ich AND je $0\square_{k-2}\square_{k-3}...\square_1\square_0$ a ich XOR je $1\square_{k-2}\square_{k-3}...\square_1\square_0$. Očividne platí, že $0\square_{k-2}\square_{k-3}...\square_1\square_0 < 1\square_{k-2}\square_{k-3}...\square_1\square_0$. To znamená, že sú rozdielnejšie ako podobnejšie. Z toho vyplýva, že nám stačí o čísle vedieť jeho najľavejší jednotkový bit aby sme vedeli, koľko čísel je s ním podobnejších ako rozdielnejších.

Optimálne riešenie

Vytvoríme si pole s veľkosťou b, kde b je počet bitov najväčšieho čísla. V tomto poli si budeme pamätať, koľko čísel s daným najľavejším bitom sme videli.

Najľavejší bit čísla vieme v C++ zistiť pomocou funkcie `__builtin_clz()`, ktorá spočíta počet nulových bitov na začiatku čísla (v $O(1)$!). V Pythone zasa môžeme využiť funkciu `bin()`, ktorá prevedie číslo do binárnej sústavy a vráti ho ako string. Pozíciu najľavejšieho jednotkého bitu zisíme ako `len(bin(x)) - 2`.

Keď už máme vstup prejdený a pole naplnené, zostáva nám spočítať počet dvojíc. Vieme, že ľubovoľné dve čísla s rovnakým najľavejším bitom sú viac podobné ako rozdielne. Na zistenie počtu dvojíc môžeme použiť vzorec $\frac{n_i*(n_i-1)}{2}$, kde n_i je číslo na i-tom mieste v našom poli. Alternatívne si vieme počet dvojíc počítať počas plnenia pola. Vždy pred tým, než zvýšime hodnotu pola o jedna k finálnemu výsledku pripočítame túto hodnotu pola. Zamyslite sa, že týmto postupom dostaneme rovnaký výsledok.

Časová zložitosť riešenia je $O(bn)$ a pamäťová je $O(b)$. Ale keďže b je vo všetkých vstupoch rovnaké a počet bitov sa väčšinou ako premenná vynecháva, môžeme povedať, že časová zložitosť je $O(n)$ a pamäťová je $O(1)$.

Listing programu (Python)

```python
n = int(input())
v = map(int, input().split())

p = [0 for _ in range(32)]
res = 0
for x in v:
    for i in range(31, -1, -1):
        if (x & (1 << i)) != 0:
            res += p[i]
            p[i] += 1
            break
print(res)
```

Listing programu (C++)

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

int main() {
    long long n, res = 0;
    cin >> n;
    vector<int> p(35, 0);
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        res += p[__builtin_clz(x)]++;
    }
```

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

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

4. Školské pomôcky

Bruteforce

Najjednoduchšie riešenie tejto úlohy je vyskúšať každú kombináciu pomôcok, ktorú Peťko vie v obchode nájsť. Pre prvú sadu je to jednoduché, pretože vyskúšame každú dvojicu pomôcok typu 1 a 2, a vyberieme dvojicu, ktorá sa zmestí do rozpočtu a má najvyššiu kvalitu. Pre druhú sadu je takéto riešenie pomalé a stačiť nebude.

Pre tretiu sadu vieme použiť rovnaký princíp – vyškúšať každú kombináciu pomôcok. Bude to však ťažšie ako pri prvej sade, lebo máme rôzny počet typov. Takéto niečo vieme pomerne v pohode vyriešiť rekurzívnym DFS prehľadávaním všetkých možností. Pre štvrtú sadu je takéto riešenie pomalé.

Rýchlejšie riešenie pre dva typy

Pozrime sa na trochu lepšie riešenie, ktoré prejde cez druhú sadu. Naše riešenie bude počítať s tým, že sú iba dva typy pomôcok. Pomôcky si rozdelíme do dvoch polí podľa ich typov a tieto polia zoradíme podľa kvality. Náš program pôjde zozadu oboch polí a bude sa snažiť kupovať najkvalitnejšie pomôcky. Vezmeme najkvalitnejšiu pomôcku z jedného typu a dáme ju do páru s najlacnejšou pomôckou druhého typu, ktorá má aspoň takú kvalitu ako tá najkvalitnejšia z prvého typu.

Najlacnejšiu pomôcku s aspoň takou kvalitou vieme nájsť jednoducho. Pôjdeme zozadu poľa druhého typu cez všetky pomôcky, ktoré sú väčšie alebo rovné kvalite prvej pomôcky a zistíme, ktorá je najlacnejšia. Keď sa v poli prvého typu posunieme o jednu pomôcku doľava, nemusíme v druhom poli začínať znova od konca. Stačí pokračovať od miesta kde sme naposledy prestali (premyslite si prečo). Tento prístup sa nazýva dvaja bežci. Alternatívne, keďže sme si polia utriedili, môžeme, bez zhoršenia časovej zložitosti, pomôcky v druhom poli binárne vyhľadávať.

Ak je cena tejto dvojice pomôcok vyššia ako je rozpočet, tak musíme hľadať ďalej. Takéto riešenie má časovú zložitosť $O(n \log(n) + n)$. Avšak, toto riešenie prejde iba cez prvé dve sady. My potrebujeme riešenie, ktoré počíta s tým, že počet typov pomôcok je variabilný.

Ak nevieme prísť na vzorové riešenie, ale chceme maximaizovať náš počet bodov. Vieme skombinovať toto riešenie a bruteforce pre $t > 2$ a získať body za prvé tri sady. Samozrejme treba dávať pozor, ktoré riešenie používame pri akej sade.

Vzorové riešenie

Úlohu vyriešime pomocou binárneho vyhľadávania, ktoré nám pomôže nájsť najvyššiu kvalitu pomôcok za cenu, ktorú si vie Peťko dovoliť. Binárne vyhľadáme kvalitu k, ktorá spĺňa naše požiadavky. Ak chceme dosiahnúť aspoň kvalitu k, tak sa pre každý typ pomôcky pokúsime nájsť najlacnejšiu pomôcku, ktorá má aspoň kvalitu k. Ak je súčet týchto najlacnejších pomôcok väčší ako počet peňazí, tak musíme hľadať ďalej a skúsime menšie k. Ak je súčet pomôcok menší alebo rovný ako počet peňazí, tak sme našli jedno možné riešenie, ale stále chceme pokračovať ďalej a vyskúšať väčšie k. Pokračujeme až kým sa nám neminú možnosti. V takom prípade sme buď našli najvyššiu kvalitu pomôcok, ktorú si vie Peťko dovoliť, alebo sme zistili, že také pomôcky neexistujú. Riešenie má časovú zložitosť $O(n \log(n) + n)$.

Existuje ešte ďalšie riešenie, ktoré je zovšeobecnením rýchleho riešenia pre dva typy. Pomôcky si zoradíme podľa kvality. Použijeme úplne ten istý prístup ako pri dvoch typoch – pozerieme sa na pomôcku z nejakého typu a snažíme sa nájsť súčet najlacnejších pomôcok všetkých zvyšných typov, ktoré majú aspoň takú kvalitu. Avšak, toto hľadanie musíme robiť v $O(1)$. Ak spravíme niečo ako techniku t bežcov, tak si nám stačí tento súčet aktualizovať pri posune bežca. Takéto riešenie má časovú zložitosť $O(n \log(n))$. Ak to chceme ešte úplne vyhrotiť, tak na zoradenie polí môžeme použiť bucketsort a tak bude mať naše riešenie časovú zložitosť $O(n)$. Detaily si môžete pozrieť v kóde.

Listing programu (Python)

```
def pc(a, v):
    best = -1
    for x in a:
```

```
        if x["v"] >= v:
            if best == -1:
                best = x["p"]
            else:
                best = min(best, x["p"])
    return best

def ok(a, k, q, money):
    sm = 0
    for t in range(k):
        price = pc(a[t], q)
        if price == -1:
            return False
        sm += price
    return sm <= money

t, n, m = map(int, input().split())

a = [list() for _ in range(t)]
b = 0
for i in range(n):
    it, ip, iv = map(int, input().split())
    a[it - 1].append({"p": ip, "v": iv})
    b = max(b, iv)

lo, hi = 0, b + 1
while hi - lo > 1:
    mid = (hi + lo) // 2
    if ok(a, t, mid, m):
        lo = mid
    else:
        hi = mid

print(lo)
```

Listing programu (Python)

```
# BUCKET sort the items by quality, then search items from the best quality
# Time complexity: O(n)

INF = 10 ** 12
n_typov, n_pomocok, vreckove = map(int, input().split())

kvalitove_pomocky = [[] for _ in range(5 * n_pomocok + 1)]
vstup = (map(int, input().split()) for _ in range(n_pomocok))
for typ, cena, kvalita in vstup:
    kvalitove_pomocky[kvalita].append((cena, typ - 1))

chyba_typov = n_typov
sucet_cien = 0
min_ceny = [INF] * n_typov
for kvalita in range(5 * n_pomocok, -1, -1):
```

```
    for cena, typ in kvalitove_pomocky[kvalita]:
        if min_ceny[typ] == INF:
            sucet_cien += cena
            min_ceny[typ] = cena
            chyba_typov -= 1
        elif cena < min_ceny[typ]:
            sucet_cien += cena - min_ceny[typ]
            min_ceny[typ] = cena
        if chyba_typov == 0 and sucet_cien <= vreckove:
            print(kvalita)
            exit()

print(0)
```

5. Koho ešte nepoznám?

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

Bruteforce

V prvej sade sú limity celkom malé, teda si môžeme dovoliť vygenerovať všetky možné výsledné plány Matfyzu a následne zistiť, či aspoň jeden z nich vyhovuje zadaniu. Na takéto generovanie môžeme použiť napríklad techniku bitmasiek, kde si vygenerujeme postupne všetky $n \cdot m$ bitové čísla a podľa nich vygenerujeme zábrany. Ak je nejaký bit 1, tak na dané miesto v pláne Matfyzu dáme zábranu (ak sa to teda dá), inak ju tam nedáme. Potom nám už len zostáva skontrolovať podmienky zo zadania.

Pre každého Matfyzáka z jeho pozície môžeme spustiť prehľadávanie do šírky[2] (alebo aj do hĺbky[3]). Ak sa týmto prehľadávaním dostaneme do pravého dolného rohu Matfyzu, tak sa vedia dostať von. Podobne to spravíme aj pre každého FIITáka. Ak sa každý Matfyzák a žiaden FIITák dokáže dostať von, tak sme našli správne riešenie, inak je aktuálne riešenie nesprávne. Pre každého človeka potrebujeme spustiť prehľadávanie, ktoré má časovú zložitosť $O(n \cdot m)$. Ľudí ale môže byť až $n \cdot m$, teda časová zložitosť kontroly jedného plánu je $O(n^2 \cdot m^2)$. Keďže všetkých podmnožín $n \cdot m$ prvkovej množiny je $2^{n \cdot m}$, tak celková časová zložitosť je $O(n^2 \cdot m^2 \cdot 2^{n \cdot m})$.

Listing programu (C++)

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

vector<int> dx{0, 0, 1, -1}, dy{1, -1, 0, 0};

int main(){
    int n,m;cin>>n>>m;
    vector<string> mapa(n);
    for(string &riadok : mapa)
        cin>>riadok;
    // zisti, ci je policko v mriezke
    auto v_mriezke = [&](int x,int y){return x < m && x >= 0 && y < n && y >= 0;};

    // z vyplnenej tabulky zisti, ci je podla zadania spravna
    auto check = [&](vector<string> &tmp_mapa){
        for(int start_y = 0;start_y < n;start_y++){
            for(int start_x = 0;start_x < m;start_x++){
                char policko = tmp_mapa[start_y][start_x];
```

[2] https://www.ksp.sk/kucharka/bfs/
[3] https://www.ksp.sk/kucharka/dfs/

```
                if(policko != 'M' && policko != 'F')
                    continue;
                bool dostanem_sa_von = false;
                vector<vector<int>> preskumane(n,vector<int>(m,false));
                queue<pair<int,int>> q;
                q.push({start_y,start_x});
                while(!q.empty()){
                    int x,y;
                    tie(y,x) = q.front();
                    q.pop();
                    if(preskumane[y][x]  tmp_mapa[y][x] == '#')
                        continue;
                    preskumane[y][x] = true;
                    if(x == m-1 && y == n-1)
                        dostanem_sa_von = true;
                    for(int smer = 0;smer < 4;smer++){
                        int x2 = x + dx[smer],y2 = y + dy[smer];
                        if(v_mriezke(x2,y2) && !preskumane[y2][x2])
                            q.push({y2,x2});
                    }
                }
                // ak sa vie FIITak dostat von alebo sa matfyzan nevie dostat von
                                // tak mame nespravne riesenie
                if((!dostanem_sa_von && policko == 'M')
                                        (dostanem_sa_von && policko == 'F'))
                    return false;
            }
        }
        return true;
    };

    for(int mask = 0;mask < 1<<(n*m);mask++){
        vector<string> tmp_mapa = mapa;
        for(int y = 0;y < n;y++){
            for(int x = 0;x < m;x++){
                if(mask & 1<<(y*m + x) && mapa[y][x] == '.'){
                    tmp_mapa[y][x] = '#';
                }
            }
        }
        if(check(tmp_mapa)){
            cout << "Plan uspesny\n";
            for(string r : tmp_mapa)
                cout << r << "\n";
            return 0;
        }
    }
    cout << "Neda sa\n";
}
```

Ako lepšie zablokovať FIITákov?

Ako prvé si môžeme všimnúť, že ak má byť plán úspešný, tak vo výslednom pláne Matfyzu nemôže existovať nezablokovaná cesta medzi FIITákom a Matfyzákom. Ak by taká existovala, tak FIITák príde za Matfyzákom

a od teraz ho bude len nasledovať. Potom sa buď nevie dostať Matfyzák von z budovy alebo sa FIITák dokáže dostať z budovy. Teda aspoň jedna z podmienok zo zadania nebude splnená. Teda ak sa na začiatku vedľa seba nachádza nejaký FIITák a Matfyzák, tak riešenie určite neexistuje.

Skúsme sa teraz zamyslieť nad tým, ako zabrániť FIITákom dostať sa von z budovy. Povedzme, že to spravíme najjednoduchším spôsobom, akým to ide a všetky 4 políčka okolo každého FIITáka zablokujeme (ak sa tam už niečo nenachádza). Takto od seba oddelíme FIITákov od Matfyzákov (ak nejakí nezačínali vedľa seba) a zablokujeme im východ zo školy.

Nájsť takýmto spôsobom zátarasy má časovú zložitosť $O(n \cdot m)$, lebo stačí raz prejsť celú tabuľku.

Kontrolovať riešenie môžeme rovnakým spôsobom, ako v bruteforce riešení. Výsledná časová zložitosť potom bude $O(n^2 \cdot m^2)$, čo stačí na prejdenie prvých 2 až 3 sád.

Listing programu (C++)

```cpp
#include <bits/stdc++.h>
using namespace std;
#define f first
#define s second

vector<int> dx{0, 0, 1, -1}, dy{1, -1, 0, 0};

int main() {
    int n, m;
    cin >> n >> m;
    vector<string> mapa(n);
    for (auto &riadok: mapa)
        cin >> riadok;

    auto v_mriezke = [&](int x, int y) { return x >= 0 && x < m && y >= 0 && y < n; };

    // obkolesenie FIITakov
    for (int y = 0; y < n; y++) {
        for (int x = 0; x < m; x++) {
            if (mapa[y][x] != 'F')
                continue;
            for (int smer = 0; smer < 4; smer++) {
                int x2 = x + dx[smer], y2 = y + dy[smer];
                if (!v_mriezke(x2, y2))
                    continue;
                if (mapa[y2][x2] == 'M') {
                    cout << "Neda sa\n";
                    return 0;
                } else if (mapa[y2][x2] != 'F')
                    mapa[y2][x2] = '#';
            }
        }
    }

    // pomocou BFS zistime, ci sa vie dany clovek dostat von
    for (int start_y = 0; start_y < n; start_y++)
        for (int start_x = 0; start_x < m; start_x++) {
            if (mapa[start_y][start_x] == 'F'  mapa[start_y][start_x] == 'M') {
                bool dostanem_sa_von = false;
                vector<vector<int>> preskumane(n, vector<int>(m, false));
                queue<pair<int, int>> q;
```

```
                q.push({start_y, start_x});
                while (!q.empty()) {
                    int y, x; tie(y, x) = q.front(); q.pop();
                    if (preskumane[y][x]  mapa[y][x] == '#')
                        continue;
                    if (y == n - 1 && x == m - 1) { // dokazeme sa dostat k vychodu
                        dostanem_sa_von = true;
                        break;
                    }
                    preskumane[y][x] = true;
                    for (int smer = 0; smer < 4; smer++) {
                        int x2 = x + dx[smer], y2 = y + dy[smer];
                        // prehladavame len ak sa nachadzame v mriezke
                        if (v_mriezke(x2, y2) && !preskumane[y2][x2])
                            q.push({y2, x2});
                    }
                }
                if ((mapa[start_y][start_x] == 'F' && dostanem_sa_von)
                    (mapa[start_y][start_x] == 'M' && !dostanem_sa_von)) {
                    cout << "Neda sa\n";
                    return 0;
                }
            }
        }
    cout << "Plan uspesny\n";
        for (auto &riadok: mapa)
            cout << riadok << "\n";
}
```

Prečo toto funguje?

Predstavme si, že sme takto obkolesili každého FIITáka, teda sa žiaden z nich nedokáže dostať von. Teda ak nie je naše riešenie správne, tak sme museli zablokovať aspoň jedného z Matfyzákov (takého, ktorý sa predtým vedel dostať von). Ak by sme chceli tohto Matfyzáka odblokovať, tak musíme aspoň jednu zo zábran, ku ktorej sa tento Matfyzák vie dostať odstrániť, aby mohol prejsť ďalej. Potom ale určite existuje nezablokovaná cesta medzi FIITákom a Matfyzákom, lebo nami pridané zábrany susedia priamo z FIITákmi. Teda riešenie by aj tak neexistovalo.

Optimálne riešenie

Zisťovanie, či je dané riešenie správne, je zatiaľ príliš pomalé. Ako by sa teda dalo zrýchliť? Asi by bolo treba spúšťať len jedno prehľadávanie. Spravíme to teda opačne a prehľadávanie spustíme z pravého dolného rohu Matfyzu. Počas tohto prehľadávania budeme počítať, na koľko Matfyzákov sme zatiaľ narazili. Keďže sa dá dostať z východu k danému Matfyzákovi, tak sa dá aj od toho Matfyzáka dostať k východu. Teda stačí len porovnať celkový počet Matfyzákov na začiatku s počtom Matfyzákov, ktorých sme prešli pri prehľadávaní. Keďže spúšťame už len jedno prehľadávanie, tak toto riešenie už bude mať časovú zložitosť $O(n \cdot m)$ a pamäťovú zložitosť $O(n \cdot m)$, teda by malo dostať 8 bodov.

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;
#define f first
#define s second

```

```cpp
vector<int> dx{0, 0, 1, -1}, dy{1, -1, 0, 0};
int main() {
    int n, m;
    cin >> n >> m;
    vector<string> mapa(n);
    for (auto &riadok: mapa)
        cin >> riadok;

    auto v_mriezke = [&](int x, int y) { return x >= 0 && x < m && y >= 0 && y < n; };

    // obkolesenie FIITakov a spocitanie poctu Matfyzakov
    int pocet_matfyzakov = 0;
    for (int y = 0; y < n; y++) {
        for (int x = 0; x < m; x++) {
            if (mapa[y][x] == 'F') {
            for (int smer = 0; smer < 4; smer++) {
                int x2 = x + dx[smer], y2 = y + dy[smer];
                if (!v_mriezke(x2, y2))
                    continue;
                if (mapa[y2][x2] == 'M') {
                    cout << "Neda sa\n";
                    return 0;
                } else if (mapa[y2][x2] != 'F')
                    mapa[y2][x2] = '#';
            }
            } else if (mapa[y][x] == 'M')
                pocet_matfyzakov++;
        }
    }

    // BFS z vychodu
    int dosiahnutelni_matfyzaci = 0;
    vector<vector<int>> preskumane(n, vector<int>(m, false));
    queue<pair<int, int>> q;
    q.push({n - 1, m - 1});
    while (!q.empty()) {
        int y, x; tie(y, x) = q.front(); q.pop();
        if (preskumane[y][x]  mapa[y][x] == '#')
            continue;
        preskumane[y][x] = true;
        if (mapa[y][x] == 'M')
            dosiahnutelni_matfyzaci++;
        for (int smer = 0; smer < 4; smer++) {
            int x2 = x + dx[smer], y2 = y + dy[smer];
            // prehladavame len ak sa nachadzame v mriezke
            if (v_mriezke(x2, y2) && !preskumane[y2][x2])
                q.push({y2, x2});
        }
    }

    if (dosiahnutelni_matfyzaci != pocet_matfyzakov)
        cout << "Neda sa\n";
    else {
```

```
        cout << "Plan uspesny\n";
        for (auto &riadok: mapa)
            cout << riadok << "\n";
    }
}
```

Listing programu (Python)

```python
#!/usr/bin/env python3
from collections import deque

n, m = map(int, input().split())
mapa = [list(input()) for _ in range(n)]

# pomocne polia na jednoduchsie hladanie susedov daneho policka
dx = (0, 0, 1, -1)
dy = (1, -1, 0, 0)


# pomocna funkcia na zistenie, ci je dane policko vnutry mriezky
def v_mriezke(x: int, y: int):
    return 0 <= x < n and 0 <= y < m


# obkolesenie FIITakov a spocitanie poctu Matfyzakov
pocet_matfyzakov = 0
for y in range(n):
    for x in range(m):
        if mapa[y][x] == "F":
            for smer in range(4):
                x2, y2 = x + dx[smer], y + dy[smer]
                if not v_mriezke(y2, x2):
                    continue
                if mapa[y2][x2] == "M":
                    # ak je vedla seba Matfyzak a FIITak, tak riesenie neexistuje
                    print("Neda sa")
                    quit()
                elif mapa[y2][x2] != "F":
                    # treba si dat pozor, aby sme nedali zabranu na policko s FIITakom
                    mapa[y2][x2] = "#"
        elif mapa[y][x] == "M":
            pocet_matfyzakov += 1

# pustime BFS z vychodu
dosiahnutelni_matfyzaci = 0
preskumane = [[0] * m for _ in range(n)]
fronta = deque([(n - 1, m - 1)])

while fronta:
    y, x = fronta.popleft()
    if preskumane[y][x] or mapa[y][x] == "#":
        continue
```

```
    preskumane[y][x] = 1
    if mapa[y][x] == "M":
        dosiahnutelni_matfyzaci += 1
    for smer in range(4):
        x2, y2 = x + dx[smer], y + dy[smer]
        if v_mriezke(y2, x2):
            fronta.append((y2, x2))

if dosiahnutelni_matfyzaci != pocet_matfyzakov:
    print("Neda sa")
else:
    print("Plan uspesny")
    for riadok in mapa:
        print("".join(riadok))
```

Iné zaujímavé riešenie

Môžeme si všimnúť, že to, ako má vyzerať nejaký riadok vo výsledku je jednoznačne určené tým, ako vyzerá riadok nad ním a riadok pod ním. Teda zistiť, kde chceme pridať v aktuálnom riadku zábrany je pomerne jednoduché. Zaujímavejší problém je ale zistiť, či je toto vyplnenie správne alebo nie, ak máme k dispozícii len 3 po sebe idúce riadky. Rozdeľme si políčka do komponentov tak, že z každého políčka v komponente sa dá dostať do každého iného v tom komponente. Potom si pre každý riadok stačí pamätať, ktoré políčko patrí do ktorého komponentu a koľko Matfyzákov sa nachádzaz v ktorom komponente. Aby bolo riešenie správne, tak komponent obsahujúci východ z Matfyzu musí obsahovať všetkých Matfyzákov. Ostáva nám už len dopočítať z toho, ako vyzerajú komponenty v predošlom riadku to, ako majú vyzerať v aktuálnom. Komponenty si budeme pamätať v dátovej štruktúre UnionFind, v ktorej vieme v čase $O(\alpha(n))$ zistiť, v ktorom komponente je prvok alebo spojiť 2 komponenty. (kde $\alpha(n)$ je inverzná Ackermannova funkcia[4], ktorá je pre ľubovoľné rozumne predstaviteľné čísla menšia ako 5) Ak sa na políčku i v aktuálnom riadku nenachádza stena, tak bude v rovnakom komponente, ako políčko i v predošlom riadku. Potom nám už len stačí spojiť komponenty susedných políčok v aktuálnom riadku a pridať Matfyzákov ku komponentom, do ktorých patria.

Úlohu teda vyriešime v dvoch prechodoch. V prvom zistíme pomocou vyššie uvedeného algoritmu, či riešenie existuje a ak hej, tak ho v druhom prechode nájdeme a vypíšeme. Toto riešenie má síce zanedbateľne horšiu časovú zložitosť $O(\alpha(m) \cdot n \cdot m)$, ale okrem vstupu si musí pamätať len $O(m)$ pamäte navyše. A hlavne, ak by ako výstup stačilo iba určiť úspešnosť plánu (keďže výsledný plán je triviálne skonštruovať), postačoval by nám jeden prechod a pamäťová zložitosť by bola iba $O(m)$.

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;
#define len(x) ((int)(x.size()))

// The time complexity is O(Row*Col*alpha(Col)).
// Take notice of how we often we use find() and join() in the code:
// The UF trees will be at most just a couple of nodes deep at any given time.
// The running time is in our test data between 1x-1.2x of the O(Row*Col) solution.

// negative number parent[x] means root with abs(parent[x]) nodes
// positive number parent[x] means child of parent[x]
struct UF {
    vector<int> parent;
    vector<int> has_matfyz;
    UF() {}
```

[4] https://cs.wikipedia.org/wiki/Ackermannova_funkce#Inverzn%C3%AD_funkce

```
16      int find(int x) {
17          if (parent[x] < 0)
18              return x;
19          return parent[x] = find(parent[x]);
20      }
21      void join(int x, int y) {
22          if (x == -1  y == -1)
23              return;
24          x = find(x);
25          y = find(y);
26          if (x == y)
27              return;
28          if (parent[x] > parent[y])
29              swap(x, y);
30          parent[x] += parent[y];
31          parent[y] = x;
32          has_matfyz[x] += has_matfyz[y];
33      }
34      void make_root(int x) { // make x root of its tree
35          int y = find(x);
36          if (x == y)
37              return;
38          parent[x] = parent[y]; // set size
39          parent[y] = x;
40          has_matfyz[x] = has_matfyz[y];
41      }
42      void add(int n) {
43          parent.resize(len(parent) + n, -1);
44          has_matfyz.resize(len(parent), 0);
45      }
46  };
47
48  enum {
49      EMPTY = 0,
50      WALL = -1,
51      FIIT = -2,
52      MATFYZ = -3,
53  };
54
55  void merge(vector<int>& row1, vector<int>& row2, UF& uf) {
56      uf.add(len(row1));
57      int Col = len(row1) - 2;
58      for (int x = 1; x <= Col; x++) { // fill new row
59          int ui = len(uf.parent) / 2 + x;
60          uf.parent[ui] = -1;
61          uf.has_matfyz[ui] = row2[x] == MATFYZ;
62          if (row2[x] != WALL)
63              row2[x] = ui;
64      }
65
66      for (int x = 1; x <= Col; x++) // merge vertically
67          uf.join(row1[x], row2[x]);
68      for (int x = 1; x <= Col; x++) // merge horizontally
```

```
        uf.join(row2[x], row2[x - 1]), uf.join(row2[x], row2[x + 1]);

    int shift = len(row1);
    for (int i = len(uf.parent) - 1; i > shift; i--) // lets move roots to second row
        if (uf.find(i) < shift)
            uf.make_root(i);
    for (int x = 1; x <= Col; x++) // lets relabel them in preparation for shifting
        if (row2[x] != WALL)
            row2[x] = uf.find(row2[x]) - shift;

    for (int i = shift; i < len(uf.parent); i++) { // shift uf row2 to row1
        uf.parent[i - shift] = uf.parent[i] < 0 ? uf.parent[i] : uf.parent[i] - shift;
        uf.has_matfyz[i - shift] = uf.has_matfyz[i];
    }
    uf.parent.resize(len(uf.parent) - shift); // delete uf row2
}

// wall out empty cells around Fs
bool block_rows(vector<int>& row1, vector<int>& row2, bool wall_fs) {
    int Col = len(row1) - 2;
    for (int x = 1; x <= Col; x++) {
        if (row2[x - 1] == FIIT  row2[x + 1] == FIIT  row1[x] == FIIT) {
            if (row2[x] == 0)
                row2[x] = WALL;
            else if (row2[x] == MATFYZ) // we would block out MATFYZ
                return false;
            if (wall_fs && row1[x] == FIIT) // for simplicity, we replace Fs with walls
                row1[x] = WALL;
        }
        if (row2[x] == FIIT) {
            if (row1[x] == 0)
                row1[x] = WALL;
            else if (row1[x] == MATFYZ) // we would block out MATFYZ
                return false;
        }
    }
    return true;
}

const string chars = ".#FM";
void char_to_int(vector<char>& row, vector<int>& row2) {
    for (int x = 1; x < len(row) - 1; x++)
        row2[x] = -chars.find(row[x]);
}

void print_row(vector<int>& row) {
    for (int x = 1; x < len(row) - 1; x++)
        cout << chars[-row[x]];
    cout << "\n";
}

int main() {
    ios_base::sync_with_stdio(false);
```

```
122     cin.tie(NULL);
123
124     int Row, Col;
125     cin >> Row >> Col;
126
127     // We will read it into memory, but use it in single pass when checking
128     // and then in single pass when printing
129     vector<vector<char>> grid(Row + 2, vector<char>(Col + 2, '#'));
130     for (int y = 1; y <= Row; y++)
131         for (int x = 1; x <= Col; x++)
132             cin >> grid[y][x];
133
134     bool doable = true;
135     UF uf;
136     uf.add(Col + 2);
137     vector<int>
138         row1(Col + 2, -1), // done row
139         row2(Col + 2, -1), // processing row
140         row3(Col + 2, -1); // helper row for preparing row2 (because Fs will be walled out in row2)
141     int matfyz_cnt = 0;
142     for (int y = 1; y <= Row + 1; y++) { // fully walled last row included
143         string chars = ".#FM";
144         for (int x = 1; x <= Col; x++) {       // initial fill
145             row3[x] = -chars.find(grid[y][x]); // empty=0, wall=-1, fiit=-2, matf=-3
146             matfyz_cnt += row3[x] == MATFYZ;
147         }
148         doable &= block_rows(row2, row3, true);
149         if (!doable)
150             break;
151
152         merge(row1, row2, uf);
153         row1 = row2;
154         row2 = row3;
155     }
156
157     // notice that corner_group can be -1 if corner is wall, but that can be ok if there are no Ms
158     int corner_group = uf.find(row1[Col]);
159     doable &= (corner_group == -1 ? 0 : uf.has_matfyz[corner_group]) == matfyz_cnt;
160     cout << (doable ? "Plan uspesny\n" : "Neda sa\n");
161     if (!doable)
162         return 0;
163
164     row1.assign(Col + 2, -1);
165     row2.assign(Col + 2, -1);
166     for (int y = 1; y <= Row + 1; y++) { // same as above, but instead of merging we print
167         string chars = ".#FM";
168         for (int x = 1; x <= Col; x++)
169             row3[x] = -chars.find(grid[y][x]);
170         block_rows(row2, row3, false);
171
172         row1 = row2;
173         row2 = row3;
174         if (y >= 2)
```

```
175             print_row(row1);
176         }
177     }
```

Jakub Konc

6. Oči veľké

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

Bruteforce

Najjednoduchší bruteforce je jednoducho vyskúšať všetky možné priradenia študovaných predmetov ku dňom a skontrolovať, ktoré predmety by sme urobili. Toto je však samozrejme príliš pomalé. Časová zložitosť je $O(nD^n)$, kde $D = \max d_i$.

Ako toto vieme zrýchliť?

Lepší bruteforce

Môžeme si všimnúť, že keď už raz začneme študovať nejaký predmet, oplatí sa nám ho celý doštudovať a až potom začínať s ďalším.

Predstavme si totiž, že by v optimálnom riešení existovala dvojica predmetov A a B, také, že A sme dokončili skôr ako B, ale ešte pred jeho dokončením sme začali študovať B. Tu vidíme, že keby sme najprv dokončili A, stále by sme mali platné riešenie - A by sme doštudovali skôr a koniec B by sa nezmenil. Tu teda vidíme, že existuje optimálne riešenie, v ktorom každý predmet študujeme v jednom kuse.

Toto znamená, že predmety môžeme vnímať ako súvislé bloky. Zaujíma nás teda len to, ktoré predmety sa rozhodneme študovať a v akom poradí.

Z tohoto sa nám formuje rýchlejšie riešenie. Vygenerujeme si všetky podmnožiny predmetov a pre každú z nich všetky jej permutácie. Pre každú z nich si skontrolujeme, že zo všetkých predmetoch v nej urobíme skúšku. Zo všetkých, pre ktoré je odpoveď áno zoberieme tú s najviac kreditmi a tá je riešením. Podľa toho, ako šikovne toto implementujeme sa bude časová zložitosť pohybovať medzi $O(n!)$ a $O(n^2 n!)$, čo stále nepostačuje na získanie bodov.

Ďalšie pozorovanie

Tu ale prichádza na radu druhé kritické pozorovanie. Dokážme si, že v optimálnom riešení budeme predmety študovať v poradí ich skúšok. Keby sme mali predmety A a B kde A má termín skôr ako B ale B sa učíme skôr ako A, môžeme ich poradie vymeniť. A tak dokončíme skôr a B vtedy, čo pôvodne A. To je ale v poriadku, keďže vieme, že A má termín skôr ako B. V nasledujúcich riešeniach si teda predmety na začiatku zoradíme a tým pádom nám vypadáva nutnosť riešiť v akom poradí ich budeme študovať.

A tu sa nám pomerne priamočiaro rodí ďalšie riešenie. Prejdeme si všetkými podmnožinami predmetov a pre každú skontrolujeme, či dokážeme všetky jej predmety urobiť. Takéto riešenie má časovú zložitosť okolo $O(n2^n)$ a po jeho odovzdaní sme odmenení dvoma bodmi.

Dynamika

A v tomto bode už skúseného KSP-čkara neprekvapí, že optimálnym riešením bude dynamické programovanie.

Zamyslime sa teda, čo by sme chceli mať ako stav tejto dynamiky. Keďže ideme robiť dynamiku, budeme mať nejaké malé podproblémy a budeme ich rozširovať na väčšie. Teda, aby sme vedeli, či vieme do riešenia pridať nejaký predmet budeme potrebovať skontrolovať či by sme ho vôbec stíhali, a teda v stave určite budeme potrebovať mať počet dní zatiaľ strávených štúdiom j. Tiež sa však potrebujeme uistiť, že predmet do jedného riešenia nepridáme dvakrát, tak si pamätajme aj posledný študovaný predmet i. Ak $i = 0$, znamená to, že sme žiadny predmet ešte nevybrali.

Pre $i = 0$ je riešenie jasné - žiadne kredity získať nevieme. Ako teraz spočítame hodnoty stavov s nejakým $i \geq 1$ ak vieme riešenia pre všetky stavy s menším i? Najprv si skontrolujeme, či by sme takto vôbec predmet doštudovali pred skúškou, teda či $j \leq d_i$. Ak nie, hodnota stavu je samozrejme 0. Inak si prejdeme všetkými možnými predošlými predmetmi a zoberieme najlepšiu možnosť - $\mathrm{dp}[i][j] = \max_{0 \leq x \leq i-1} \mathrm{dp}[x][j - t_i] + k_i$. Toto nám dáva časovú zložitosť $O(n^2 D)$, čo je dostatočne rýchle na získanie šiestich bodov.

Optimálne riešenie

Optimalizácia, ktorá nám získa posledné dva body je pomerne jednoduchá a podobá sa klasickému 0-1 knapsacku. Prestaňme vynucovať, že i je posledný predmet – nech len hovorí, že žiadny predmet po ňom

sme už nepoužili. Toto ale znamená, že pri prechode nebudeme musieť prechádzať všetky x menšie ako i, ale len $x = i - 1$, pretože všetky ostatné v ňom budú zahrnuté. Nový prechod tak bude jednoducho $\mathrm{dp}[i][j] = \max(\mathrm{dp}[i-1][j], \mathrm{dp}[i-1][j-t_i] + k_i)$ ak $j \leq d_i$, inak proste $\mathrm{dp}[i-1][j]$. Toto nám dáva časovú aj pamäťovú zložitosť $O(n \log n + nD)$. Môžeme si však všimnúť, že pri počítaní stavov s i potrebujeme len stav $i - 1$, čo znamená, že ostatné vieme postupne zahadzovať a tak mať pamäťovú zložitosť $O(n + D)$.

Listing programu (C++)

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

struct Predmet {
    int k, d, t;
    Predmet(int ki, int di, int ti) : k(ki), d(di), t(ti) {}
};

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

    vector<Predmet> predmety;
    for (int i = 0; i < n; i++) {
        int ki, di, ti;
        cin >> ki >> di >> ti;
        predmety.push_back(Predmet(ki, di, ti));
    }

    // predmety zoradíme podľa di
    sort(predmety.begin(), predmety.end(),
         [](Predmet a, Predmet b) { return a.d < b.d; });

    int D = 0;
    for (auto i : predmety) D = max(D, i.d);

    vector<int> dp(D + 1, 0);

    for (int i = 0; i < n; i++) {
        vector<int> new_dp(D + 1, 0);  // spravíme si novy vektor, aby sme
                                       // nepouzili jeden predmet dvakrat
        for (int t = 0; t <= D; t++) {
            int zaciatok = t - predmety[i].t;
            if (t > predmety[i].d  zaciatok < 0) {
                new_dp[t] = dp[t];
            } else {
                new_dp[t] = max(dp[t], dp[zaciatok] + predmety[i].k);
            }
        }
        dp = new_dp;
    }
    int res = 0;
    // riesenim je maximum celej tabulky
    for (auto i : dp) res = max(res, i);
    cout << res << endl;
```

```
}
```

JaNo

7. Logistika usadenia

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

Trocha netradične si najprv popíšeme vzorové riešenie a až potom pomalšie riešenie. Pomalšie riešenie navyše funguje aj na všeobecnejšiu úlohu, takže aj keď už viete vzorák, môže byť pre vás stále zaujímavé.

Vzorové riešenie

Máme popísané nejaké pravidlo, ktoré vraví že nemôže byť zároveň obsadená aj stolička A aj stolička B, ak A a B sú v susedných stĺpcoch a v rovnakých alebo susedných radoch.

V zadaní je to napísane trocha inak, ale význam je rovnaký. A všimnime si, že formulácia v tomto odseku je viac očividne symetrická, ak vo vete vymeníme "A" a "B" tak sa nezmení význam. Symetria je dôležitá, pretože, keď si triedu pred predstavíme ako graf – dobré stoličky sú vrcholy grafu a susedné stoličky, ktorých obsadenie sa navzájom vylučuje, sú hrany grafu – tak dostaneme jednoduchý neorientovaný graf.

Druhá dôležitá vec je, že tento graf je bipartitný ("jáááj", počujeme ozývať sa vzdychy riešiteľov, ktorým práve napadlo riešenie.). Bipartitný graf je taký, ktorý nemá cyklus nepárnej dĺžky.

No a zistiť, koľko najviac detí vieme usadiť do triedy je ekvivalentné tomu, koľko najviac vrcholov v grafe vieme ofarbiť zelenou farbou tak, aby žiadne dva susedné vrcholy neboli ofarbené (zelené vrcholy = obsadené stoličky). Ešte trochu formálnejšie napísané, chceme vybrať najväčšiu množinu vrcholov S tak, aby žiadne dva vrcholy v S neboli susedné.

Toto je už známy grafový problém, ktorý sa volá "Najväčšia nezávislá množina", a ktorý sa v bipartitných grafoch rieši pomerne ľahko.

Vzorové riešenie tohto problému nájdete v našej kuchárke[5], najskôr sa potrebujete prehrýzť cez "Maximálne párenie v bipartitnom grafe" a z neho sa dá potom spočítať aj najväčšia nezávislá množina. Všetko je tam pekne popísané, ja osobne najviac odporúčam Hopcroftov-Karpov algoritmus, lebo má veľmi jednoduchú a efektívnu implementáciu, keď viete ako na to (ak neviete, pozrite kód nižšie).

Vo všeobecnosti má tento algoritmus časovú zložitosť $O(E\sqrt{V})$, kde E je počet hrán a V počet vrcholov. V našom prípade máme $O(mn)$ hrán a $O(mn)$ vrcholov, takže zložitosť je $O((mn)^{1.5})$. Pamäťová zložitosť je $O(mn)$.

Listing programu (Python)

```python
import sys
sys.setrecursionlimit(40234)

m, n = [int(x) for x in input().split()]
grid = [input() for _ in range(m)]
pair = [[None for _ in range(n)] for _ in range(m)]

# Returns True if found an augmenting path from (i, j)
# and updates the matching if such path was found
def find_improvement(i: int, j: int) -> bool:
    visited[i][j] = True  # only needed for calculating max_IS
    for ii in (i - 1, i, i + 1):
        for jj in (j - 1, j + 1):
            if (
                0 <= ii < m
                and 0 <= jj < n
                and grid[ii][jj] == "."
                and not visited[ii][jj]
            ):
                visited[ii][jj] = True
                if pair[ii][jj] is None:
```

[5] https://www.ksp.sk/kucharka/parenie/%22

```
                    pair[ii][jj] = (i, j)
                    pair[i][j] = (ii, jj)
                    return True
                else:
                    mi, mj = pair[ii][jj]
                    if find_improvement(mi, mj):
                        pair[ii][jj] = (i, j)
                        pair[i][j] = (ii, jj)
                        return True
    return False


# Find maximum matching using Simplified Hopcroft-Karp.
# In each round we find "maximal disjoint set of augmenting paths".
# Note the set does not need to be maximum in size.
improved = True
while improved:
    improved = False
    visited = [[0 for _ in range(n)] for _ in range(m)]
    for i in range(m):
        for j in range(0, n, 2):
            if grid[i][j] == "." and pair[i][j] is None:
                if find_improvement(i, j):
                    improved = True

# Conveniently the last matching pass, found all visited vertices needed to calculate max_IS
max_is = 0
for i in range(m):
    for j in range(n):
        if grid[i][j] == ".":
            max_is += (j % 2) ^ visited[i][j]

print(max_is)
```

Pomalšie riešenie – dynamické programovanie

Ale čo mám ja robiť, keď som nikdy nepočul o bipartitných grafoch a nezávislých množinách? V tejto časti si ukážeme pomalšie riešenie, ktoré neuvažuje probléme ako o grafe a navyše by fungovalo, aj keby daný graf nebol bipartitný. (Napríklad, keby sa nebolo možné sedieť vedľa nikoho vo všetkých ôsmych smeroch, alebo keby iba nebolo možné sedieť na pozícii (x, y), ak je obsadené ľubovoľné z políčok $\{(x-1, y-1), (x-1, y-2), (x, y-3)\}$.)

Dalo by sa začať obyčajným bruteforcom, postupne prechádzame políčka po stĺpcoch. Ak sa na dané políčko nedá posadiť (lebo je pokazené, alebo je obsadené niektoré z ľavých susedov) musíme nechať políčko voľné, v opačnom prípade vyskúšame obe možnosti obsadenia. Toto riešenie by v najhoršom prípade skúšalo dokopy 2^{mn} možností, ale dokážeme ho zlepšiť memoizáciou či prerobením na dynamické programovanie.

Kľúčové pozorovanie je, že keď už sme spracovali k políčok, obsadenosť políčok $[0..k-m)$ už nebude ovplyvňovať, ktoré budúce políčka môžeme alebo nemôžeme obsadiť neskôr.

Len pre ilustráciu si predstavme, že máme triedu s $m = 5$ radmi:

```
?????????A--------
????????AA--------
????????A---------
????????A---------
????????A---------
```

Keď sme spracovali prvých 47 políčok (označené `?` a `A`), tak políčka označené `?` už nezasahujú do toho, kto môže sedieť na políčkach označených `-`.

Ak sa chceme posunúť v dynamickom programovaní o políčko ďalej, viď druhý obrázok, vyskúšame všetkých 2^{m+1} možností, ako môžu vyzerať políčka `B`.

```
??????????B--------
??????????B--------
????????BB--------
????????B---------
????????B---------
```

Na odpovedanie otázky "Koľko najviac detí môžeme usadiť na prvých $k+1$ políčok ak zároveň obsadenie posledných $m+1$ políčok zodpovedá binárnemu reťazcu B?" nám stačí poznať už spočítané odpovede pre k políčok a všetky možné binárne reťazce A dĺžky $m+1$.

Domyslenie detailov necháme na čitateľa ako cvičenie, pri implementácii použijeme bitové operácie na efektívnu prácu s binárnymi reťazcami.

Celková časová zložitosť riešenia je $O(2^m mn)$, pamäťová zložitosť je $O(2^m)$, lebo si stačí pamätať len posledné dve vrstvy (k a $k+1$) tabuľky dynamického programovania.

Listing programu (C++)

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

int main() {
    int m, n;
    cin >> m >> n;
    vector<string> trieda(m);
    for (int i = 0; i < m; i++) cin >> trieda[i];

    int m2 = 1 << (m + 1);
    vector<int> ans(m2, 0);
    for (int j = 0; j < n; j++)
         for (int i = 0; i < m; i++) {
            vector<int> newans(m2, 0);

            int check = 1 << (m - 1);
            if (i != 0) check = 1 << m;
            if (i != m - 1) check = 1 << (m - 2);

            bool free = trieda[i][j] == '.';
            for (int k = 0; k < m2; k++) {
                int shifted = (k << 1) & (m2 - 1);
                newans[shifted] = max(newans[shifted], ans[k]);
                if (free && !(k & check)) {
                    newans[shifted + 1] = max(newans[shifted + 1], ans[k] + 1);
                }
            }
            ans = move(newans);
        }
    int best = 0;
    for (int x : ans) best = max(best, x);
    cout << best << "\n";
}
```

Hodobox

8. Ako prežiť domáce úlohy

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

Riešenia na našu úlohu sa líšia v tom, koľko práce potrebujú spraviť na aktualizáciu trvania domácej úlohy (a všetkej informácie, ktorú potrebujú mať vypočítanú, aby rýchlo odpovedali na otázku), a za akých podmienok to zvládnu.

V prvej sade si vieme dovoliť pre každú otázku zrátať informácie pomaly.

V druhej aj tretej sade si potrebujeme informácie zrátať rýchlo, no máme to uľahčené: v druhej sade bude otázka vždy rovnaká, a v tretej sade budú otázky rôzne, nebudeme však musieť aktualizovať trvania úloh.

Na plný počet potom už musíme vedieť aktualizovať dĺžky úloh rýchlo, a zároveň vedieť rýchlo odpovedať na všetky možné otázky.

Vymieňanie úloh

Pri zodpovedaní konkrétnej otázky x nás nezaujíma konkrétne poradie úloh v zozname, keďže Samo vie vymieňať úlohy bezohľadu na ich umiestnenie. Relevantné je len to, koľko úloh každej dĺžky 1 až 3 je vrámci prvých x, a koľko vrámci zvyšku.

Ozbrojení týmito šiestimi číslami potom vieme zrátať najmenší počet výmen (alebo že to nie je možné) pažravo: najprv chceme povymieňať úlohy dĺžky 3 za úlohy dĺžky 1, keďže tieto výmeny zmenia rozdiel súčtov oboch častí zoznamu o 4, a ak to nestačí, povymieňame potrebný počet úloh dĺžky 3 za 2, a 2 za 1.

Ako sa môžeme uistiť, že takáto pažravá stratégia funguje? Zvolíme prístup, ktorý sa často hodí pri riešení úloh: dokážeme, že ak existuje *nejaké* riešenie, tak ho vieme transformovať na naše *pažravé* riešenie.

Pozrime sa teda na *nejaké* riešenie (BÚNV je súčet úloh vľavo väčší ako vpravo). Ak sme v ňom nejakú úlohu vymenili viac ako raz, teda vymeníme úlohy a, b a niekedy neskôr c, a, tak namiesto toho sme mohli rovno vymeniť c, b. Dôsledok je aj to, že nikdy nevymeníme najprv 3 za 2 a potom 2 za 1. Ak niekedy vymeníme 1 za 2 (kratšiu za dlhšiu), musíme to nejakou výmenou odčiniť; ak neskôr vymeníme 2 za 1 alebo 3 za 1, platí predošlá úvaha, inak vieme vymienať len 3 za 2, čo sme však mohli aj predtým (nezískali sme ani 3ku vľavo ani 2ku vpravo navyše), a teda vieme túto výmenu 1,2 a jednu 3,2 vynechať. Podobne vieme ukázať, že nemusíme nikdy vymeniť 2 za 3.

Z týchto dvoch pozorovaní vieme, že ľubovoľné riešenie vieme prepísať na také, v ktorom vymieňame len 3 za 1, a buď 3 za 2 alebo 2 za 1 (ale nie oboje). No a ak máme dve výmeny 3-za-2 (2-za-1), ak máme napravo 1ku nazvyš (naľavo 3ku nazvyš), môžeme namiesto toho vymeniť 3-za-1. Ak nemáme napravo 1ku nazvyš (naľavo 3ku nazvyš), tak naše riešenie už spravilo všetky výmeny 3-za-1, teda už bolo pažravé.

Takto sme ukázali, že ľubovoľné riešenie ktoré zarovná trvanie úloh v oboch častiach zoznamu vieme prepísať na pažravé, teda pažravý prístup nikdy nespôsobí, že riešenie nenájdeme. Zároveň sme to spravili len tým, že sme niektoré výmeny pomazali, a tak vidíme že pažravé riešenie spraví menej výmen, ako iné s ktorým začneme. Hurá!

Ako naše pažravé riešenie nájdeme? Pri každom druhu výmen (3 za 1, 3 za 2, 2 za 1) si zrátame, koľko výmen môžeme urobiť (minimum výskytov v prvej a druhej časti zoznamu), a koľko ich chceme urobiť (rozdiel trvania oboch častí zoznamu, deleno koľko ho výmena zmení). Následne túto výmenu spravíme buď koľko krát môžeme, alebo koľko krát chceme, podľa toho ktoré je menšie.

Ak po vykonaní výmen všetkých troch typov obe časti nemajú rovnaké trvanie (buď lebo nevychádza parita, alebo pretože nevieme nasúkať dosť dlhých úloh do kratšej časti), odpovieme -1, inak povieme koľko výmen sme vykonali.

Odporúčame si toto zrátanie naprogramovať ako osobitnú funkciu, ktorú potom zavoláte s pripravenými počtami úloh rôznej dĺžky v oboch častiach zoznamu. Potom môžete vylepšovať to, ako tieto počty počas prepisovania dĺžok úloh získavate, pre rôzne vstupné sady. Dbajte na to, aby vaša funkcia netrvala dlhšie, ak sú počty úloh vyššie - mala by všetko zrátať v $O(1)$.

Listing programu (C++)

```cpp
// A[i] je pocet uloh dlzky i vlavo, B[i] vpravo
int zrataj_vymeny(vector<int> A, vector<int> B) {
    int rozdiel = A[1] - B[1] + 2 * (A[2] - B[2]) + 3 * (A[3] - B[3]);
    // nech presuvame z vacsieho A do mensieho B
    if (rozdiel < 0) {
        rozdiel = -rozdiel;
        swap(A, B);
```

```
    }
    // lubovolna vymena zmeni rozdiel o parnu hodnotu
    // ak je rozdiel neparny, neda sa
    if (rozdiel % 2) {
        return -1;
    }
    rozdiel /= 2;

    // zratame kolko 3-za-1 vymen chceme, vieme
    // spravime ich minimum z toho
    int chcem_3_za_1 = rozdiel / 2;
    int viem_3_za_1 = min(A[3], B[1]);
    int vymenim_3_za_1 = min(chcem_3_za_1, viem_3_za_1);
    rozdiel -= 2 * vymenim_3_za_1;
    A[3] -= vymenim_3_za_1;
    B[1] -= vymenim_3_za_1;

    // to iste pre 3-za-2 a 2-za-1
    // mozeme to zratat sucasne, kedze sme dokazali
    // ze nebudeme nikdy robit obe
    int viem_3_za_2 = min(A[3], B[2]);
    int viem_2_za_1 = min(A[2], B[1]);
    int viem_jednotkovych_vymen = max(viem_3_za_2, viem_2_za_1);
    int chcem_jednotkovych_vymen = rozdiel; // uplne nepotrebne, ale pre pochopenie
    int vymenim_jednotkove = min(chcem_jednotkovych_vymen, viem_jednotkovych_vymen);
    rozdiel -= vymenim_jednotkove;

    // ak ostal nejaky rozdiel, tak sa to neda
    if (rozdiel != 0)
        return -1;
    // inak vratime kolko vymen sme spravili
    return vymenim_3_za_1 + vymenim_jednotkove;
}
```

Hoďme na to for cyklus

Najjednoduchšie riešenie: budeme si udržovať pole s dĺžkami úloh, a aby sme zistili koľko akých úloh je v prvých x a vo zvyšku, jednoducho ho prejdeme a zrátame si to.

Udržujeme si teda pole veľkosti n. Každú zmenu dĺžku úlohy vieme priamo vykonať v našom poli v $O(1)$. Pre každú otázku musíme prejsť celé naše pole aby sme zrátali počty rôznych úloh v oboch častiach nášho zoznamu, čo nám trvá $O(n)$.

Keďže otázok aj aktualizácií je q, toto nám zaberie $O(nq)$ času. Naše pole úloh nám zaberie $O(n)$ pamäte.

Listing programu (C++)

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

int zrataj_vymeny(vector<int> A, vector<int> B) {
    // ... //
}

int main() {
    int t;
```

```
    cin >> t;
    while (t--) {
        int n, q;
        cin >> n >> q;
        vector<int> v(n);
        for (int i = 0; i < n; i++)
            cin >> v[i];
        for (int i = 0; i < q; i++) {
            int a, b, x;
            cin >> a >> b >> x;
            a--;
            v[a] = b;
            vector<int> A = {0, 0, 0, 0};
            vector<int> B = {0, 0, 0, 0};
            // prejdeme vsetky ulohy a zratame kolko akych je
            // vlavo a vpravo od x
            for (int j = 0; j < x; j++)
                A[v[j]]++;
            for (int j = x; j < n; j++)
                B[v[j]]++;

            cout << zrataj_vymeny(A, B) << "\n";
        }
    }
}
```

Keď si úlohy vieme dopredu rozdeliť

V prvých dvoch sadách máme síce príliš veľa úloh a otázok, aby sme si mohli dovoliť prepočítavať niečo na všetkých úlohách, máme však nejaké záruky navyše.

V druhej sade sa vždy pýtame otázku $x = \frac{n}{2}$. Dve časti zoznamu, ktoré budeme musieť vyvážiť, budú teda vždy rovnaké, a potrebujeme si len udržiavať počet úloh dĺžky 1, 2 a 3 v nich.

Nech A_1, A_2, A_3 je počet úloh dĺžky 1 až 3 v prvej polovici zoznamu, a B_1, B_2, B_3 v druhej polovici. Na začiatku si tieto hodnoty vieme prejdením celého poľa zrátať - pripočítavame do A pre prvú polovicu úloh, a do B pre druhú. Následne, keď sa zmení dĺžka úlohy a_i na b_i, pozrieme sa do poľa úloh, akú má úloha a_i momentálne dĺžku, odpočítame ju z A alebo B podľa toho do ktorej polovice patrí a_i, a následne ju zmeníme a pripočítame späť. Potom zavoláme našu pripravenú funkciu ktorá zistí či a na koľko výmen vieme A a B vyrovnať, a zmlsneme 2 body za program navyše.

Pamäťová zložitosť zostáva $O(n)$ keďže si pamätáme pole úloh, a konštatne veľa čísel navyše. Časová zložitosť je $O(n+q)$, keďže pre každú zmenu spravíme konštatne veľa operácií (odpočítanie - zmenenie - pripočítanie) a zavoláme našu pripravenú funkciu.

Listing programu (C++)

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

int zrataj_vymeny(vector<int> A, vector<int> B) {
    // ... //
}

int main() {
    int t;
    cin >> t;
```

```
    while (t--) {
        int n, q;
        cin >> n >> q;
        vector<int> v(n);
        vector<int> A = {0, 0, 0, 0};
        vector<int> B = {0, 0, 0, 0};

        // predratame si, kolko akych uloh je
        // v prvej a druhej polovici zoznamu
        for (int i = 0; i < n; i++) {
            cin >> v[i];
            if (2 * i < n)
                A[v[i]]++;
            else
                B[v[i]]++;
        }

        for (int i = 0; i < q; i++) {
            int a, b, x;
            cin >> a >> b >> x;
            a--;
            // preratame si hodnotu v tej polovici
            // do ktorej uloha patri
            if (a < x) {
                A[v[a]]--;
                A[b]++;
            } else {
                B[v[a]]--;
                B[b]++;
            }
            // prepiseme si ju
            v[a] = b;

            cout << zrataj_vymeny(A, B) << "\n";
        }
    }
}
```

Keď sa úlohy nemenia

V tretej sade máme inú garanciu: dĺžky úloh sa nemenia. Musíme však zodpovedať rôzne otázky.

Môžeme zvoliť jeden z dvoch prístupov - predpočítať si všetky otázky, alebo pripraviť sa odpovedať na otázky rýchlo.

Prvé riešenie bude veľmi podobné riešeniu druhej sady - majme polia A a B, v ktorých si budeme udržiavať počty úloh všetkých dĺžok, v A po úlohu x, a v B napravo. Na začiatku sú všetky úlohy v B (akoby $x = 0$). Teraz budeme pomyselné x posúvať doprava: postupne prejdeme úlohy do prvej po n-tú, odpočítame ju z B, pripočítame do A, našou funkciou zrátame výsledok, a zapíšeme si ho do poľa odpovedí. Nakonci máme odpovede pre všetky x od 1 po n, a vypisujeme ich ako si ich vstup pýta.

Druhé riešenie je použiť prefixové súčty (viď kuchárka[6]) - zrátame si ich osobitne pre všetky tri dĺžky úloh. Potom pomocou nich vieme v $O(1)$ zistiť koľko ktorých úloh je do x a po x, a opäť len funkciou zrátať výsledok.

Obe riešenia si nazačiatku niečo pre všetky úlohy predpočítajú, a pre každú úlohu spravia $O(1)$ práce. Následne vedia odpovedať na otázky v $O(1)$. Majú teda pamäťovú zložitosť $O(n)$ a časovú $O(n + q)$

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

Listing programu (C++)

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

int zrataj_vymeny(vector<int> A, vector<int> B) {
    // ... //
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    int t;
    cin >> t;
    while (t--) {
        int n, q;
        cin >> n >> q;
        vector<int> v(n);
        vector<vector<int>> prefixy = vector<vector<int>>(4, vector<int>(n + 1, 0));
        // predratame si prefixove sucty pre vsetky dlzky uloh
        for (int i = 0; i < n; i++) {
            cin >> v[i];
            for (int j = 0; j < 4; j++)
                prefixy[j][i + 1] = prefixy[j][i];
            prefixy[0][i + 1] += v[i];
            prefixy[v[i]][i + 1] += 1;
        }

        for (int i = 0; i < q; i++) {
            int a, b, x;
            cin >> a >> b >> x;
            // nemusime nic preratavat, kedze mame zaruku ze v[a-1] == b
            vector<int> A(4, 0);
            vector<int> B(4, 0);
            // zratame si z nich, kolko ktorych uloh je pred a po x
            for (int j = 1; j <= 3; j++) {
                A[j] = prefixy[j][x];
                B[j] = prefixy[j][n] - A[j];
            }

            cout << zrataj_vymeny(A, B) << "\n";
        }
    }
}
```

Keď to treba zvládnuť naraz...

Posledné spomenuté riešenie má do plného počtu bodov len jediný háčik - ak by sa zmenila dĺžka nejakej úlohy, aby bol náš prefixový súčet správny, museli by sme všetko napravo od nej prepočítať, a to trvá dlho.

Kiežby existovala dátová štruktúra, ktorá vie na nejakom intervale čísla posčítavať, a aj ich postupne po jednom meniť...

A takých dátových štruktúr existuje dokonca viacero!

Jeden dobrý kandidát je intervalový strom[7]. Môžeme si v jeho vrcholoch udržiavať počty všetkých dĺžok

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

úloh, alebo si postaviť viacero jednoduchých stromov, každý na jedno trvanie úloh. Postavíme ich zo vstupu, pri prepisovaní úloh voláme update, a pri odpovedaní na otázky zrátame úlohy do x a od x query operáciami, a odpoveď dorátame našou funkciou.

Pamäťová zložitosť intervalového stromu je $O(n)$. Jeho vytvorenie nás trvá $O(n)$, každá zmena nám zaberie $O(\log n)$ na spracovanie, a zrátať úlohy na dvoch intervaloch tiež $O(\log n)$. Dokopy je teda časová zložitosť $O(n + q \log n)$.

Pre záujemcov naučiť sa dačo nové, ďalším dobrým kandidátom na dátovú štruktúru je Binary Indexed Tree, tzv. Fínsky strom. Žiaľ, chýba k nemu článok v kuchárke, môžete si o ňom však prečítať na wikipédii, a pozrieť si naše vzorové riešenie, ktoré ho využíva.

Listing programu (C++)

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

int zrataj_vymeny(vector<int> A, vector<int> B) {
    // ... //
}

// Finsky strom
struct FT {
    vector<int> s;
    FT(int n) : s(n) {}
    void update(int pos, int dif) {
        for (; pos < s.size(); pos |= pos + 1)
            s[pos] += dif;
    }
    int query(int pos) {
        int res = 0;
        for (; pos > 0; pos &= pos - 1)
            res += s[pos - 1];
        return res;
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    int t;
    cin >> t;
    while (t--) {
        int n, q;
        cin >> n >> q;
        vector<int> v(n);
        // finsky strom pre vsetky dlzky uloh
        FT fts[] = {FT(0), FT(n), FT(n), FT(n)};
        for (int i = 0; i < n; i++) {
            cin >> v[i];
            fts[v[i]].update(i, 1);
        }

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

```
43
44              a--;
45              fts[v[a]].update(a, -1);
46              v[a] = b;
47              fts[b].update(a, 1);
48
49              vector<int> A(4, 0), B(4, 0);
50
51              for (int j = 1; j <= 3; j++) {
52                  A[j] = fts[j].query(x);
53                  B[j] = fts[j].query(n) - A[j];
54              }
55
56              cout << zrataj_vymeny(A, B) << "\n";
57          }
58      }
59  }
```