Korešpondenčný seminár z programovania
XXXIX. ročník, 2021/22
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

# Vzorové riešenia 2. kola zimnej časti

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

## 1. Tajná aplikácia

### Myšlienka riešenia

Na to, aby sme vedeli vytvoriť cestu, ktorá bude viesť čo najďalej od tej pôvodnej, musíme najprv zistiť, kde končí cesta na vstupe. Prečítame vstup a pre každý znak posunieme Paulínku na štvorčekovej sieti. Paulínka začína v bode $(0, 0)$. Ak na vstupe máme `>`, pripočítame k jej x-ovej súradnici 1, ak `<`, jednotku odčítame. Podobne to bude fungovať aj s `^` a `v`. Po prečítaní celého vstupu vieme súradnice bodu, v ktorom Paulínka mala skončiť.

Teraz sa na vstup pozrieme ešte raz a budeme vymieňať šípky tak, že všetky šípky, ktoré sú orientované smerkom **k cieľu** vymeníme za opačné. Teda, ak na vstupe máme šípku `<` a cieľ je naľavo od $(0, 0)$, vymeníme ju za `>`. Keby bol cieľ napravo, šípku necháme tak, ako bola.

### Časová a pamäťová zložitosť

Keďže sa iba niekoľkokrát pozrieme na celý vstup, časová zložitosť bude lineárne závislá od dĺžky vstupu – $O(n)$.

Pamäťová zložitosť bude tiež $O(n)$, keďže si musíme pamätať celý vstup a konštantný počet premenných – súradnice pôvodného cieľa.

### Listing programu (Python)

```
arrows = input()
position_x = 0
position_y = 0

for arrow in arrows:
    if arrow == "<":
        position_x -= 1
    if arrow == ">":
        position_x += 1
    if arrow == "^":
        position_y += 1
    if arrow == "v":
        position_y -= 1

for arrow in arrows:
    if arrow == "v" or arrow == "^":
        if position_y > 0:
            print("v", end="")
        else:
            print("^", end="")
    else:
        if position_x > 0:
            print("<", end="")
        else:
            print(">", end="")

print()  # newline na konci
```

## 2. Absurdne drahá pizza

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

### Priamočiare riešenie

Najjednoduchšie riešenie je krok po kroku odsimulovať, ako sa mení počet kváskov. Vieme to urobiť tak, že budeme mať pole, kde na indexe $i$ budeme mať čas kvásku $i$. Potom stačí $t$ krát prejsť celé pole, a každému kvásku znížiť čas o 1. Ak je nový čas kvásku rovný 0, tak na koniec poľa pridáme $k$ nových kváskov s časom 8, a tomuto kvásku nastavíme čas opäť na 6. Je treba si dať pozor, aby sme v tomto prechode poľom časy novopridaných kváskov už nezmenšovali, alebo aby sme pridávali nové kvásky s vekom 9 a teda po prejdení celého poľa mali tieto kvásky správny čas.

Výsledok, počet kváskov po $t$ dňoch je rovný dĺžke poľa.

Čo sa týka časovej zložitosti, tak náš program $t$ krát prejde celé pole. Pole bude mať na konci dĺžku najviac $nk^{t/6}$. Prečo? Ak si predstavíme, že by mali všetky kvásky rovnaký čas, a nové kvásky by vznikali s časom 6, tak každých 6 dní by z sa počet kváskov zväčšil $k$ krát. To znamená, že po 6 dňoch by sa počet kváskov zväčšil na $k$ násobok, po 12 dňoch na $k \cdot k = k^2$ násobok, a po $t$ dňoch na $k^{t/6}$ násobok. (Všimnite si, že sme urobili horný odhad, keďže sme predpokladali, že nové kvásky vznikajú s menším časom, a teda v skutočnosti z nich nové kvásky vzniknú skôr.) Pole teda prejdeme $t$ krát, a vždy bude mať dĺžku najviac $nk^{t/6}$.

Časová zložitosť je teda rovná tomu, že $t$ krát prejdeme pole s maximálnou dĺžkou $nk^{t/6}$, takže $O(t \cdot nk^{t/6})$. Pamäťová zložitosť je rovná výslednej dĺžke poľa, teda $O(nk^{t/6})$.

Takéto riešenie stačí na prvé 2 sady.

### Listing programu (C++)

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

using namespace std;

int main(){
        long long n,k,t;
        int max_cas_kvasku = 7;
        int cas_noveho_kvasku = 8;
        cin>>n>>k>>t;

        vector<long long> kvasky;

        for(int i=0;i<n;i++){
                int tmp;
                cin>>tmp;
                kvasky.push_back(tmp);
        }

        for(int c=0;c<t;c++){
        for(int i=kvasky.size()-1;i>=0;i--){
            if(kvasky[i]==0){
                for(int j=0;j<k;j++)
                    kvasky.push_back(cas_noveho_kvasku);
                kvasky[i]=max_cas_kvasku;
            }
            kvasky[i]--;
        }
    }

        cout<<kvasky.size()<<endl;

        return 0;
}
```

### Listing programu (Python)

```
#!/usr/bin/env python3

max_cas_kvasku = 7
cas_noveho_kvasku = 8

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

for i in range(t):
    pocet_novych_kvaskov = 0

    for j in range(len(kvasky)):
        if kvasky[j] == 0:
            kvasky[j] = max_cas_kvasku - 1
            pocet_novych_kvaskov += 1
        else:
            kvasky[j] -= 1

    kvasky += [cas_noveho_kvasku for i in range(pocet_novych_kvaskov*k)]


print(len(kvasky))
```

### Vzorové riešenie

Môžeme si uvedomiť, že v predchádzajúcom riešení robíme pre všetky kvásky v zásade tú istú vec: zmenšíme ich čas, a poprípade pridáme nejaké nové. Takéto niečo ale nemusíme robiť pre každý kvások zvlášť, môžeme to robiť pre všetky kvásky s rovnakým časom spolu. Ak máme napríklad tri kvásky, a z každého z nich ide akurát vzniknúť 6 nových, tak dokopy vieme povedať, že z nich vznikne 18 nových kváskov s vekom 8.

Skúsme si teda namiesto poľa, v ktorom na indexe $i$ máme čas kvásku $i$ pamätať kvásky inak. Pamätajme si ich tak, že v poli na indexe $i$ budeme mať počet kváskov s časom $i$. Takto dosiahneme, že dĺžka poľa sa nebude zväčšovať, ale bude mať vždy dĺžku 9 (keďže najväčší možný čas kvásku je 8).

Ako teda presne budeme simulovať rast kváskov?

Opäť $t$ krát prejdeme celé pole, a kvásky na indexoch 1 až 8 posunieme na o 1 menší index (zmenšíme im čas o 1), kvásky na indexe 0 pripočítame ku kváskom na indexe 6 a vytvoríme príslušný počet nových kváskov s časom 8. Treba si dať pozor na to v akom poradí to robíme, aby sme kvásky z indexu 0, ktoré presunieme na index 6 už viac neposúvali.

Výsledok, teda počet kváskov zistíme tak, že spočítame počty kváskov s jednotlivými časmi.

Náš program $t$ krát prejde celé pole, a každý raz urobí niekoľko málo operácií (pripočítanie a vynásobenie zopár čísel). Pole má celý čas dĺžku 9, a prejdeme ho $t$ krát, takže časová zložitosť je $O(9t) = O(t)$.

Čo sa týka pamäťovej zložitosti, tak náš program si okrem niekoľko málo premenných pamätá len pole dĺžky 9, takže pamäťová zložitosť je $O(9) = O(1)$.

### Listing programu (C++)

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

using namespace std;

int main(){
        long long n,k,t;
        int max_cas_kvasku = 7;
        int cas_noveho_kvasku = 8;
        cin>>n>>k>>t;

        vector<long long> pocty_kvaskov(cas_noveho_kvasku+1, 0);
```

```
        for(int i=0;i<n;i++){
                int tmp;
                cin>>tmp;
                pocty_kvaskov[tmp]++;
        }

        for(int i=0;i<t;i++){
                long long nove_kvasky = pocty_kvaskov[0];
                for(int j=0;j<cas_noveho_kvasku;j++){
                        pocty_kvaskov[j] = pocty_kvaskov[j+1];
                }
                pocty_kvaskov[cas_noveho_kvasku] = nove_kvasky*k;
                pocty_kvaskov[max_cas_kvasku-1] += nove_kvasky;
        }

        long long pocet_kvaskov = 0;
        for(int i=0;i<=cas_noveho_kvasku;i++)
                pocet_kvaskov += pocty_kvaskov[i];
        cout<<pocet_kvaskov<<endl;

        return 0;
}
```

**Listing programu (Python)**

```
#!/usr/bin/env python3

max_cas_kvasku = 7
cas_noveho_kvasku = 8

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

pocty_kvaskov = {i:0 for i in range(0, cas_noveho_kvasku+1)}
for i in kvasky:
    pocty_kvaskov[i] += 1


for i in range(t):
    nove_kvasky = pocty_kvaskov[0]

    for j in range(1, max_cas_kvasku):
        pocty_kvaskov[j-1] = pocty_kvaskov[j]

    pocty_kvaskov[max_cas_kvasku-1] = nove_kvasky

    pocty_kvaskov[max_cas_kvasku-1] += pocty_kvaskov[max_cas_kvasku]
    pocty_kvaskov[max_cas_kvasku] = pocty_kvaskov[cas_noveho_kvasku]
    pocty_kvaskov[cas_noveho_kvasku] = nove_kvasky * k


print(sum(pocty_kvaskov.values()))
```

Marianka

## 3. Kapustnica Trojstenu

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

Prvá vec, ktorú si treba uvedomiť pri riešení tejto úlohy je, že ak má čokoláda viac alebo menej horkých tabličiek ako je počet Trojstenákov, tak odpoveď je automaticky `nie`, pretože nevieme splniť všetky podmienky.

### Pomalé riešenie

Asi najviac priamočiare riešenie je vyskúšať postupne všetky možné rozdelenia čokolády a skontrolovať, či v každom súvislom kúsku je práve jedna horká tablička. Takéto niečo vieme urobiť jednoduchou rekurziou. Začneme pri tom, že naša rekurzia odkrojí z čokolády súvislý kus, ktorý má $l$ tabličiek. Akonáhle sme čokoládu rozdelili na $n$ kusov – aby každý Trojstenák dostal jeden kus – skontrolujeme, či sú všetky podmienky splnené a či nám žiadna čokoláda nezvýši. Ak je všetko v poriadku, môžeme vypísať `ano`. Ak niektorá z podmienok splnená nie je, pokračujeme ďalej v rekurzii. Rekurzia bude rozdelovať čokoládu na kúsky s $l$ až $r$ tabličkami. Akonáhle sme prešli všetky možnosti ako sa dá čokoláda rozdeliť a ani raz sme nenašli riešenie, ktoré spĺňa všetky podmienky, vypíšeme `nie`.

### Vzorové riešenie

Na vyriešenie tejto úlohy však nepotrebujeme zbytočne skúšať všetky existujúce možnosti. Pre každú horkú tabličku nás zaujíma, na akom indexe sa nachádza, aby sme vedeli odvodiť intervaly, v ktorých môžeme rozdeliť čokoládu, aby sme určite mali v každom kúsku práve jednu horkú tabličku. Preto si prejdeme celú čokoládu a zapíšeme si indexy horkých tabličiek do poľa $h$. Na čokoládu pozeráme zľava doprava a počiatočný interval je $(0, 0)$, čo je na začiatku čokolády. Čísla v intervaloch ukazujú na miesta medzi dvoma tabličkami. Ak máme napríklad interval $(2, 5)$, tak čokoládu môžeme rozdeliť pred tabličkou na druhom indexe až za tabličkou na štvrtom indexe.

Vytvoríme si funkciu, ktorá pre každú horkú tabličku vypočíta interval, v ktorom sa môže rozdeliť čokoláda za danou tabličkou. Začiatok intervalu musí byť vždy aspoň o $l$ väčší ako predchádzajúci začiatok intervalu, pretože najmenej $l$ tabličiek môžeme oddeliť od zvyšku čokolády. Avšak, môže sa nám stať, že aj keby oddelíme kus čokolády s $l$ tabličkami, tak medzi nimi nebude horká tablička. Takýmto možnostiam sa chceme vyhnúť, pretože hneď vieme, že nespĺňajú všetky podmienky a nemá zmysel s nimi ďalej pokračovať. V takomto prípade musí byť začiatok intervalu za horkou tabličkou. Takže začiatok intervalu sa dá napísať ako $\max(\text{zaciatok\_predchadzajuceho\_intervalu} + l, h[i] + 1)$.

Koniec intervalu musí byť vždy maximálne o $r$ väčší ako koniec predchádzajúceho intervalu, pretože môžeme od čokolády oddeliť kus s maximálne $r$ tabličkami. Môže však nastať situácia, kedy v týchto $r$ tabličkách už bude aj druhá horká tablička, čo nechceme, pretože by sme nesplnili podmienky. Takže chceme čokoládu rozdeliť najviac pred nasledujúcou horkou tabličkou. Koniec intervalu preto vieme zapísať ako $\min(\text{koniec\_predchadzajuceho\_intervalu} + r, h[i + 1])$.

Aby kód fungoval poriadne, musíme si na koniec poľa $h$ pridať aj index $k$, čo je koniec čokolády, pretože to je najďalej, kde vieme rozdeliť čokoládu. Chceme sa vyhnúť tomu, že by sme chceli rozdeliť čokoládu za jej koncom, pretože toľko čokolády na rozdávanie nemáme.

Vždy, keď si vypočítame interval pre danú horkú tabličku, skontrolujeme či začiatok je menší alebo rovný koncu intervalu. Ak by bol začiatok väčší, tak môžeme rovno vypísať `nie`, pretože neexistuje miesto, kde by sme dokázali čokoládu rozdeliť, aby sme splnili všetky podmienky.

Úplne na koniec nás zaujíma posledný interval. Ak koniec tohoto intervalu je menší ako celková dĺžka čokolády $k$, tak Naďka nevie rozdať celú čokoládu a vypíšeme `nie`. Ak však prešlo hladko a nestretli sme sa so žiadnym problémom, môžeme vypísať `ano`, pretože sme splnili všetky podmienky a každý Trojstenák dostal svoj kus čokolády, bez toho, aby Naďke nejaká zvýšila.

### Časová a pamäťová zložitosť

Na začiatku sme prešli celú čokoládu, aby sme zistili, na akých indexoch sú horké tabličky – $O(k)$. Potom sme $n$-krát vypočítali interval rezu – $O(n)$. Dokopy je teda časová zložitosť $O(k + n)$.

Pamätáme si čokoládu a nejaké konštantne veľké premenné k tomu, takže pamäťová zložitosť je $O(k)$.

### Listing programu (Python)

```
import sys

n, k, l, r = map(int, input().split())
cokolada = input()
```

```
horke = list()
pocet_horkych = 0
interval = (0, 0)

def rez(interval, i):
    return (max(horke[i] + 1, interval[0] + l), min(horke[i + 1], interval[1] +
        ↪ r))

for i in range(k):
    if cokolada[i] == '1':
        pocet_horkych += 1
        horke.append(i)
# na koniec ľpoa nesmieme ťzabudnú ťprida koniec čokolády
horke.append(k)

if pocet_horkych != n:
    print('nie')
    sys.exit()

for i in range(n):
    interval = rez(interval, i)
    if interval[0] > interval[1]:
        print('nie')
        sys.exit()

if interval[1] < k:
    print('nie')
else:
    print('ano')
```

## 4. Ťahanie Kalerábu

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

### Pomalý bruteforce

Tento príklad vieme spraviť celkom jednoducho v exponenciálnom čase. Pôjdeme od začiatku cez všetky čísla, a po každom čísle sa rozvetvíme na dve možnosti: s použitím Adama a bez použitia. Ak Adama použijeme, dáme mu silu o jedna väčšiu, ako je naše aktuálne číslo, a v tejto vetve ho už použiť nemôžeme. Takto vieme prejsť všetky možnosti použitia Adama a stačí si nám zapamätať najdlhšiu takúto vetvu. Takéto riešenie bude mať časovú zložitosť $O(2^N)$, pretože na každom čísle musíme prejsť cez dve vetvy. Dajú sa tu rôzne veci zoptimalizovať, avšak zložitosť bude stále $O(2^N)$, a takéto riešenie bude teda veľmi pomalé.

### Lepší bruteforce

Čo ak by sme si pre každého človeka v rade pamätali dve čísla? Prvé číslo bude vyjadrovať, koľko ľudí je za ním takých, že spolu tvoria reťaz, ktorá spĺňa naše pravidlá. Toto číslo si označíme písmenom $A$. Druhé číslo bude vyjadrovať to isté, až na to, že v nejakom momente je v tomto rade aj Adam. Toto číslo označíme ako $B$. Budeme takto prechádzať cez všetkých ľudí. Pre každého nového človeka nájdeme posledného človeka v rade, ktorý je o jedna slabší a človeka, ktorý je o dva slabší. Novému človeku nastavíme číslo $A$ na silu človeka o 1 slabšieho plus jedna, pretože sa pridal do radu. Číslo $B$ je troška komplikovanejšie.

Číslo $B$ má dve možnosti: buď to môže byť číslo $B$ človeka o jedna slabšieho plus jedna alebo to môže byť číslo $A$ človeka o dva slabšieho plus dva. Druhá možnosť vyjadruje možnosť, kde sme Adama doteraz nepoužili a až teraz sme ho vsunuli do radu. Takto prejdeme cez všetkých ľudí a riešenie je najväčšie číslo s použitím Adama, pretože to vyjadruje Adam vždy predĺži rad o jedna. Samozrejme sa môže stať, že v rade nie je človek o jedna alebo o dva slabší. V takom prípade je tento človek začiatkom reťaze a začne teda od nuly.

Toto riešenie má kvadratickú časovú zložitosť v závislosti od počtu ľudí, pretože pri pridaní každého človeka musíme v najhoršom prípade prejsť všetkých predchádzajúcich ľudí, teda je časová zložitosť $O(N \cdot N) = O(N^2)$

teda kvadratická. Pamäťová zložitosť je lineárna od počtu ľudí ($O(N)$), keďže si pre každého človeka pamätáme dve čísla.

### Optimálne riešenie

Od lepšieho bruteforcu k optimálnemu riešeniu je už len jeden krok. Namiesto toho, aby sme si pamätali tieto údaje pre každého človeka, budeme si tieto údaje pamätať pre každú možnú silu človeka. Teraz namiesto toho, aby sme prehľadávali posledného človeka s nejakou silou, stačí nám použiť hľadanú silu ako index poľa, ktorú vieme nájsť v konštantnom čase. Polia vieme na začiatku naplniť nasledovne: pole reprezentujúce rad bez Adama naplníme nulami, pretože Adama sme nepoužili a človek, ktorý hľadá takúto silu bude prvý v tomto rade. Pole reprezentujúce čísla $B$ naplníme jednotkami, pretože v tomto prípade použijeme Adama hneď na začiatku.

### Prečo nám netreba max()

Môže sa nám stať, že v rade stoja dvaja alebo viacerí ľudia s rovnakou silou. Keď vyrátame prvého takého človeka, jednoducho uložíme čísla do poľa. Ak nájdeme ďalšieho človeka s takouto silou, vieme, že teraz pracujeme s číslami, ktoré sú rovnaké alebo lepšie, ako keď sme túto silu rátali naposledy, pretože teraz pracujeme s nadmnožinou čísel s akou sme pracovali minule. Táto optimalizácia nebola potrebná na dosiahnutie plného počtu bodov, avšak je to zaujímavé pozorovanie, ktoré môže byť pre vás v budúcnosti užitočné.

### Listing programu (Python)

```
def sol(nums, biggest):
    values_without_adam = [0 for n in range(biggest+1)]
    values_with_adam = [1 for n in range(biggest+1)]

    for ind, n in enumerate(nums):
        one_lower = n-1
        two_lower = n-2

        without_one = values_without_adam[one_lower]
        without_two = values_without_adam[two_lower]
        with_one = values_with_adam[one_lower]

        values_without_adam[n] = without_one + 1
        values_with_adam[n] = max(with_one + 1, without_two + 2)

    return max(values_with_adam + values_without_adam)


n_people, biggest = list(map(int, input().split()))
answer = sol(list(map(int, input().split())), biggest)
print(answer)
```

### Listing programu (C++)

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

using namespace std;

int main() {
    long long n, m;
    cin >> n >> m;
    m++;

    int without_adam[m];
    int with_adam[m];
```

```
    for (int i = 0; i < m; i++) {
        without_adam[i] = 0;
        with_adam[i] = 1;
    }

    int sol = 1;
    long long current;

    for (int i = 0; i < n; i++) {
        cin >> current;

        without_adam[current] = without_adam[current-1]+1;
        with_adam[current+1] = without_adam[current-1] + 2;
        with_adam[current] = with_adam[current-1] + 1;

        sol = max({sol, without_adam[current], with_adam[current+1], with_adam[
            ↪ current]});
    }

    cout << sol << endl;
    return 0;
}
```

Aďo

## 5. Úžasná lyžovačka

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

Už na prvý pohľad je asi vcelku zrejmé, že lyžiarske stredisko si vieme predstaviť ako neorientovaný ohodnotený graf, v ktorom stanice lanoviek budú vrcholmi a svahy hranami grafu. Ceny hrán budú strmosti svahov, ktoré si vypočítame už pri načítavaní vstupu.

### Priamočiare riešenie

Môžeme si všimnúť, že na nájdenú cestu nemáme žiadne ďalšie nároky, okrem toho že existuje. Najjednoduchšie riešenie teda spočíva v tom, že z grafu odstránime všetky hrany a budeme ich postupne pridávať v poradí od najmenej strmej. Po každom pridaní hrany skontrolujeme, či neexistuje cesta medzi $s$ a $f$. V momente, ako takúto cestu nájdeme, môžeme skončiť, keďže všetky ďalšie hrany, ktoré by sme pridali by mali vyššiu strmosť. Naopak, lepšie riešenie tiež nemôže existovať, keďže by sme ho už našli skôr. Zároveň už poznáme aj najstrmšiu hranu v nájdenej ceste – je ňou posledná pridaná hrana.

V našej implementácii nám bude teda stačiť, keď si po načítaní uriedime hrany podľa strmosti, v cykle ich budeme pridávať do grafu (ideálne reprezentovaného ako zoznam susedov) a existenciu cesty overíme napríklad pomocou BFS (prehľadávania do šírky). Časová zložitosť takéhoto riešenia je $O(e \cdot (v+e))$. V najhoršom prípade totiž postupne pridáme do grafu všetky hrany, čo znamená, že BFS budeme musieť spustiť až $e$-krát. Pamäťová zložitosť je $O(v+e)$.

Za implementáciu s maticou susedov ste mohli dostať až 4 body, za šikovnú implementáciu so zoznamom susedov dokonca až 6 bodov.

### Listing programu (Python)

```
from collections import deque

v, e, start, end = (int(x) for x in input().split())
heights = [int(input()) for _ in range(v)]

edges = []
for _ in range(e):
    a, b, d = (int(x) for x in input().split())
```

```
    slope = abs(heights[a] - heights[b]) / d
    edges.append((slope, a, b))
edges.sort()

neighbors = {x: {} for x in range(v)}
for slope, a, b in edges:
    neighbors[a][b] = neighbors[b][a] = (slope, d)

    # BFS
    explored = [False] * v
    d = deque([start])
    explored[start] = True

    while len(d):
        next = d.popleft()
        if next == end:
            break

        for neighbor, edge in neighbors[next].items():
            if not explored[neighbor]:
                explored[neighbor] = True
                d.append(neighbor)

    if next == end:
        print(f'{slope:0.2f}')
        break
```

**Listing programu (C++)**

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <set>
#include <deque>
#include <iomanip>
#include <algorithm>
using namespace std;

typedef tuple<double, int, int> Edge;

int main() {
    int v, e, s, f;
    cin >> v >> e >> s >> f;

    vector<int> heights(v);
    for (int i = 0; i < v; i++) {
        cin >> heights[i];
    }

    vector<Edge> edges(e);
    for (int i = 0; i < e; i++) {
        int a, b, d;
        cin >> a >> b >> d;
        double slope = abs((heights[a] - heights[b])) / double(d);
        edges[i] = make_tuple(slope, a, b);
    }
    sort(edges.begin(), edges.end());
```

```
    unordered_map<int, set<int>> neighbors;
    for (const auto &edge : edges) {
        neighbors[get<1>(edge)].insert(get<2>(edge));
        neighbors[get<2>(edge)].insert(get<1>(edge));

        // BFS
        vector<bool> explored(v, false);
        deque<int> d;
        d.push_back(s);
        explored[s] = true;

        while (!d.empty()) {
            int next = d.front();
            d.pop_front();

            if (next == f) {
                cout << fixed << showpoint << setprecision(2) << get<0>(edge) <<
                    ↪ endl;
                return 0;
            }

            for (const auto &neighbor : neighbors[next]) {
                if (!explored[neighbor]) {
                    explored[neighbor] = true;
                    d.push_back(neighbor);
                }
            }
        }
    }

    return 0;
}
```

### Ako to zrýchlime?

Z predchádzajúceho riešenia už asi tušíme, že hlavný problém je v počte spustení BFS. Naozaj musíme pridávať hrany po jednej a zakaždým skúšať, či existuje riešenie? Samozrejme, že nie!

To, čo sme robili v predchádzajúcom riešení si môžme predstaviť aj trochu inak: V podstate sme hľadali posledný prvok najmenšieho prefixu **utriedeného** poľa hrán $[e_0, e_1, \ldots, e_{max}]$ takého, že s pomocou týchto hrán je možné vytvoriť cestu z $s$ do $f$. No, a keďže je pole už utriedené, namiesto skúšania všetkých hrán si tú správnu hranu môžeme nájsť binárnym vyhľadávaním.

Samotné binárne vyhľadávanie bude prebiehať takmer rovnako ako na obyčajnom poli: Prehľadávanú oblasť rozdelíme v polovici a vyberieme prostrednú hranu $e_{mid}$. Vyskúšame, či podgraf s hranami $e_0$ až $e_{mid}$ obsahuje cestu z $s$ do $f$. Ak áno, budeme znovu binárne prehľadávať ľavú polovicu (keďže chceme nájsť najmenej strmú hranu, ktorú potrebujeme).

Vďaka binárnemu vyhľadávaniu potrebujeme spustiť BFS už len $\log e$-krát, čiže výsledná časová zložitosť bude $O((v+e) \cdot \log e)$. Pamäťová zložitosť sa oproti predchádzajúcemu riešeniu nezmenila.

### Listing programu (Python)

```
from collections import deque

v, e, s, f = (int(x) for x in input().split())
heights = [int(input()) for _ in range(v)]

edges = []
for _ in range(e):
    a, b, d = (int(x) for x in input().split())
```

```
    slope = abs(heights[a] - heights[b]) / d
    edges.append((slope, a, b))
edges.sort()

def bfs(edges, v, s, f):
    neighbors = {x: {} for x in range(v)}
    for slope, a, b in edges:
        neighbors[a][b] = neighbors[b][a] = slope

    # BFS
    explored = [False] * v
    d = deque([s])
    explored[s] = True

    while len(d):
        next = d.popleft()
        if next == f:
            return True

        for neighbor, _ in neighbors[next].items():
            if not explored[neighbor]:
                explored[neighbor] = True
                d.append(neighbor)

    return False


l, h = 0, e
best = -1
while h - l > 1:
    m = l + (h-l) // 2

    if bfs(edges[:m+1], v, s, f):
        h = m
        best = edges[m][0]
    else:
        l = m

print(f'{best:0.2f}')
```

**Listing programu (C++)**

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
#include <deque>
#include <map>
#include <set>
using namespace std;

struct Edge {
    int a, b, d;
    int o;
    double slope;
};

struct Graph {
```

```
    int v, e, s, f;
    vector<int> heights;
    vector<Edge> edges;
};

bool cmp(const Edge &a, const Edge &b) {
    return a.slope < b.slope;
}

bool bfs(Graph &g, int maxEdge) {
    map<int, set<int>> matrix;
    for (int i = 0; i < maxEdge; i++) {
        auto &e = g.edges[i];
        matrix[e.a].insert(e.b);
        matrix[e.b].insert(e.a);
    }

    vector<bool> visited(g.v, false);
    deque<int> d;

    visited[g.s] = true;
    d.push_back(g.s);

    while (!d.empty()) {
        auto next = d.front();
        d.pop_front();

        if (next == g.f) {
            return true;
        }

        for (auto &&neighbor : matrix[next]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                d.push_back(neighbor);
            }
        }

    }

    return false;
}

int main() {
    Graph g;
    cin >> g.v >> g.e >> g.s >> g.f;

    g.heights.resize(g.v);
    for (int i = 0; i < g.v; i++) {
        cin >> g.heights[i];
    }

    for (int i = 0; i < g.e; i++) {
        Edge e;
        cin >> e.a >> e.b >> e.d;
        e.slope = abs(g.heights[e.a] - g.heights[e.b]) / double(e.d);
        g.edges.push_back(e);
```

```
    }

    sort(g.edges.begin(), g.edges.end(), cmp);

    int l = 0, h = g.e;
    double bestslope = -1;
    while (h - l > 1) {
        int m = l + (h-l) / 2;

        if (bfs(g, m+1)) {
            h = m;
            bestslope = g.edges[m].slope;
        } else {
            l = m;
        }
    }

    cout << fixed << showpoint << setprecision(2) << bestslope << endl;

    return 0;
}
```

### Riešenie s kostrami

Táto úloha sa dala riešiť aj trochu inak: Ak si na začiatku nájdeme minimálnu kostru grafu reprezentujúceho lyžiarske stredisko, dostaneme vlastne graf, v ktorom už nie sú žiadne nadbytočné drahšie hrany. Medzi každou dvojicou vrcholov teda bude existovať už len jedna cesta, ktorej maximálna strmosť bude najmenšia možná. Na to, aby sme túto cestu našli nám opäť bude stačiť BFS alebo podobný algoritmus.

Navyše, ak si mierne upravíme Kruskalov algoritmus, nebudeme dokonca potrebovať ani BFS. Rovnako ako v Kruskalovom algoritme budeme postupne pridávať najlacnejšie hrany, pričom každá nová hrana nám spojí dva komponenty grafu. Ak sa nám teda niekedy vyskytnú vrcholy $s$ a $f$ v rovnakom komponente, vieme, že existuje medzi nimi cesta. Stačí nám teda po každej pridanej hrane skontrolovať v akých komponentoch sú $s$ a $f$. A keďže nehľadáme minimálnu kostru, cykly v grafe nám nevadia a nemusíme dokonca ani vynechávať hrany.

Takéto riešenie má časovú zložitosť iba $O(e \log e)$. Toto riešenie by sa dalo ešte zrýchliť efektívnejšou implementáciou Union-Find až na $O(e \log^* e)$[1]. Pamäťová zložitosť zostáva stále rovnaká, čiže $O(v + e)$.

### Listing programu (Python)

```
v, e, start, end = (int(x) for x in input().split())
heights = [int(input()) for _ in range(v)]
parents = list(range(v))

edges = []
for _ in range(e):
    a, b, d = (int(x) for x in input().split())
    slope = abs(heights[a] - heights[b]) / d
    edges.append((slope, a, b))
edges.sort()

def find(i):
    if parents[i] == i:
        return i
    else:
        parents[i] = find(parents[i])
        return parents[i]
```

[1] $\log^*$ je iterovaný logaritmus

```
def union(a, b):
    a, b = find(a), find(b)
    if a == b:
        return
    parents[a] = b

for slope, a, b in edges:
    union(a, b)
    if find(start) == find(end):
        print(f'{slope:0.2f}')
        break
```

**Listing programu (C++)**

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

typedef tuple<double, int, int> Edge;
vector<int> parents;

int find(int i) {
    return parents[i] == i ? i : parents[i] = find(parents[i]);
}

void uni(int a, int b) {
    a = find(a); b = find(b);
    if (a == b) return;
    parents[a] = b;
}

int main() {
    int v, e, s, f;
    cin >> v >> e >> s >> f;

    vector<int> heights(v);
    parents.resize(v);
    for (int i = 0; i < v; i++) {
        cin >> heights[i];
        parents[i] = i;
    }

    vector<Edge> edges(e);
    for (int i = 0; i < e; i++) {
        int a, b, d;
        cin >> a >> b >> d;
        double slope = abs((heights[a] - heights[b])) / double(d);
        edges[i] = make_tuple(slope, a, b);
    }
    sort(edges.begin(), edges.end());

    for (const auto &edge : edges) {
        uni(get<1>(edge), get<2>(edge));
        if (find(s) == find(f)) {
            cout << fixed << showpoint << setprecision(2) << get<0>(edge) <<
                ↪ endl;
```

```
            break;
        }
    }

    return 0;
}
```

Jožko F

## 6. Strieborné bubáky

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

### Dynamika

Prvá vec, ktorá pravdepodobne skúsenému riešiteľovi napadne, keď sa pozrie na túto úlohu je: "dynamické programovanie". Otázkou len je, ako si dobre zadefinovať stavy, aby sme si z menších stavov vedeli jednoducho vypočítať väčšie. Jednoduchý plán by bol definovať $dp[t]$ = najväčší počet bubákov, ktorý viem mať v deň $t$ (t.j. po $t$ použitiach nejakého krompáča, hneď po tom ako sme sa rozhodovali, či kupujeme krompáč v deň $t$). Avšak, $dp[t]$ nevieme priamočiaro vypočítať iba z predošlých stavov. Pri úvahách záleží na tom, ktorý krompáč použijeme zo dňa $t-1$ na deň $t$. Preto sa núka zadefinovať stavy nasledovne:
$dp[t][j]$ = náš najväčší možný počet bubákov v deň $t$, ak aktuálne máme krompáč $j$ (ten, ktorý bolo možné kúpiť v deň $j$)

Môžeme to vnímať tak, že na začiatku máme krompáč 0, ktorý ťaží 0 bubákov za deň. Teraz vieme medzi stavmi jednoducho prechádzať:

- pre $j < t$: $dp[t][j] = dp[t-1][j] + b_j$,
- pre $j = t$: $dp[t][j] = -c_t + \max\{dp[t][0], \ldots, dp[t][t-1]\}$,
- pre $j > t$: $dp[t][j] = -\infty$.

Takže môžeme počítať odpoveď pre jednotlivé stavy podľa $t$ vzostupne a pre fixné $t$ vzostupne podľa $j$. Nakoniec finálnu odpoveď nájdeme ako $\max\{dp[N+1][0], \ldots dp[N+1][N]\}$. Časová zložitosť tohoto riešenia je $O(N^2)$.

### Už to len zrýchliť

Pokúsime sa vylepšiť naše doterajšie riešenie. Chceli by sme dátovú štruktúru. Tá by si udržovala aktuálny deň $t$, pre tento aktuálny deň naše možnosti zodpovedajúce dvojiciam $(j, dp[t][j])$, kde $j \leq t$ reprezentuje náš aktuálny krompáč (posledný, ktorý sme sa rozhodli kúpiť). Navyše chceme, aby sa táto štruktúra dala ľahko upraviť pri prechode do dňa $t+1$.

Ak by sme si pre každý krompáč $j$ skutočne stále pamätali $dp[t][j]$, tak by bol update z $t$ na $t+1$ časovo náročný (priamočiaro to nejde lepšie ako v $O(N)$). Skúsme sa toho teda vzdať. Napriek tomu však budeme chcieť byť schopný (pri troche úsilia) v aktuálnom dni vypočítať hodnotu $dp[t][j]$ pre ľubovoľné $j$.

Pre konkrétne $j$ sa $dp$ v čase vyvíja priamočiaro. $dp[j][j]$ treba vypočítať náročne, ale potom je to už jednoduché: $dp[t][j] = (t-j)b_j + dp[j][j]$. To nás navádza na myšlienku, že v našej štruktúre bude stačiť mať pre fixné $j$ iba hodnotu $dp[j][j]$.

Update z času $t$ na $t+1$ sa tak zvrháva na to, ako pridať $t+1$. Ako spočítať $dp[t+1][t+1]$? Toto je ťažká vec. Na to, aby sme to vedeli rátať by bolo dobré mať krompáče v našej štruktúre utriedené podľa ich aktuálnej hodnoty $dp$. Potom by výpočet bol triviálny. Chceš $dp[t+1][t+1]$? Zober zo štruktúry (v čase $t+1$, pred pridaním krompáča $t+1$) krompáč $j_0$ s najväčším $dp[t+1][j_0]$. Potom $dp[t+1][t+1] = dp[t+1][j_0] - c_{t+1}$. Ako však udržiavať štruktúru utriedenú?

Vezmime fixný deň $t$. Krompáč $j \leq t$ nazvime *nadbytočný*, ak

- existuje krompáč $k \leq t$ taký, že ($dp[t][k] > dp[t][j]$ a $b_k \geq b_j$) alebo ($dp[t][k] = dp[t][j]$ a $b_k > b_j$), ALEBO
- existuje krompáč $k \leq t$ taký, že $dp[t][k] = dp[t][j]$ a $b_k = b_j$ a $k > j$.

Ak sa nám v nejakom čase stane, že je nejaký krompáč nadbytočný, tak na neho môžeme navždy zabudnúť. A presne to aj budeme robiť.

Navyše si všimnime, že ak v čase $t$ máme v štruktúre iba nenadbytočné krompáče a utriedime ich podľa $b_j$, tak ich hodnoty $dp[t][j]$ musia v tomto poradí (nie nutne ostro) klesať. Skutočne, ak $b_j \leq b_k$ a zároveň $dp[t][j] \leq dp[t][k]$, tak aspoň jeden z nich je nadbytočný. Takže nám stačí udržovať nenadbytočné krompáče

utriedené podľa ich $b_j$, v prípade zhody podľa $j$ vzostupne. Konkrétna implementácia môže byť napríklad ako `set` v C++ (binárny vyvažovaný strom) s prvkami tvaru $(b_j, j)$.

Ako si však udržovať v štruktúre iba nenadbytočné krompáče? Deň sa práve zmenil z $t$ na $t+1$, ešte sme však nepridali krompáč $t+1$. Potrebujeme spraviť dve úpravy:

- Odstránenie nadbytočných krompáčov: V štruktúre máme krompáče $j_0, \ldots, j_m$ zoradené podľa $b_j$. Ak sa nejaký krompáč $j_i$ stal nadbytočným, tak si všimnime, že je nadbytočný s krompáčom $k = j_{i+1}$ (v definícii vyššie). Budeme preto udržiavať set udalostí $U$, v ktorom pre každý krompáč $j_i$ v našej štruktúre budeme mať udalosť $(t_*, j_i)$ hovoriacu, že v čase $t_*$ sa $j_i$ stane nadbytočným (s $k = j_{i+1}$). Potom vieme ľahko spraviť úpravu poradia: Pokiaľ je v $U$ udalosť s časom $\leq$ ako $t+1$, vezmi príslušný krompáč, odstráň ho z poradia a updatni udalosti $U$.
- Pridanie krompáču $t+1$ a odstránenie tým vzniknutých nadbytočných krompáčov: Toto je jednoduché. Pozrieme sa kam do poradia by patril $t+1$. Ten je buď hneď nadbytočný (v takom prípade ho nepridáme vôbec), alebo nie je (v takom prípade ho pridáme a prípadne povyhadzujeme krompáče, ktoré sa tým stali nadbytočné (tie sú nutne v poradí susedné pod ním)).

Poznamenajme, že nám nevadí, keď v konkrétnej implementácii budú pre nejaké $j_*$ v $U$ zostávať aj zastaralé udalosti.

### Zhrnutie

Udržujeme si aktuálny deň $t$, nenadbytočné krompáče v poradí podľa $(b_j, j)$ a set udalostí $U$.

Update z $t$ na $t+1$ vyzerá nasledovne:

- Pokiaľ je v $U$ udalosť s $t_* \leq t+1$, vyhoď príslušný krompáč zo štruktúry (stal sa nadbytočný) a pridaj nové udalosti do $U$. Opakuj.
- Nájdi $j_0$ s najmenším $b_{j_0}$ (a preto najväčším $dp[t+1][j_0]$), z toho spočítaj $dp[t+1][t+1]$.
- Ak $t+1$ nie je nadbytočný, pridaj ho a povyhadzuj krompáče, ktoré sa stali nadbytočné (v poradí susedné pod ním).

Na konci (v dni $(N+1)$), jednoducho nájdeme $j_0$ s najmenším $b_{j_0}$ a odpoveď bude $dp[N+1][j_0]$.

Celkovo spravíme $O(N)$ operácii s oboma setmi a tieto operácie sú v $O(\log N)$. Celková zložitosť bude $O(N \log N)$. Pamäťová zložitosť bude $O(N)$.

### Listing programu (C++)

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

using namespace std;

typedef long long ll;

#define ff first
#define ss second

struct State {
    int time;
    // optimal miners: {miner_income, miner_id}
    // redundant miners: {time, miner_id}
    set<pair<ll, int>> optimal_miners, redundant_miners;
    vector<ll> price, income, base_income;

    ll miner_value(int id) {
        return (time-id)*income[id]+base_income[id];
    }

    ll get_swap_time(int down_id, int up_id) {
        if(income[up_id] == income[down_id]) return 0;
        double temp_time = (base_income[down_id]-down_id*income[down_id])-(
            ↪ base_income[up_id]-up_id*income[up_id]);
```

```
        temp_time /= income[up_id]-income[down_id];
        ll T = max((ll) 0, (ll) ceil(temp_time));
        return T;
    }

    void add_miner(int id) {
        ll val = miner_value(optimal_miners.begin()->ss);
        if(val < price[id]) return;
        base_income[id] = val-price[id];
        auto up_it = optimal_miners.upper_bound({income[id], INT_MIN});
        if(up_it != optimal_miners.end()) {
            if(miner_value(up_it->ss) >= miner_value(id)) return;
            ll T = get_swap_time(id, up_it->ss);
            redundant_miners.insert({T, id});
        }
        if(up_it == optimal_miners.begin()) {
            optimal_miners.insert({income[id], id});
            return;
        }
        auto down_it = prev(up_it);
        ll T = get_swap_time(down_it->ss, id);
        redundant_miners.insert({T, down_it->ss});
        optimal_miners.insert({income[id], id});
    }

    void remove_miner(int id) {
        pair<ll, int> curr = {income[id], id};
        if(optimal_miners.find(curr) == optimal_miners.end()) return;
        optimal_miners.erase(curr);
        auto up_it = optimal_miners.upper_bound(curr);
        if(up_it == optimal_miners.end() || up_it == optimal_miners.begin())
            ↪ return;
        auto down_it = prev(optimal_miners.upper_bound(curr));
        int up_id = up_it->ss, down_id = down_it->ss;
        ll T = get_swap_time(down_id, up_id);
        redundant_miners.insert({T, down_id});
    }

    void simplify() {
        while(redundant_miners.size() > 0) {
            auto curr = *redundant_miners.begin();
            if(curr.ff > time) break;
            redundant_miners.erase(curr);
            int id = curr.ss;
            remove_miner(id);
        }
    }
};

int main() {
    int N; ll B;
    cin >> N >> B;
    State s;
    s.time = 0;
    s.optimal_miners.insert({0, 0});
    s.price.resize(N+1);
    s.income.resize(N+1);
    s.base_income.resize(N+1, 0);
```

```
    s.base_income[0] = B;
    for(int t = 1; t <= N; t++) {
        cin >> s.price[t] >> s.income[t];
        s.time = t;
        s.simplify();
        s.add_miner(t);
        s.simplify();
    }
    s.time++;
    s.simplify();
    cout << s.miner_value(s.optimal_miners.begin()->ss) << "\n";
    return 0;
}
```

Mato

## 7. Osvetlenie

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

V tejto úlohe bolo treba určiť, ktoré tlačidlá, prepínajúce žiarovku spolu s jej susedmi, treba stlačiť, aby sme dosiahli požadovanú konfiguráciu zapnutých žiaroviek.

### Pomalšie riešenie

Dôležité pozorovanie je, že keď si zafixujeme, ktoré tlačidlá v prvom riadku stlačíme, zvyšok je už jednoznačný. Pozrime sa na žiarovky v prvom riadku potom, ako stlačíme tlačidlá v prvom riadku. Niektoré sú v správnom stave a niektoré sú v nesprávnom. Tie žiarovky, ktoré sú nesprávne, musíme ešte prepnúť a jediné tlačidlo ktoré ešte môžeme použiť je to priamo pod ňou (tlačidlá v prvom riadku máme zafixované). Naopak, tlačidlo pod správnou žiarovkou nesmieme prepnúť, lebo by sme ju prepli do nesprávneho stavu. To nám jednoznačne určuje, ktoré tlačidlá v druhom riadku treba stlačiť.

Takto môžeme pokračovať. Teraz každú žiarovku v druhom riadku vieme prepnúť len tlačidlom priamo pod (tie v prvom a druhom riadku sú určené) a teda vieme, ktoré tlačidlá v treťom riadku treba stlačiť.

Toto opakujeme postupne pre všetky riadky, až kým určíme ktoré všetky tlačidlá budú stlačené.

Stačí nám teda vyskúšať všetky možnosti pre prvý riadok, podľa vždy toho určiť zvyšok a skontrolovať, či dosiahneme, že všetky žiarovky sú správne.

Dokopy je $2^m$ možností pre prvý riadok. Každú možnosť vyskúšame v $O(nm)$. Časová zložitosť tohto riešenia je $O(2^n nm)$.

### Listing programu (C++)

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

int main() {
        int n, m;
    cin >> n >> m;
    vector<vector<int>> ziarovky(n, vector<int>(m));
    for(int i = 0; i < n; i++) for(int j = 0; j < m; j++) cin >> ziarovky[i][j];
    int N = (1<<m);
    for(int start = 0; start < N; start++) {
        bool ok = true;
        vector<vector<int>> prep(n, vector<int>(m, 0));
        for(int i = 0; i < m; i++) prep[0][i] = (bool)(start&(1<<i));
        vector<vector<int>> z(n, vector<int>(m));
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < m; j++) {
                if(prep[i][j]) {
                    z[i][j] = !z[i][j];
                    if(j) z[i][j-1] = !z[i][j-1];
                    if(j < m - 1) z[i][j+1] = !z[i][j+1];
```

```
                    if(i < n - 1) z[i+1][j] = !z[i+1][j];
                }
            }
            for(int j = 0; j < m; j++) {
                if(i < n - 1) prep[i+1][j] = (z[i][j] != ziarovky[i][j]);
                else {
                    if(z[i][j] != ziarovky[i][j]) ok = false;
                }
            }
        }
        if(ok) {
            for(auto x : prep)
                for(int i = 0; i < m; i++)
                    cout << (bool)(x[i]) << (i == m - 1 ? '\n' : ' ');
            return 0;
        }
    }
    cout << "-1\n";
}
```

### Zrýchlenie

Toto riešenie vieme zrýchliť pomocou bitovej mágie. Každý riadok žiaroviek budeme reprezentovať jedným číslom, ktorého binárna reprezentácia hovorí o tom, ktoré žiarovky sú zapnuté a ktoré sú vypnnuté. Tak isto budeme reprezentovať stlačené a nestlačené tlačidlá.

Potom pomocou bitových operácií (`^`, `<<`, `>>`, `&`, `|`) vieme vyhodnotiť každý riadok v konštantnom čase: 1. Máme číslo ktoré reprezentuje tlačidlá ktoré chceme stlačiť, nazvime ho $s$. Na základe toho chceme aktualizovať aktuálny stav žiaroviek. Riadok nad a riadok pod nám stačí zmeniť na **xor** (`^`) hodnoty toho riadku a $s$. Takto sa zmenia bity na tých miestach, kde boli jednotky v $s$. V aktuálnom riadku potrebujeme prepnúť žiarovky na miestach kde bolo stlačené tlačidlo a ešte naľavo a napravo od nich. Najprv na riadok aplikujeme **xor** s $s$, čím prepneme tie na stlačených miestach. Potom môžeme použiť bitový posun doprava a doľava (`>>`, `<<`) na to, aby sme dostali jednotky na pozíciach napravo a naľavo od stlačených tlačidiel. Riadok teda ešte pre**xor**ujeme `s>>1` a `s<<1`, čím prepneme tieto žiarovky. Nakoniec si treba dať pozor aby sme odstránili jednotky za $m$-tou pozíciou, ktoré mohli vzniknúť bitovým posunom. Na to použijeme **and** (`&`) s číslom `(1<<m)-1`, ktoré má jednotky na prvých $m$ pozíciach. 2. Ako druhý krok musíme porovnať aktuály stav žiaroviek s očakávaným stavom žiaroviek, aby sme zistili, ktoré tlačidlá v ďalšom riadku treba stlačiť. Na to použijeme znova **xor** aktuálneho stavu s očakávaným stavom. Na miestach kde sa líšia dostaneme 1 a na ostatných miestach 0.

### Listing programu (C++)

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

int main() {
        int n, m;
    cin >> n >> m;
    vector<int> correct(n);
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++)
        {
            int x;
            cin >> x;
            correct[i] += x<<j;
        }
    }
    int N = (1<<m);
    for(int start = 0; start < N; start++) {
        vector<int> prep(1, start);
```

```
        int next = 0;
        for(int i = 0; i < n; i++) {
            int cur = next;
            cur ^= prep[i] ^ (prep[i]>>1) ^ (prep[i]<<1) & (N-1);
            next = prep[i];
            prep.push_back(cur ^ correct[i]);
        }
        if(prep.back() == 0) {
            for(int i = 0; i < n; i++) {
                for(int j = 0; j < m; j++) {
                    cout << (bool)(prep[i] & (1<<j)) << (i == m - 1 ? '\n' : ' '
                        ↪ );
                }
            }
            return 0;
        }
    }
    cout << "-1\n";
}
```

### Vzorové riešenie

Označme si $z_{i,j}$ požadovaný stav žiarovky (0 = vypnutá, 1 = zapnutá) v riadku $i$ a stĺpci $j$, ktorý dostaneme na vstupe. Podobne si označme $t_{i,j}$ stav tlačidla (0 = stlačené, 1 = nestlačené) v riadku $i$ a stĺpci $j$.

Celú úlohu teraz vieme zapísať pomocou sústavy lineárnych rovníc modulo 2. Hľadáme také hodnoty premenných $t_{i,j}$, aby platili rovnice typu:

$t_{i,j} + t_{i+1,j} + t_{i-1,j} + t_{i,j+1} + t_{i,j-1} \equiv z_{i,j} \pmod 2$

To znamená, že z tých piatich tlačidiel, ktoré vedia prepnúť danú žiarovku, musí byť počet stlačených modulo 2 rovnaký, ako požadovaný stav žiarovky.

Na vyriešenie takejto sústavy rovníc môžeme použiť Gaussovu eliminačnú metódu[2]. Treba len dať pozor na to, že sme modulo 2, teda $1 + 1 = 0$ a podobne.

Gaussova eliminácia pri $k$ rovniciach o $k$ neznámych má časovú zložitosť $O(k^3)$. V našom prípade je $k = nm$, teda časová zložitosť celého prgramu je $O((nm)^3)$.

### Listing programu (C++)

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

vector<int> solve(vector<vector<int>> matrix) {
    int n = matrix.size();
    vector<int> finished(n, -1);
    for(int j = 0; j < n; j++) {
        for(int i = 0; i < n; i++) {
            if(matrix[i][j] && finished[i] == -1) {
                for(int ii = 0; ii < n; ii++) {
                    if(ii == i) continue;
                    if(matrix[ii][j] == 0) continue;
                    for(int jj = j; jj <= n; jj++) {
                        matrix[ii][jj] ^= matrix[i][jj];
                    }
                }
                finished[i] = j;
                break;
            }
            if(i == n-1) {
                for(int ii = 0; ii < n; ii++) matrix[ii][j] = 0;
```

[2] https://en.wikipedia.org/wiki/Gaussian_elimination

```
            }
        }
    }
    vector<int> solutions(n, 0);
    for(int i = 0; i < n; i++) if(finished[i] != -1) solutions[finished[i]] =
        ↪ matrix[i][n];
    for(int i = 0; i < n; i++) {
        if(finished[i] == -1) {
            int x = 0;
            for(int j = 0; j < n; j++) x ^= (matrix[i][j] & solutions[j]);
            if(x != matrix[i][n]) return {};
        }
    }
    return solutions;
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> matrix(n*m, vector<int>(n*m+1, 0));
    for(int i = 0; i < n; i++) for(int j = 0; j < m; j++) {
        int id = i*m+j;
        cin >> matrix[id][n*m];
        matrix[id][id] = 1;
        if(i > 0) matrix[id][id-m] = 1;
        if(i < n-1) matrix[id][id+m] = 1;
        if(j > 0) matrix[id][id-1] = 1;
        if(j < m-1) matrix[id][id+1] = 1;
    }
    vector<int> sol = solve(matrix);
    if(sol.size() == 0) cout << "-1\n";
    else for(int i = 0; i < n*m; i++) cout << sol[i] << ((i+1) % m ? " " : "\n")
        ↪ ;
}
```

Tellegar

## 8. Masaker v Kratarii

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

### Počet operácii

Konštanta $K$ nech predstavuje počet jednotiek na vstupe. Ak je vstup validný, respektíve ak existuje aspoň jedna permutácia, ktorá sa po konečnom počte operácií zmení na postupnosť na vstupe, tak počet vykonaných operácií je rovno $K - 1$, pretože po každej operácií sa úsek jednotiek zväčši o jedna.

### Úseky

Na základe počtu operácií sa pre každé číslo na vstupe dá určiť úsek pozícií, kde sa v pôvodnej permutácií toto číslo mohlo nachádzať. Treba si však všimnúť, že číslo sa nemohlo nachádzať na pozícii, kde do $K$ pozícií od neho sa na vstupe nachádza väčšie číslo. Na základe tohto sa nemôžu dva úseky prekrývať, pretože by to znamenalo, že na rovnakej pozícii môžu byť dve rôzne čísla, ktoré sa v postupnosti zo vstupu nachádzajú. To by znamenalo, že do $K$ pozícií od tejto spoločnej sa na vstupe nachádzajú obe čísla, ale pritom platí, že jedno z týchto čísel je menšie a nemôže sa teda na tejto pozícii nachádzať.

```
vstup: 1 1 1 2 2 4
úseky: 1   2 4 4 4
       1 · 1 · 3
```

Tieto úseky vieme nájsť jedným prechodom vstupu: Ak narazíme na prechod z väčšieho čísla na menšie, znamená to, že úsek väčšieho čísla môže začínať najskôr $K$ pozícií pred ním a končiť najneskôr na jeho pozícii,

a že úsek menšieho čísla môže začínať najskôr a končiť najneskôr na svojej pozícii. A podobne naopak, ak narazíme na prechod z menšieho na väčšie, úsek menšieho môže začínať najskôr a končiť najneskôr $K$ pozícií pred jeho pozíciou a úsek väčšieho môže začať najskôr $K$ pozícií pred svojou pozíciou a končiť najneskôr na svojej pozícii.

### Dolné hranice

Spomínané úseky však nijako neurčujú kde sa môžu nachádzať zvyšné čísla, ktoré sa na vstupe nevyskytujú. Pre tieto čísla vieme určiť dolné hranice, teda pre každú pozíciu najmenšie číslo ktoré sa tam mohlo nachádzať v pôvodnej permutácií. Teda dolná hranica pre nejakú pozíciu je určená maximom na intervale dĺžky $K$ začínajúcom na tejto pozícii. Toto sa dá nájsť pomocou okienkového minima.

```
        vstup: 1 1 1 2 2 4
        úseky: 1   2 4 4 4
dolné hranice: 1 2 2 4 4 4
```

Všimnime si, že pre každú pozíciu v úseku je rovnaká dolná hranica. Je to tak, pretože dolná hranica nemôže byť menšia ako číslo, ktorého je to úsek, a zároveň na týchto pozíciach môže byť aspoň to číslo. Z tohto dôvodu, keď si zvolíme niektorú možnú pozíciu pre číslo z jeho úseku, dolné hranice sa nezmenia a teda môžme voliť pozície a čísla ktoré sa na vstupe nenachádzajúcu podľa dolných hraníc nezávislo na sebe.

Potrebujeme si však ešte rozmyslieť, ako vyberať čísla, ktoré na vstupe nie sú. Môžu byť kdekoľvek, kde nie je žiaden úsek a ak tam úsek je, tak jednu (ľubovoľnú) pozíciu vynecháme, pričom musia vyhovovať dolným hraniciam. Ak by sme tieto čísla vyberali akokoľvek, mohlo by sa stať, že by pre pozície s vyššou dolnou hranicou nezostalo číslo. Keďže platí, že pre pozíciu s vyššou dolnou hranicou je na výber podmnožina čísel ako pre pozíciu s nižšou, tak môžme vyberať náhodne od pozícií s väčšou. Na to však treba tieto dolné hranice zoradiť podľa veľkosti, pričom si stačí uvedomiť, že na ne vieme použiť counting sort.

```
        vstup: 1 1 1 2 2 4
        úseky: 1   2 4 4 4
               1 · 1 · 3
dolné hranice: 1 2 2 4 4 4
                 3
 zvyšné čísla:   5   5 5
                 6   6 6
```

Výsledok je tým pádom $1 \cdot 1 \cdot 3$ (úseky) krát $2 \cdot 1 \cdot 1$ (zvyšné čísla), teda dokopy $3 \cdot 2 = 6$

Možné postupnosti:

```
1 3 2 4 5 6
1 3 2 4 6 5
1 3 2 5 4 6
1 3 2 6 4 5
1 3 2 5 6 4
1 3 2 6 5 4
```

### Neplatné vstupy

Na vstupe musí byť aspoň jedna jednotka. Všetky rovnaké čísla na vstupe musia tvoriť jeden súvislý interval. Pri hľadaní úsekov, nemusí existovať neprázdny úsek alebo pri vyberaní zvyšných čísel nemusí byť možné čísla rozdeliť. V týchto prípadoch neexistuje žiadna permutácia ktorá sa po $K - 1$ operáciach zmení na zadanú postupnosť, teda výsledok je 0.

### Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;
#define pn 1000000007


int main() {
```

```
int n; cin >> n;
deque <int> A(n);
int k = 0;
for (int i = 0; i < n; i++) { // O(n)
        cin >> A[i];
        if (A[i] == 1) k++;
}

if (k == 0) {
        cout << 0 << endl;
        return 0;
}

if (k == n) { // out = n!
        uint64_t out = 1;
        for (int i = 2; i <= n; i++) out = out * i % pn;
        cout << out << endl;
        return 0;
}

while (A.front() != 1 || A.front() == A.back()) {
        A.push_front(A.back());
        A.pop_back();
}
A.shrink_to_fit();

bool possible = true;
vector <bool> notInA(n+1, true);
for (int i = 1; i < n; i++) { // O(n)
        if (A[i-1] != A[i] && !notInA[A[i]]) possible = false;
        notInA[A[i]] = false;
}
if (!possible) {
        cout << 0 << endl;
        return 0;
}

deque <array <int, 3>> u;   // {num, left, right}
deque <int> mxw;            // max window {pos}
vector <int> l(n, 1); // lower bounds
vector <int> lc(n+1);    // lc[i] - count of numbers not in A and at
   ↪ least i
lc[n] = notInA[n];
u.push_front({A.back(), n-k, n-1});
for (int i = n-1; i >= 1; i--) {                                    // O
   ↪ (n)
        while (!mxw.empty() && A[mxw.front()] <= A[i]) mxw.pop_front();
           ↪ // O(1)
        mxw.push_front(i);
        if (mxw.back() == i+k) mxw.pop_back();
        l[i] = A[mxw.back()];

        if (A[i-1] != A[i]) {
                u.front()[2] = min(u.front()[2], i);
                if (A[i-1] < A[i]) {
                        u.push_front({A[i-1], i-k, i-k});
                } else if (A[i-1] > A[i]) {
                        u.front()[1] = max(u.front()[1], i);
```

```
                                u.push_front({A[i-1], i-k, i-1});
                        }
                        if (u.front()[1] > u.front()[2]) {
                                cout << 0 << endl;
                                return 0;
                        }
                }

                lc[i] = lc[i+1] + notInA[i];
        }

        /*
        for (int i = 0; i < n; i++) cout << A[i] << ' ';
        cout << endl;
        for (int i = 0, j = 0; i < n; i++) {
                if (j < u.size() && i >= u[j][1]) {
                        cout << u[j][0] << ' ';
                        cout.flush();
                } else {
                        cout << "  ";
                }
                if (i == u[j][2]) j++;
        }
        cout << endl;
        for (int i = 0; i < n; i++) cout << l[i] << ' ';
        cout << endl;
        */
//      for (auto i : u) cout << i[0] << ' ' << i[1] << ' ' << i[2] << endl;

        uint64_t out = 1;
        vector <int> countSort(n+1, 0);
        for (int i = 0, j = 0; i < n; i++) {// |u[i]| < 0 - not possible
//              if (i == u[j][2]) out = out * max(0, u[j][2]-u[j][1]+1) % pn, j
    ↪ ++;
                if (i == u[j][2]) {
                        if (j < (int)u.size()) out = out * max(0, u[j][2]-u[j
                            ↪ ][1]+1) % pn, j++;
                } else countSort[lc[l[i]]]++;
        }
//      for (int i = 0; i < n+1; i++) cout << i << ' ' << lc[i] << ' ' <<
    ↪ countSort[i] << endl;
        for (int i = 0, removed = 0; i < (int)countSort.size() && out; i++) { //
            ↪  O(n - |u|)
                while (countSort[i]) {
                        out = out * (i-removed) % pn;
                        countSort[i]--, removed++;
                }
        }

        cout << out << endl;

        return 0;
}
```