Korešpondenčný seminár z programovania
XLIII. ročník, 2025/26
FMFI UK,
Mlynská Dolina, 842 48 Bratislava

# Vzorové riešenia 2. kola zimnej časti

Konco

## 1. O vrecku čo zradilo ostatné

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

### Priame prehľadávnie

Najjednoduchším riešením, ktoré by nám mohlo napadnúť, je postupne vždy z jediného vrecka zobrať jednu mincu a odvážiť ju. Potom podľa hodnoty na váhe hneď vieme, či sú mince v danom vrecku falošné alebo nie.

Takéto riešenie však v najhoršom prípade potrebuje $n$ vážení, čo postačuje na vyriešenie prvej sady.

Môžeme si ale všimnúť, že keď už sme odvážili všetky vrecká okrem jedného a všetky obsahovali pravé mince, tak falošné musia byt v tom poslednom. Takto si vieme ušetriť jedno váženie a tak vyriešiť aj druhú sadu.

### Zaujímavejšie prístupy

Tretia sada sa dá vyriešiť riešeniami založenými na rôznych dekompozíciách.

Jednou možnosťou je rozdeliť si vrecúška na niekoľko skupín (napríklad mať skupiny po 10 vrecúškach) a najprv zistiť, v ktorej skupine sa falošné vrecko nachádza. Následne si individuálne prejdeme len vrecká v tejto skupine a zistíme, ktoré konkrétne vrecko obsahuje falošné mince.

Takéto riešenie nám zaberie $S + K$ vážení, kde $S$ je počet skupín a $K$ je maximálny počet vreciek v skupine. To by sa dalo ešte optimalizovať rovnakým trikom ako v minulej sekcii, ale na vyriešenie tretej sady to nebolo potrebné.

Táto myšlienka sa dá rozšíriť aj na viacero úrovní – skupinu, kde sa nachádza falošné vrecko môžeme opäť rozdeliť na menšie skupiny a takto pokračovať, až kým nám neostane len jediné vrecko. Ak vám táto myšlienka príde zaujímavá, mohli by vás zaujať Intervaláče[1].

V každom prípade vám však takéto riešenia s poslednou sadou vstupov nepomôžu pretože...

### Optimálne riešenie

Na vyriešenie celej úlohy však stačí použiť iba jediné váženie.

Po chvíľke rozmýšľania si uvedomíme, že tu nám už nebude stačiť z každého vrecka brať maximálne jednu mincu. V skutočnosti si môžeme všimnúť silnejšiu vec – z každej dvojice vreciek musíme zobrať iný počet mincí. Ak by sme tak nespravili a náhodou by sa stalo, že by v jednom z nich boli falošné mince, nemali by sme šancu určiť, ktoré z nich to je.

Predpokladajme teda, že sme z každého vrecka zobrali iný počet mincí, z prvého sme ich zobrali $a_1$, z druhého $a_2$, ..., z $n$-tého $a_n$. Keby boli všetky mince pravé, váha by ukázala $P = 50 \cdot (a_1 + a_2 + ... + a_n)$ gramov. Ale ak bolo $i$-te vrecko falošné, váha nám v skutočnosti ukázala hodnotu $W = 50 \cdot (a_1 + a_2 + ... + a_{i-1} + a_{i+1} + ... + a_n) + 55 \cdot a_i$. To znamená, že rozdiel medzi skutočnou a očakávanou hmotnosťou je $W - P = 5 \cdot a_i$. $W$ ale poznáme z váženia a $P$ vieme jednoducho spočítať. Teda potom vieme, že $a_i = \frac{W-P}{5}$, a keďže sme z každého vrecka zobrali iný počet mincí, vieme jednoznačne určiť, ktoré vrecko je falošné.

Prakticky je najjednoduchšie zobrať z $i$-teho vrecka $i$ mincí, potom je $a_i = i$ a teda $i = \frac{W-P}{5}$.

### Listing programu (Python)

```
def odvaz(vahy_minci):
    assert len(vahy_minci)==n
    print("?", *vahy_minci, flush=True)
    result = int(input())
    return result

```

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

```python
def odpovedaj(index):
    print("!", index, flush=True)
    exit(0)

n = int(input())

while True:
    # Z i-teho vrecka vezmeme i minci
    pocty_minci = list(range(1, n+1))

    #  Ak by boli vsetky prave, tak by vazili presne p gramov
    p = sum(pocty_minci) * 50

    w = odvaz(pocty_minci)

    odpovedaj((w-p)//5)
```

## 2. Bazalky a čili papričky

12 b za popis, 8 b za program

### Riešenie hrubou silou

Úlohu vieme riešiť pomocou hrubej sily. Iterujeme zľava doprava po prvých $K$ pozíciách. Ak sa na nejakej pozícii nachádza koreníčka s číslom väčším než $K$, nájde sa najbližšia *správna* koreníčka s číslom $K$ napravo a pomocou postupných susedných výmen sa presunie na túto pozíciu. Každá takáto výmena sa započíta do výsledku. Algoritmus pokračuje, kým prvých K pozícií neobsahuje iba koreníčky 1 až K, pričom ich vnútorné poradie nie je dôležité. Týmto postupom sa nikdy nerobia zbytočné výmeny, pretože každá výmena rieši konkrétny problémový pár. Problémový pár je dvojica koreníčiek, kde koreníčka s číslom väčším než K stojí pred koreníčkou s číslom menším alebo rovným K, a každá takáto inverzia musí byť odstránená aspoň jednou susednou výmenou.

### Optimálne riešenie

Kľúčové je uvedomiť si, že koreníčky na indexoch $i$ a $j$ vieme vymeniť pomocou $|i-j|$ susedných výmen, takže nie je potrebné jednotlivé výmeny simulovať. Nech $p_i$ označuje index $i$-tej koreníčky s číslom $\leq K$ v pôvodnom poradí poľa (číslované zľava doprava). Koreníčka na pozícii $p_i$ má pred sebou presne $p_i - i$ koreníčiek s číslom väčším než $K$, s ktorými tvorí problémové dvojice. Pole prechádzame zľava doprava. Pri každej koreníčke s číslom $\leq K$ pripočítame do výsledku jej aktuálny index $i$, čím získame hodnotu

$$\sum p_i.$$

Z tejto hodnoty následne odčítame

$$\sum_{i=0}^{K-1} i = \frac{(K-1)K}{2}.$$

Výsledkom je presný počet problémových dvojíc, a teda aj minimálny počet potrebných susedných výmen.

### Časová a pamäťová zložitosť

Takéto riešenie bude mať časovú zložitosť $O(n)$ a pamäťovú zložitosť $O(1)$

V riešení sa využíva iba lineárne iterovanie cez prvky, pričom ich nie je potrebné ukladať do pamäte. A zopár premenných.

### Listing programu (C++)

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

using namespace std;
using ll = long long;
```

```cpp
using vi = vector<int>;
using vvi = vector<vector<int>>;
using pii = pair<int, int>;
using vll = vector<long long>;
using vvll = vector<vector<long long>>;
using vstr = vector<string>;
using str = string;

#define PB push_back
#define ALL(x) (x).begin(), (x).end()
#define FOR(i, s, f) for (int(i) = (s); i < (f); (i)++)
#define ROF(i, s, f) for (int(i) = (s); i > (f); (i)--)
#define loop(x) for (int i = 0; i < (x); i++)

const int INF = 1e9;
const long long LLINF = 1e18;
const int MOD = 1e9 + 7;
const long long LLMOD = 1e18 + 7;


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

    vll lst(n);
    FOR(i,0,n) cin >> lst[i];
    vi pos;
    FOR(i,0,n){
        if (lst[i] <= k){
            pos.PB(i);
        }
    }
    ll ans = 0;
    FOR(i,0,k){
        ans += pos[i] - i;
    }
    cout <<ans<<'\n';
}
```

### Listing programu (Python)

```python
n,k = map(int, input().split())
pole = [int(i) for i in input().split()]
print(sum([i if pole[i] <= k else 0 for i in range(n)])-(k-1)*k//2)
```

## 3. Eww, niečo je tu rozliate

12 b za popis, 8 b za program

### Pozorvanie

Zadanie hovorí dve zaujímavé veci:

- Filipko môže začať na ľubovoľnom z prvých dvoch schodov.
- Môže vykročiť ľubovoľnou nohou, ale od toho momentu musí striedať nohy pri každom kroku.

To znamená, že na každom schode nás zaujímajú dve hodnoty: aké je minimálne zašpinenie, ak naň Filipko stúpi ľavou nohou, a to isté pre pravú nohu.

### Naivné riešnie

Najpriamočiarejšie riešenie je prejsť všetky možné cesty, ktorými sa vieme dostať na posledný schod. Začneme na jednom z prvých dvoch schodov a ľubovoľnou nohou. V každom kroku máme na výber ísť o jeden alebo o dva schody vyššie; tým sa automaticky určí, ktorou nohou stúpime na nový schod a akú hodnotu zašpinenia k výsledku pripočítame. Takto rekurzívne preskúmame všetky kombinácie krokov, na konci si vyberieme najmenší nájdený súčet. Tento postup síce vždy nájde správnu odpoveď, ale má exponenciálnu časovú zložitosť $O(2^n)$.

### Vzorové riešenie

Túto úlohu budeme riešiť dynamickým programovaním[2].

Pre každú nohu si budeme postupne dorátavať optimálne hodnoty. A to tak, že postupne budeme dorávať hodnotu na aktuálnom schode pomocou hodnoty na predchádzajúcich schodoch.

Definujeme si dve polia:

- `dl[i]` — najmenšie zašpinenie, ak Filipko skončí na i-tom schode a stúpi naň ľavou nohou
- `dp[i]` — to isté, ale pre pravú nohu

Aby sme mohli vypočítať `dl[i]`, musíme sa sem nejako dostať. Keďže striedame nohy, vieme sem prísť len s pravou nohou, a to:

- buď zo schodu i-1, teda `dp[i-1] + l[i]`
- alebo zo schodu i-2, teda `dp[i-2] + l[i]`

Analogicky pre pravú stranu schodov.

Ešte potrebujeme vyriešiť začiatočné prípady pre prvé dva schody na oboch stranách. O tých vieme rovno povedať, že to budú určite hodnoty len daných schodov, keďže sa na ne dokážeme dostať jedným krokom – teda priamo na ne. Teda `dp[0] = p[0]; dp[1] = p[1]` a podobne pre ľavú nohu.

Potom dynamicky zostavujeme riešenie od najnižšieho schodu k najvyššiemu. Na posledný schod n-1 musíme došliapnuť, ale môžeme naň stúpiť ľavou alebo pravou nohou — vyberieme teda minimum: $min(d_l[-1], d_p[-1])$.

### Prečo to funguje ?

Správnosť riešenia vyplýva z toho, že `dl[i]` a `dp[i]` vždy uchovávajú minimálne možné zašpinenie pri skončení na $i$-tom schode ľavou, resp. pravou nohou. Pre prvé dva schody vieme tieto hodnoty určiť priamo, keďže na ne môžeme vstúpiť jediným krokom. Pre každý ďalší schod sa môžeme naň ľavou nohou dostať len z pravou nohou na $i-1$ alebo $i-2$ (aby sme dodržali striedanie nôh a povolené kroky), takže optimálna hodnota `dl[i]` je $min(dp[i1], dp[i2]) + l[i]$ , a analogicky pre dp[i]. Keďže rekurencia zohľadňuje všetky možné prípady a vždy berie minimum, výsledok $min(dl[n1], dp[n1])$ je skutočne globálne najmenšie možné zašpinenie.

### Zložitosť

Časová zložitosť: $O(n)$ — pre každý schod spravíme konštantný počet operácií.

Pamäťová zložitosť: $O(n)$ — ukladáme si dve hodnoty pre každý schod.

*Optimalizácia*: Ak potrebujeme ešte menej pamäte, môžeme si namiesto celých polí dl a dp pamätať len posledné dve hodnoty. Keďže na výpočet `dl[i]` a `dp[i]` používame iba hodnoty z $i-1$ a $i-2$, staršie nepotrebujeme. Preto si stačí pamätať len tieto dve posledné hodnoty pre každú nohu a zvyšok môžeme zahodiť.

### Listing programu (Python)

```
n = int(input())
l = list(map(int, input().split()))
p = list(map(int, input().split()))

# DP zoznamy pre kazdu z noh
dpL = [float('inf')] * n
```

[2] https://school.ksp.sk/courses/techniques-2/dynamic-programming/intro/

```
dpP = [float('inf')] * n

# zaciatocne pripady dynamiky pre prve dva schody.
dpL[0], dpP[0] = l[0], p[0]
dpL[1], dpP[1] = l[1], p[1]

# naplname dynamiku pre kazdy schod podla nohy, ktorou nan stupame
for i in range(2, n):
    dpL[i] = min(dpP[i-1], dpP[i-2]) + l[i]
    dpP[i] = min(dpL[i-1], dpL[i-2]) + p[i]

# vysledok ako minimum posledneho schodu lavou a pravou nohou
print(min(dpL[-1], dpP[-1]))
```

Autor

## 4. Delíme sa s jedlom

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

### Prvé nápady

Situáciu z tejto úlohy si vieme predstaviť pomocou orientovaného grafu, kde vrcholy zodpovedajú vedúcim a od vedúceho $i$ k vedúcemu $j$ vedie hrana práve vtedy, keď vedúci $i$ je ochotný podeliť sa s vedúcim $j$. Chceme vybrať takú množinu vrcholov, aby bol každý vrchol v tejto množine alebo aby susedil s nejakým vrcholom z tejto množiny.

Najľahšie riešenie samozrejme dosiahneme tak, že každý vedúci donesie jedlo. Takto však bude musieť doniesť jedlo až všetkých $n \leq 1000$ vedúcich, čo je veľmi neoptimálne.

O niečo lepšie riešenie vieme dosiahnuť tak, že zaručíme, aby každý vedúci, ktorý donesie jedlo, pokryl ešte aspoň nejakého jedného ďalšieho vedúceho, ktorý tak nebude musieť doniesť jedlo. To vieme spraviť tak, že popárujeme vedúcich do dvojíc $(1, 2)$, $(3, 4)$ a tak ďalej, pričom v prípade nepárneho počtu vedúcich nám ostane na konci ešte jeden nespárovaný vedúci.

Vieme, že v každej dvojici je práve jeden vedúci ochotný podeliť sa s tým druhým, takže v každej dvojici sa nám stačí pozrieť na to, ktorý vedúci z danej dvojice to je. Potom stačí, aby tento vedúci doniesol jedlo, a pokryje tým aj druhého vedúceho z danej dvojice. Pre nepárne $n$ posledného nespárovaného vedúceho vieme vyriešiť zrejme tak, že aj on samotný ešte prinesie jedlo.

Týmto postupom vieme vybrať $\lceil n/2 \rceil$ vedúcich tak, že pokryjeme každého vedúceho, čo nám stačí na zisk 2 bodov. Časová aj pamäťová zložitosť tohto postupu je $O(n^2)$ kvôli načítavaniu vstupu, zvyšok už prebehne v čase lineárnom od $n$, keďže počet dvojíc je lineárny od $n$ a v každej sa nám stačí pozrieť len na to, z ktorého vrcholu do ktorého vedie hrana.

### Optimálne riešenie

Keď sa pokúsime vyššie uvedené riešenie vylepšiť a zaručiť tak napríklad pokrytie viacerých než dvoch vedúcich jedným vedúcim, zistíme, že to už takto ľahko zlepšiť nejde. Napríklad spomedzi trojice vedúcich už nemusíme vedieť vybrať jedného vedúceho, ktorým pokryjeme zvyšných dvoch.

Môže nám ale napadnúť pažravé (greedy) riešenie, v ktorom vždy vyberieme nejakého nepokrytého vedúceho, ktorý nám pokryje čo najviac ešte nepokrytých vedúcich. Takto vyberáme ďalších vedúcich s jedlom až dokým nepokryjeme úplne všetkých vedúcich. Toto riešenie znie prirodzene, no bez ďalšieho dôkazu nie je vôbec jasné, ako dobré v skutočnosti je.

Poďme sa bližšie pozrieť na jeden krok tohto postupu. Nech je ešte nepokrytých $x$ vedúcich a my z nich vyberáme toho vedúceho, ktorý pokryje čo najviac nepokrytých vedúcich. Poďme spočítať, koľko vedúcich takto pokryjeme.

Tu nám pomôže grafová interpretácia tejto úlohy. Ľubovoľný nepokrytý vrchol totiž určite pokryje sám seba a okrem toho pokryje práve toľko vrcholov, koľko hrán z neho vedie k ešte nepokrytým vrcholom.

V podgrafe celého grafu tvorenom len nepokrytými vrcholmi a hranami medzi nimi pritom máme hranu medzi každou dvojicou rôznym vrcholom, a to orientovanú práve jedným smerom, takže hrán v tomto podgrafe je $\binom{x}{2} = \frac{1}{2}x(x-1)$. Každá táto hrana vychádza z nejakého z $x$ ešte nepokrytých vrcholov, takže z jedného vrcholu vychádza priemerne $\frac{1}{2}(x-1)$ hrán.

Keď potom vyberieme nepokrytý vrchol, z ktorého vychádza najviac hrán do nepokrytých vrcholov, tak z neho bude určite vychádzať aspoň priemerný počet hrán, teda aspoň $\frac{1}{2}(x-1)$. V opačnom prípade by totiž

z každého nepokrytého vrcholu vychádzalo menej než $\frac{1}{2}(x-1)$ hrán do nepokrytých vrcholov, a tak celkovo zo všetkých $x$ nepokrytých vrcholov by dokopy vychádzalo menej než $\frac{1}{2}x(x-1)$ hrán do ďalších nepokrytých vrcholov, teda hrán medzi nepokrytými vrcholmi by bolo menej než $\frac{1}{2}x(x-1)$, čo je ale spor s tým, že hrán medzi danými vrcholmi je práve $\frac{1}{2}x(x-1)$.

Vybraný vrchol tak pokryje sám seba a okrem toho ešte aspoň $\frac{1}{2}(x-1)$ ďalších nepokrytých vrcholov, teda v jednom kroku pokryjeme aspoň $1+\frac{1}{2}(x-1)=\frac{1}{2}(x+1)$ ešte nepokrytých vrcholov. Všimnime si, že v jednom kroku pokryjeme vždy viac než polovicu všetkých ešte nepokrytých vrcholov, takže počet potrebných krokov ako aj počet vybraných vrcholov bude približne logaritmický oproti celkovému počtu vrcholov.

Poďme presne určiť potrebnú veľkosť vybranej podmnožiny vedúcich pre $n \leq 1000$. Zrejme nám stačí určovať tento počet pre maximálny počet vedúcich, teda pre $n = 1000$. Po vybraní 0 vedúcich tak máme $x = 1000$ nepokrytých vedúcich a v každom ďalšom kroku nám počet nepokrytých vedúcich klesne z $x$ aspoň na $x-\frac{1}{2}(x+1)=\frac{1}{2}(x-1)$, pričom tento počet musí ostať celočíselný, takže v skutočnosti najväčší možný počet nepokrytých vedúcich po ďalšom kroku je $\lfloor\frac{1}{2}(x-1)\rfloor$. Maximálne počty nepokrytých vedúcich po $k$ vybraných vedúcich týmto postupom tak vieme postupne odsimulovať a zapísať do nasledovnej tabuľky:

| $k$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $x_{\max}$ | 1000 | 499 | 249 | 124 | 61 | 30 | 14 | 6 | 2 | 0 |

Vidíme, že pre $n \leq 1000$ skutočne stačí vybrať 9 vedúcich na pokrytie všetkých vedúcich, čo už stačí na zisk plného počtu bodov.

Zamyslime sa ešte nad tým, akú časovú a pamäťovú zložitosť má tento postup. Pri najjednoduchšom spôsobe implementácie prechádzame v každom kroku cez $n$ vrcholov a pre každý nepokrytý vrchol zrátame, do koľkých ďalších nepokrytých vrcholov z neho vedú hrany, prechodom cez všetkých $n$ možností na jeho suseda. Následne vrchol s maximálnym počtom susedov a aj všetkých jeho susedov označíme za pokrytých a pokračujeme ďalej. Takýchto krokov je $O(\log n)$, takže takto dostávame celkovo časovú zložitosť $O(n^2 \log n)$.

Toto vieme ešte mierne vylepšiť tak, že si v hešovacej tabuľke udržiavame všetky ešte nepokryté vrcholy a v každom kroku prechádzame len cez ne. Takto krok, v ktorom je nepokrytých vrcholov $x$, zaberie čas $O(x^2)$. Keďže zároveň v každom kroku klesne veľkosť nepokrytej množiny aspoň na polovicu, tak tento postup uskutoční rádovo $n^2+(n/2)^2+(n/4)^2+\cdots+(n/n)^2$ operácií. Túto hodnotu vieme zhora ohraničiť $n^2$ násobkom súčtom nekonečného geometrického radu s počiatočnou hodnotou 1 a kvocientom $1/4$, teda hodnotou $n^2 \cdot \frac{1}{1-\frac{1}{4}} = \frac{4}{3}n^2$, takže časová zložitosť je teraz dokonca $O(n^2)$.

Po hlbšom zamyslení vieme dokonca zistiť, že aj časová zložitosť pôvodného algoritmu bez hešovacej tabuľky je v skutočnosti $O(n^2)$. V každom kroku totiž lineárne prechádzame všetkými vrcholmi, ale ďalej počítame počty hrán len pre tie nepokryté vrcholy. Keďže je krokov $O(\log n)$, tak lineárne prechody všetkými vrcholmi vyjdú dokopy len na čas $O(n \log n)$ a prechody všetkými hranami, kde za prvý vrchol vezmeme vždy len nejaký nepokrytý vrchol, zaberie čas rádovo $n \cdot n + n/2 \cdot n + n/4 \cdot n + \cdots + n/n \cdot n$, čo je opäť zhora ohraničené súčtom nejakého nekonečného geometrického radu, ktorého hodnota je tentokrát $2n^2$, takže aj časová zložitosť pôvodného algoritmu je $O(n^2)$

Pamäťová zložitosť je $O(n^2)$, keďže si musíme pamätať informácie o všetkých hranách.

### Listing programu (Python)

```
n, s = map(int, input().split())
A = [list(map(int, input().split())) for _ in range(n)]

hungry_count = n
hungry = [True for _ in range(n)]
food_bringers = []

while hungry_count > 0:
    max_outdegree = -1
    best_choice = -1

    for i in range(n):
        if hungry[i]:
```

```
            outdegree = 0

            for j in range(n):
                if hungry[j] and A[i][j]:
                    outdegree += 1

            if outdegree > max_outdegree:
                max_outdegree = outdegree
                best_choice = i

    food_bringers.append(best_choice + 1)
    hungry[best_choice] = False
    hungry_count -= 1

    for j in range(n):
        if hungry[j] and A[best_choice][j]:
            hungry[j] = False
            hungry_count -= 1

print(len(food_bringers))
print(*food_bringers)
```

### Listing programu (C++)

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

int main() {
    cin.tie(0)->sync_with_stdio(0);

    int n; cin >> n; int s; cin >> s;

    vector<vector<bool>> A(n, vector<bool>(n));
    for(int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            bool b; cin >> b;
            A[i][j] = b;
        }
    }

    int hungry_count = n;
    vector<bool> hungry(n, true);
    vector<int> food_bringers;

    while (hungry_count > 0) {
        int max_outdegree = -1;
        int best_choice = -1;

        for (int i = 0; i < n; i++) {
            if (hungry[i]) {
                int outdegree = 0;

```

```
                for (int j = 0; j < n; j++) {
                    if (hungry[j] && A[i][j]) {
                        outdegree++;
                    }
                }

                if (outdegree > max_outdegree) {
                    max_outdegree = outdegree;
                    best_choice = i;
                }
            }
        }

        food_bringers.push_back(best_choice + 1);
        hungry[best_choice] = false;
        hungry_count--;

        for (int j = 0; j < n; j++) {
            if (hungry[j] && A[best_choice][j]) {
                hungry[j] = false;
                hungry_count--;
            }
        }
    }

    cout << food_bringers.size() << "\n";
    for (unsigned int i = 0; i < food_bringers.size(); i++) {
        if (i > 0) {cout << " ";}
        cout << food_bringers[i];
    }
    cout << "\n";
}
```

Miška, credits to MichalS

## 5. Už nie je viac jedla

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

Zabudnime zatiaľ na obmedzujúcu podmienku v zadaní, že z každého slova musíme do skratky vybrať aspoň jedno písmeno. Keďže medzery sa v skratke nenachádzajú, bez tohto obmedzenia sa nás úloha pýta, koľkými spôsobmi možno dostať skratku – reťazec $S$ ako podreťazec reťazca $T$. Táto úloha sa dá riešiť jednoducho dynamickým programovaním.

Dynamické programovanie označuje techniku, kedy si problém vieme rozdeliť na menšie, výsledky pre menšie problémy si zapamätať a potom ich rovno používať, nepočítať ich znova. Naše podproblémy budú zodpovedať tú istú otázku, ale len pre menšie prefixy $S$ a $T$. Konkrétne: označíme si $dp[i][j]$ počet spôsobov, ako vybrať z prvých $i$ znakov $T$ podpostupnosť zhodnú s prvými $j$ znakmi $S$. Tieto hodnoty budeme počítať od menších hodnôt $i$ a $j$ k väčším.

Ako vypočítať hodnotu $dp[i][j]$, ak poznáme hodnoty pre menšie $i$ a $j$ Chceme teda získať prefix $S$ dĺžky $j$ ako podpostupnosť prefixu $T$ dĺžky $i$.

Ak je $j = 0$, potom zrejme existuje práve jedna možnosť, ako vybrať prázdnu podpostupnosť z niečoho, a to práve tá, že nevyberiem žiadny znak.

Ak je $i = 0$ a $j \neq 0$, potom nie je možné z prázdneho prefixu $T$ neprázdnu podpostupnosť.

Ak sa $j$-ty znak $S$ zhoduje s $i$-tym znakom $T$ [3], potom tento znak mohol byť z $T$ vybraný a ostáva nám vybrať prvých $j-1$ znakov $S$ z o 1 kratšieho prefixu $T$. Už vieme, koľkými spôsobmi to ide: $dp[i-1][j-1]$. Inou možnosťou je, že sme tento znak z $T$ nevybrali (nejaký rovnaký sme v $T$ vybrať museli, ale skôr). To znamená, že sa pokúšame vybrať prvých $j$ znakov $S$ ako podpostupnosť z o 1 kratšieho prefixu $T$, čo ide $dp[i-1][j]$

[3] Znaky indexujeme od 1, pretože napr. prefix dĺžky 2 končí druhým znakom, ale ten je na indexe 1 v zero-based stringu.

spôsobmi. Každá možnosť konštrukcie podpostupnosti spadá do práve jedného z týchto prípadov, a teda bude započítaná práve raz.

Ak sa $j$-ty znak $S$ nezhoduje s $i$-tym znakom $T$, musia všetky spôsoby, ako vyrobiť žiadaný prefix, vyzerať tak, že používajú iba prvých $i-1$ znakov $T$.

Vďaka pamätaniu si predchádzajúcich hodnôt $dp$ vieme každú ďalšiu hodnotu $dp$ spočítať v konštantnom čase, teda celková časová zložitosť je úmerná počtu dvojíc $i$ a $j$, teda $O(|S| \cdot |T|)$.

Ako by sme vedeli upraviť tento algoritmus, aby počítal len možnosti, kedy z každého slova zoberieme aspoň jedno písmeno?

Parametre $i$, $j$ dynamického programovania vlastne zodpovedajú akýmsi stavom. Stav je napríklad, že mám prefix $S$ dĺžky $j$ a prefix $T$ dĺžky $i$. V našej pôvodnej úlohe tiež môžeme nájsť nejaké stavy. Konkrétne je stavom to, že máme prefix $S$ dĺžky $j$, prefix $T$ dĺžky $i$ a buď sme už z aktuálneho slova (z toho, ktorému patrí $i$-ty znak $T$) vybrali nejaké písmeno, alebo nie. Týmto sa stavový priestor zväčší iba dvojnásobne, takže to časovú zložitosť nepokazí.

Nech teda $dp[i][j][N]$ značí stav, kedy máme prefix $S$ dĺžky $j$ a prefix $T$ dĺžky $i$, pričom z aktuálneho slova sme ešte nepoužili žiaden znak a $dp[i][j][P]$ značí, že sme už nejaký znak slova použili. $N$ a $P$ sú len symboly pre lepšie pochopenie. V implementácii môžeme použiť hodnoty 0 a 1 alebo mať dve polia, $dp_N$ a $dp_P$.

Ak je $i$-ty znak $T$ písmeno, do stavu $dp[i][j][N]$ sa dá dostať jedine zo stavu $dp[i-1][j][N]$. Ak by sme totiž použili daný znak, dostali by sme sa do stavu s $P$. To však nenastalo, a teda máme o 1 kratší prefix.

Do stavu $dp[i][j][P]$ sme sa mohli dostať viacerými spôsobmi:

$i$-ty znak $T$ sme nevybrali, museli sme vybrať nejaký znak predtým, čiže sme prišli zo stavu $dp[i-1][j][P]$, $i$-ty znak $T$ sme vybrali (samozrejme, len ak je zhodný s $j$-tym znakom $S$) a už predtým sme vybrali nejaký iný znak slova, teda sme prišli zo stavu $dp[i-1][j-1][P]$, $i$-ty znak $T$ sme vybrali a bol to prvý vybraný znak slova, žiaden predošlý sme nevybrali, prišli sme zo stavu $dp[i-1][j-1][N]$. Ostáva poriešiť hranice medzi slovami. Tie sa jednoducho riešia vtedy, ak je aktuálny ($i$-ty) znak $T$ medzera. Vtedy začína nové slovo, takže počet spôsobov, ako mať vybraný nejaký znak nového (aktuálneho) slova ($dp[i][j][P]$) je nula. Počet spôsobov, ako nemať vybraný znak nového slova ($dp[i][j][N]$) je rovný $dp[i-1][j][P]$, teda hodnota pre o 1 kratší prefix $T$ s použitím nejakého znaku posledného slova. Všimnime si, že nezapočítavame $dp[i-1][j][N]$, pretože to zodpovedá situácii, kedy sme z predošlého slova nič nevybrali.

Počet možností pre skúmaný stav získame ako súčet počtov možností pre stavy, z ktorých sa do aktuálneho vieme dostať.

Odpoveďou je potom odpoveď pre celý reťazec $S$ a celý reťazec $T$, pričom aj z posledného slova sme museli nejaké písmeno vybrať. Je to teda hodnota $dp[|T|][|S|][P]$.

Časová zložitosť riešenia je stále $O(|S| \cdot |T|)$, pretože máme len dvakrát viac hodnôt ako v zjednodušenom prípade a každú hodnotu vieme vypočítať v konštantnom čase.

Pamäťová zložitosť je rovnaká, ale vieme dosiahnuť aj zložitosť $O(|S|)$, pretože pri výpočte hodnôt $dp[i]$ nám stačí poznať hodnoty $dp[i-1]$, teda stačí si pamätať dva stĺpce, predošlý a aktuálny, tabuľky $dp$.

### Listing programu (Python)

```
MOD = 10**9 + 7


def count(abbrev, text):
    m = len(abbrev)
    n = len(text)
    dp = [[[0, 0] for _ in range(m + 1)] for _ in range(n + 1)]
    dp[0][0][0] = 1
    for i in range(1, n + 1):
        if text[i - 1] == " ":
            for j in range(m + 1):
                dp[i][j][1] = 0
                dp[i][j][0] = dp[i - 1][j][1]
            continue
        for j in range(m + 1):
            dp[i][j][1] = dp[i - 1][j][1]
            if j > 0 and abbrev[j - 1] == text[i - 1]:
```

```
                dp[i][j][1] = (
                    dp[i][j][1] + dp[i - 1][j - 1][1] + dp[i - 1][j - 1][0]
                ) % MOD
            dp[i][j][0] = dp[i - 1][j][0]
    return dp[n][m][1]


if __name__ == "__main__":
    nwords = input()
    abbrev = input()
    text = input()
    print(count(abbrev, text))
```

### Listing programu (C++)

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

using namespace std;

int main()
{
        ios::sync_with_stdio(false);
        cin.tie(0);
        cout.tie(0);
        const long long mod = 1000000007;
        int w;
        cin >> w;
        string s;
        cin >> s;
        int n = s.size();
        vector<long long>dp(n+1, 0);
        dp[0] = 1;
        for (int i = 0; i < w; i++)
        {
                string t;
                cin >> t;
                vector<long long>newdp(n + 1, 0);
                for (int j = 0; j < t.size(); j++)
                {
                        for (int k = n; k > 0; k--)
                        {
                                if (s[k - 1] == t[j])
                                {
                                        newdp[k] += dp[k - 1]+newdp[k-1];
                                        newdp[k] %= mod;
                                }
                        }
                }
                dp = newdp;
        }
```

```
38          cout << dp[n] << endl;
39          return 0;
40  }
```

## 6. Žeby závod?

12 b za popis, 8 b za program

### Tip na úvod

Aby sme nemuseli špeciálne riešiť začiatok a koniec parkoviska, tak si vieme pridať 2 nabíjacie stanice na začiatok a koniec, ktoré zaberajú celú šírku parkoviska.

### Bruteforce

Vytoríme si celé parkovisko ako 2D pole boolovských hodnôt, na začiatku núl, a pre každú nabíjačku zmeníme hodnoty v poli, na všetkých políčkach kam zasahuje z nuly na jednotku.

Na zistenie odpovede prejdeme každý riadok a počítame akú najväčšiu medzeru sme videli.

Časová zložitosť takéhoto riešenia je $O(w(n+l))$, pretože musíme vytvoriť a prejsť pole veľkosti $w \times l$, a pridať $n$ nabíjacích staníc s maximálnou dĺžkou $w$.

Pamäťová zložitosť je $O(wl)$, pretože si musíme pamätať celé pole parkoviska.

### Lepší bruteforce

Namiesto vytvárania celého parkoviska si vieme vždy jednoducho zistiť $x$-ové súradnice všetkých nabíjacích staníc, ktoré zasahujú do riadka ktorý aktuálne spracovávame.

To vieme spraviť tak, že si pre každý riadok prejdeme všetky nabíjačky, a zistíme, či cez ne kolobežka na tomto riadku prejde, potom tieto nabíjačky prejdeme v zoradenom poradí podľa $x$, a nájdeme medzi ktorými dvomi je najväčšia medzera.

Časová zložitosť je $O(nw \log n)$, pre každý riadok prejdeme všetky nabíjačky, a tie cez ktoré kolobežka na danom riadku prejde, zoradíme podľa $x$. Všimnime si, že riešenie vieme spraviť aj tak, že si nabíjačky zoradíme podľa $x$ na začiatku, a tak vieme dostať zložitosť $O(nw + n \log n)$, respektíve $O(nw + l)$, ak použijeme counting sort. [1][4]

Pamäťová zložitosť je $O(n)$, pretože si musíme pamätať všetky nabíjačky, respektíve $O(n + l)$, ak chceme použiť counting sort.

### Zaujímavá myšlienka

Pozrime sa, ako vyzerá $i$-ty riadok v porovnaní s $(i+1)$-vým. Vidíme, že väčšinou vyzerajú celkom podobne a vačšina nabíjačiek zostáva na rovnakých miestach.

Vieme si všimnúť, že ak prechádzame všetky riadky postupne, tak sa nabíjačky dokopy zmenia iba $2n$-krát. Každá nabíjačka sa raz pridá medzi tie aktívne, teda tie, cez ktoré prejde kolobežka na aktuálnom riadku, a raz sa z nich odoberie. Tieto zmeny pre $i$-tu kolobežku nastanú na súradnichiach $y_{0,i}$ a $y_{1,i}$.

Otázkou zostáva, ako tieto zmeny rýchlo vykonávať, a následne rýchlo zisťovať veľkosť najväčšej medzery medzi aktívnymi nabíjačkami.

### Optimálne riešenie

Použijeme metódu zametania roviny priamkou. [2][5]

Najskôr si vytvoríme zoznam zmien - udalostí. Zmena nastáva na riadku, kde končí alebo začína nejaká nabíjačka. Tieto zmeny si utriedime primárne podľa riadku a sekundárne podľa toho, či sa jedná o začiatok alebo koniec nabíjačky.

Následne zametáme rovinu, teda prechádzame riadky, a držíme si množinu všetkých aktuálne aktívnych nabíjačiek, teda tých cez ktoré prejde kolobežka na aktuálnom riadku - $A$.

Postupujeme následovne: - Na novom riadku prejdeme všetky zmeny, kde sa nám pridá nejaká nabíjačka, a pridáme ich do množiny $A$. - Zistíme, aká je najväčšia medzera medzi 2 aktuálne aktívnymi nabíjačkami. - Vymažeme všetky nabíjačky, ktoré v tomto riadku končia z množiny $A$.

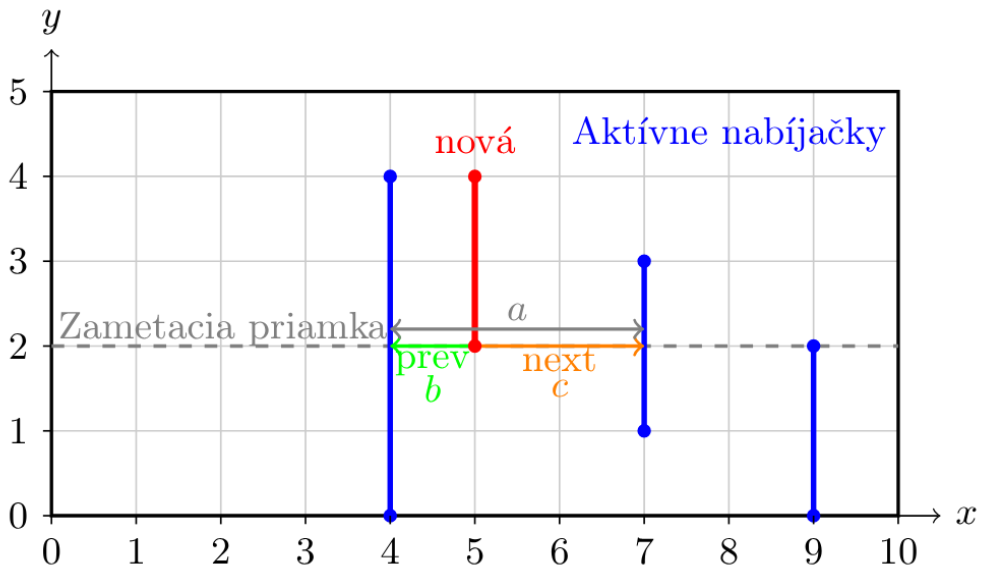
Na uchovávanie si množiny $A$ vieme použiť napríklad std::set v jazyku C++.

Poslednou otázkou zostáva, ako rýchlo sa dá zistiť aká je najväčšia medzera medzi 2 aktívnymi nabíjačkami. To urobíme tak, že si spravíme multimnožinu, teda množinu, ktorá podporuje opakovanie sa prvkov, všetkých

[4] https://sk.wikipedia.org/wiki/Counting_sort
[5] https://usaco.guide/plat/sweep-line?lang=cpp

veľkostí medzier medzi aktívnymi susediacimi nabíjačkami - $S$. Na to vieme použiť napríklad std::multiset v C++.

Ako robiť zmeny v $S$: - Pri pridaní nabíjačky medzi aktívne sa s medzerami stane to, že sa nejaká medzera, veľkosti $a$, rozdelí na 2 menšie, veľkosti $b, c$. Na to, aby sme vedeli ako vyzerala pôvodná medzera, a ako vyzerajú tie nové potrebujeme vedieť, kde sa nachádza najbližšia nabíjačka na ľavo a na pravo od tej, ktorú sme aktuálne pridali. To vieme spraviť pomocou metód prev/next od práve pridanej nabíjačky v množine $A$. Potom užiba stačí z $S$ odstrániť $a$ a pridať $b, c$. Je treba dávať si pozor, aby sme pri odstraňovaní nevymazali všetky výskyty medzier veľkosti $a$ z multimnožiny, ale iba jeden.




- Zistíme, aká je najväčšia medzera medzi 2 aktuálne aktívnymi nabíjačkami ako maximum multisetu, teda jeho posledný prvok, pretože multiset je Binárny Vyhľadávací Strom. [3][6]
- Pri odobraní nabíjačky sa stane opačná situácia, a dve medzery sa spoja do jednej. Vieme postupovať analogicky ako pri pridávaní, a len vymazať $b, c$ z $S$ a pridať $a$.

Časová zložitosť je $O((n+w)\log n)$, máme $n$ udalostí, ktoré musíme zoradiť $O(n \log n)$, pri každej udalosti hľadáme/vyberáme/pridávame do množiny $O(n \log n)$, a pre každý riadok musíme zistiť aktuálnu veľkosť najväčsej medzery z množiny $S$ - $O(w \log n)$, všimnime si, že taktiež vieme mať zložitosť $O(n \log n + w)$, ak si po každej zmene uchovávame aktuálnu najväčšiu dĺžku medzery, namiesto toho, aby sme ju pre každý riadok hľadali v $S$.

Pamäťová zložitosť je $O(n)$, pretože máme $O(n)$ udalostí, a množiny veľkosti $O(n)$.

### Riešenie pre iné jazyky

Ak nemáme usporiadanú množinu v štandardnej knižnici, ako to je v C++, tak máme dve možnosti ako si ju implementovať.

- Spravíme si vyvažovaný binárny vyhľadávací strom, na ktorom imlementujeme operácie nasledovný a predchádzajúci prvok [4][7]
- Implementujeme množinu pomocou dynamického intervalového stromu nad množinou možných hodnôt - v našom prípade čísla od 0 do $l$. Zisťovanie nasledovného a predchádzajúceho prvku vieme urobiť pomocou chodenia po intervalovom strome.[5][8]

Všimnime si že oboma spôsobmi vieme tak isto implementovať aj multimnožinu.

### Listing programu (C++)

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

[6] https://sk.wikipedia.org/wiki/Bin%C3%A1rny_vyh%C4%BEad%C3%A1vac%C3%AD_strom
[7] https://en.wikipedia.org/wiki/Binary_search_tree#Successor_and_predecessor
[8] https://usaco.guide/plat/segtree-ext?lang=cpp

```cpp
typedef long long ll;
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()
#define pb push_back
#define mp make_pair
#define F first
#define S second
ll MOD = 1e9 + 7;
const int INF = 1e9;
const ll LINF = 1e18;

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

    int n,l,w;
    cin>>n>>w>>l;

    vector<vector<int>> events_ytx;
    for (int i=0;i<n;i++){
        int x,a,b;
        cin>>x>>a>>b;

        events_ytx.push_back({a,0,x});
        events_ytx.push_back({b,1,x});
        // Vytváranie eventov
    }

    sort(all(events_ytx));
    events_ytx.push_back({INF,0,0});

    set<int> active={0,l};
    multiset<int> sizes={l};
    int c = 0;

    for (int y=0;y<=w;y++){
        while (events_ytx[c][0]<=y && events_ytx[c][1]==0){
            active.insert(events_ytx[c][2]);
            int x1 = events_ytx[c][2];
            int x0 = *prev(active.find(x1));
            int x2 = *next(active.find(x1));
            sizes.erase(sizes.find(x2-x0));
            sizes.insert(x1-x0);
            sizes.insert(x2-x1);
            c++;
        } // Pridanie všetkých začínajúcich nabíjačiek

        cout<<*prev(sizes.end())<<'\n';
        // Vypísanie najväčšej medzery

        while (events_ytx[c][0]<=y && events_ytx[c][1]==1){
            int x1 = events_ytx[c][2];
```

```
            int x0 = *prev(active.find(x1));
            int x2 = *next(active.find(x1));
            sizes.erase(sizes.find(x2-x1));
            sizes.erase(sizes.find(x1-x0));
            sizes.insert(x2-x0);
            active.erase(x1);
            c++;
        } // Odobranie všetkych končiacich nabíjačiek
    }
}
```

## 7. Jedenie v polceste

12 b za popis, 8 b za program

Táto úloha sa vlastne skladá z dvoch separátnych úloh:

- Pre každú túru potrebujeme zistiť, na ktorých miestach sa pre ňu môže nachádzať obed
- Následne musíme vybrať, na ktorých miestach sa obedy budú servírovať

Všimnite si, že ak máme len túry nepárnej dĺžky, druhý problém máme vyriešený automaticky – nemáme na výber. Zamerajme sa teda na prípad, že máme aj túry párnej dĺžky.

### Druhá časť

Poďme sa najskôr pozrieť na druhú časť úlohy. Ako sme si už povedali, pre túry nepárnej dĺžky je problém vyriešený. Povedzme, že tieto obedové miesta sú *povinné* vrcholy. Zoberme si teda túry nepárnej dĺžky. Povedzme, že vrchol *pokrýva* túru, ak by sa na ňom mohol servírovať obed pre túto túru. Povedzme, že množina vrcholov je *pokrývajúca* ak je každá túra pokrytá nejakým vrcholom.

Všimnime si, že ak nejakú túru pokrýva povinný vrchol, tak na ňu môžeme zabudnúť. Takto nám ostanú iba túry párnej dĺžky nepokryté povinnými vrcholmi a úloha sa zúži na to, nájsť najmenšiu pokrývajúcu množinu pre tieto túry.

Ďalej sa pozrime na to, čo sú to za dvojice vrcholov na ktorých môže byť servírovaný obed pre nejakú túru nepárnej dĺžky. Konkrétne, tieto vrcholy sú dva stredné vrcholy na ceste. Čiže musia byť spojené *hranou*. Teda sa úloha zúži na to, že máme podmnožinu hrán stromu, a chceme vybrať čo najmenšie množstvo vrcholov tak, aby každá z "obedových hrán" mala aspoň jeden koniec v množine.

Všimnime si, že hrany, ktoré nie sú v strede nejakej nepokrytej párnej túry nás nezaujímajú. Takže ich môžeme vyhodiť zo stromu. Strom sa nám takto rozpadne na viacero podstromov – to voláme les. Každý z týchto podstromov môžeme riešiť separátne.

Napríklad na dolnom obrázku máme strom s 10 vrcholmi, v ktorom chceme pokryť 6 hrán. Strom sa nám tak rozpadne na dva podstromy s aspoň jednou hranou (a dva izolované vrcholy).

### Nová úloha

Dostali sme teda nasledovnú novú úlohu: máme zadaný strom a chceme vybrať najmenšiu pokrývajúcu množinu vrcholov (takú, že každá hrana bude mať aspoň jeden pokrytý koniec).

Pokiaľ strom nemá žiadnu, alebo najviac jednu hranu, riešenie je jednoduché: zoberieme prázdnu množinu (ak tam nie je žiadna hrana), alebo ľubovoľný vrchol (ak je hrana presne jedna). Čo robiť, ak je strom väčší?

Samozrejme, dá sa to riešiť v exponciálnom čase skúšaním všetkých možností[9], našťastie, ide to aj oveľa rýchlejšie a jednoduchšie, a to: pažravo.

### Prvé pozorovanie

Predstavme si, že strom tvoria aspoň dve hrany. Vtedy vieme spraviť nasledovné pozorovanie: vždy existuje pokrývajúca množina optimálnej veľkosti, ktorá neobsahuje žiadny list (listy sú vrcholy z ktorých ide jediná hrana).

Prečo to platí? Zoberme si ľubovoľnú pokrývajúcu množinu minimálnej veľkosti a predstavme si, že sa v nej nachádza nejaký list. Tento list pokrýva jedinú hranu (keďže listy majú len jedného suseda). Ak vyberieme list z množiny a nahradíme ho jeho susedom, tak táto hrana bude stále pokrytá a množinu nezväčšíme (keďže odstránime jeden vrchol a pridáme jeden vrchol).

[9] a keby to nebol strom, ale ľubovoľný graf, nemali by sme moc inú možnost – vo všeobecnosti je tento problém NP-ťažký

Pokiaľ sú v strome aspoň dve hrany, list nikdy nesusedí s iným listom (schválne, skúste si to). Takže sme teda dostali množinu minimálnej veľkosti neobsahujúcu žiadne listy.

### Pažravé riešenie

Na základe tohto pozorovania získame jednoduchý algoritmus: pokiaľ má strom najviac jednu hranu, skončili sme. Inak pridáme do riešenia najskôr všetkých susedov listov (keďže hrany do listov musia byť pokryté a existuje optimálne riešenie bez listov). Vyhodíme všetky hrany, ktoré sme takto pokryli a dostaneme menší les (alebo menší strom). Spustíme algoritmus na každý zo zostávajúcich menších stromov.

Výsledné riešenie musí byť optimálne, keďže akonáhle neberieme listy (a vieme, že to môžeme urobiť), tak sú naše zvyšné voľby vynútené. Na obrázku dole môžeme vidieť dve iterácie tohto procesu.

Toto by sme mohli implementovať naivne, v kvadratickom čase. Ale stačí krátke zamyslenie, a zistíme, že nám v skutočnosti stačí jedno prehľadávanie do hĺbky. Pre každý vrchol si vrátime, či patrí do množiny. Ako toto vyhodnotíme? Ak sme v liste, povieme nie. Ak sme v nie-liste odpovieme áno, len ak musíme (máme nepokryté dieťa). Môžete si rozmyslieť, prečo to funguje.

S trochou zamyslenia takto vieme vyriešiť všetky podstromy, ktoré nám vznikli z prvej časti úlohy naraz, jedným prehľadávaním, a zaberie nám to lineárny čas.

### Prvá časť: ako nájsť stredy?

Tak sme si vyriešili druhú časť úlohy a už nám ostáva len zistiť, ktoré hrany alebo vrcholy sú to v strede ciest.

Táto časť úlohy zahŕňa dva podproblémy: chceme zistiť: ako ďaleko je od seba dvojica vrcholov, a ktorý vrchol je následne v polceste.

Mohli by sme to, samozrejme, riešiť naivne, v lineárnom čase na otázku, a získať takto 4 body.

Vo vzorovom riešení vieme tieto otázky riešiť v $O(\log n)$ čase na otázku (s $O(n \log n)$ predpočítaním). Použijeme na to dátovú štruktúru zvanú *najbližší spoločný predok* (inak známy ako LCA). Predstavme si, že si *zakoreníme strom* (vyberieme si jeden vrchol ako koreň a strom si naň zavesíme). Predkovia vrchola sú vrcholy na ceste z neho do koreňa. Všimnime si, že pri každej ceste medzi dvoma vrcholmi ideme najskôr hore do *najbližšieho spoločného predka* a potom dole. Takže na zistenie cesty chceme vedieť, ktorý je ten najbližší spoločný predok. A stred, resp. stredy cesty vieme zistiť tak, že z jedného z vrcholov ideme o pol-dĺžku cesty hore.

Ako toto zistíme rýchlo? V skratke (ak si to chcete prečítať detailnejšie, návod je v kuchárke[10]) si pre každý vrchol zistíme, aký vrchol je jeho priamy predok, vrchol o dva vyššie, o štyri vyššie, o osem vyššie (až tak ďalej, $O(\log n)$ informácií). Najbližšieho spoločného predka vieme binárne vyhľadávať v $O(\log n)$ čase (ak je $i$-ty predok rovnaký, potom aj $(i+1)$-ty bude rovnaký). A vrchol ktorý je o $l$ levelov vyššie vieme získať tak, že sa na číslo $l$ pozrieme v binárke a správne vystavame skoky.

Celé riešenie má dokopy časovú zložitosť $O((n+m)\log n)$, a pamäťovú zložitosť $O(n \log n)$, pričom si všimnime, že druhá časť úlohy nám zložitosť nemení.

### Listing programu (C++)

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

using namespace std;

#define FOR(i,n)        for(int i=0;i<(int)n;i++)
const int infinity = 2000000999 / 2;

void dfs_root(int v, vector<int> &F, int h, vector<int> &Z,
              vector<int> &euler, vector<vector<int> > &hrany) {
    Z[v] = (euler.size());
    euler.push_back(h);
    for (int w : hrany[v]) {
        if (F[v] == w) continue;
        F[w] = v;
        dfs_root(w, F, h + 1, Z, euler, hrany);
```

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

```
        euler.push_back(h);
    }
}

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

    rmq(int N, vector<int> &A) {
        n = N;
        minis.push_back(A);
        int len = 1;
        int it = 0;
        while (len * 2 <= n) {
            minis.push_back(vector<int>());
            for(int i = 0; i < n; i ++) {
                minis[it + 1].push_back(
                    min(minis[it][i],
                        (i + len < n ? minis[it][i + len] : infinity)));
            }
            len *= 2;
            it ++;
        }
    }

    int query(int z, int k) {
        int it = log2(k - z);
        return min(minis[it][z], minis[it][k - (1 << it)]);
    }
};

bool dfs_resolve(int v, vector<int> &F, vector<vector<int> > &hrany,
                 vector<bool> &taken, vector<bool> &me_or_parent, vector<int> &kde) {
    bool here = taken[v];
    for (int w : hrany[v]) {
        if (F[v] == w) continue;
        here |= dfs_resolve(w, F, hrany, taken, me_or_parent, kde);
    }

    if (here) {
        kde.push_back(v);
        return 0;
    }

    return me_or_parent[v]; // deffer up
}

int main() {
    cin.sync_with_stdio(false); cin.tie();
    cout.sync_with_stdio(false); cout.tie();
    int n, m;
    cin >> n >> m;

```

```
    vector<vector<int> > hrany(n);
    FOR(i, n - 1) {
        int a, b;
        cin >> a >> b;
        a --; b --;
        hrany[a].push_back(b);
        hrany[b].push_back(a);
    }

    vector<int> Z(n, -1), euler, F(n, -1);
    dfs_root(0, F, 0, Z, euler, hrany);
    vector<vector<int> > jumps(1, F);
    jumps[0][0] = 0;

    int max_h = 0;
    FOR(i, n) max_h = max(max_h, euler[Z[i]]);

    int len = 1;
    int it = 0;
    while (len < max_h) {
        jumps.push_back(vector<int>(n, -1));
        FOR(i, n) {
            jumps[it + 1][i] = jumps[it][jumps[it][i]];
        }
        it ++;
        len *= 2;
    }


    rmq R(euler.size(), euler);
    vector<bool> me_or_parent(n, 0), taken(n, 0);

    FOR(i, m) {
        int a, b;
        cin >> a >> b;
        a --; b --;
        int lca_h = R.query(min(Z[a], Z[b]), max(Z[a], Z[b]) + 1);

        int dist = (euler[Z[a]] - lca_h) + (euler[Z[b]] - lca_h) + 1;

        int da = euler[Z[a]] - lca_h, db = euler[Z[b]] - lca_h;

        if (da < db) {
            swap(da, db);
            swap(a, b);
        }


        int h_dist = (dist - 1) / 2;

        if (dist % 2) { // even case
            int in_half = a;
            int j = 0;
```

```
            while (1 << j <= h_dist) {
                if ((1 << j) & h_dist) {
                    in_half = jumps[j][in_half];
                }
                j ++;
            }
            taken[in_half] = true;
        }
        else { // odd case
            int in_half = a;
            int j = 0;
            while (1 << j <= h_dist) {
                if ((1 << j) & h_dist) {
                    in_half = jumps[j][in_half];
                }
                j ++;
            }
            me_or_parent[in_half] = true;
        }
    }


    vector<int> res;
    if (dfs_resolve(0, F, hrany, taken, me_or_parent, res)) {
        res.push_back(0);
    }

    cout << res.size() << "\n";
    FOR(i, res.size()) cout << (i ? " " : "") << res[i] + 1;
    cout << endl;
}
```

Viktor

## 8. Escargot

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

Keďže tvar žiadnej z dvoch množín sa nemení ani neotáča, vždy vieme stav mriežky popísať jedinou dvojicou celých čísel, ktorá označuje, ako sú obe množiny voči sebe posunuté (vzhľadom ku pôvodnému stavu $0, 0$). Konkrétne označme dvojicou parametrov $S[i, j]$ taký stav plochy, ktorý dostaneme, keď z počiatočného rozloženia množinu `@` posunieme o $i$ políčok dole a $j$ políčok doprava (ignorujúc možné kolízie, či už počas presunu, alebo vo finálnom umiestnení). Uvedomme si, že ťahy, ktoré v úlohe vykonávame, sú v takejto reprezentácií vlastne vždy len zmenou jedného z dvoch parametrov o 1, keďže buď posunieme o políčko množinu `@`, alebo množinu `#`, čo je to isté, ako posunúť množinu `@` v opačnom smere. Našou úlohou je teda, keď si všetky tieto stavy predstavíme ako nekonečnú mriežku, z počiatočného stavu $S[0, 0]$ nájsť cestu po susedných stavoch takú, že množiny oddelíme, teda dosiahneme stav $S[i, j]$, kde buď $|i| \geq m$ alebo $|j| \geq n$ (všetky takéto stavy nazývajme *vonkajšie*, a zvyšných $(2m - 1)(2n - 1)$ stavov nazývajme *vnútorné*). Problémom je, že niektoré z týchto stavov môžu byť nepriechodné, pokiaľ by daný posun množín spôsobil kolíziu na nejakom políčku (nikdy nie stav $S[0, 0]$, označujúci počiatočnú pozíciu, ani žiaden z vonkajších stavov). Pokiaľ by sme teda pre všetky vnútorné stavy vypočítali, či sú priechodné, alebo nie, potom nám už stačí jednoducho akýmkoľvek prehľadávacím algoritmom nájsť cestu z $S[0, 0]$ do akéhokoľvek vonkajšieho stavu, čo sa stihne v rozumnom čase $O(mn)$. Zároveň máme aj garantované, že ak takáto cesta existuje, očividne nebude dlhšia než celkový počet vnútorných stavov, čo sa určite zmestí do limitu v zadaní $10^7$. Až túto cestu nájdeme, skonvertujeme ju do požadovaného formátu jednoducho tak, že každý pohyb bude posun množinou `@`, a potom každý pohyb medzi stavmi v nejakom smere zodpovedá posunu tejto množiny v tom istom smere.

Teraz sa črtá hlavný problém riešenia: ako pre všetky vnútorné stavy určiť, či sú priechodné alebo nie?

Najjednoduchšie riešenie je zostrojiť pôvodne všade priechodnú mriežku, a potom pre každú dvojicu políčok na vstupe, kde jedno je `@` a druhé `#`, vypočítať, pre aký posun sa dostanú na tú istú pozíciu, a zaznačiť, že

zodpovedajúce políčko v mriežke je nepriechodné (keď `#` je v $i_a$-tom riadku a $j_a$-tom stlpci, a `@` je v $i_b$-tom riadku a $j_b$-tom stĺpci, potom vzniká kolízia medzi nimi v stave $S[i_a - i_b, j_a - j_b]$). Tento postup ale môže trvať až $O(m^2n^2)$, čo nie je dosť rýchle…

Pri zostrojení asymptoticky lepšieho riešenia nám pomôže algoritmus spomenutý v zadaní: fast fourier transform. Jedná sa o algoritmus, ktorý vynásobí dva polynómy dĺžky $n$ v čase $O(n \log n)$ (do detailov v tomto vzorovom riešení nepôjdeme). Toto možno s úlohou na prvý pohľad vôbec nesúvisí, v skutočnosti sa to ale dá na riešenie využiť. Aby sme pochopili ako, predstavme si prípad, že $m = 1$. V tomto prípade je vstupom jediný riadok. Interpretujme si tento riadok ako dvojicu vektorov boolov o dĺžke $n$: platí, že $a[i]$ značí, či je na $i$-tej pozícií znak `#`, a $b[i]$ značí, či je na nej `@`. V tomto prípade platí, že $S[0, i]$ je nepriechodný práve vtedy, keď existuje taká pozícia $j$, že $a[j] = 1$ aj $b[j - i] = 1$ (pre indexy mimo poľa predpokladáme, že všetky sú nulové, pretože tam nie je žiadne políčko danej množiny). Už to vidíte? Nie? Upravme trochu vstup: zostrojme pole $b'$ otočením poľa $b$, aby so stúpajúcim $i$ $b'[i]$ označovalo pozície bližšie a bližšie k začiatku vstupu. Zároveň, keďže $S$ má pozíciu $S[0, 0]$ v strede, a nie na začiatku, zadefinujme "poriadnejšie" pole $S'$ tak, že pre $0 \leq i \leq 2n - 2$ $S'[i] = S[0, i - n + 1]$. Teraz si všimnime, že $S'[i]$ je nepriechodný práve vtedy, keď existuje taká pozícia $j$, že $a[j] = 1$ aj $b'[j - i] = 1$, teda inak povedané, existujú dve celé čísla $j_1, j_2$, ktorých súčtom je $i$, a na týchto pozíciach v poliach $a$ a $b'$ sú jednotky. To už sa nám určite podobá na násobenie polynómov - jednoducho si interpretujeme $a$ a $b'$ ako polynómy, kde každý koeficient je buď 0 alebo 1, a potom ich súčinom získame polynóm, ktorý má nenulový koeficient práve pri takom exponente $i$, že existujú nenulové koeficienty v $a$ a $b$, ktoré sú pri exponentoch, ktoré sa sčítajú na $i$. Tento výsledný súčin teda je žiadaným $S'$.

Otázkou je teraz, ako tento postup rozšíriť na vstup pozostávajúci z viacerých riadkov. Chceli by sme zachovať vlastnosť, že dvojice indexov, ktoré majú rovnaký súčet, korešpondujú k vždy rovnako posunutým pozíciám na vstupe (keďže po vynásobení polynómov prispejú tieto dvojice k tomu istému posunu). Prvým očividným prístupom je jednoducho do polynómu $a$ vložiť zaradom všetky pozície na vstupe - najprv prvý riadok, potom druhý, a tak ďalej, a do $b'$ to isté, len reverznuté. Potom naozaj pre pozície, ktorých súčet indexov je $i$, ak $i < n$, platí, že sú všetky rovnako posunuté (prvá pozícia je v prvom riadku na mieste $j_1$, druhá v poslednom na mieste $n-1-j_2$, potom je vždy ich horizontálny rozdiel rovný $n-1-j_2-j_1 = n-1-i$). Len čo sa ale $i$ rovná $n$, nastane problém. Súčet indexov $i$ totiž začnú mať aj políčka, ktoré sú v druhom a poslednom riadku (prvé políčko druhého riadku má index $n$, posledné posledného má index 0), ale aj políčka, ktoré sú stále len v okrajových riadkoch (posledné políčko prvého má index $n - 1$, predposledné posledného má 1). A tieto očividne nemajú rovnaký rozdiel, líšia sa už vo vertikálnom rozdieli. Našim problémom teda je, že nejaký súčet indexov sa dá vyskladať z políčok v rôzne vzdialených riadkoch. Pomôže nám akoby si vytvoriť "nárazníkové zóny" medzi riadkami. Čo to znamená? Zostrojme $a$ a $b'$ trochu inak: Na pozíciach $a[0]$ až $a[n - 1]$ je uložená informácia o prvom riadku vstupu, na $a[2n - 1]$ až $a[3n - 2]$ o druhom, a tak ďalej, vždy o $i$-tom riadku je informácia na pozíciách $a[i(2n-1)]$ až $a[i(2n-1)+n-1]$, štandardne zľava doprava. Podobne na pozíciach $b'[0]$ až $b'[n-1]$ je informácia o poslednom riadku, zprava doľava, a na $b'[i(2n-1)]$ až $b'[i(2n-1)+n-1]$ je informácia o $i$-tom riadku od konca. Dĺžka oboch týchto polí je $2mn-m-n+1$. Zvyšok týchto polí nech je prázdny - teda akoby označovali políčka, ktoré sú voľné (v polynóme tam bude 0). Čo teraz dostaneme vynásobením týchto polí ako polynómov? Označme ich súčin $c$, toto pole bude veľkosti $4mn - 2m - 2n + 1 = (2m - 1)(2n - 1)$. Toto je rovnako veľa, ako je počet všetkých vnútorných stavov. Ukážeme, že toto nie je náhoda, a $c$ presne popisuje priechodnosť všetkých vnútorných stavov, konkrétne $c[y(2n - 1) + x]$ je nenulový práve vtedy, keď je $S[y - m + 1, x - n + 1]$ nepriechodný.

Pozrime sa na ľubovoľnú dvojicu políčok, ktorá môže spôsobiť kolíziu. Konkrétne nech je v riadku $y$ a stĺpci $x$ vstupu znak `#`, a v riadku $y'$ a stĺpci $x'$ vstupu znak `@`. Potom je korešpondujúca hodnota 1 na pozíciach $a[y(2n-1)+x]$ a $b'[(m-1-y')(2n-1)+n-1-x']$. To znamená, že po vynásobení táto dvojica zaručí nenulovosť v koeficiente na indexe rovnom súčtu dvoch pôvodných: $c[y(2n-1)+x+(m-1-y')(2n-1)+n-1-x'] = c[(m-1+y-y')(2n-1)+(n-1+x-x')]$. Vidíme, že každá dvojica symbolov kolidujúca pri posune $S[\Delta_y, \Delta_x]$ spôsobuje nenulovosť hodnoty $c[(m-1+\Delta_y)(2n-1)+(n-1+\Delta_x)]$, čo znamená, že sme skonštruovali presne rovnakú konštrukciu, ako v pomalom riešení, ale namiesto prechádzania každej dvojice manuálne sme museli iba vynásobiť dva veľké polynómy.

Vidíme, že vhodným transformovaním vstupu na dvojicu polynómov o veľkosti $2mn - m - n + 1$ a ich vynásobením sme dostali tabuľku veľkosti $(2m - 1)$ krát $(2n - 1)$ (pozor! Tu sa dá urobiť chyba - FFT vytvorí polynóm rovnakej veľkosti ako tie vstupné - teda najprv musíme vstupné polynómy predĺžiť na túto dvojnásobnú veľkosť a novopridanú polovicu vyplniť nulami), v ktorej nám stačí pre vyriešenie úlohy jednoducho nájsť cestu spájajúcu stred s okrajom, prípadne rozhodnúť, že neexistuje.

Toto sa stíha v čase $O((2mn - m - n + 1)\log(2mn - m - n + 1) + (2m - 1)(2n - 1)) = O(mn \log(mn))$ (ale FFT má pomerne vysokú konštantu, takže v praxi to až tak rýchle nebude… no pre vstup veľkosti danej v zadaní je rozdiel už dosť podstatný). Pamäťová zložitosť je $O(mn)$, pretože FFT stačí lineárny priestor, a potom si pamätáme tabuľku priechodnosti posunov, ktorú prehľadávať dokážeme bez potreby väčšieho priestoru.

### Listing programu (C++)

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

vector<complex<double>> f(vector<complex<double>> A, vector<complex<double>>& omega){
    ll n = A.size();
    if(n == 1){
        return A;
    }

    vector<complex<double>> Ae(n/2);
    vector<complex<double>> Ao(n/2);
    for(ll i = 0; i < n; i++){
        if(i%2){
            Ao[i/2] = A[i];
        }else{
            Ae[i/2] = A[i];
        }
    }

    vector<complex<double>> fAo = f(Ao,omega);
    vector<complex<double>> fAe = f(Ae,omega);
    for(ll i = 0; i < n/2; i++){
        fAo.push_back(fAo[i]);
        fAe.push_back(fAe[i]);
    }
    vector<complex<double>> result(n);
    ll jump = omega.size() / n;
    for(ll i = 0; i < n; i++){
        result[i] = fAe[i] + fAo[i]*omega[i*jump];
    }
    return result;
}

vector<complex<double>> inv_f(vector<complex<double>> A, vector<complex<double>>& omega){
    ll n = A.size();
    if(n == 1){
        return A;
    }

    ll jump = omega.size() / n;

    vector<complex<double>> average(n/2);
    vector<complex<double>> radius(n/2);
    for(ll i = 0; i < n/2; i++){
        average[i] = (A[i] + A[i+n/2])*0.5;
        radius[i] = A[i]-average[i];
        radius[i] /= omega[i*jump];
    }
    vector<complex<double>> even = inv_f(average,omega);
```

```
52      vector<complex<double>> odd = inv_f(radius,omega);
53      vector<complex<double>> result(n);
54      for(ll i = 0; i < n; i++){
55          if(i%2){
56              result[i] = odd[i/2];
57          }else{
58              result[i] = even[i/2];
59          }
60      }
61
62      return result;
63
64
65  }
66
67  bool dfs(vector<vector<bool>>& overlaps, vector<vector<bool>>& visited, ll row, ll col,
    ↪  vector<char>& way){
68      ll m = overlaps.size();
69      ll n = overlaps[0].size();
70      if(row < 0  col < 0  row >= m  col >= n){
71          return true;
72      }
73
74      if(visited[row][col]  (!overlaps[row][col])){
75          return false;
76      }
77
78      visited[row][col] = true;
79
80      way.push_back('W');
81      if(dfs(overlaps,visited,row,col-1,way)){
82          return true;
83      }
84      way.pop_back();
85
86      way.push_back('E');
87      if(dfs(overlaps,visited,row,col+1,way)){
88          return true;
89      }
90      way.pop_back();
91
92      way.push_back('S');
93      if(dfs(overlaps,visited,row+1,col,way)){
94          return true;
95      }
96      way.pop_back();
97
98      way.push_back('N');
99      if(dfs(overlaps,visited,row-1,col,way)){
100         return true;
101     }
102     way.pop_back();
103
```

```
104     return false;
105 
106 }
107 
108 int main() {
109     cin.tie(0)->sync_with_stdio(0);
110 
111     ll m,n;
112     cin >> m >> n;
113     vector<string> vstup(m);
114     for(ll i = 0; i < m; i++){
115         cin >> vstup[i];
116     }
117 
118     //takto velke budu polynomy co nasobime
119     ll poly_velkost = m*n*4 - 2*m - 2*n + 1;
120     //actually chcu mat velkost ktora je mocninou 2
121     ll power = 2;
122     while(power < poly_velkost){
123         power *= 2;
124     }
125 
126     vector<complex<double>> A(power,0);
127     vector<complex<double>> B(power,0);
128 
129     for(ll i = 0; i < m; i++){
130         for(ll j = 0; j < n; j++){
131             char znak = vstup[i][j];
132             if(znak == '#'){
133                 A[i*(2*n-1) + j] = 1;
134             }else if(znak == '@'){
135                 B[(m-1-i)*(2*n-1) + (n-1-j)] = 1;
136             }
137         }
138     }
139 
140     //predpocitame odmocniny 1
141     vector<complex<double>> omega(power);
142     for(ll i = 0; i < power; i++){
143         omega[i] = polar(1.0 , 3.14159265358979323846264338327950288419716939937510582097 * 2 * i /
↪   power);
144     }
145 
146     //skonvertuj
147     vector<complex<double>> Af = f(A,omega);
148     vector<complex<double>> Bf = f(B,omega);
149 
150     //vynasob po prvkoch
151     vector<complex<double>> Cf(power);
152     for(ll i = 0; i < power; i++){
153         Cf[i] = Af[i] * Bf[i];
154     }
155     //inverzne
```

```
    vector<complex<double>> C = inv_f(Cf,omega);
    //tabulka pozicii
    vector<vector<bool>> overlaps(m*2-1);
    vector<vector<bool>> visited(m*2-1);
    for(ll i = 0; i < 2*m-1; i++){
        for(ll j = 0; j < 2*n-1; j++){
            overlaps[i].push_back( abs(C[i*(2*n-1) + j]) < 0.1 );
            visited[i].push_back(false);
        }
    }

    vector<char> way;
    bool result = dfs(overlaps,visited,m-1,n-1,way);

    if(result){
        cout << way.size() << "\n";
        for(ll i = 0; i < way.size(); i++){
            cout << "@ " << way[i] << "\n";
        }
    }else{
        cout << "-1\n";
    }



}
```