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

# Vzorové riešenia 1. kola letnej časti

## 1. Národné toaletné centrum

12 b za popis, 8 b za program

### Čo bolo našou úlohou

Našou úlohou bolo prejsť každý riadok vstupu a vypísať z neho $k$-te písmeno. Bolo treba ignorovať medzery, a ak bol riadok bez medzier kratší ako naše $k$, vypísať prázdny riadok.

### Ako sme to urobili

Mohli sme zvoliť viacero postupov. Tým najprimitívnejším bolo prechádzať každý znak z riadku, zapamätať si koľko nemedzerových znakov sme už prešli, a keď sa tento počet rovnal $k$, vypísať aktuálny znak. Ak sme sa dostali na koniec riadku a naše počítadlo bolo menšie ako $k$, vypíšeme prázdny riadok. Mohli sme taktiež využiť vstavané funkcie na nahradenie znakov medzery v riadku "ničím" a skontrolovať, či je riadok dlhší ako $k$. Ak áno vypísať jeho $k$-ty znak.

Časová zložitosť riešenia je $O(n \cdot k)$. Pamäťová zložitosť je priamo úmerná dĺžke riadku($l$), teda $O(l)$.

### Listing programu (Python)

```
numberOfLines,k=map(int, input().split())
out = ''
for i in range(numberOfLines):
    data = input()
    data = data.replace(' ', '')
    if len(data)>k:
        print(data[k])
    else: print()
```

### Listing programu (C++)

```
//
// Created by filip on 5/12/22.
//
#include <cstdio>
#include <string>
#include <iostream>

using namespace std;

void kCharacter() {
    string input;
    int numberOfLines = 0;
    int k = 0;
    cin >> numberOfLines >> k;
    getline(cin, input);
    for (int j = 0; j < numberOfLines; j++) {
        getline(cin, input);
        int c = -1;
        for (int i = 0; i < input.length(); ++i) {
            if (input[i] != ' ') {
```

```
                c++;
            }
            if (c == k) {
                cout << input[i];
                break;
            }
        }
            cout << endl;

    }

}

int main() {
    kCharacter();
}
```

## 2. Architektonický problém

12 b za popis, 8 b za program

Na vyriešenie tejto úlohy nebolo treba vymýšlať žiadne komplikované postupy. Stačí nám iba postupne hľadať záchodové misy, kontrolovať či sa v ich vnútri nič nenachádza a spočítať ich.

To urobíme nasledovne: Budeme prechádzať načítaný nákres po riadkoch a hľadať dvojice znakov `O` medzi ktorými sú iba znaky `.` (bodka), a žiadne iné . V prípade, že k niektorému znaku `O` nenájdeme jeho dvojicu, záchod nie je správne zakreslený a teda aj celý nákres je nesprávny.

Každá nájdená dvojica znakov `O` je potenciálne záchodovou misou, ktorej veľkosť $k$ vieme určiť zo vzdialenosti týchto znakov. Vďaka tomu vieme vypočítať, kde sa má nachádzať spodný vrchol trojuholníka reprezentujúceho misu. Následne si už len overíme, či sa na jeho mieste naozaj nachádza znak `O`.

Predtým, ako vyhlásime že je táto záchodová misa zakreslená správne, musíme ešte skontrolovať, či sa v jej vnútri naozaj nič nenachádza. Stačí nám jednoduchým cyklom skontrolovať plochu trojuholníka na nasledujúcich $k$ riadkoch. Ak nájdeme iný znak ako `.`, záchod sa s niečím kryje a nákres je nesprávny.

Aby sme sa pri hľadaní ďalších mís nepomýlili, preznačíme si spodný vrchol na nejaký iný znak (napríklad `#`), ktorý budeme pri hľadaní ľavého horného vrcholu ignorovať. Nebudeme ho ale ignorovať pri kontrole vnútra trojuholníka, vďaka čomu zistíme prípadné prekrývanie s už skontrolovaným záchodom.

### Listing programu (Python)

```
w, h = [int(x) for x in input().split()]
m = [list(input()) for _ in range(h)]

def check(r, c, s):
    global m
    if s <= 1:
        return False

    k = s // 2
    if m[r+k][c+k] != 'O':
        return False

    m[r][c] = '.'
    m[r][c+s] = '.'
    m[r+k][c+k] = '#'

    for y in range(r, r+k):
        for x in range(c+y, c+s-y):
            if m[y][x] != '.':
                return False

    return True
```

```
def abort():
    print('Zly nakres')
    exit()

start, cnt = None, 0
for r, row in enumerate(m):
    for c, char in enumerate(row):
        if start is None and char == 'O':
            start = c
        elif char == 'O':
            if check(r, start, c-start):
                cnt += 1
                start = None
            else:
                abort()

    if start is not None:
        abort()

print(cnt)
```

**Listing programu (C++)**

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

using namespace std;

typedef vector<vector<char>> vv;

bool check(vv &v, int r, int c, int s) {
    if (s <= 1) return false;

    int k = s / 2;
    if (v[r+k][c+k] != 'O') return false;

    v[r][c] = '.';
    v[r][c+s] = '.';
    v[r+k][c+k] = '#';

    for (int y = r; y < r+k; y++) {
        for (int x = c+y; x < c+s-y; x++) {
            if (v[y][x] != '.') return false;
        }
    }

    return true;
}

int q() {
    cout << "Zly nakres\n";
    return 0;
}

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

```
    int w, h;
    cin >> w >> h;
    vv v(h, vector<char>(w));

    for (int i = 0; i < h; i++) {
        for (int j = 0; j < w; j++) {
            cin >> v[i][j];
        }
    }

    int start = -1, cnt = 0;
    for (int i = 0; i < h; i++) {
        for (int j = 0; j < w; j++) {
            if (start == -1 && v[i][j] == 'O') {
                start = j;
            } else if (v[i][j] == 'O') {
                if (check(v, i, start, j-start)) {
                    cnt++;
                    start = -1;
                } else {
                    return q();
                }
            }
        }

        if (start != -1) {
            return q();
        }
    }

    cout << cnt << "\n";

    return 0;
}
```

Horné vrcholy trojuholníkov budeme hľadať v celom nákrese, čiže postupne prejdeme všetkých $r \cdot s$ políčok. Pri kontrole vnútier trojuholníkov skontrolujeme určite menej ako $r \cdot s$ políčok, keďže plocha pod záchodmi je určite menšia ako celý nákres, a pri prvom zistenom prekrývaní program končí. Výsledná časová zložitosť teda bude $O(rs)$.

Pamäťová zložitosť bude rovnako $O(rs)$ - pamätáme si celý nákres a zopár ďalších premenných.

## 3. Zahrabaná Kika

12 b za popis, 8 b za program

Riešenie tejto úlohy nie je vôbec zložité. Jediné, čo nám stačí spraviť, je prejsť celú miestnosť a zistiť pre každého vedúceho, kam vidí. Ukážeme si dva spôsoby ako to spraviť.

### Prechádzanie vedúcich

Budeme sa postupne posúvať po políčkach a pozerať sa na to, či na danom políčku stojí vedúci. V prípade, že stojí, označíme si všetky políčka na ktoré vidí nad a pod ním, vľavo a vpravo od neho ako "ohrozené". Teda všetky, ktoré nie sú už za nejakou prekážkou alebo iným vedúcim.

Keď toto spravíme pre všetkých vedúcich, tak nám na plániku ostanú neoznačené len miesta, na ktoré nevidí ani jeden vedúci. Stačí ich už len zrátať a dostaneme počet miest, na ktoré sa môže Kika bezpečne skryť.

### Časová a pamäťová zložitosť

Keďže rozmery nášho plániku zo vstupu, ktorý si musíme zapamätať, sú $m \times n$ a prechádzame iba tento plánik, tak aj časová zložitosť bude $O(m \cdot n)$. Teraz sa možno zdá, že to nevychádza, lebo vždy, keď prídeme na vedúceho, musíme označiť *všetky* políčka v stĺpci a riadku, na ktoré vidí a tých môže byť až $n + m - 1$.

Treba si však uvedomiť, že ak takto označíme napríklad veľkú časť jedného riadu, tak to znamená že na tom úseku nebol žiadny vedúci a teda ho následne prejdeme rýchlo. Dá sa teda povedať, že na každé políčko sa "pozeráme" najviac 5 krát. Raz keď kontrolujeme či je tam vedúci a potom najviac raz z každého smeru, podla toho či na to políčko z daného smeru vidí vedúci. Z toho vychádza časová zložitosť $O(5 \cdot m \cdot n)$, ale keďže 5 je len konštanta, tak $O(m \cdot n)$.

Pamäťová zložitosť je $O(m \cdot n)$, lebo si pamätáme iba plánik, ktorý v priebehu prepisujeme.

### Listing programu (Python)

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

pole = [list(input()) for i in range(n)]

for r in range(n):
    for s in range(m):
        if pole[r][s] != "V":
            continue
        # doprava
        for i in range(r + 1, n):
            if pole[i][s] == "#" or pole[i][s] == "V":
                break
            pole[i][s] = "S"
        # dolava
        for i in range(r - 1, -1, -1):
            if pole[i][s] == "#" or pole[i][s] == "V":
                break
            pole[i][s] = "S"
        # dole
        for j in range(s + 1, m):
            if pole[r][j] == "#" or pole[r][j] == "V":
                break
            pole[r][j] = "S"
        # hore
        for j in range(s - 1, -1, -1):
            if pole[r][j] == "#" or pole[r][j] == "V":
                break
            pole[r][j] = "S"

ans = 0
for i in range(n):
    for j in range(m):
        if pole[i][j] == ".":
            ans += 1

print(ans)
```

### Druhý spôsob - zametanie

Zametanie je prístup, ktorý spočíva v tom, že cez nejakú plochu (pole) prechádzame v jednom smere a ťaháme so sebou nejakú informáciu – ako keď metla ťahá špinu. My budeme zametať postupne vo všetkých 4 smeroch: zhora, zdola, zľava a sprava, aby sme si v plániku zaznačili, ktoré políčka sú videné nejakým vedúcim. Bližšie si popíšeme jeden z nich.

Ideme zametať zľava doprava. Môžeme si teda predstaviť že každý vedúci sa pozerá iba doprava a chceme označiť videné políčka. Po takomto zjednodušení môžeme prechádzať riadok po riadku a pamätať si len či sa niekto pozerá, alebo nie.

Nakoniec opäť prejdeme plánik a zistíme, ktoré políčka sú "neohrozené".

### Časová a pamäťová zložitosť

Na tomto riešení ešte jasnejšie vidieť, že 4 krát prejdeme celé pole a nerobíme pri tom nič časovo náročnejšie, ako že sa pozrieme na celý plánik, teda časová aj pamäťová zložitosť je opäť $O(m \cdot n)$.

### Listing programu (Python)

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

pole = [list(input()) for i in range(n)]

# z lava doprava
for i in range(n):
    pozera = False
    for j in range(m):
        if pole[i][j] == "#":
            pozera = False
        if pole[i][j] == ".":
            if pozera:
                pole[i][j] = "S"
        if pole[i][j] == "V":
            pozera = True

# z prava ľdoava
for i in range(n):
    pozera = False
    for j in range(m - 1, -1, -1):
        if pole[i][j] == "#":
            pozera = False
        if pole[i][j] == ".":
            if pozera:
                pole[i][j] = "S"
        if pole[i][j] == "V":
            pozera = True

# z hora dole
for j in range(m):
    pozera = False
    for i in range(n):
        if pole[i][j] == "#":
            pozera = False
        if pole[i][j] == ".":
            if pozera:
                pole[i][j] = "S"
        if pole[i][j] == "V":
            pozera = True

# z dola hore
for j in range(m):
    pozera = False
    for i in range(n - 1, -1, -1):
        if pole[i][j] == "#":
            pozera = False
        if pole[i][j] == ".":
            if pozera:
                pole[i][j] = "S"
        if pole[i][j] == "V":
            pozera = True
```

```
ans = 0
for i in range(n):
    for j in range(m):
        if pole[i][j] == ".":
            ans += 1

print(ans)
```

Gardener

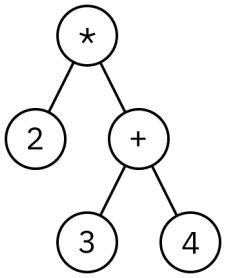
## 4. Ako sa to ráta?

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

Naše riešenie si rozdelíme na dve časti – v prvej sa pokúsime spracovať vstup a v druhej zistiť, či sa dá daný výraz vypočítať.

### Načítanie vstupu

Bežné aritmetické výrazy je vhodné v počítači reprezentovať tzv. aritmetickým stromom. Aritmetický strom je binárny strom, kde v listoch sa nachádzajú operandy (čísla) a v ostatných vrcholoch sa nachádzajú operácie. Aritmetický strom pre výraz `2 * (3 + 4)` by vyzeral takto:




Príjemná vlastnosť takéhoto stromu je, že z neho vieme pomerne jednoducho vyčítať výrazy v rôznych zápisoch. Nás ale bude zaujímať prefixový zápis, ktorý z takéhoto stromu dostaneme napríklad týmto jednoduchým rekurzívnym algoritmom: pre každý vrchol vypíšeme svoju hodnotu a následne rekurzívne pokračujeme do ľavého a pravého potomka. Môžeš sa doma zamyslieť, ako by sme z takéhoto stromu získali infixový a postfixový zápis.

Keď už toto vieme, stačí sa nám zamyslieť, ako vieme takýto strom zostrojiť zo zápisu.

### Spracovanie vstupu

Keď už máme aritmetický strom vytvorený, potrebujeme z neho zistiť, či ho Julka dokáže vyrátať na svojej kalkulačke. Pozrime sa najprv na výrazy zo zadania. Výraz `+ + 1 2 3` vieme prepísať na infixový zápis `(1 + 2) + 3` a výraz `+ + 1 2 + 3 4` zas na `(1 + 2) + (3 + 4)`. Môžeš si všimnúť, že sme v zápise naschvál nechali zátvorky, ktoré tam implicitne z prefixového zápisu sú. Takýto výraz na našej kalkulačke vypočítať nevieme – Julka má zakázané robiť úpravy výrazu na vstupe, takže ani tieto zátvorky nemôže dať preč. Z toho si už vieme všimnúť, že Julka nevie vypočítať také výrazy, v ktorých sú zátvorky na oboch stranách operácie.

Ako takéto situácie nájdeme v našom aritmetickom strome? Postupne prejdeme stromom (napríklad algoritmom DFS[1]) a ak nastane situácia, že máme aj v ľavom aj v pravom potomkovi operáciu, vieme, že sa daný výraz vypočítať nedá. Za takéto riešenie bolo možné získať dva body.

### Nie všetky operácie sú rovnaké

Na plný počet bodov bolo potrebné si ešte uvedomiť jednu vlastnosť tejto kalkulačky. Výraz `4 * (2 + 3)` vieme vypočítať. Ak ste niekedy vlastnili takúto najlacnejšiu formu kalkulačky, pravdepodobne viete, že zadaním `4 * 2 + 3 =` nedostaneme správny výsledok (lebo výrazy vyhodnocuje zľava doprava bez ohľadu na prioritu operácií). Bežne by ste si proste vypočítali najprv `2 + 3` a potom výsledok vynásobili číslom `4`. Výsledok `2 + 3` si budete musieť buď pamätať, alebo si vo výraze prehodíte poradie operandov: `(2 + 3) * 4`. Toto už viete bez problémov vypočítať zadaním `2 + 3 * 4 =`.

Toto ale nefunguje pre všetky operácie. Násobenie a súčet vieme takto upraviť, ale delenie a rozdiel nie. `4 - (2 + 3)` si už nevieme prehodiť tak, aby sa to dalo na takejto kalkulačke vypočítať. Vieme takýto výraz

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

upraviť ale to, ako sme si už spomenuli vyššie, nie je dovolené. Rovnako to platí aj pre delenie. Preto si musíme do nášho riešenia pridať ešte túto situáciu: ak je vo vrchole delenie/rozdiel a jeho pravý potomok je operácia, nepôjde to.

Časová zložitosť riešenia je $O(n)$, keďže vytvorenie stromu nám zaberie $O(n)$ času a prejdenie všetkých jeho vrcholov tiež $O(n)$. Pamäťová zložitosť je $O(n)$, keďže si pamätáme len vrcholy stromu, ktorých počet je rovný $n$.

**Listing programu (Python)**

```
n = int(input())
vstup = input().split()

ops = ['+', '-', '/', '*']

class Vrchol:
        def __init__(self, value=None):
                self.value = value
                self.L = None
                self.R = None

        def __str__(self):
                s = self.value
                if self.L:
                        s += f" {self.L}"
                if self.R:
                        s += f" {self.R}"
                return s

root = Vrchol(vstup[0])
root.L = Vrchol()
root.R = Vrchol()
stack = [root.R, root.L]

if root.value not in ops:
        print("ANO")
        raise SystemExit(0)

for v in vstup[1:]:
        m = stack.pop()
        m.value = v

        if v in ops:
                m.L = Vrchol()
                m.R = Vrchol()
                stack.append(m.R)
                stack.append(m.L)

c_on_right = root.value in ["-", "/"]
v = root
while v:
        op = None
        nxt = None
        if v.L:
                if v.L.value in ops:
                        op = v.L.value
                else:
                        nxt = v.L
```

```
        if v.R:
                if v.R.value in ops:
                        if op or c_on_right:
                                print("NIE")
                                raise SystemExit(0)
                        op = v.R.value
                else:
                        nxt = v.R

        v = nxt
        c_on_right = op in ["-", "/"]

print("ANO")
```

## 5. Cyklus v potrubí

12 b za popis, 8 b za program

Začnime tým, že si formálnejšie pomenujeme, čo presne je zadaním tejto úlohy. Na vstupe máme systém potrubí, ktorý spolu tvorí graf. Vieme, že potrubie je trochu iné ako bežne poznáme – voda cez potrubie vie tiecť iba jedným smerom. To znamená, že graf, ktorý reprezentuje potrubie je orientovaný. Niektoré potrubia sú špeciálne, keďže sa na ich stene nachádza KSP logo.

Našou úlohou je zistiť, či v potrubí existuje taká postupnosť potrubí (ktorá začína na spojení číslo 0), ktorá obsahuje logo KSP a zároveň po nej voda vie chodiť donekonečna. To znamená, že máme zistiť, či graf obsahuje cyklus, ktorý obsahuje hranu, ktorá má na sebe KSP logo. Tento cyklus ale musí byť dosiahnuteľný z vrcholu 0.

### Existuje cyklus, ktorý prechádza cez túto hranu?

Predstavme si, že máme zistiť, či v grafe existuje cyklus, ktorý obsahuje nejakú konkrétnu hranu. Povedzme, že táto hrana vedie z vrcholu $u$ do vrcholu $v$. To, čo nám stačí urobiť je, že spustíme nejaké naše obľúbené prehľadávanie z vrcholu $v$ (z konca hrany), a budeme zisťovať, že či sa počas prehľadávania dostaneme do vrcholu $u$, teda na začiatok hrany. Ak sa dostaneme, tak to znamená, že sa z vrcholu $u$ vieme dostať do vrcholu $v$, odkiaľ sa našou hranou vieme dostať zase do vrcholu $u$, a teda v grafe existuje cyklus, ktorý obsahuje túto hranu.

To je už takmer to, čo potrebujeme zistiť. Jediná drobnosť ktorá chýba, je overiť, že či sa vieme k tomuto cyklu dostať z vrcholu 0. To vieme tiež urobiť tak, že spustíme naše obľúbené prehľadávanie z vrcholu 0. Ak sa nám týmto prehľadávaním podarí dosiahnuť vrchol $u$ (alebo $v$), tak je celý cyklus dosiahnuteľný z vrcholu 0.

Týmto už máme všetky ingrediencie na prvé riešenie. Tým riešením je, že prejdeme cez všetky hrany s KSP logom. Pre každú z nich zistíme, či sa z koncového vrcholu tejto hrany vieme dostať do počiatočného. Ak áno, tak zistíme, že či je tento cyklus dosiahnuteľný z vrcholu 0. V prípade, že toto platí aspoň pre jednu hranu zo vstupu, tak je odpoveď `obsahuje`, inak je odpoveď `NEobsahuje`.

Keďže hrán s KSP logom môže byť najviac $m$, a pre každú z nich v najhoršom prípade prehľadáme celý graf (v zložitosti $O(n+m)$), tak celková zložitosť je najviac $O(m(n+m))$. Pamätať si okrem vstupného grafu nemusíme nič, takže pamäťová zložitosť zostáva $O(n+m)$.

### Detekcia cyklov s nejakými špeciálnymi hranami, to mi znie nejako povedome...

Ďalšie riešenie je založené na nasledujúcej myšlienke: V prípade, že ste niekedy počuli o Dijkstrovom alebo Floyd-Warshallovom algoritme, tak ste možno počuli o problémoch, ktoré nastávajú ak v grafe v ktorom zisťujeme vzdialenosti sú medzi vrcholmi záporné hrany (hrany so zápornou dĺžkou).

Konkrétnejšie, teraz nás bude zaujímať to, že občas sa nám môže stať, že v grafe existuje záporný cyklus (cyklus so zápornou dĺžkou). Ten môže spôsobiť, že najkratšia cesta medzi vrcholom $a$ a $b$ obsahuje tento cyklus, a teda dĺžka najkratšej cesty je rovná $-\infty$ (keďže najkratšia cesta obsahuje nekonečno prejdení záporného cyklu).

Takéto niečo je ale veľmi podobné tomu, čo máme zistiť. Chceli by sme vedieť, že či na ceste z vrcholu 0 do nejakého iného vrcholu existuje cyklus, ktorý obsahuje špeciálnu hranu (zápornú hranu, resp. hranu s KSP logom).

To, čo teda vieme urobiť je, že zo vstupného grafu urobíme ohodnotený graf. To urobíme tak, že hrany, ktoré neobsahujú KSP logo ohodnotíme nejakým malým kladným číslom – napríklad 1. Hrany, ktoré obsahujú KSP logo ohodnotíme $-\infty$, resp. dostatočne veľkým záporným číslom.

Potom spustíme nejaké prehľadávanie z vrcholu 0, a ak nájdeme nejaký záporný cyklus, tak môžeme vyhlásiť, že potrubie obsahuje cyklus s nápisom KSP.

Záporné cykly vieme napríklad detegovať Floyd-Warshallovovym algoritmom, a to tak, že najprv spustíme tento algoritmus, ktorý nájde najkratšie vzdialenosti medzi každou dvojicou vrcholov. Záporný cyklus sa nám na výsledných vzdialenostiach prejaví tak, že najkratšia vzdialenosť z vrcholu do samého seba je záporná. Následne zistíme, či je nejaký z vrcholov, cez ktorý prechádza záporný cyklus dosiahnuteľný z vrcholu 0.

Floyd-Warshallov algoritmus má časovú zložitosť $O(n^3)$ a následne ešte potrebujeme prejsť celý graf, aby sme zistili, ktoré vrcholy sú dosiahnuteľné z vrcholu 0. Výsledná časová zložitosť je teda rovná $O(n^3+n+m) = O(n^3)$. Pamätať si okrem celého grafu potrebujeme ešte maticu vzdialeností medzi vrcholmi (ktorá síce môže byť porovnateľne veľká s počtom hrán, ale pre korektnosť ju spomíname). To znamená, že pamäťová zložitosť je $O(n + m + n^2)$.

### Listing programu (Python)

```
import sys
from math import inf

def dfs(v):
    for u in range(len(G[v])):
        if G[v][u]!=inf:
            if not visited[u]:
                visited[u] = True
                dfs(u)
    return False

n, m = map(int, input().split())
G = [ [ inf for i in range(n) ] for j in range(n) ]
visited = [ False for _ in range(n) ]
for i in range(n):
    G[i][i] = 0;

for i in range(m):
    v, u, sympaticka = input().split()
    v = int(v)
    u = int(u)
    if sympaticka=='KSP':
        G[v][u] = -inf
    else:
        G[v][u] = 1

for k in range(n):
    for i in range(n):
        for j in range(n):
            G[i][j] = min(G[i][j], G[i][k]+G[k][j])


dfs(0)

for i in range(n):
    if G[i][i]<0:
        if visited[i]:
            print("obsahuje")
            sys.exit(0)
print("NEobsahuje")
```

### Vzorové riešenie

Ako na väčšinu takýchto úloh, vzorové riešenie je použiť prehľadávanie do hĺbky. Ako prvé si je treba

uvedomiť, ako pri prehľadávaní do hĺbky vieme detegovať cykly.

Najprv si uvedomme, že počas prehľadávania grafu môžu byť vrcholy v trochu rôznych "stavoch": nenavštívené, otvorené a zatvorené. Ako *nenavštívené* označujeme také vrcholy, že v nich sme ešte neboli (náš algoritmus ich nenavštívil), a ešte čakajú na svoje objavenie. *Otvorenými*, nazvime také vrcholy, že algoritmus aktuálne prehľadáva niektoré vrcholy v ich podstrome, teda že sme už tieto vrcholy objavili, ale ešte sme neprezreli všetky hrany, ktoré z nich vedú. Tretím druhom sú *uzavreté* vrcholy. To sú také, že sme už prezreli všetky hrany, ktoré z nich vedú a odišli sme z nich prehľadávať niečo iné (a už sa sem nikdy nevrátime).

Následne, podľa toho, do ktorého vrcholu smeruje hrana, DFS počas prehľadávania stromu nachádza 3 druhy hrán:

- dopredné/stromové. Tento druh hrán sa vyskytne vtedy, keď prejdením po tejto hrane prídeme do ešte neobjaveného vrcholu.
- spätné. Tento druh hrán sa vyskytne vtedy, keď hrana smeruje do otvoreného vrcholu. Môžete si rozmyslieť, že tento vrchol musí byť nejaký predok (v DFS strome) daného vrcholu..
- priečne/kolmé. Tento druh hrán smeruje do už uzatvorených vrcholov. Môžete si rozmyslieť, že tento vrchol nie je predok (v DFS strome) daného vrcholu.

V prípade, že by sa v grafe nevyskytovali žiadne cykly, tak by graf obsahoval iba dopredné a priečne hrany (opäť si premyslite prečo). Tá vec, ktorá nás teda pri hľadaní cyklov bude zaujímať je, že či v grafe existujú spätné hrany. V prípade, že v grafe existuje nejaká spätná hrana, tak v grafe existuje aj cyklus.

Otázka je, že ako zistiť, že či sa niekde na tomto cykle vyskytla hrana s nápisom KSP. Mohli by sme samozrejme nejako prejsť celú postupnosť hrán, a zisťovať, že či tam taká hrana nebola, ale ide to aj výrazne jednoduchšie. Stačí nám zaznamenávať si počet hrán s nápisom KSP, ktoré sme doteraz videli po ceste z vrcholu 0. V prípade, že nájdeme spätnú hranu, tak nám stačí zistiť, koľko hrán s nápisom sme videli po ceste do vrcholu, kde aktuálne sme, a koľko ich bolo vo vrchole, kam vedie spätná hrana. Ak sa tento počet líši (alebo má samotná spätná hrana nápis), tak sa niekde na cykle musela vyskytnúť spätná hrana.

Takéto riešenie okrem bežného DFS robí pre každý vrchol iba konštantne viac operácií, a teda celý algoritmus beží v zložitosti ako bežné DFS, teda $O(n+m)$. Pamätať si potrebujeme iba celý vstupný graf, a teda zložitosť je $O(n+m)$.

**Listing programu (Python)**

```
import sys
from math import inf

sys.setrecursionlimit(10000)

def dfs(v):
    doteraz_sympatickych = sympaticke[v]
    visited[v] = None
    for u, sympaticka in G[v]:
        if visited[u] is False:
            sympaticke[u] = doteraz_sympatickych+int(sympaticka)
            if dfs(u):
                return True
        elif visited[u]==None and sympaticke[u] < doteraz_sympatickych+int(
            ↪ sympaticka):
            return True
    visited[v] = True
    return False

n, m = map(int, input().split())
G = [ [] for j in range(n) ]
visited = [ False for _ in range(n) ]
sympaticke = [ 0 for _ in range(n) ]

for i in range(m):
```

```
    v, u, sympaticka = input().split()
    v = int(v)
    u = int(u)
    sympaticka = sympaticka=='KSP'
    G[v].append((u, sympaticka))

visited[0] = None
sympaticke[0] = 0
if dfs(0):
    print("obsahuje")
else:
    print("NEobsahuje")
```

**Listing programu (C++)**

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

# define NOT_VISITED 0
# define OPEN 1
# define CLOSED 2

using namespace std;

vector<vector<pair<long long, long long> > > G;
vector<long long> sympaticke;
vector<int> visited;

bool dfs(long long v){
    visited[v] = OPEN;
    long long doteraz_sympatickych = sympaticke[v];
    for(auto & i : G[v]){
        long long u = i.first;
        bool sympaticka = i.second;
        if(visited[u]==NOT_VISITED){
            sympaticke[u] = doteraz_sympatickych + sympaticka;
            if(dfs(u))
                return true;
        } else if(visited[u]==OPEN && sympaticke[u]<doteraz_sympatickych+
            ↪ sympaticka){
            return true;
        }
    }
    visited[v] = CLOSED;
    return false;
}

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

    G.resize(n);
    sympaticke.resize(n, 0);
    visited.resize(n, NOT_VISITED);

    for(long long i=0;i<m;i++){
        long long u, v;
```

```
        string sympaticka;

        cin>>u>>v>>sympaticka;

        G[u].push_back({v, "KSP"==sympaticka});
    }

    sympaticke[0] = 0;
    if(dfs(0))
        cout<<"obsahuje"<<endl;
    else
        cout<<"NEobsahuje"<<endl;

    return 0;
}
```

danza

## 6. Hlúpa chata

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

Ked súťažná programátorka počuje "počet možností", okamžite by jej malo napadnúť slovo "dynamika".

### Hrubá sila

Pár bodov sa dalo získať za riešenie hrubou silou. Mohli sme napríklad vyskúšať všetky možné žrebovania a zrátať vyhovujúce.

### Obmedzenia v sadách

Niektoré sady mali malé obmedzenia na $n$, alebo $k$. Tieto sady sa pravdepodobne dali riešiť odvodením vhodných matematických vzorcov.

### Takmer vzorové riešenie

Nuž, ako by sme sa na túto úlohu mohli pozerať? Ako funkčný sa ukáže prístup, kde postupne skúsime všetky možnosti pre počet návštev záchoda najviackrát vyžrebovaného účastníka.

Keď vieme, koľkokrát bol vyžrebovaný najšťastnejší účastník, nech je to $w$, potrebujeme zodpovedať na nasledovnú otázku. Koľkými spôsobmi možno rozdeliť $k - w$ návštev toalety medzi zvyšných $n - 1$ účastníčiek tak, že každá získa nanajvýš $w - 1$ návštev?. Pre účastníčku, ktorá bola vyžrebovaná $w$ krát je $n$ možností, preto odpoveď na našu otázku potrebujeme prenásobiť $n$. Odpoveď na celú úlohu potom bude súčet cez všetky možné maximálne počty vyžrebovaní najúspešnejšieho účastníka.

Budeme teda potrebovať zrátať dynamiku `dp[x][y][z]`, ktorá bude hovoriť, koľkými možnosťami možno rozdeliť `x` vyžrebovaní medzi `y` účastničiek tak, že každá bude vyžrebovaná nanajvýš `z` krát. Triviálne prípady, kde aspoň jedno z `x`, `y`, `z` je 0, vieme určiť jednoducho. Ako následne zrátame `dp[x][y][z]`? Skúsime všetky možnosti pre to, koľkokrát bol vyžrebovaný posledný účastník. Bude teda platiť:

$$dp[x][y][z] = \sum_{i=0}^{z} dp[x-i][y-1][z]$$

Priamočiary pohľad na toto riešenie hovorí, že jeho časová zložitosť bude $O(nk^3)$. Pamäťová zložitosť je $O(nk^2)$. Keďže nám vždy stačia posledné dva riadky tabuľky `dp`, vieme túto zložitosť zlepšiť na $O(nk)$.

### Listing programu (C++)

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

using namespace std;

const int mod = 1000000007;

int main() {
```

```
    int TC, n, k;
    cin >> TC;
    while(TC--) {
        cin >> n >> k;

        if(n == 1) {
            cout << 1 << '\n';
            continue;
        }

        // Inicializujeme dp: kolkymi sposobmi vieme rozdelit `total`
        // bodov medzi `people` ludi zatialco kazdy z nich dostane nanajvys `
            ↪ mx_points` bodov
        long long dp[k + 1][n + 1][k + 1]; // total, people, mx_points
        for(int total = 0; total <= k; total++)
            for(int people = 0; people <= n; people++)
                for(int mx_points = 0; mx_points <= k; mx_points++)
                    dp[total][people][mx_points] = !total;

        // Vypocitame netrivialne pripady dp
        for(int total = 1; total <= k; total++) {
            for(int people = 1; people <= n; people++) {
                for(int mx_points = 1; mx_points <= k; mx_points++) {
                    dp[total][people][mx_points] = 0;
                    // Skusime vsetky moznosti pre pocet bodov ziskanych
                        ↪ poslednou ucastnickou
                    for(int last = 0; last <= mx_points && total - last >= 0;
                        ↪ last++) {
                        dp[total][people][mx_points] += dp[total - last][people
                            ↪ - 1][mx_points];
                        if(dp[total][people][mx_points] >= mod)
                            dp[total][people][mx_points] -= mod;
                    }
                }
            }
        }

        // Skusime vsetky moznosti pre pocet bodov ziskanych vitazom
        long long ans = 0;
        for(int winner_points = 1; winner_points <= k; winner_points++) {
            ans += dp[k - winner_points][n - 1][winner_points - 1];
            if(ans >= mod)
                ans -= mod;
        }

        cout << (ans * n) % mod << '\n';
    }

    return 0;
}
```

**Listing programu (Python)**

```
mod = 1000000007

TC = int(input())
for _ in range(TC):
    n, k = map(int, input().split())
```

```
    if n == 1:
        print(1)
        continue

    dp = []
    for total in range(k + 1):
        dp.append([])
        for people in range(n + 1):
            dp[-1].append([])
            for mx_points in range(k + 1):
                dp[-1][-1].append(1 if total == 0 else 0)

    for total in range(1, k + 1):
        for people in range(1, n + 1):
            for mx_points in range(1, k + 1):
                dp[total][people][mx_points] = 0
                for last in range(mx_points + 1):
                    if total - last < 0:
                        break
                    dp[total][people][mx_points] += dp[total - last][people -
                    ↪ 1][mx_points]
                    if dp[total][people][mx_points] >= mod:
                        dp[total][people][mx_points] -= mod

    ans = 0
    for winner_points in range(1, k + 1):
        ans += dp[k - winner_points][n - 1][winner_points - 1]
        if ans >= mod:
            ans -= mod

    print((ans * n) % mod)
```

### Vzorové riešenie

Ako pri dynamikách býva, aj predchádzajúce riešenie sa dá zoptimalizovať. Takáto optimalizácia často spočíva v tom, že sa zbavíme jedného rozmeru a možno pri tom nejak pomiešame zvyšné rozmery tak, aby sme tretí rozmer nepotrebovali.

Môžeme iterovať cez maximálny povolený počet vyžrebovaní jedného účastníka. Počas toho budeme počítať dynamiku iba so zvyšnými dvoma rozmermi.

Časová zložitosť riešenia po takejto optimalizácii bude $O(nk^2)$. Pamäťová zložitosť je $O(nk)$.

### Listing programu (C++)

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

using namespace std;

const int mod = 1000000007;

int main() {
    int TC, n, k;
    cin >> TC;
    while(TC--) {
        cin >> n >> k;

        if(n == 1) {
            cout << 1 << '\n';
```

```
            continue;
        }

        long long dp[n][k]; // people, total
        long long ans = 0;
        for(int mx_points = 0; mx_points < k; mx_points++) {
            long long k_limit = k - mx_points - 1;
            for(int people = 0; people < n; people++)
                for(int total = 0; total <= k_limit; total++)
                    dp[people][total] = !total;

            for(int people = 1; people < n; people++) {
                for(int total = 1; total <= k_limit; total++) {
                    long long &current = dp[people][total];
                    current +=
                        dp[people][total - 1] +
                        dp[people - 1][total] -
                        (total > mx_points ? dp[people - 1][total - mx_points -
                            ↪ 1] - mod : 0);
                    current -= mod * (current >= mod);
                    current -= mod * (current >= mod);
                }
            }

            ans += dp[n - 1][k_limit];
            ans -= mod * (ans >= mod);
        }

        cout << (ans * n) % mod << '\n';
    }
}
```

**Listing programu (Python)**

```
mod = 1000000007

TC = int(input())
for _ in range(TC):
    n, k = list(map(int, input().split()))

    if n == 1:
        print(1)
        continue

    dp = [[-1 for j in range(k)] for i in range(n)]
    ans = 0
    for mx_points in range(k):
        k_limit = k - mx_points - 1
        for people in range(n):
            for total in range(k_limit + 1):
                dp[people][total] = 1 if total == 0 else 0
        for people in range(1, n):
            for total in range(1, k_limit + 1):
                dp[people][total] += dp[people][total - 1] + dp[people - 1][
                    ↪ total] - (dp[people - 1][total - mx_points - 1] if total >
                    ↪  mx_points else 0)
                for i in range(2):
```

```
                dp[people][total] -= mod * (1 if dp[people][total] >= mod
                    ↪ else 0)
        ans += dp[n - 1][k_limit]
        ans -= mod * (1 if ans >= mod else 0)
    print((ans * n) % mod)
```

Paulinka

## 7. Otoč sa!

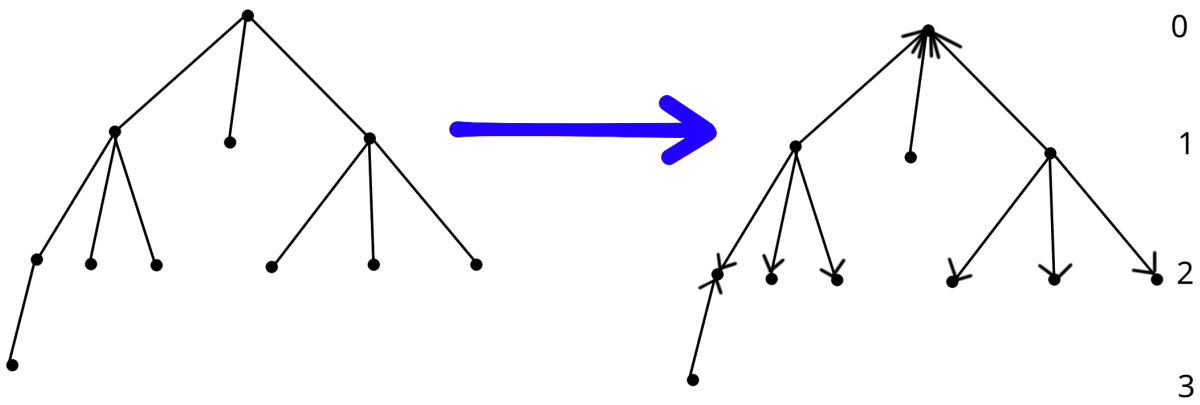
(max. 12 b za popis, 8 b za program)

### Keď je kaktus stromom

Pozrime sa najskôr na to, ako vyriešiť tretiu sadu: keď zadaný graf je strom.
Ukáže sa, že stromy vieme orientovať vždy nasledovným algoritmom:

- Zakoreňme strom a pre každý vrchol spočítame jeho hĺbku (vzdialenosť od vrchola).
- Všetky hrany orientujeme tak, aby išli z nepárnej hĺbky do párnej hĺbky

Každý vrchol po tomto orientovaní má buď všetky hrany idúce do neho, alebo z neho. Takto neexistuje cesta (po orientovaných hranách) s dĺžkou dva (alebo viac). Taktiež sa nevyskutuje cyklus, keďže stromy žiadne cykly nemajú.



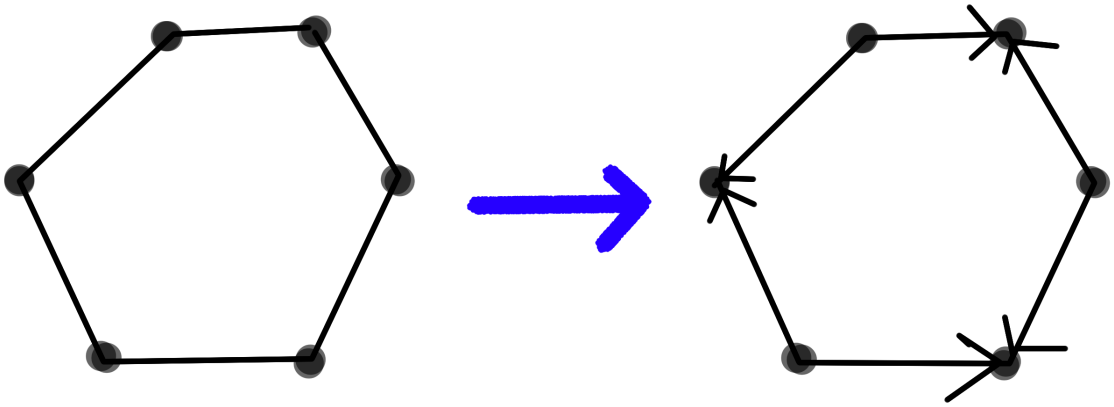

Tento algoritumus vieme implementovať v čase aj pamäti $O(n)$ , napríklad pomocou prehľadávania do hĺbky.

### Čo s cyklami?

Najskôr sa zamyslime, čo sa deje, ak je zadaný kaktus cyklom.

Všimnime si, že akonáhle máme cyklus veľkosti viac ako 3, nemôžeme mať cestu dĺžky dva, teda hranu z $a$ to $b$ a zároveň hrany z $b$ do $c$. Je to preto, že potom by sme museli mať aj hranu z $a$ do $c$. Lenže, ak už hrana z $a$ do $b$ leží na väčšom cykle, nemôže ležať na ďalšom cykle $(a - b - c)$, keďže zadaný graf je kaktus. Takže, každý vrchol ležiaci v cykle dĺžky väčšej než 3 má buď všetky hrany idúce z neho, alebo do neho.

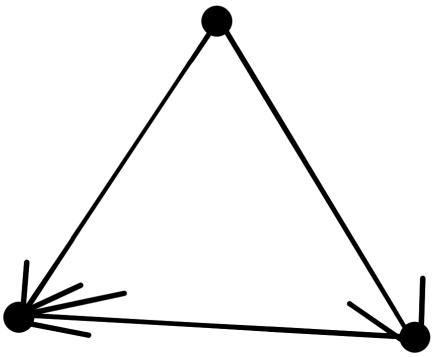
Keď je dĺžka cyklu párna, potom vieme orientovať hrany na striedačku: každý druhý vrchol má všetky jeho hrany idúce z neho, a ostatné vrcholy v cykle majú všetky hrany idúce z nich, ako môžme vidieť na obrázku.



Čo ak je cyklus nepárnej dĺžky? Potom tento istý postup nemôžeme spraviť. Všimnime si navyše, že akonáhle orientujeme jednu hranu v cykle, orientácia ostatných hrán v cykle je daná - tak, aby sme nespravili cyklus. Problém je, že orientácia hrán "na striedačku" v nepárnom cykle nefunguje (skúste si to). Takže akonáhle máme nepárny cyklus dĺžky väčšej ako tri, nevieme ho orientovať.

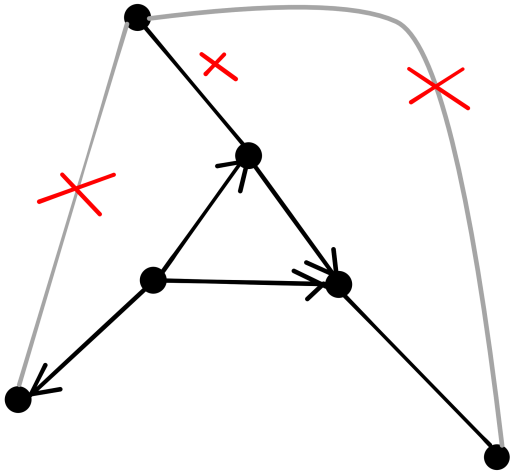
Z toho vyplýva, že keď graf obsahuje nepárny cyklus dlhší než tri, nejde ho zorientovať.

Čo s cyklami dĺžky tri? Všimnime si, že ich hrany vieme orientovať hocijako. Bez ohľadu na ich orientovanie, vždy ostane jeden vrchol do ktorého hrany prichádzajú, jeden z ktorého odchádzajú a jeden, do ktorého jedna hrana prichádza a druhá odchádza. Rozmyslite si, prečo sú tak všetky podmienky zo zadania splnené.



### Všetko dokopy

Vyššie v texte sme si všimli, že ak graf obsahuje nepárny cyklus veľkosti 5 alebo viac, potom hrany orientovať nejde. Ide to inak vždy? Nie, ukáže sa, že je ešte jeden prípad, v ktorom to nejde. Predstavme si, že graf obsahuje cyklus dĺžky tri, taký, že z každého vrchola tohto cyklu ide aspoň jedna ďalšia hrana. Lenže, ako sme pozorovali vyššie, akokoľvek orientujeme hrany v trojcykle, bude mať presne jeden vrchol, ktorý je v strede cesty dĺžky dva (teda jedna hrana troj-cyklu ide doňho, a ďalšia z neho). Problém je s hranami v tomto vrchole, ktoré nie sú hrany trojcyklu. Keďže zadaný graf je kaktus, hrany trojcyklu neležia na žiadnom ďalšom cykle. Z toho nám vyjde, že tieto ďalšie hrny orientovať nevieme (rozmyslite si to).



Ukáže sa, že toto je naozaj posledný prípad kedy to nejde. Na algoritmus nám ďalej poslúži nasledovné pozorovanie: keď zoberieme dobre-orientovaný graf (spĺňajúci podmienky v zadaní), a otočíme *všetky* hrany, dostaneme taktiež dobre-orientovaný graf. Teda nezáleží na tom, ako zorientujeme prvú hranu.

Algoritmus bude podobný ako pre stromy: zvolíme si ľubovoľne orientáciu prvej hrany, a takto sa nám jednoznačne zadá orientácia (takmer) všetkých ostatných hrán. Jediné, kde môžeme mať voľbu sú troj-cykly. V nich treba vybrať ktorý vrchol bude mať jednu príchodziu a jednu odchádzajúcu hranu, a tento vrchol musí mať stupeň presne 2. Jediné, na čo si treba dávať pozor sú trojuholníky - tam treba vybrať orientáciu hrán, (teda aby z neho išli len grafy trojcyklu).

- Najskôr skontrolujme (napríklad prehľadávaním do hĺbky), či v grafe neexistuje nepárny cyklus dĺžky aspoň 5, alebo troj-cyklus, ktorý neobsahuje vrchol stupňa dva.
- Prehľadávame do hĺbky: pre vrcholy vo vrstvách dfs-stromu alternujeme, či idú hrany z nich, či do nich. Treba si dávať pozor na trojuholníky, tie ošetríme špeciálne: vyberieme jeden – ak ich je viac, z jeho vrcholov ktorý má stupeň dva, jednu hranu orientujeme doňho, a druhú z neho.

Všetko sa dá spraviť napríklad dvoma prehľadávaniami do hĺbky, v časovej aj pamäťovej zložitosti $O(n)$.

### Listing programu (C++)

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

using namespace std;

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

typedef complex<long double> point;
```

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

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

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

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

    out << "}"; return out;
}

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

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

pair<ii, bool> treeify(int v, int f, int h, vector<int> &H, vector<ii> &parcyc,
    ↪ vector<vector<int> > &hrany) {
    H[v] = h;
    DBG cout << "In " << v << " with height " << h << endl;

    for (int w : hrany[v]) {
        if (w == f) continue;
        if (H[w] != -1) {
            if (H[w] > H[v]) continue;
            int cycle = h - H[w] + 1;
            DBG cout << "Cycle detected in " << v << " backedge to " << w << "
                ↪ of len " << cycle << endl;
            if (cycle > 3 && (cycle % 2) == 1) return {{v, w}, 0};
```

```
            parcyc[v] = {v, w};

        }
        else {
            auto res = treeify(w, v, h + 1, H, parcyc, hrany);
            if (!res.second) return {res.first, 0};
            // if f-v is in the cycle
            if (res.first.first != v && res.first.second != v && res.first.first
                ↪  >= 0) {
                parcyc[v] = res.first;
            }
        }
    }
    return {parcyc[v], 1};
}

bool otoc(int v, int f, bool into, vector<int> &H, vector<ii> &parcyc, vector<
    ↪ vector<int> > &hrany, set<ii> &oriented) {
    int cycle_len = (parcyc[v].first >= 0 ? H[parcyc[v].first] - H[parcyc[v].
        ↪ second] + 1 : 1);

    DBG cout << "In  " << v << " pointing into ? " << into << " and cycle len "
        ↪ << cycle_len << endl;

    if (cycle_len != 3 || hrany[v].size() > 2) {
        for (int w : hrany[v]) {
            if (w == f) {
                DBG cout << "vf ? " << oriented.count({v,w}) << " and fv " <<
                    ↪ oriented.count({w, v}) << endl;
                if (into && oriented.count({v, w})) return 0;
                if ((!into) & oriented.count({w, v})) return 0;
                continue; // sorted
            }
            if (into) {
                if (oriented.count({v, w})) {
                    DBG cout << "Problem: {" << v << ", " << w << "} in oriented
                        ↪ " << endl;
                    return 0;
                }
                oriented.insert({w, v});
            }
            else {
                if(oriented.count({w, v})) {
                    DBG cout << "Problem: {" << w << ", " << v << "} in oriented
                        ↪ " << endl;
                    return 0;
                }
                oriented.insert({v, w});
            }
        }
        for (int w : hrany[v]) {
            if (parcyc[v] == MP(v, w) || w == f) continue;
            if (parcyc[w] != MP(w, v)) { // o/w it's taken care of
                DBG cout << v << "-> " << w << endl;
                if (!otoc(w, v, !into, H, parcyc, hrany, oriented)) {
                    DBG cout << "From " << w << " reported failure " << endl;
                    return 0;
```

```
                }
            }
        }
    }
    else {
        DBG cout << "in 3-cycle " << endl;
        int w1 = hrany[v][0], w2 = hrany[v][1];

        if (oriented.count({w1, v})) {
            if (oriented.count({w2, v}) + oriented.count({v, w2})) {
                DBG cout << "SEEN BOTH NEIGBOURS" << endl;
                return 1; // both were seen
            }
            oriented.insert({v, w2});
            if (!otoc(w2, v, 1, H, parcyc, hrany, oriented)) return 0;
        }
        else if (oriented.count({v, w1})) {
            if (oriented.count({w2, v}) + oriented.count({v, w2})) return 1; //
                ↪ both were seen
            oriented.insert({w2, v});
            if (!otoc(w2, v, 0, H, parcyc, hrany, oriented)) return 0;
        }
        else if (oriented.count({w2, v})) {
            oriented.insert({v, w1});
            if (!otoc(w1, v, 1, H, parcyc, hrany, oriented)) return 0;
        }
        else if (oriented.count({v, w2})) {
            oriented.insert({w1, v});
            if (!otoc(w1, v, 0, H, parcyc, hrany, oriented)) return 0;
        }
        else {
            // first seen, must be a 3-cycle
            DBG cout << "FIRST SEEN" << endl;
            oriented.insert({v, w2});
            oriented.insert({w1, v});
            oriented.insert({w1, w2});
            return 1;
        }
    }
    return 1;

}

int main() {
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    int n = get<int>();
    int m = get<int>();

    vector<vector<int> > hrany(n);
    FOR(i, m) {
        int a = get<int>() - 1;
        int b = get<int>() - 1;
        hrany[a].push_back(b);
        hrany[b].push_back(a);
    }
    int v0 = 0;
    FOR(i, n) {
```

```
        if (hrany[i].size() > 2) {
            v0 = i;
            break;
        }
    }

    DBG cout << "v0 = " << v0 << endl;

    vector<int> H(n, -1);
    vector<ii> parcyc(n, {-1, -1});

    auto res = treeify(v0, -1, 0, H, parcyc, hrany);
    if (!res.second) {
        cout << "nie" << endl;
        return 0;
    }


    DBG cout << "H: " << H << "parcyc: " << parcyc;

    set<ii> oriented;
    if (!otoc(v0, -1, 1, H, parcyc, hrany, oriented)) {
        cout << "nie" << endl;
    }
    else {
        cout << "ano" << endl;
        for (ii e : oriented) cout << e.first + 1 << " " << e.second + 1 << endl
            ↪ ;
    }



}
```

Maťo

## 8. Dežo

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

V tejto úlohe bolo treba nájsť počet spôsobov, ktorými sa vie Dežo dostať stopom z Trnavy do Nitry.

### Hrubá sila

Môžeme vyskúšať všetky možné podmnožiny áut. Zostáva zistiť pre danú podmnožinu či funguje.

Prvá vec čo potrebujeme zistiť je, v akom poradí tieto autá použijeme. Čas, v ktorom $i$-te auto prejde nejakým bodom $x$, vieme vypočítať ako $t_i + x - a_i$. To znamená, že z $i$-teho auta vieme prestúpiť do $j$-teho iba ak $t_i + x - a_i < t_j + x - a_j$, teda $t_i - a_i < t_j - a_j$. Na začiatku si teda môžeme všetky autá usporiadať podľa $t_i - a_i$ a potom vieme, že autá môžeme použiť iba v poradí v akom sú v tomto zozname.

Keď už máme určené poradie pre danú podmnožnu, musíme ešte skontrolovať, či náhodou nemáme $t_i - a_i == t_j - a_j$ pre nejaké dve po sebe idúce autá. To by znamenalo, že autá idú naraz a teda nestíhame prestúpiť.

Ak nám to časovo sedí, treba ešte zistiť či to sedí aj priestorovo. Prejdeme autá a pamätáme si interval $(l_i; r_i\rangle$ v ktorom vieme skončiť po použití prvých $i$ áut. Prvé auto musí mať $a_1 = 0$ aby sme vedeli nastúpiť a potom $l_1 = a_1$ a $r_1 = b_1$. Do auta $i+1$ vieme prestúpiť práve vtedy, ak $a_{i+1} <= r_i \wedge b_{i+1} > l_i$. Ak $a_{i+1} <= l_i$, vieme presúpiť a potom aj vystúpiť v nejakom bode za $l_i$. Ak $a_{i+1} > l_i$, vieme prestúpiť najskôr v bode $a_{i+1}$ a teda vystúpiť v nejakom bode neskôr ako $a_{i+1}$ (lebo sa musíme viezť nenulový čas). Teda $l_{i+1} = \max(l_i, a_{i+1})$. Najneskôr vieme vystúpiť vždy v $b_{i+1}$, teda $r_{i+1} = b_{i+1}$. Ak sa nám podarí prísť až do konca a $r_n = s$, je daná množina áut použiteľná.

Časová zložitosť tohoto riešenia je $O(n \cdot 2^n + n \log(n))$. Pamäťová zložitosť je $O(n)$.

### Listing programu (C++)

```
#include <bits/stdc++.h>
#define MOD 1000000007

using namespace std;

struct Car
{
    int a, b;
    int t;
};

int main()
{
    int n, s;
    cin >> n >> s;
    vector<Car> v(n);
    for (int i = 0; i < n; i++)
        cin >> v[i].a >> v[i].b >> v[i].t;
    sort(v.begin(), v.end(), [](Car a, Car b)
         { return (a.t - a.a) < (b.t - b.a); });

    int result = 0;
    for (int i = 0; i < (1 << n); i++)
    {
        int l = 0, r = 0;
        bool fail = false;
        int t = -1;
        for (int j = 0; j < n; j++)
        {
            if (!(i & (1 << j)))
                continue;
            Car c = v[j];
            if (c.t - c.a <= t || c.a > r || c.b <= l)
            {
                fail = true;
                break;
            }
            r = c.b;
            l = max(l, c.a);
            t = c.t - c.a;
        }
        if (!fail && r == s)
            result++;
    }
    cout << result << "\n";
}
```

### Dynamika

Pre každú neprázdnu podmnožinu áut, ak ich vieme postupne všetky použiť, vieme definovať interval v ktorom môžeme skončiť po tejto ceste. Bude to polootvorený interval v tvare $(l; r\rangle$.

Pre každú trojicu $k, l, r$ si spočítame, koľko existuje podmnožín z prvých $k$ áut (máme ich usporiadané v poradí v akom idú), ktorých cesta končí v intervale $(l; r\rangle$. Túto hodnotu si označime $D(k, l, r)$. Očividne $D(k, l, r) = 0$ ak $l >= r$.

Povedzme, že to máme spočítané pre všetky $k < i$ a ideme to počítať pre $k = i$. To vypočítame ako $D(i, l, r) = D(i - 1, l, r) + d(i, l, r)$, kde $d(i, l, r)$ je počet nových ciest, ktoré použijú aj auto $i$.

Každá cesta končiaca autom $i$ končí najneskôr v bode $b_i$, teda pre všetky $l, r$ kde $r \neq b_i$ máme $d(i, l, r) = 0$.

Tak isto pre $l < a_i$ nevieme použiť $i$-te auto, keďže z neho vieme vystúpiť iba neskôr ako $a_i$, teda pre ne tiež $d(i, l, r) = 0$.

Nech $j$ je posledné auto ktoré ide ostro skôr ako $i$ (teda sa z neho dá prestúpiť). Zvyšné hodnoty teraz vieme vypočítať ako:

- $d(i, a_i, b_i) = \sum_{l=0}^{a_i} \sum_{r=a_i}^{s} D(j, l, r)$, $+1$ ak $a_i = 0$
- $d(i, l, b_i) = \sum_{r=l+1}^{s} D(j, l, r)$; pre $a_i < l < b_i$

Keď $a_i = 0$, môžeme auto $i$ použiť aj ako úplne prvé. Takáto cesta končí v intervale $(a_i; b_i\rangle$, preto pripočítame 1 k $d(i, a_i, b_i)$.

Výpočet začneme tým, že $D(0, l, r) = 0$ a potom postupne počítame $D(i, l, r)$ pre $i = 1, 2, 3 \ldots$.

Keď pridáme nové auto, nové hodnoty vieme vypočítať v čase $O(s^2)$, teda celková časová zložitosť je $O(n \cdot s^2)$. Pamäťová zložitosť je tiež $O(n \cdot s^2)$.

Pamäťová zložitosť sa zlepšiť na $O(s^2)$. Stačí si pamätať hodnoty $D(i, l, r)$ pre predchádzajúce $i$ a k tomu ešte hodnoty $D(j, l, r)$ pre posledné auto $j$, ktoré ide ostro skôr. Ďalšie hodnoty budú vždy zavisieť iba od týchto dvoch.

### Listing programu (C++)

```
#include <bits/stdc++.h>
#define MOD 1000000007

using namespace std;

struct Car
{
    int a, b;
    int t;
};

int main()
{
    int n, s;
    cin >> n >> s;
    vector<Car> v(n);
    for (int i = 0; i < n; i++)
        cin >> v[i].a >> v[i].b >> v[i].t;
    sort(v.begin(), v.end(), [](Car a, Car b)
         { return (a.t - a.a) < (b.t - b.a); });

    vector<vector<long long>> dyn;
    for (int i = 0; i <= s; i++)
        dyn.push_back(vector<long long>(i));

    for (int i = 0; i < n;)
    {
        auto last = dyn;
        int t = v[i].t - v[i].a;
        vector<Car> cars;
        while (i < n && v[i].t - v[i].a == t)
            cars.push_back(v[i++]);
        for (Car c : cars)
        {

            if (c.a == 0)
            {
                dyn[c.b][c.a]++;
                dyn[c.b][c.a] %= MOD;
```

```
            }

            for (int b = 0; b <= s; b++)
                for (int a = 0; a < b; a++)
                {
                    if (c.a > b || c.b <= a)
                        continue;

                    dyn[c.b][max(a, c.a)] += last[b][a];
                    dyn[c.b][max(a, c.a)] %= MOD;
                }
        }
    }

    long long result = 0;
    for (int i = 0; i < s; i++)
        result += dyn[s][i];
    result %= MOD;
    cout << result << "\n";
}
```

### Intervaláč

Lepšie riešenie vieme dosiahnuť tým, že si hodnoty budeme pamätať v súčtovom intervalovom strome. Keď vypočítame hodnoty pre $i$, budeme ich mať zapísané v $s$ intervaláčoch: jeden intervaláč pre každé $l$, v ktorom budeme mať hodnoty $D(i,l,r)$ pre $0 \leq r \leq s$. Tieto intervaláče budeme upravovať pritom ako spracúvame ďalšie autá.

Vždy naraz spracujeme všetky autá ktoré idú naraz (podľa $t-a$). Nech sú to autá $i+1$ až $j$. Vieme, že $D(j,l,r) = D(i,l,r) + d(i+1,l,r) + d(i+1,l,r) \ldots d(j,l,r)$. Najprv teda vypočítame všetky $d(x,l,r)$ ($x \in \{i+1,\ldots,j\}$) a potom v intervaláči pričítame tieto hodnoty.

Nemusíme počítať tie $d(x,l,r)$, o ktorých vieme, že sa vždy rovnajú 0, teda stačí vždy vypočítať iba $d(x,l,b_x)$ pre $a_x \leq l < b_x$. Hodnotu $d(x,a_x,b_x)$ vieme vypočítať pomocou $s$ otázok na intervaláč v $O(s \log s)$ a každú z $d(x,l,b_x)$ pre $a_x < l < b_x$ vypočítame jednou otázkou v $O(\log s)$.

Časová zložitosť tohto riešenia bude $O(ns \log s + s^2)$, pričom $s^2$ je kvôli skonštruovaniu intervaláča. Tohoto člena sa vieme zbaviť tým, že intervaláč budeme konštruovať dynamicky počas každej query. Pamäťová zložitosť je $O(s^2)$ alebo $O(ns \log s)$ pri dynamickom intervaláči.

### Listing programu (C++)

```
#include <bits/stdc++.h>
#define MOD 1000000007

using namespace std;

struct Car
{
    int a, b;
    int t;
};

struct Inter
{
    long long val = 0;
    int l, r;
    Inter *L = nullptr;
    Inter *R = nullptr;
    Inter(int l, int r) : l(l), r(r) {}
    long long get(int ll, int rr)
```

```
    {
        if (ll >= r || rr <= l)
            return 0;
        if (ll <= l && rr >= r)
            return val;
        lazy_propagate();
        return (L->get(ll, rr) + R->get(ll, rr)) % MOD;
    }
    void add(int p, long long v)
    {
        if (p >= r || p < l)
            return;
        val += v;
        val %= MOD;
        if (r > l + 1)
        {
            lazy_propagate();
            L->add(p, v);
            R->add(p, v);
        }
    }
    void lazy_propagate()
    {
        if (L == nullptr)
            L = new Inter(l, (l + r) / 2);
        if (R == nullptr)
            R = new Inter((l + r) / 2, r);
    }
};

int main()
{
    int n, s;
    cin >> n >> s;
    vector<Car> v(n);
    for (int i = 0; i < n; i++)
        cin >> v[i].a >> v[i].b >> v[i].t;
    sort(v.begin(), v.end(), [](Car a, Car b)
         { return (a.t - a.a) < (b.t - b.a); });

    vector<Inter> dyn;
    for (int i = 0; i < s; i++)
        dyn.push_back(Inter(0, s + 1));

    for (int i = 0; i < n;)
    {
        int t = v[i].t - v[i].a;
        vector<Car> cars;
        while (i < n && v[i].t - v[i].a == t)
            cars.push_back(v[i++]);
        vector<vector<long long>> add;
        for (Car c : cars)
        {
            add.push_back(vector<long long>(s));
            if (c.a == 0)
                add.back()[c.a]++;
            for (int a = 0; a < c.b; a++)
                add.back()[max(a, c.a)] += dyn[a].get(c.a, s + 1);
```

```
        }
        for (int ci = 0; ci < cars.size(); ci++)
            for (int a = 0; a < s; a++)
            {
                add[ci][a] %= MOD;
                dyn[a].add(cars[ci].b, add[ci][a]);
            }
    }

    long long result = 0;
    for (long long i = 0; i < s; i++)
        result += dyn[i].get(s, s + 1);
    result %= MOD;
    cout << result << "\n";
}
```

### Optimálne riešenie

V optimálnom riešení budeme používať iba dva intervaláče. Prvý $L(l)$ bude obsahovať počty ciest, ktorých interval začína v bode $l$, teda $L(l) = \sum_{r=0}^{s} D(i,l,r)$ (vždy pre nejaké konkrétne $i$) a druhý $R(r)$ bude obsahovať počty ciest, ktorých interval končí v bode $r$, teda $R(r) = \sum_{l=0}^{s} D(i,l,r)$. Súčty intervalov budeme označovať $L(a,b) = \sum_{i=a}^{b} L(i)$ a $R(a,b) = \sum_{i=a}^{b} R(i)$

Keď pridávame nejaké nové auto $i$, potrebujeme vypočítať počet ciest z ktorých sa naňho dá prestúpiť. To budú také cesty, ktorých interval sa prekrýva s intervalom $\langle a_i; b_i \rangle$. To vieme vypočítať ako $R(a_i, s) - L(b_i, s)$. Toľko bude nových ciest, ktoré použijú toto auto. Keďže interval každej z nich končí v bode $b_i$, intrvaláču $R$ stačí k $R(b_i)$ pričítať túto hodnotu.

Aktualizovať intervaláč $L$ bude o niečo komplikovanejšie, lebo začiatky intervalov ciest sú rôzne. Ak na auto $i$ prestúpime z cesty, ktorej interval začína skôr ako $a_i$, interval novej cesty bude začínať v bode $a_i$. Počet takýchto ciest vieme vypočítať ako $R(a_i, s) - L(a_i, s)$. Teda k $L(a_i)$ musíme pričítať túto hodnotu.

Zostávajú intervaly ktoré začínajú v $a_i$ a neskôr. Takéto intervaly majú prekryv práve vtedy, ak začínajú skôr ako $b_i$. Ak takýto interval začína v $l$, nový interval bude tiež začínať v $l$. To znamená, že pre všetky $a_i \leq l < b_i$ musíme spraviť $L(l) += L(l)$, alebo teda $L(l)$ *= 2. Takúto operáciu "plus svoja hodnota" vieme robiť lazy intervaláčom.

Vždy si najprv niekde zapíšeme aké operácie chceme na intervaláčoch spraviť a potom ich vykonáme, aby sme si počas výpočtu neprepísali staré hodnoty novými. Ak máme niekoľko áut, ktoré idú naraz, zapíšeme si najprv operácie pre všetky autá a až potom ich vykonáme. Treba si pri tom dať pozor ako vyhodnotíme operácie typu "plus svoja hodnota". Ak viacero áut chce aplikovať túto operáciu na rovnakej pozícii, vo výsledku musíme na tejto pozícii spraviť $\times 3$. Ak by sme iba postupne dvakrát aplikovali túto operáciu, dostali by sme $\times 4$. Toto vieme vyriešiť tým, že si v zvlášť dátovej štruktúre udržiavame akým číslom treba ktoré pozície vynásobiť a potom na konci to aplikovať.

Na to môžeme použiť napríklad mapu ($M$) a keď nám príde nejaké auto $i$, spravíme $M[a_i]$ += 1, $M[b_i]$ -= 1. Keď chceme zistiť, akým číslom treba vynásobiť pozíciu $x$, vieme to vypočítať ako $\sum_{i=0}^{x} M[i]$. Na konci teda iba preiterujeme túto mapu, pričom si pamätáme doterajší súčet a vždy týmto číslo vynásobíme interval medzi dvoma po sebe idúcimi kľúčmi tejto mapy. Ak máme takto $k$ áut, týchto intervalov bude najviac $2k$ a teda len toľko operácií musíme na intervaláči vykonať.

Keď prejdeme všetky autá, odpoveď zistíme jednoducho ako $R(s)$.

Pre každé auto spravíme iba konštantne veľa operácií s intervaláčom. teda časová zložitosť je $O(n \log s)$. Pamäťová zložitosť je tiež $O(n \log s)$. Obe zložitosti sú za predpokladu, že používame dynamické intervaláče.

### Listing programu (C++)

```
#include <bits/stdc++.h>
#define MOD 1000000007

using namespace std;

struct Car
{
```

```
    int a, b;
    int t;
};

struct Inter
{
    long long val = 0;
    int l, r;
    long long lazy_mul = 1;
    Inter *L = nullptr;
    Inter *R = nullptr;
    Inter(int l, int r) : l(l), r(r) {}
    long long get(int ll, int rr)
    {
        if (ll >= r || rr <= l)
            return 0;
        if (ll <= l && rr >= r)
            return val;
        lazy_propagate();
        return (L->get(ll, rr) + R->get(ll, rr)) % MOD;
    }
    void add(int p, long long v)
    {
        if (p >= r || p < l)
            return;
        val += v;
        val %= MOD;
        if (r > l + 1)
        {
            lazy_propagate();
            L->add(p, v);
            R->add(p, v);
        }
    }
    void mul(int ll, int rr, long long v)
    {
        if (ll >= r || rr <= l)
            return;
        if (ll <= l && rr >= r)
        {
            lazy_update(v);
            return;
        }
        lazy_propagate();
        L->mul(ll, rr, v);
        R->mul(ll, rr, v);
        val = (L->val + R->val) % MOD;
    }
    void lazy_update(long long v)
    {
        val = (val * v) % MOD;
        lazy_mul = (lazy_mul * v) % MOD;
    }
    void lazy_propagate()
    {
        if (L == nullptr)
            L = new Inter(l, (l + r) / 2);
        if (R == nullptr)
```

```
            R = new Inter((l + r) / 2, r);

        if (lazy_mul == 1)
            return;
        L->lazy_update(lazy_mul);
        R->lazy_update(lazy_mul);
        lazy_mul = 1;
    }
};

int main()
{
    int n, s;
    cin >> n >> s;
    vector<Car> v(n);
    for (int i = 0; i < n; i++)
        cin >> v[i].a >> v[i].b >> v[i].t;
    sort(v.begin(), v.end(), [](Car a, Car b)
         { return (a.t - a.a) < (b.t - b.a); });

    Inter ends(0, s + 1);
    Inter starts(0, s + 1);

    for (int i = 0; i < n;)
    {
        int t = v[i].t - v[i].a;
        vector<Car> cars;
        while (i < n && v[i].t - v[i].a == t)
            cars.push_back(v[i++]);

        vector<long long> pred;
        vector<long long> prekryv;

        for (Car c : cars)
        {
            long long p = (ends.get(c.a, s + 1) - starts.get(c.a, s + 1) + MOD)
                ↪ % MOD;
            pred.push_back(p);
            prekryv.push_back((p + starts.get(c.a, c.b)) % MOD);
        }

        map<int, int> intervals;
        for (Car c : cars)
        {
            intervals[c.a]++;
            intervals[c.b]--;
        }
        int last_mul = 1;
        int last_start = 0;
        for (auto x : intervals)
        {
            starts.mul(last_start, x.first, last_mul);
            last_start = x.first;
            last_mul += x.second;
        }

        for (int ci = 0; ci < cars.size(); ci++)
        {
```

```
            ends.add(cars[ci].b, prekryv[ci]);
            starts.add(cars[ci].a, pred[ci]);
            if (cars[ci].a == 0)
            {
                starts.add(cars[ci].a, 1);
                ends.add(cars[ci].b, 1);
            }
        }
    }

    long long result = ends.get(s, s + 1);
    cout << result << "\n";
}
```