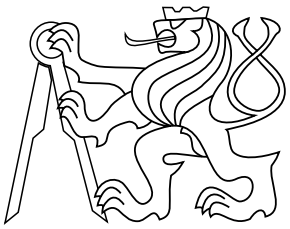
Programovací úlohy v Algoritmizaci

Elektronická skripta předmětu B4B33ALG

Daniel Průša

26. září 2025



Katedra kybernetiky
Fakulta elektrotechnická
České vysoké učení technické v Praze

Obsah

1 Poznámkové lístky

Kvido je zvyklý organizovat svoji práci pomocí velkého množství poznámkových lístků. V kanceláři má částečně prosklené dveře, přes které je dobře vidět jak z místnosti na chodbu, tak z chodby do místnosti. Svoje poznámkové lístky umisťuje na spodní okraj skla dveří (ne však výše, aby o výhled na chodbu nepřišel). Vzhledem k velkému množství lístků je lepí různě přes sebe. Aby byl schopen najít jakýkoliv lístek, píše si poznámky o datu jejich vylepení, rozměrech a vzdálenosti od levého okraje skla.

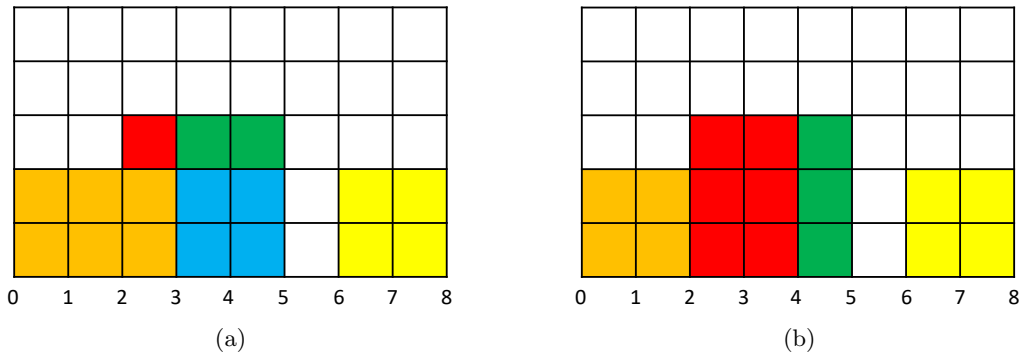
Po nějakém čase Kvida zajímá, zda je každý lístek alespoň částečně vidět z jeho kanceláře nebo z chodby. Mohl by viditelné lístky spočítat, přemýšlí ale, že bude lepší, když vyhodnocení provede automaticky, na základě seznamu lístků.

Úloha

Je dán seznam obdélníkových poznámkových lístků, uspořádaných podle data a času jejich vylepení. Pro každý lístek jsou zadány jeho rozměry a vzdálenost od levého okraje skleněné stěny. Předpokládáme, že jak rozměry, tak vzdálenosti jsou vyjádřeny jako celočíselné násobky určité jednotkové délky.

Lístek je považován za viditelný (z kanceláře nebo z chodby), pokud není zcela překryt jinými lístky – tedy pokud z něj zůstane odkrytá alespoň jedna čtvercová ploška o velikosti 1×1. Příklad je uveden na Obrázku 1.

Určete, kolik lístků je viditelných zároveň z chodby i z kanceláře, kolik pouze z jedné strany, a kolik není viditelných vůbec.



(a) (b)

Obrázek 1: Pohled na vylepené lístky: (a) z kanceláře a (b) z chodby. Předpokládáme, že Kvido vylepil pět lístků v tomto pořadí: červený (rozměry 3×2) ve vzdálenosti 2 od levého okraje skla; žlutý (2×2) ve vzdálenosti 6; zelený (3×2) ve vzdálenosti 3; modrý (2×2) ve vzdálenosti 3; a oranžový (2×3) ve vzdálenosti 0. Z kanceláře jsou viditelné všechny lístky, z chodby je zakrytý modrý, ostatní zůstávají viditelné.

Vstup

První řádek obsahuje dvě celá čísla O, N oddělená mezerou. Číslo O je šířka skla dveří, číslo N je počet vylepených lístků. Následuje N řádků, kde každý řádek popisuje jeden vylepený lístek, přičemž je zachováno pořadí vylepení lístků, tj., i-tý z těchto řádků popisuje lístek vylepený jako i-tý. Na jednotlivých řádcích jsou tři celá čísla X, V, S oddělená mezerami. Číslo X je vzdálenost lístku od levého okraje skla, číslo V je výška lístku a číslo S je šířka lístku. Pravý okraj lístku nikdy nepřesahuje pravý okraj skla. Veškeré vstupní rozměry a vzdálenosti jsou vyjádřeny jako celočíselné násobky blíže nespecifikované jednotkové délky.

Nechť $V_{\max}$ je výška nejvyššího lístku a $S_{\max}$ je šířka nejširšího lístku. Platí

$$1 \leq O \leq 3 \cdot 10^6, \quad 1 \leq N \leq 2 \cdot 10^5, \quad V_{\max}, S_{\max} \leq 6 \cdot 10^4$$

Výstup

Výstup obsahuje jeden textový řádek, na kterém jsou tři čísla oddělená mezerami. První číslo udává počet poznámkových lístků, které jsou viditelné z kanceláře i z chodby, druhé číslo udává počet lístků viditelných pouze z jedné strany skla a třetí číslo udává počet lístků, které nejsou viditelné ani z jedné strany skla.

Příklad 1

Vstup

```
8 5
2 3 2
6 2 2
3 3 2
3 2 2
0 2 3
```

Výstup

```
4 1 0
```

Data Příkladu 1 jsou vizualizována na Obrázku 1.

Příklad 2

Vstup

```
10 6
5 4 4
5 3 4
3 7 4
1 1 2
4 3 2
5 1 1
```

Výstup

```
3 3 0
```

Příklad 3

Vstup

```
15 4
4 14 8
5 4 6
4 9 6
2 11 8
```

Výstup

```
2 1 1
```

Rozbor řešení

Nejprve určíme, které lístky jsou viditelné z kanceláře. K tomu použijeme celočíselné pole `height` délky O, jež pro každou horizontální pozici na skle (tj. vzdálenost od levého okraje) uchovává dosud nejvyšší dosaženou výšku vylepeného lístku. Lístky procházíme ve vzestupném pořadí podle času jejich vylepení.

Pro každý lístek na pozici p, výšky h a šířky w zkontrolujeme všechny x-ové souřadnice, do kterých zasahuje, tedy $x = p, p+1, \ldots, p+w-1$. Pokud pro některou z těchto souřadnic platí $h >$ `height`$[x]$, znamená to, že daná část lístku není zcela překrytá a lístek je tedy viditelný z kanceláře. V takovém případě aktualizujeme `height`[x] na hodnotu h. Informaci o viditelnosti si pro každý lístek zaznamenáme do samostatného pole `visibleFromFront`.

Viditelnost z chodby určíme obdobným způsobem, pouze lístky procházíme v opačném pořadí (tj. od posledního vylepeného k prvnímu). Pole `height` před použitím vynulujeme. Pokud je lístek podle detekce viditelný zezadu, kontrolujeme jeho předchozí viditelnost zepředu:

- pokud již byl viditelný z kanceláře, zvýšíme čítač `visibleTwoSides`,
- pokud nebyl, zvýšíme čítač `visibleOneSide`.

Zbylé lístky, které nebyly označeny jako viditelné ani v jednom průchodu, dopočítáme jako neviditelné.

Popsaný algoritmus má časovou složitost $\mathcal{O}(N \cdot S_{\max})$, protože každý lístek zpracováváme v čase úměrném jeho šířce.

Java kód

```
package alg;

import java.io.*;
import java.util.*;

public class Notes {

    public static int W, N;
    public static int[] sx, height, width;
    public static int[] top;
    public static boolean[] visibleFromFront;

    public static int[] solve() {
        top = new int[W];
        visibleFromFront = new boolean[N];
        for (int i = 0; i < N; i++) {
            boolean isVisible = false;
            for (int x = sx[i]; x < sx[i] + width[i]; x++) {
                if (top[x] < height[i]) {
                    isVisible = true;
                    top[x] = height[i];
                }
            }
            visibleFromFront[i] = isVisible;
        }

        top = new int[W];
        int visibleOneSide = 0, visibleTwoSides = 0;
        for (int i = N - 1; i >= 0; i--) {
            boolean isVisible = false;
            for (int x = sx[i]; x < sx[i] + width[i]; x++) {
                if (top[x] < height[i]) {
                    isVisible = true;
                    top[x] = height[i];
                }
            }
            if (isVisible && visibleFromFront[i])
                visibleTwoSides++;
            else if (isVisible || visibleFromFront[i])
                visibleOneSide++;
        }
        return new int[] {visibleTwoSides, visibleOneSide, N - visibleTwoSides -
            visibleOneSide};
    }

    public static void loadData() throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        StringTokenizer tokenizer = new StringTokenizer(reader.readLine());
        W = Integer.valueOf(tokenizer.nextToken());
        N = Integer.valueOf(tokenizer.nextToken());

        sx = new int[N];
        height = new int[N];
        width = new int[N];

        for (int i = 0; i < N; i++) {
            tokenizer = new StringTokenizer(reader.readLine());
            sx[i] = Integer.valueOf(tokenizer.nextToken());
            height[i] = Integer.valueOf(tokenizer.nextToken());
            width[i] = Integer.valueOf(tokenizer.nextToken());
        }
    }

    public static void main(String[] args) throws IOException {
```

```
64          loadData();
65          int res[] = solve();
66          System.out.println(res[0] + " " + res[1] + " " + res[2]);
67      }
68
69  }
```

2 Fototrofní organismy

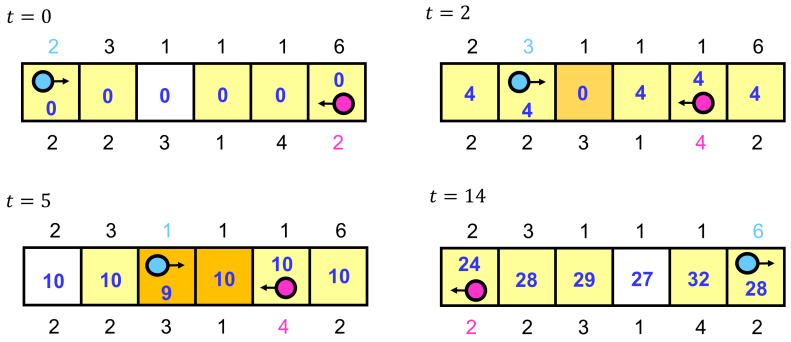
Skupina biologů zkoumá možnosti pěstování fototrofních organismů v nehostinném prostředí. Pro experiment zvolili dlouhou chodbu v jeskyni, kterou pro své účely pomyslně rozdělili do N úseků číslovaných z levého konce chodby k pravému postupně od 1 do N. Organismy na stěnách jeskyně zamýšlejí osvětlovat pomocí dvou robotů, z nichž každý je vybaven svítilnou s dlouhým dosvitem. Kromě úseku, ve kterém se první robot právě nachází, osvítí ještě D_1 nejbližších úseků v každém směru. Znamená to tedy, že první robot osvítí $1 + 2 \times D_1$ úseků, pokud je dostatečně vzdálen od obou konců chodby. Analogicky osvítí druhý robot D_2 nejbližších úseků v každém směru, přičemž typicky platí $D_1 \neq D_2$.

Roboty se po chodbě pohybují. První začíná v úseku číslo 1 a přesouvá se až k úseku číslo N, druhý se naopak přesouvá z úseku číslo N až k úseku číslo 1. Platí, že první robot při svém pohybu čeká v k-tém úseku čas T_k, zatímco druhý robot čas S_k. Zároveň platí $T_1 + \cdots + T_N = S_1 + \cdots + S_N$. Znamená to, že oba roboty ukončí průchod chodbou ve stejném okamžiku. Poté se mohou opět podobným způsobem vracet a díky tomu pravidelně osvětlovat jednotlivé úseky jeskyně. Čas na přesun z jednoho úseku do druhého je zanedbatelný vůči době čekání v úsecích, proto ho považujeme za nulový.

Biologové se zajímají, jaká bude kvalita osvětlení v jednotlivých úsecích při jednom průchodu robotů chodbou, tedy v čase $T_1 + \cdots + T_N$. Kvalitu osvětlení úseku vyjadřují celým číslem, které je odvozené od toho, po jakou dobu je daný úsek osvětlován pouze jedním robotem (prvním nebo druhým, označme tuto dobu jako O_1) a po jakou dobu oběma roboty současně (doba O_2). Kvalita osvětlení je stanovena výrazem $2 \times O_1 + 3 \times O_2$.

Úloha

Jsou dány dosvity robotů a jejich časy strávené v jednotlivých úsecích chodby. Pro jeden průchod robotů určete, jaká je minimální a maximální hodnota kvality osvětlení úseků.



Obrázek 2: Schémata zachycují průběh průchodu robotů chodbou postupně v časech 0, 2, 5 a 14. Platí $N = 6$, $D_1 = 1$, $D_2 = 2$. Úseky jsou reprezentovány čtverečky, první robot světle modrým kolečkem, druhý robot červeno-fialovým kolečkem. Úsek osvětlený jedním robotem je zvýrazněn žlutě, úsek osvětlený oběma roboty oranžově. Časy T_k jsou uvedené nad úseky, časy S_k pod úseky. Tmavě modrá čísla uvnitř úseků odpovídají doposud dosažené kvalitě osvětlení. Roboty dokončí pohyb v čase 14.

Vstup

První řádek vstupu obsahuje tři celá čísla N, D_1 a D_2, oddělená mezerou, kde N je počet úseků chodby, D_1 je dosvit prvního robotu a D_2 je dosvit druhého robotu.

Platí $1 \leq N \leq 8 \times 10^6$, $0 \leq D_1, D_2 < N$. Kvalita osvětlení každého z úseků během průchodu robotů nepřesáhne hodnotu 10^8.

Výstup

Výstup obsahuje jeden textový řádek s celými čísly $K_{\min}$ a $K_{\max}$ oddělenými mezerou, kde $K_{\min}$ je minimální kvalita osvětlení v úsecích po dokončení jednoho průchodu robotů a $K_{\max}$ je maximální kvalita osvětlení.

Příklad 1

Vstup

```
4 1 1
1 2 3 4
2 3 4 1
```

Výstup

```
16 27
```

Příklad 2

Vstup

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

Vstup

```
24 32
```

Data a řešení Příkladu 2 můžeme vidět na Obrázku 2.

Rozbor řešení

Klíčové je efektivně určit pro každý úsek časový interval, kdy do něj dopadá světlo od prvního a od druhého robota. Protože pohyb obou robotů je deterministický a doba čekání v jednotlivých úsecích je známá, lze pro každý úsek předpočítat časy:

- $tin[i]$ – okamžik, kdy světelný kužel daného robota začne pokrývat úsek i,
- $tout[i]$ – okamžik, kdy světelný kužel přestane pokrývat úsek i.

U prvního robota (pohybujícího se zleva doprava) se tyto časy určují postupným sčítáním dob čekání. U druhého robota (pohybujícího se zprava doleva) se situace obrací: jeho pokrytí úseku i odpovídá časovému intervalu, který lze vyjádřit jako transformaci podle celkové doby průchodu T.

Pro každý úsek máme tedy dva časové intervaly: $[in_1, out_1]$ od prvního robota a $[in_2, out_2]$ od druhého robota. Celková kvalita se rozdělí na:

$$O_1 = (out_1 - in_1) + (out_2 - in_2) - O_2,$$

$$O_2 = \text{délka průniku intervalů } [in_1, out_1] \cap [in_2, out_2].$$

Poté dosadíme do definice kvality:

$$K = 2 \cdot O_1 + 3 \cdot O_2.$$

Stačí spočítat K pro všechny úseky $1, \ldots, N$. Během průchodu se vypočítá pro každý úsek přesná kvalita osvětlení a z těchto hodnot se určí:

$$K_{\min} = \min_{1 \le i \le N} K(i), \qquad K_{\max} = \max_{1 \le i \le N} K(i).$$

Protože je třeba projít všechny úseky a pro každý úsek se počítají pouze konstantně složité operace (sčítání a porovnání časů), je výsledná složitost algoritmu $\mathcal{O}(N)$. To je vyhovující i pro maximální hodnotu $N = 8 \times 10^6$.

Java kód

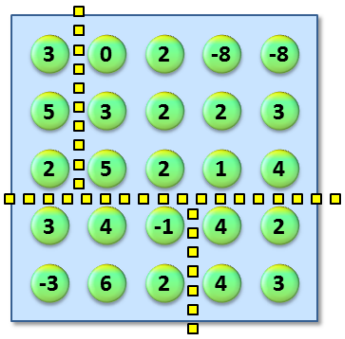
```
package alg;

import java.io.*;
import java.util.*;

public class Cave {

    public static final int K1 = 2;
    public static final int K2 = 3;
    public static final int DK = K2 - K1;
    public static int N, T;
    public static int[] D;
    public static int[][] times;
    public static int[][] tin, tout;

    public static int[] solve() {
        tin = new int[2][N];
        tout = new int[2][N];
        int minGain = Integer.MAX_VALUE;
        int maxGain = Integer.MIN_VALUE;
        int minIndex = 0, maxIndex = 0;
        T = 0;
        for (int i = 0; i < N; i++)
            T += times[0][i];

        computeTimes(0);
        computeTimes(1);

        for (int i = 0; i < N; i++) {
            int in1 = tin[0][i];
            int out1 = tout[0][i];
            int in2 = T - tout[1][i];
            int out2 = T - tin[1][i];
            int gain = K1*(out1 - in1) + K1*(out2 - in2) - DK*(intersect(in1, out1, in2,
                out2));
            if (gain > maxGain) {
                maxGain = gain;
                maxIndex = i;
            }
            if (gain < minGain) {
                minGain = gain;
                minIndex = i;
            }
        }
        return new int[] {minGain, maxGain};
    }

    public static int intersect(int as, int ae, int bs, int be) {
        if (as > be || bs > ae)
            return 0;
        return Math.min(ae, be) - Math.max(as, bs);
    }

    public static void computeTimes(int index) {
        for (int i = 0; i <= D[index]; i++)
            tout[index][0] += times[index][i];
        for (int i = 1; i < N; i++) {
            if (i + D[index] < N)
                tout[index][i] = tout[index][i-1] + times[index][i+D[index]];
            else
                tout[index][i] = T;
            if (i <= D[index])
                tin[index][i] = 0;
            else
```

```
                tin[index][i] = tin[index][i-1] + times[index][i - D[index]-1];
        }
    }

    public static void loadData() throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        StringTokenizer tokenizer = new StringTokenizer(reader.readLine());
        N = Integer.valueOf(tokenizer.nextToken());
        D = new int[2];
        D[0] = Integer.valueOf(tokenizer.nextToken());
        D[1] = Integer.valueOf(tokenizer.nextToken());
        times = new int[2][N];
        for (int r = 0; r <= 1; r++) {
            tokenizer = new StringTokenizer(reader.readLine());
            for (int i = 0; i < N; i++)
                times[r][i] = Integer.valueOf(tokenizer.nextToken());
        }
    }

    public static void main(String[] args) throws IOException {
        loadData();
        int[] result = solve();
        System.out.println(result[0] + " " + result[1]);
    }

}
```

3 Rozdělení sadu

Kvido má čtvercový sad plný různých ovocných stromů. Stromy jsou vysázeny do pravidelných řad a sloupců tvořících čtvercovou mřížku. Počet řad je stejný jako počet sloupců. Aby mohl Kvido snadněji udržovat sad, chce ho rozdělit na čtyři obdélníkové části. Nejprve rozdělí sad na severní a jižní část jednou přímou cestou vedoucí od východu k západu mezi dvěma řadami stromů. Poté rozdělí severní část jednou přímou cestou vedoucí od severu kolmo na cestu východ-západ. Jižní část bude rozdělena podobně přímou cestou vedoucí od jihu kolmo na cestu východ-západ. Tyto dvě další cesty nemusí vést ze stejného bodu. Výsledná situace může vypadat jako na obrázku níže.

Kvido přiřadil každému stromu kvalitativní index, který může být kladný, záporný nebo nula. Kvalita každé ze čtyř částí sadu je součtem kvalit všech stromů v dané části. Kvido chce, aby výsledné části měly podobnou kvalitu, tedy aby kvality čtyř částí byly co nejbližší. Proto musí zvolit polohu tří dělících cest podle tohoto požadavku.



Obrázek 3: Příklad rozdělení sadu podle požadavků Kvida. Hodnoty udávají kvalitu stromů. Kvality čtyř částí (po směru hodinových ručiček od severozápadu) jsou 10, 8, 13, 11. Maximální rozdíl kvality mezi libovolnými dvěma částmi je 5.

Úloha

Dostanete kvalitu každého stromu v sadu. Úkolem je rozdělit sad podle výše popsaného postupu tak, aby rozdíl mezi kvalitou nejkvalitnější části a nejméně kvalitní části byl co nejmenší.

Vstup

Vstup obsahuje několik řádků popisujících sad. První řádek obsahuje jediné celé číslo N, které udává velikost sadu. V sadu je $N \times N$ stromů. Každý z následujících N řádků obsahuje N celých čísel oddělených mezerou, přičemž každé číslo udává kvalitu jednoho stromu. Pořadí řádků i hodnot odpovídá přesně pořadí stromů v sadu.

Všechny kvality stromů jsou celá čísla z intervalu $\langle -10^6, 10^6 \rangle$. Hodnota N nepřekročí 3025.

Výstup

Výstup obsahuje jedno celé číslo, které představuje minimální možný rozdíl mezi kvalitou nejkvalitnější a nejméně kvalitní části sadu podle Kvidova rozdělení. Hodnota výstupu je nezáporná a nepřekročí 10^8.

Příklad 1

Vstup

```
5
 3   0   2  -8  -8
 5   3   2   2   3
 2   5   2   1   4
 3   4  -1   4   2
-3   6   2   4   3
```

Výstup

```
5
```

Příklad 1 odpovídá situaci z Obrázku 3.

Příklad 2

Vstup

```
6
1  2  3   4   5   6
2  4  6   8  10   8
3  6  9  12   9   6
4  8 12   8   4   0
5 10  5   0  -5 -10
6  0 -6 -12 -18 -24
```

Výstup

```
16
```

Sad bude rozdělen podle následujícího schématu, kvality čtyř částí (po směru hodinových ručiček od severozápadu) jsou: 30, 29, 14, 18.

```
  1   2   3   4|  5   6
  2   4   6   8| 10   8
.......................
  3|  6   9  12   9   6
  4|  8  12   8   4   0
  5| 10   5   0  -5 -10
  6|  0  -6 -12 -18 -24
```

Příklad 3

Vstup

```
8
100 -1 -2 -3 -4 -5 -6 -7
 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1
 30 -1 -1 -1 -1 -1 -1 -30
```

Výstup

```
67
```

Sad bude rozdělen podle následujícího schématu, kvality čtyř částí (po směru hodinových ručiček od severozápadu) jsou: 37, -13, -30, 24.

```
100  -1  -2  -3  -4  -5  -6| -7
 -1  -1  -1  -1  -1  -1  -1| -1
 -1  -1  -1  -1  -1  -1  -1| -1
 -1  -1  -1  -1  -1  -1  -1| -1
 -1  -1  -1  -1  -1  -1  -1| -1
 -1  -1  -1  -1  -1  -1  -1| -1
 -1  -1  -1  -1  -1  -1  -1| -1
...............................
 30  -1  -1  -1  -1  -1  -1|-30
```

Rozbor řešení

Klíčové je pozorování, že pro zafixovanou cestu od východu k západu lze hledat optimální rozdělení severní i jižní části nezávisle na sobě – v obou případech hledáme takové rozdělení, které minimalizuje rozdíl mezi kvalitou levé a pravé části.

Pozorování dokážeme formálně pro severní část. Předpokládejme, že celá má kvalitu S. Uvažujme optimální rozdělení severní části na levou část kvality a a pravou část kvality b. Bez újmy na obecnosti předpokládejme, $a \geq b$. Nechť jiné rozdělení severní části má kvality c, d, kde $c \geq d$. Z optimality plyne

$$a - b \leq c - d\,.$$

Vzhledem k tomu, že

$$a + b = S, \quad c + d = S,$$

obdržíme dosazením do nerovnosti

$$a - S + a \leq c - S + c \Rightarrow a \leq c$$

a analogicky

$$S - b - b \leq S - d - d \Rightarrow b \geq d\,.$$

Z nerovností $d \leq b \leq a \leq c$ již plyne, že rozdělení severní části s kvalitami c, d nevede ke zmenšení intervalu, který obsahuje kvality všech čtyř částí.

Nyní můžeme navrhnout algoritmus využívající prefixové součty v řádcích a sloupcích, který postupně zkouší všechny možné řádkové hranice. Pro každou hranici pak provede následující:

1. Upraví částečné součty sloupců:
   - `NorthColSum[j]` obsahuje kvalitu stromů v severní části až po aktuální řádek pro sloupec j.
   - `SouthColSum[j]` obsahuje kvalitu stromů v jižní části (od řádku $i+1$ dolů) pro sloupec j.
2. Hledá optimální svislé rozdělení pro severní část:
   - Postupně posouváme svislou hranici zleva doprava.
   - Aktualizujeme součty NW a NE částí.
   - Sledujeme rozdíl $|NW - NE|$ a ukládáme nejlepší dělení (minimální rozdíl).
3. Stejně postupujeme pro jižní část (SW, SE).
4. Pro aktuální řádkové rozdělení vypočítáme rozdíl mezi největší a nejmenší kvalitou čtyř částí a případně aktualizujeme celkový minimální rozdíl `mindiff`.

Před zkoušením všech řádků se provede předzpracování:

- `rowsSum[i]` = součet stromů v řádku i.
- Prefixový součet `rowsSum` umožňuje rychlé získání součtu severní části až po řádek i.
- Celkový součet `allsum` = součet všech stromů.

Díky prefixovým součtům a postupnému posouvání hranic je možné pro každou řádku určit optimální svislé rozdělení v čase $O(N)$, místo zkoušení všech možností dvojic sloupců. Výsledná složitost celého algoritmu je proto $O(N^2)$.

Java kód

```java
package alg;

import java.io.*;

public class OrchardDivision {

    public static class Orchard {
        public int N;
        public int[][] matrix;
```

```
10          public long[] NorthColSum;
11          public long[] SouthColSum;
12          public long[] rowsSum;
13          public long allsum;
14          public long mindiff;
15
16          public Orchard(int n) {
17              this.N = n;
18              this.NorthColSum = new long[n];
19              this.SouthColSum = new long[n];
20              this.rowsSum = new long[n];
21              this.allsum = 0;
22              this.mindiff = Long.MAX_VALUE;
23          }
24
25          public void load(BufferedReader reader) throws IOException {
26              matrix = new int[N][N];
27              for (int i = 0; i < N; i++) {
28                  String[] tokens = reader.readLine().trim().split("\\s+");
29                  for (int j = 0; j < N; j++) {
30                      matrix[i][j] = Integer.parseInt(tokens[j]);
31                  }
32              }
33          }
34
35          public void preprocess() {
36              for (int iRow = 0; iRow < N; iRow++) {
37                  for (int jCol = 0; jCol < N; jCol++) {
38                      SouthColSum[jCol] += matrix[iRow][jCol];
39                      rowsSum[iRow] += matrix[iRow][jCol];
40                  }
41              }
42              for (int iRow = 1; iRow < N; iRow++) {
43                  rowsSum[iRow] += rowsSum[iRow - 1];
44              }
45              allsum = rowsSum[N - 1];
46          }
47
48          public void compute() {
49              for (int iRow = 0; iRow < N - 1; iRow++) {
50
51                  // update partial sums
52                  for (int jCol = 0; jCol < N; jCol++) {
53                      NorthColSum[jCol] += matrix[iRow][jCol];
54                      SouthColSum[jCol] -= matrix[iRow][jCol];
55                  }
56
57                  // northern part
58                  long NWsum = NorthColSum[0];
59                  long optNWsum = NWsum;
60                  long NEsum = rowsSum[iRow] - NWsum;
61                  long optNEsum = NEsum;
62                  long northMinDiff = Math.abs(NWsum - NEsum);
63                  for (int jCol = 1; jCol < N - 1; jCol++) {
64                      NWsum += NorthColSum[jCol];
65                      NEsum -= NorthColSum[jCol];
66                      long diff = Math.abs(NWsum - NEsum);
67                      if (diff < northMinDiff) {
68                          northMinDiff = diff;
69                          optNWsum = NWsum;
70                          optNEsum = NEsum;
71                      }
72                  }
73
74                  // southern part
75                  long SWsum = SouthColSum[0];
```

```
                long optSWsum = SWsum;
                long SEsum = (allsum - rowsSum[iRow]) - SWsum;
                long optSEsum = SEsum;
                long southMinDiff = Math.abs(SWsum - SEsum);
                for (int jCol = 1; jCol < N - 1; jCol++) {
                    SWsum += SouthColSum[jCol];
                    SEsum -= SouthColSum[jCol];
                    long diff = Math.abs(SWsum - SEsum);
                    if (diff < southMinDiff) {
                        southMinDiff = diff;
                        optSWsum = SWsum;
                        optSEsum = SEsum;
                    }
                }

                // evaluate current row
                long rowOptimum = Math.max(Math.max(optSWsum, optSEsum),
                                           Math.max(optNWsum, optNEsum))
                                - Math.min(Math.min(optSWsum, optSEsum),
                                           Math.min(optNWsum, optNEsum));
                mindiff = Math.min(mindiff, rowOptimum);
            }

            System.out.println(mindiff);
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(reader.readLine().trim());
        Orchard orch = new Orchard(n);
        orch.load(reader);
        orch.preprocess();
        orch.compute();
    }
}
```

4 Nanoroboty shromažďují vzorky

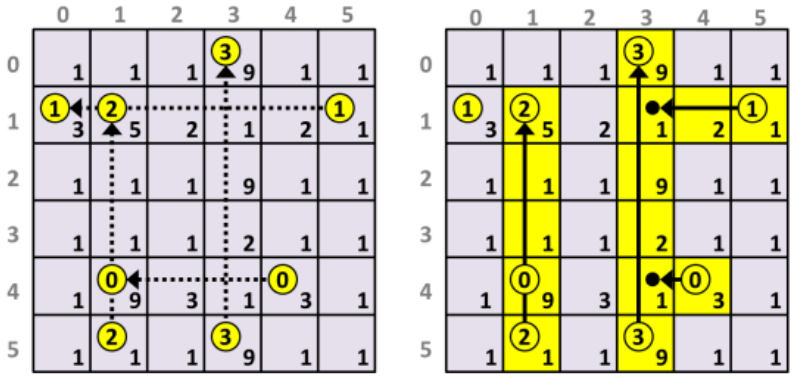
Pokusné obdélníkové pole je rozděleno do $M \times N$ čtvercových sektorů. V každém sektoru je umístěno několik vzorků. Pokusné nanoroboty, které mají za úkol vzorky shromažďovat, pracují v poli podle pravidel uvedených dále.

Každý nanorobot má určen startovní a koncový sektor. Koncový sektor vždy leží buď ve stejném řádku, nebo ve stejném sloupci jako startovní sektor. Úkolem nanorobota je projít všemi sektory na přímé dráze ze startovního do koncového sektoru a shromáždit všechny vzorky ve všech navštívených sektorech. Když nanorobot shromáždí vzorky ve svém cílovém sektoru, zastaví se a nepokračuje v žádné činnosti. Je zaručeno, že jakmile nanorobot začne pracovat v určitém sektoru, shromáždí v něm všechny vzorky.

Činnost kteréhokoli nanorobota v kterémkoli sektoru kontaminuje prostředí sektoru natolik, že když do sektoru navštíveného dříve jedním nanorobotem vstoupí další nanorobot, ztrácí tento druhý nanorobot orientaci, zastaví se a už nepokračuje v pohybu a shromažďování dalších vzorků.

Nanoroboty jsou do svých startovních sektorů umisťovány jeden po druhém. Další nanorobot je do svého startovního sektoru umístěn až poté, co se předchozí nanorobot zastaví. Nezáleží přitom, zda se předchozí nanorobot zastavil po shromáždění vzorků ve svém koncovém sektoru nebo po vstupu do některého kontaminovaného sektoru. Pokud nanorobot kontaminuje svou činností startovní pole jiného nanorobota, který bude umístěn do pole později, nevykoná tento další nanorobot již žádnou činnost a neshromáždí žádné vzorky.

Experimentátoři chtějí nanoroboty do pokusného pole umisťovat v takovém pořadí, aby maximalizovali celkový počet vzorků shromážděných všemi nanoroboty.



Obrázek 4: Obrázek odpovídá situaci z Příkladu 1 níže. V pravém dolním rohu každého ze 36 sektorů je vepsán počet vzorků, které se v něm původně nacházejí. V levém schématu je přerušovanými čarami znázorněna možná dráha každého nanorobota z jeho startovního sektoru do jeho cílového sektoru. Nanoroboty jsou označeny čísly 0, 1, 2, 3. V pravém schématu jsou plnými čarami znázorněny dráhy nanorobotů pro každé pořadí, ve kterém nanoroboty 0 a 1 budou umístěny do pole až po nanorobotu 3. Všechny navštívené sektory jsou zvýrazněny žlutě, celkový počet shromážděných vzorků z těchto sektorů je 54 a je maximální možný. (Černé body představují vstup nanorobotů 0 a 1 do sektoru kontaminovaného nanorobotem 3.)

Úloha

Je znám počet vzorků v každém sektoru pole a je znám startovní a koncový sektor každého robota. Určete maximální možný celkový počet vzorků, které mohou nanoroboty shromáždit, pokud budou do pole umisťovány ve vhodném pořadí.

Vstup

První řádek vstupu obsahuje dvě celá čísla M, N oddělená mezerou, určující počet sektorů podél vertikální a horizontální hrany pole. Předpokládáme, že souřadnice jednotlivých sektorů jsou určeny jejich vertikální a horizontální vzdáleností (v tomto pořadí) od sektoru v levém horním rohu pole, který má souřadnice $(0, 0)$. Následuje M řádků, každý s N hodnotami, určujícími počet vzorků v jednotlivých sektorech. Pokud bychom tyto řádky vstupu indexovali od 0 a hodnoty na každém řádku rovněž od 0, pak by j-tá hodnota v i-tém řádku představovala počet vzorků v sektoru se souřadnicemi (i, j). Dále je na vstupu řádek s jediným číslem R představujícím počet nanorobotů.

Následuje R řádků, přičemž každý obsahuje čtyři čísla a, b, c, d oddělená mezerou, popisující startovní a cílový sektor jednoho z nanorobotů. Startovní sektor má souřadnice (a, b), cílový sektor má souřadnice (c, d), kde $a = c$ nebo $b = d$.

Platí $2 \leq M, N \leq 50$, $2 \leq R \leq 10$, počet vzorků v jednom sektoru je nejvýše 1000.

Výstup

Na výstupu je jeden textový řádek s jedním číslem představujícím maximální možný celkový počet vzorků, které mohou nanoroboty shromáždit, když budou do pole umisťovány v co nejvýhodnějším pořadí.

Příklad 1

Vstup

```
6 6
1 1 1 9 1 1
3 5 2 1 2 1
1 1 1 9 1 1
1 1 1 2 1 1
1 9 3 1 3 1
1 1 1 9 1 1
4
4 4 4 1
1 5 1 0
5 1 1 1
5 3 0 3
```

Výstup

```
54
```

Příklad 2

Vstup

```
4 5
10 10 50 10 90
10 10 40 90 10
20 30 60 50 20
10 10 80 90 90
2
2 0 2 4
0 2 3 2
```

Výstup

```
280
```

Příklad 3

Vstup

```
2 9
1 3 4 5 3 1 2 1 9
5 3 4 5 3 7 7 2 1
5
0 1 1 1
0 2 1 2
0 3 1 3
0 4 1 4
1 1 1 7
```

Výstup

```
46
```

Rozbor řešení

Pro nalezení nejvýhodnějšího pořadí, v jakém mají být nanoroboti do pole umisťováni, použijeme metodu prohledávání s návratem (backtracking), doplněnou o heuristiku a ořezávání stavového prostoru.

Nejprve nanoroboty seřadíme podle počtu vzorků, které mohou samostatně shromáždit, pokud by byli aktivováni jako první. Tento počet určíme průchodem jejich plánované trasy a sečtením vzorků v navštívených sektorech. Roboty pak zkoušíme umisťovat v pořadí od nejvýnosnějších k méně výnosným. Například v situaci z Obrázku 4 je výsledné pořadí 3, 2, 0, 1, protože odpovídající maximální výnosy jsou 31, 17, 3 a 3.

Rekurzivní procedura má podobu `search(robots, score)`, kde `robots` je počet dosud aktivovaných nanorobotů a `score` aktuální celkový počet shromážděných vzorků. Jakmile aktivujeme všech N robotů, aktualizujeme dosud nejlepší známý výsledek `bestScore`.

Pro efektivní prohledávání využíváme horní odhad maximálního počtu vzorků, které lze ještě získat. Označme A množinu již aktivovaných nanorobotů a B množinu zbývajících. Výsledek `score` rozšíříme o optimistický příspěvek robotů z B, přičemž předpokládáme, že každý z nich začne pohyb hned po robotech z A. Ignorujeme přitom možné kolize uvnitř množiny B. Každý nanorobot z B sbírá vzorky, dokud nenarazí na sektor kontaminovaný robotem z A, nebo dokud nedojde do svého cílového sektoru. Výpočet tohoto odhadu je rychlý a nevyžaduje rekurzi. Díky odhadu můžeme předčasně ukončit větve prohledávání, které již nemohou vést k lepšímu řešení.

Uveďme konkrétní příklad, opět vycházející z Obrázku 4. Pokud aktivujeme nanorobota 0 jako prvního, shromáždí 16 vzorků a zároveň kontaminuje část pole. Pro zbývající nanoroboty můžeme spočítat optimistický odhad jejich výnosů jako $1 + 9 + 14 = 24$. Horní odhad celkového výtěžku tak bude $16 + 24 = 40$, což je méně než maximum 54 dosažené při pořadí navrženém heuristikou.

Java kód

```java
package alg;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.StringTokenizer;

public class Nanorobots {
    public static int M, N; // number of rows and columns in the 2D grid of sectors
    public static int R; // number of nanorobots
    public static int[][] samples; // number of samples available in each sector
    public static Robot[] robots; // array of all nanorobots

    public static int[][] sectorMarks; // for each sector, stores the ID of the robot that
        visited it
    public static int bestScore; // best total number of collected samples found so far

    public static class Robot implements Comparable {

        public int id; // nanorobot ID (from 1 to R)
        public boolean activated = false; // set to true when the nanorobot is activated
        public int startRow, startCol, endRow, endCol; // coordinates defining the robot's
            trajectory
        public int deltaRow, deltaCol; // unit direction vector from start sector to end
            sector
        public int maxSamples; // maximum number of samples this robot can collect (
            assuming no interference)

        public Robot(int id, int start_row, int start_col, int end_row, int end_col) {
            this.id = id;
            startRow = start_row;
            endRow = end_row;
            startCol = start_col;
            endCol = end_col;
            if (startRow == endRow) {
```

```
34                deltaRow = 0;
35                deltaCol = startCol < endCol ? 1 : -1;
36            } else {
37                deltaCol = 0;
38                deltaRow = startRow < endRow ? 1 : -1;
39            }
40            maxSamples = countReachableSamples(false);
41        }
42
43        /**
44         * @param pickUp If true, the robot is activated and collects all reachable samples
                .
45         *               If false, only the number of reachable samples is computed without
               activation.
46         * @return The number of samples the robot collects or could collect.
47         */
48        public final int countReachableSamples(boolean pickUp) {
49            int samplesToCollect = 0;
50            int row = startRow, col = startCol;
51            while (sectorMarks[row][col] == 0) {
52                samplesToCollect += samples[row][col];
53                if (pickUp)
54                    sectorMarks[row][col] = id;
55                if (row == endRow && col == endCol)
56                    break;
57                // move to the next sector
58                row += deltaRow;
59                col += deltaCol;
60            }
61            return samplesToCollect;
62        }
63
64        /**
65         * Reverts the contamination caused by this nanorobot (undoes its activation).
66         */
67        public final void undo() {
68            int row = startRow, col = startCol;
69            while (sectorMarks[row][col] == id) {
70                sectorMarks[row][col] = 0;
71                if (row == endRow && col == endCol)
72                    break;
73                // move to the next sector
74                row += deltaRow;
75                col += deltaCol;
76            }
77        }
78
79        @Override
80        public int compareTo(Object o) {
81            return -maxSamples + ((Robot) o).maxSamples; // descending order by maxSamples
82        }
83    }
84
85    /**
86     * @param r    Number of robots activated so far.
87     * @param score Total number of samples collected by the activated robots.
88     */
89    public static void search(int r, int score) {
90        if (r == robots.length) {
91            // no more robots to activate
92            if (score > bestScore)
93                bestScore = score; // a better solution was found
94            return;
95        }
96
97        // compute an upper bound on the score we can still achieve
```

```
        int scoreUpperBound = score;
        for (Robot robot : robots) {
            if (!robot.activated) {
                scoreUpperBound += robot.countReachableSamples(false);
            }
        }
        if (scoreUpperBound <= bestScore)
            return; // ***PRUNING***: stop if no better result is possible

        // try activating one more robot
        for (Robot robot : robots) {
            if (!robot.activated) {
                robot.activated = true;
                int robotScore = robot.countReachableSamples(true); // mark visited sectors
                search(r + 1, score + robotScore); // recurse
                robot.activated = false;
                robot.undo(); // restore sector state
            }
        }
    }

    public static void loadData(String fileName) throws IOException {
        BufferedReader reader = fileName == null
            ? new BufferedReader(new InputStreamReader(System.in))
            : new BufferedReader(new FileReader(fileName));

        StringTokenizer tokenizer = new StringTokenizer(reader.readLine());
        M = Integer.parseInt(tokenizer.nextToken());
        N = Integer.parseInt(tokenizer.nextToken());

        samples = new int[M][N];
        sectorMarks = new int[M][N];

        for (int row = 0; row < M; row++) {
            tokenizer = new StringTokenizer(reader.readLine());
            for (int col = 0; col < N; col++) {
                samples[row][col] = Integer.parseInt(tokenizer.nextToken());
            }
        }

        R = Integer.parseInt(reader.readLine().trim());
        robots = new Robot[R];

        for (int r = 0; r < R; r++) {
            tokenizer = new StringTokenizer(reader.readLine());
            int start_row = Integer.parseInt(tokenizer.nextToken());
            int start_col = Integer.parseInt(tokenizer.nextToken());
            int end_row = Integer.parseInt(tokenizer.nextToken());
            int end_col = Integer.parseInt(tokenizer.nextToken());
            robots[r] = new Robot(r + 1, start_row, start_col, end_row, end_col);
        }
    }

    public static void main(String[] args) throws IOException {
        loadData(null);
        Arrays.sort(robots); // ***HEURISTIC***: search state space in this order
        bestScore = 0;
        search(0, 0);
        System.out.println(bestScore);
    }

}
```

5 Průzkum povodí v deštném pralese

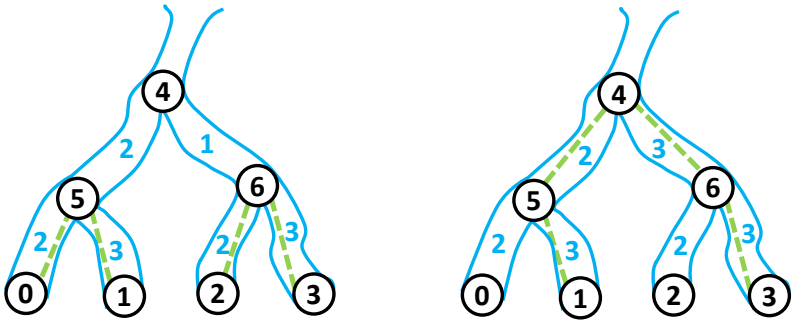
Společnost Ekosystems Ltd. se chystá provést floristický průzkum v povodí velké řeky v deštném pralese. Kvůli specifikům prostředí chce na tuto práci najmout a zaškolit domorodce.

Povodí sestává z úseků. Průzkum bude proveden pouze na některých z nich. Úsekem se rozumí část řeky od pramene k nejbližšímu soutoku s jinou řekou, nebo od soutoku k bezprostředně následujícímu soutoku. Mezi úseky tak není zařazena část hlavní řeky tekoucí od posledního soutoku k jejímu ústí do moře. To však nevadí, protože příslušná oblast byla v minulosti již dobře zmapována. Platí, že v každém soutoku se v povodí setkávají právě dvě řeky.

K průzkumu bude z domorodců sestaveno několik samostatně pracujících skupin. Každá skupina prozkoumá sekvenci na sebe navazujících úseků řek, začínající u pramene nějaké řeky a končící u pramene jiné řeky. O domorodcích je známo, že při zvolené organizaci práce panuje mezi skupinami velká rivalita. Z tohoto důvodu společnost musí přidělit skupinám po dvou navzájem disjunktní trasy, aby se při práci v terénu členové z různých skupin nemohli potkat. Dvě trasy jsou disjunktní, pokud nesdílejí žádný úsek.

Úloha

Je dána struktura úseků povodí a jejich délky. Určete, jaká je maximální sumární délka úseků, které mohou sestavené týmy prozkoumat, za předpokladu, že na počet týmů není žádné omezení.



(a) (b)

Obrázek 5: (a) Schéma povodí se třemi soutoky (identifikátory 4 až 6) a čtyřmi prameny (identifikátory 0 až 3). Modrá čísla udávají délky jednotlivých úseků. Celková délka prozkoumaných úseků je maximalizována, pokud dva týmy prozkoumají trasy znázorněné zelenými přerušovanými úsečkami (jeden tým trasu 0-5-1, druhý tým trasu 2-6-3). (b) Podobná situace, pouze s odlišnou délkou úseku mezi soutoky 4 a 6. Celková délka prozkoumaných úseků je maximalizována, pokud je sestaven pouze jeden tým, který prozkoumá úseky podél znázorněné trasy.

Vstup

První řádek vstupu obsahuje dvě celá čísla P a S, oddělená mezerou, kde P je počet pramenů a S počet soutoků v povodí.

Následuje S řádků, kde každý řádek reprezentuje jeden soutok pomocí pěti celých čísel I, I_1, D_1, I_2 a D_2 oddělených mezerami. I je identifikátor reprezentovaného soutoku, I_1 a I_2 jsou identifikátory bezprostředních soutoků nebo pramenů, ze kterých do soutoku I přitékají dvě řeky, D_1 je délka úseku mezi I a I_1, a analogicky, D_2 je délka úseku mezi I a I_2. Identifikátory pramenů a soutoků jsou navzájem různá celá čísla od 0 do $P + S - 1$. Jako první je reprezentovaný soutok, který je nejblíže k ústí hlavní řeky do moře. Pořadí dalších soutoků může být na vstupu libovolné.

Platí $3 \leq P + S \leq 5.5 \times 10^6$. Délka každého z úseků je vždy celé číslo od 1 do 200.

Výstup

Výstup obsahuje jeden textový řádek s jedním celým číslem, jež je rovno maximální možné sumární délce úseků, které mohou skupiny za daných podmínek v povodí prozkoumat.

Příklad 1

Vstup

```
4 3
4 5 2 6 1
5 0 2 1 3
6 2 2 3 3
```

Výstup

```
10
```

Data a řešení Příkladu 1 můžeme vidět na **Obrázku** 1a).

Příklad 2

Vstup

```
4 3
4 5 2 6 3
5 0 2 1 3
6 2 2 3 3
```

Výstup

```
11
```

Data a řešení Příkladu 2 můžeme vidět na **Obrázku** 1b).

Rozbor řešení

Na vstupu je dán binární zakořeněný strom T s P listy a S vnitřními uzly. Hrany jsou ohodnocené kladným číslem. Požaduje se nalézt množinu po dvou uzlově disjunktních cest, kde každá cesta spojuje dva různé listy. Zároveň je cílem vybrat takové cesty, jejichž součet ohodnocení bude maximální.

Úlohu řešíme pomocí průchodu stromem, při kterém pro každý uzel v spočítáme dvě hodnoty (nechť T_v označuje podstrom zakořeněný ve v):

- `openSum` je hodnota optimálního dílčího řešení v T_v, které sestává z cest z listu do listu v T_v a kromě toho ještě z části cesty, která vede z listu T_v přes v do listu mimo T_v; do `openSum` započítáme cenu každé vybrané hrany v T_v,
- `closedSum` je hodnota optimálního dílčího řešení v T_v, které sestává z cest z listu do listu v T_v. V tomto případě žádná vybraná cesta nevede přes v ven z T_v.

Předpokládejme, že v.`leftCost` označuje cenu hrany z v do levého potomka, a podobně, v.`rightCost` označuje cenu hrany z v do pravého potomka. Potom platí:

```
v.openSum = max(v.left.openSum + v.leftCost + v.right.closedSum,
    v.left.closedSum + v.right.openSum + v.rightCost)

v.closedSum = max(v.left.closedSum + v.right.closedSum,
    v.left.openSum + v.right.openSum + v.leftCost + v.rightCost)
```

Hledaným výsledkem je hodnota `closedSum` vypočítaná pro kořen.

Vzhledem k tomu, že pro každý uzel proběhne výpočet pomocných hodnot v konstantním čase, časová složitost algoritmu odpovídá časové složitosti průchodu grafem a je tudíž rovna $\mathcal{O}(P + S)$.

Java kód

```
1  package alg;
2
3  import java.io.*;
4  import java.util.*;
5
6  public class WatershedSurvey {
7
8      static int N, I;
9      static Node[] nodes;
```

```
    static int rootId;

    public static class Node {
        int id;
        Node left = null;
        Node right = null;
        int leftCost, rightCost;
        int openSum, closedSum;

        Node(int id) {
            this.id = id;
        }
    }

    public static void dfs(Node node) {
        if (node.left == null) { // leaf
            node.openSum = node.closedSum = 0;
            return;
        }
        // inner node
        dfs(node.left);
        dfs(node.right);
        node.closedSum = Math.max(node.left.closedSum + node.right.closedSum,
                node.left.openSum + node.right.openSum + node.leftCost + node.rightCost);
        node.openSum = Math.max(node.left.openSum + node.leftCost + node.right.closedSum,
                node.left.closedSum + node.right.openSum + node.rightCost);
    }

    public static void loadData() throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        StringTokenizer tokenizer = new StringTokenizer(reader.readLine());
        N = Integer.valueOf(tokenizer.nextToken());
        I = Integer.valueOf(tokenizer.nextToken());
        N += I;

        nodes = new Node[N];
        for (int i = 0; i < N; i++)
            nodes[i] = new Node(i);

        rootId = -1;
        for (int i = 0; i < I; i++) {
            tokenizer = new StringTokenizer(reader.readLine());
            int id = Integer.valueOf(tokenizer.nextToken());
            Node node = nodes[id];
            node.left = nodes[Integer.valueOf(tokenizer.nextToken())];
            node.leftCost = Integer.valueOf(tokenizer.nextToken());
            node.right = nodes[Integer.valueOf(tokenizer.nextToken())];
            node.rightCost = Integer.valueOf(tokenizer.nextToken());
            if (rootId < 0)
                rootId = id;
        }
    }

    public static void main(String[] args) throws IOException {
        loadData();
        dfs(nodes[rootId]);
        System.out.println(nodes[rootId].closedSum);
    }

}
```

6 Počítačová hra

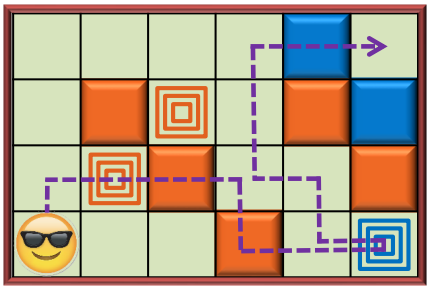
V počítačové hře ovládáme postavičku zvanou Tom, která se pohybuje po čtvercové mřížce. Naším úkolem je pomocí přesunů vlevo, vpravo, nahoru a dolů (ne diagonálně) přemístit Toma z levého dolního rohu do pravého horního rohu. Každý přesun Toma na sousední pole nazýváme tahem.

V pohybu Tomovi brání barevné překážky (zdi), které se na některých polích mřížky vyskytují. Zároveň některá pole mřížky fungují jako spínače, které všechny zdi jedné barvy, totožné s barvou spínače, zasunou do země. Pole s takovými překážkami se změní na průchozí a zůstávají v tomto stavu, dokud Tom nevstoupí na spínač jiné barvy B. Když se tak stane, původně zasunuté zdi se opět vysunou a zasunou se zdi barvy B.

Platí, že na začátku jsou všechny zdi vysunuté. Vysunutí a zasunutí zdí je okamžité. Tom se celým svým objemem vejde do jednoho pole. Hrací plochu nemůže opustit. Pokud Tom vstoupí opakovaně na spínač jedné barvy, aniž by mezitím navštívil spínač jiné barvy, nic se nestane. Na výchozím ani cílovém poli není zeď ani spínač. Spínač se nevyskytuje na poli se zdí.

Úloha

Pro danou hrací plochu nalezněte nejkratší posloupnost tahů, která přemístí Toma z výchozí pozice do pozice koncové.



Obrázek 6: Příklad hrací plochy rozměrů 4×6. Tom je ve své výchozí pozici znázorněn smajlíkem. Jsou přítomné překážky dvou barev - oranžové a modré. Plochy se spínači jsou znázorněné pomocí tří soustředných čtverců stejné barvy. Fialová přerušovaná trasa reprezentuje možný optimální postup, který přemístí Toma s použitím 14-ti tahů do cílové pozice. Vidíme, že po vykonání druhého tahu Tom vstoupí na pole se spínačem oranžové barvy, proto se může ve třetím a pátém tahu přesunout na pole se zasunutou oranžovou zdí. Vzhledem k tomu, že přístup k pravému hornímu rohu blokují dvě modré zdi, je potřeba navštívit pole s modrým spínačem. To způsobí, že se modré zdi zasunou, ale oranžové se vysunou a je nezbytné je během zbývajících tahů obejít.

Vstup

Na prvním vstupním řádku jsou tři celá čísla M, N, C oddělená mezerami. Číslo M, resp. N je počet řádků, resp. sloupců čtvercové mřížky reprezentující hrací plochu. Barvy překážek mají identifikátory 1 až C.

Následuje M řádků vstupu, kde i-tý z těchto řádků reprezentuje i-tý řádek čtvercové mřížky pomocí N celých čísel oddělených mezerami, přičemž číslo 0 reprezentuje prázdné pole, kladné číslo od 1 do C reprezentuje barvu zdi na daném poli a záporné číslo od $-C$ do -1 reprezentuje barvu spínače na daném poli; spínač, jehož barva má identifikátor B, je reprezentovaný číslem opačným k B, tedy $-B$.

Je zaručena dosažitelnost cílového pole.

Platí $1 \leq M, N \leq 1000$, $2 \leq C \leq 10$.

Výstup

Výstup sestává z jednoho textového řádku, který obsahuje číslo, jež je rovno minimálnímu počtu tahů potřebných na přesun Toma z levého dolního rohu do pravého horního rohu.

Příklad 1

Vstup

```
4 6 2
 0  0  0  0  1  0
 0  2 -2  0  2  1
 0 -2  2  0  0  2
 0  0  0  2  0 -1
```

Výstup

```
14
```

Příklad 2

Vstup

```
6 8 2
-1  1  0  0 -1  0  2  0
-1  1 -2  1  0  0  2 -2
 0  1  0  1  1  0  2  0
 0  0  0  1  0  0  2  1
 0  0  0  1  0  2  2  1
 0  0  0  1  0  0  0  0
```

Výstup

```
20
```

Rozbor řešení

Úloha lze převést na prohledávání orientovaného grafu, který je uspořádán do $C + 1$ vrstev číslovaných od 0 do C. Vrstva 0 představuje herní plochu ve výchozím stavu, kdy všechny spínače jsou neaktivní a všechny zdi jsou vysunuté. Dále pak, pro $1 \leq c \leq C$, vrstva c reprezentuje herní plochu, na které jsou zasunuty všechny zdi barvy c a ostatní zdi jsou vysunuté. Každá vrstva obsahuje $M \times N$ vrcholů, přičemž každý vrchol odpovídá jednomu poli mřížky.

Orientované hrany mezi vrcholy reprezentují přípustné pohyby Toma mezi sousedními poli mřížky, stejně jako případnou aktivaci spínače. Nechť $v(P, i)$ označuje vrchol grafu, který reprezentuje pole P ve vrstvě i. Pokud jsou P_1 a P_2 sousední pole mřížky, ve vrstvě i přidáme hranu z $v(P_1, i)$ do $v(P_2, i)$, pokud na polích P_1 a P_2 není zeď jiné barvy než i a pokud na poli P_2 není spínač jiné barvy než i. V případě, že se na poli P_2 nachází spínač barvy $j \neq i$ (a zároveň na P_1 není zeď jiné barvy než i), přidáme hranu z $v(P_1, i)$ do $v(P_2, j)$, čímž reprezentujeme aktivaci spínače posunem mezi vrstvami.

Ve zkonstruovaném grafu hledáme nejkratší cestu, která začíná ve vrcholu reprezentujícím výchozí pozici ve vrstvě 0 a končí vrcholem v libovolné vrstvě reprezentujícím koncové pole.

Pro zjištění nejkratší cesty v grafu aplikujeme prohledávání do šířky (BFS) z výchozí pozice za použití fronty. Samotný graf není potřeba explicitně konstruovat; stačí uchovávat matici herní plochy a alokovat třírozměrné pole `distances`, kde `distances[r][s][c]` reprezentuje nejkratší vzdálenost od počátku do vrcholu ve vrstvě c, který odpovídá poli herní plochy na řádku r a sloupci s.

Do fronty přidáváme trojice (r, s, c), což opět odpovídá poloze v herní ploše a vrstvě. Všechny hodnoty `distances` jsou na začátku -1 ("nenavštíveno"), kromě `distances[M-1][0][0]`, která je nastavena na 0. Frontu inicializujeme vložením záznamu $(M - 1, 0, 0)$.

BFS končí, jakmile poprvé dosáhneme cílového pole (v libovolné vrstvě). Časová složitost celého algoritmu je $\mathcal{O}(MNC)$.

Java kód

```java
package alg;

import java.io.*;
import java.util.*;

public class ComputerGame {

```

```
    public static int M, N, S;
    public static int[][] maze;
    public static int[][][] grid;

    public static Point[] queue;
    public static int head, tail;

    public static int max_dist;
    public static Point fPoint;

    public static class Point {
        int row, col, sw;
        int total_switches;

        public Point(int row, int col, int sw, int tsw) {
            this.row = row;
            this.col = col;
            this.sw = sw;
            this.total_switches = tsw;
        }
    }

    public static void enqueue(int row, int col, int sw, int tsw) {
        Point p = new Point(row, col, sw, tsw);
        queue[tail] = p;
        tail++;
        if (tail == queue.length)
            tail = 0;
    }

    public static Point dequeue() {
        int index = head;
        head++;
        if (head == queue.length)
            head = 0;
        return queue[index];
    }

    public static boolean isEmptyQueue() {
        return head == tail;
    }

    public static void checkNeighbor(Point p, int drow, int dcol, int dist) {
        int row = p.row + drow;
        int col = p.col + dcol;
        if (row >= 0 && row < M && col >= 0 && col < N) {
            int sw = p.sw;
            int tsw = p.total_switches;
            if (maze[row][col] < 0) {
                sw = -maze[row][col];
                tsw++;
            }
            if (grid[row][col][sw] == 0) { // not yet visited & not a wall (-1)
                grid[row][col][sw] = dist;
                enqueue(row, col, sw, tsw);
            }
        }
    }

    public static int solve() {
        grid = new int [M][N][S+1];
        for (int r = 0; r < M; r++)
            for (int c = 0; c < N; c++) {
                if (maze[r][c] > 0) {
                    for (int s = 0; s <= S; s++)
                        if (maze[r][c] != s)
```

```
                        grid[r][c][s] = -1;
                }
            }
        max_dist = 0;
        queue = new Point[M*N*(S+1)];
        head = tail = 0;
        grid[M-1][0][0] = 1; // visited
        enqueue(M-1, 0, 0, 0);
        while (!isEmptyQueue()) {
            Point p = dequeue();
            if (p.row == 0 && p.col == N-1) {
                return grid[p.row][p.col][p.sw]-1;
            }
            int dist = grid[p.row][p.col][p.sw] + 1;
            checkNeighbor(p, -1, 0, dist);
            checkNeighbor(p, 1, 0, dist);
            checkNeighbor(p, 0, -1, dist);
            checkNeighbor(p, 0, 1, dist);
        }
        return -1;
    }

    public static void loadData() throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        StringTokenizer tokenizer = new StringTokenizer(reader.readLine());
        M = Integer.valueOf(tokenizer.nextToken());
        N = Integer.valueOf(tokenizer.nextToken());
        S = Integer.valueOf(tokenizer.nextToken());
        maze = new int[M][N];
        for (int r = 0; r < M; r++) {
            tokenizer = new StringTokenizer(reader.readLine());
            for (int c = 0; c < N; c++)
                maze[r][c] = Integer.valueOf(tokenizer.nextToken());
        }
    }

    public static void main(String[] args) throws IOException {
        loadData();
        System.out.println(solve());
    }

}
```

7 Karavana v poušti

Karavana se v poušti pohybuje mezi vesnicemi, kde každá vesnice je *spřátelená* nebo *neutrální*. Přesun karavany začíná v jedné výchozí vesnici a dále probíhá po *trasách*. Trasa vždy spojuje dvě vesnice a karavana se po ní může pohybovat v obou směrech.

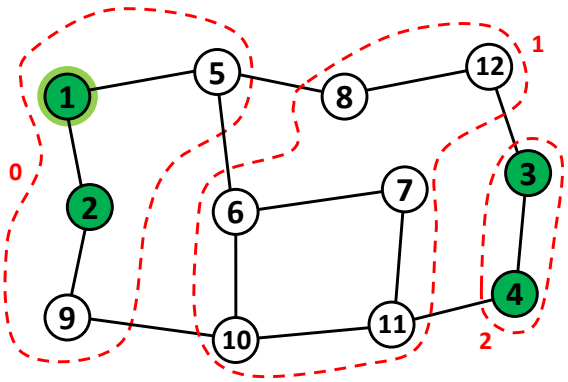
Ve spřátelených vesnicích je o karavanu dobře postaráno, co se týče proviantu. Když se karavana nachází ve spřátelené vesnici, je hodnota *nasycení* karavany vždy rovna danému celému kladnému číslu D. Pokud se karavana po libovolné trase přesune do neutrální vesnice, klesne hodnota nasycení o 1. Jinými slovy, hodnota nasycení karavany v neutrální vesnici V je vždy číslo o 1 menší, než byla hodnota nasycení v předchozí navštívené vesnici, ze které po jedné trase karavana do V docestovala.

Výjimkou je pouze situace, kdy je hodnota nasycení v libovolné neutrální vesnici V_1 rovna 0. V tomto případě karavana během pohybu po další trase z vesnice V_1 do nějaké vesnice V_2 musí nutně spotřebovat jeden balík přídavných zásob, bez ohledu na to, zda je V_2 spřátelená nebo neutrální. Výsledkem takového přesunu se spotřebováním balíku zásob je to, že ve vesnici V_2 je hodnota nasycení karavany rovna D, bez ohledu na to, zda V_2 je či není neutrální.

Aby karavana dosáhla cílové vesnice, může se pohybovat libovolně po navazujících trasách. To znamená, že stejnou trasu lze v případě potřeby projít i opakovaně, pokud to přináší výhodu. Předpokládáme, že balíky přídavných zásob nelze doplňovat ve vesnicích – karavana jimi musí být vybavena už na začátku cesty.

Úloha

Je dán seznam spřátelených a neutrálních vesnic, seznam tras, číslo D a výchozí vesnice, která je spřátelená. Máme určit, kolik balíků přídavných zásob karavana minimálně potřebuje k tomu, aby byla schopná se přepravit do libovolné cílové vesnice. Zajímá nás také, kolik vesnic představuje pro karavanu cíl dostupný bez použití přídavných zásob.



Obrázek 7: Na obrázku uzly reprezentují vesnice a hrany reprezentují trasy mezi vesnicemi. Výchozí vesnice má číslo 1. Všechny spřátelené vesnice jsou zvýrazněné zeleně. Předpokládáme, že $D = 1$. Potom platí, že bez použití přídavných zásob jsou z výchozí vesnice dosažitelné právě vesnice číslo 1, 2, 5 a 9. Dále, vesnice číslo 6, 7, 8, 10, 11 a 12 jsou dosažitelné z vesnice číslo 1 s použitím jednoho balíku zásob, a konečně vesnice číslo 3 a 4 jsou dosažitelné se dvěma balíky zásob (ne však s jedním). Rozklad množiny vesnic na uvedené podmnožiny je znázorněn pomocí čárkovaných červených hranic.

Vstup

Na prvním vstupním řádku jsou čísla N, M, S, D, kde N je počet vesnic, M je počet tras mezi dvojicemi vesnic, S je počet spřátelených vesnic a D je hodnota nasycení karavany ve spřátelené vesnici. Vesnice jsou číslované od 1 do N. Výchozí vesnice má číslo 1. Spřátelené vesnice mají čísla 1 až S. Následuje M řádků, kde každý řádek reprezentuje jednu trasu pomocí dvojice čísel V_1 a V_2. Tato čísla označují vesnice, které daná trasa napřímo spojuje.

Pro každou dvojici vesnic V_1 a V_2 je na vstupu maximálně jedna trasa. Trasy jsou na vstupu v náhodném pořadí. Předpokládáme, že všechny vesnice jsou z výchozí vesnice dostupné (s případným použitím přídavných zásob).

Platí $1 \leq D \leq 8$, $1 \leq S \leq N$, $1 \leq N \leq 4 \cdot 10^5$, $M < 4N$.

Výstup

Výstup obsahuje jeden řádek, na kterém jsou dvě čísla oddělená mezerou. První číslo je minimální počet balíků přídavných zásob, který postačuje k tomu, aby karavana docestovala z výchozí vesnice do libovolně zvolené cílové vesnice. Druhé číslo je počet vesnic, které jsou pro karavanu dostupné z výchozí vesnice bez použití přídavných zásob. Mezi tyto vesnice počítáme i výchozí vesnici.

Příklad 1

Vstup

```
12 14 4 1
1 5
5 8
8 12
6 7
9 10
10 11
11 4
1 2
2 9
5 6
6 10
7 11
12 3
3 4
```

Výstup

```
2 4
```

Data a řešení Příkladu 1 jsou znázorněna na **Obrázku** 7.

Příklad 2

Vstup

```
10 10 1 1
2 3
7 3
10 2
10 7
4 6
9 2
5 9
1 8
8 5
8 6
```

Výstup

```
3 2
```

Rozbor řešení

Vstup reprezentujeme neorientovaným grafem $G = (V, E)$, kde vrcholy odpovídají vesnicím a hrany trasám mezi nimi. Protože graf je řídký, zvolíme reprezentaci pomocí seznamů sousedů.

Označme V_0 množinu všech vesnic, do kterých se karavana z výchozí vesnice dostane bez spotřebování přídavných zásob. Pro $i \geq 1$ označme V_i množinu všech vesnic, do kterých se karavana dostane při spotřebování právě i balíků přídavných zásob, ale nedostane se do nich s menším počtem balíků.

K nalezení minimálního počtu přídavných balíků, které karavana potřebuje k dosažení všech vesnic, použijeme vícefázové prohledávání grafu do šířky (BFS). Označíme-li fáze $0, 1, 2, \ldots$, pak i-tá fáze detekuje právě množinu V_i. Jakmile po k-té fázi platí

$$V_0 \cup V_1 \cup \cdots \cup V_k = V,$$

tedy že všechny vesnice jsou dosažitelné, algoritmus končí. Výstupem je dvojice čísel $(|V_0|, k)$.

Před spuštěním BFS pro fázi $i+1$ jsou všechny dosud dosažené vesnice označeny jako *uzavřené* a je vytvořen seznam `starts`, který obsahuje všechny dosud neuzavřené sousedy vesnic z množiny V_i. Na začátku nulté fáze obsahuje `starts` pouze výchozí vesnici.

BFS zahájíme s frontou obsahující prvky `starts`. Pro každou dosud neuzavřenou vesnici zjišťujeme, s jakou maximální nasyceností se do ní lze dostat, aniž by byl spotřebován další balík zásob. Vesnici lze tedy během BFS navštívit opakovaně, pokud to vede ke zlepšení této hodnoty; v takovém případě ji znovu zařadíme do fronty.

Po vyprázdnění fronty označíme všechny právě dosažené vesnice jako uzavřené a určíme nové počáteční vesnice pro následující fázi (pokud ještě existují neuzavřené vesnice).

Časová složitost i-té fáze je $\mathcal{O}(D\,|E_i|)$, kde E_i je množina všech tras incidentních s vesnicemi ve V_i. Celý algoritmus má proto složitost $\mathcal{O}(DM)$.

Java kód

```
package alg;

import java.io.*;
import java.util.*;

public class Caravan {

    public static int N, M, D;
    public static Node[] nodes;
    public static LinkedList<Node> starts;
    public static int finishedNodes;

    public static class Node {
        public int id;
        public ArrayList<Integer> edges;
        public int d;
        public boolean isFriendly;
        public boolean isStart;

        public Node(int id, boolean isFriendly) {
            this.id = id;
            this.isFriendly = isFriendly;
            d = Integer.MAX_VALUE;
            edges = new ArrayList<>();
        }

        public void addEdge(int to) {
            edges.add(to);
        }
    }

    public static void bfs(LinkedList<Node> list) {
        while (!list.isEmpty()) {
            Node node = list.removeFirst();
            for (int to : node.edges) {
                Node n = nodes[to - 1];
                if (n.d > node.d + 1) {
                    n.d = n.isFriendly ? 0 : node.d + 1;
                    if (n.d < D)
                        list.addLast(n);
                }
            }
        }
    }

    public static void findStarts(LinkedList<Node> list) {
        while (!list.isEmpty()) {
            Node node = list.removeFirst();
            if (node.d == Integer.MAX_VALUE) {
```

```
                node.d = 0;
                node.isStart = true;
                starts.add(node);
            } else if (node.d != -1 && !node.isStart) {
                node.d = -1;
                finishedNodes++;
                for (int to : node.edges)
                    list.addLast(nodes[Math.abs(to) - 1]);
            }
        }
        for (Node n : starts)
            n.isStart = false;
    }

    public static int[] solve() {
        int reachableNodes = -1;
        finishedNodes = 0;
        starts = new LinkedList<>();
        nodes[0].d = 0;
        starts.add(nodes[0]);
        int count = 0;
        while (!starts.isEmpty()) {
            count++;
            LinkedList<Node> old_starts = (LinkedList<Node>)starts.clone();
            bfs(old_starts);
            old_starts = starts;
            starts = new LinkedList<>();
            findStarts(old_starts);
            if (reachableNodes == -1)
                reachableNodes = finishedNodes;
        }
        return new int[] {count-1, reachableNodes};
    }

    public static void loadData() throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        StringTokenizer tokenizer = new StringTokenizer(reader.readLine());
        N = Integer.valueOf(tokenizer.nextToken());
        M = Integer.valueOf(tokenizer.nextToken());
        int friendlyTotal = Integer.valueOf(tokenizer.nextToken());
        D = Integer.valueOf(tokenizer.nextToken());
        nodes = new Node[N];
        for (int i = 0; i < N; i++) {
            nodes[i] = new Node(i + 1, i < friendlyTotal);
        }
        for (int i = 0; i < M; i++) {
            tokenizer = new StringTokenizer(reader.readLine());
            int from = Integer.valueOf(tokenizer.nextToken());
            int to = Integer.valueOf(tokenizer.nextToken());
            nodes[from - 1].addEdge(to);
            nodes[to - 1].addEdge(from);
        }
    }

    public static void main(String[] args) throws IOException {
        loadData();
        int[] result = solve();
        System.out.println(result[0] + " " + result[1]);
    }

}
```

8 Knihovna

Soňa a Marek si chtějí pořídit novou knihovnu, přou se však o její podobu. Bude tvořena několika policemi stejné délky. Marek navrhuje, aby rozestupy mezi nimi byly variabilní, přizpůsobené nejvyšší knize na dané polici. Pokud se knihy seřadí podle velikosti, může celkový rozměr knihovny klesnout.

Soňa s tím nesouhlasí. Chce, aby knihy byly seřazeny abecedně podle autora a názvu. Marek z podmínky není nadšený, ale musí ji přijmout. Po chvíli přemýšlení věří, že i při dodržení abecedního pořadí dokáže najít optimální řešení. Uvědomuje si, že někdy může být výhodnější nevyužít celou šířku police. To se nelíbí Soně, jelikož se obává, že vzniknou na konci polic nevzhledné mezery. Marek ji však ujišťuje, že dokáže zohlednit i velikost mezer a navrhne řešení, které bude vypadat přijatelně.

Úloha

Je dána délka police L a posloupnost rozměrů knih $(H_i \times W_i)_{i=1}^{n}$, kde H_i, respektive W_i je výška, respektive tloušťka i-té knihy v abecedním pořadí. Rozmístění knih na $k \geq 1$ policích lze reprezentovat jako rozklad množiny $\{1, \ldots, N\}$ na podmnožiny $S_1, S_2, \ldots, S_k$, kde S_i obsahuje indexy knih umístěných na i-té polici.

Pro všechny podmnožiny S_i platí:

- $|S_i| \geq 1$ (na každé polici je alespoň jedna kniha),
- $\sum_{j \in S_i} W_j \leq L$ (knihy se vejdou na polici),
- pokud $i < j$, pak $\max S_i < \min S_j$ (je zachováno abecední pořadí).

Počet polic k není nijak omezen a hloubka polic je dostatečná pro jakoukoliv knihu.

Náklady pro dané rozmístění knih definujeme pomocí

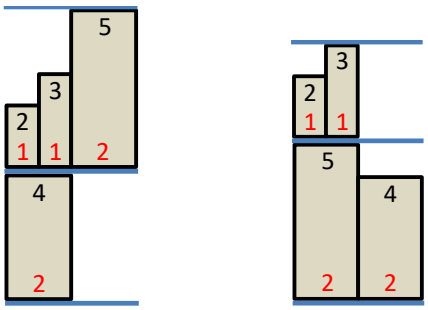
$$\text{cost}(S_1, \ldots, S_k) = \sum_{i=1}^{k} \max_{j \in S_i} H_j,$$

tedy jako součet nejvyšších knih na jednotlivých policích. Tento údaj odpovídá celkové výšce knihovny (bez započtení tloušťky polic).

Cílem je nalézt uspořádání s minimálními náklady. Pokud existuje více takových uspořádání, pak mezi nimi hledáme takové, které minimalizuje největší mezeru na konci polic, což je funkce

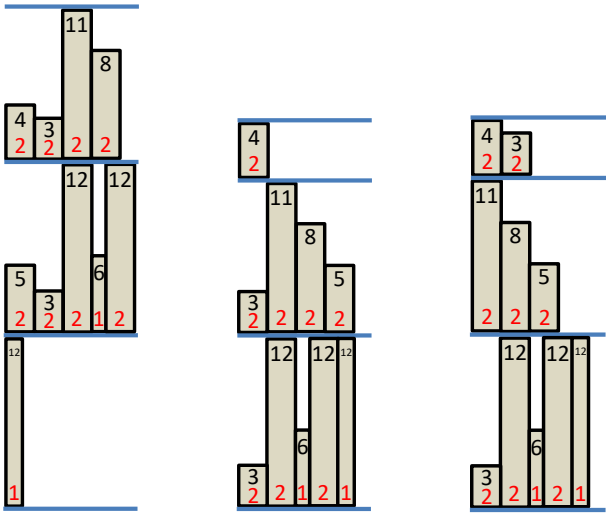
$$\text{maxGap}(S_1, \ldots, S_k) = \max_{i=1}^{k} \left(L - \sum_{j \in S_i} W_j \right).$$

Optimálního řešení navíc porovnáme s výsledkem naivního přístupu, kdy vkládáme knihy v abecedním pořadí na aktuální polici, dokud se vejdou. V okamžiku, kdy se již nevejdou, otevřeme novou polici a pokračujeme na ní.



(a) (b)

Obrázek 8: Knihovna s abecedně seřazenými knihami. Modré úsečky znázorňují police, béžové obdélníky knihy. Černé číslo v obdélníku je výška příslušné knihy, červené číslo je její tloušťka. Knihy v abecedním pořadí mají rozměry (výška × tloušťka): 2×1, 3×1, 5×2, 4×2. Délka police je $L = 4$. (a) Rozmístění knih podle naivního algoritmu s cenou $5 + 4 = 9$. (b) Jediné optimální řešení s cenou $3 + 5 = 8$ a maximální mezerou 2.



Obrázek 9: Abecedně seřazené knihy mají rozměry (výška × tloušťka): 4×2, 3×2, 11×2, 8×2, 5×2, 3×2, 12×2, 6×1, 12×2, 12×1. Délka police je $L = 9$. (a) Rozmístěný podle naivního algoritmu s cenou $11 + 12 + 12 = 35$. (b) Řešení s cenou $4 + 11 + 12 = 27$ a maximální mezerou 7. (c) Optimální řešení s cenou 27 a minimální maximální mezerou 5.

Vstup

První řádek vstupu obsahuje dvě celá čísla N a L oddělená mezerou, kde N je počet knih a L je délka police.

Následuje N řádků, přičemž každý z nich obsahuje celá čísla H a W, oddělená mezerou, reprezentující výšku a tloušťku jedné knihy. Tyto rozměry jsou uvedeny v abecedním pořadí knih.

Platí: $1 \leq N \leq 6 \cdot 10^5$, $1 \leq L \leq 3 \cdot 10^4$, $1 \leq H_i \leq 135$, $1 \leq W_i \leq 55$, $W_i \leq L$.

Výstup

Výstup tvoří jeden řádek se třemi celými čísly oddělenými mezerou: C_I, C_O, G, kde C_I je náklad pro rozmístění knih vytvořené naivním algoritmem, C_O je náklad optimálního rozmístění knih a G je nejmenší možná hodnota maxGap mezi všemi rozmístěními knih s nejmenším nákladem.

Příklad 1

Vstup

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

Výstup

```
9 8 2
```

Příklad 2

Vstup

```
10 9
4 2
3 2
11 2
8 2
5 2
3 2
```

```
12 2
6 1
12 2
12 1
```

Výstup

```
35 27 5
```

Příklad 3

Vstup

```
16 8
7 3
10 1
1 2
4 3
8 3
14 1
12 3
11 4
1 3
10 2
15 3
13 2
6 2
14 4
16 2
15 4
```

Výstup

```
81 77 2
```

Rozbor řešení

Úlohu lze přirozeně řešit dynamickým programováním. Definujme si pro $i = 1, \ldots, N$:

- $\text{cost}[i]$ = minimální cena uspořádání knih $i, i+1, \ldots, N$,
- $\text{gap}[i]$ = minimální hodnota maxGap mezi všemi optimálními uspořádáními knih $i, i+1, \ldots, N$.

Okrajové podmínky jsou

$$\text{cost}[N+1] = 0, \quad \text{gap}[N+1] = 0.$$

(Upozornění: V přiloženém Java kódu je indexování od 0.)

Pro výpočet $\text{cost}[i]$ uvažujeme, že knihy od i-té po j-tou ($i \le j \le N$) umístíme na jednu polici, pokud se jejich celková tloušťka vejde do police:

$$\sum_{t=i}^{j} W_t \le L.$$

Výška této police je pak

$$h_{i,j} = \max_{t=i}^{j} H_t.$$

Pokud první police končí na pozici j, zbytek optimálně uspořádáme od knihy $j+1$. Celkové náklady jsou

$$h_{i,j} + \text{cost}[j+1].$$

Z těchto možností vybíráme minimum:

$$\text{cost}[i] = \min_{j \,:\, \sum_{t=i}^{j} W_t \le L} \left(h_{i,j} + \text{cost}[j+1] \right).$$

Abychom kromě nákladů minimalizovali i největší mezeru na policích, sledujeme zároveň i funkci gap. Pokud dosáhneme stejné hodnoty cost[i] různými volbami j, pak z nich vybereme takovou, která minimalizuje

$$\text{gap}[i] = \max\Big(L - \sum_{t=i}^{j} W_t,\ \text{gap}[j+1]\Big).$$

Tímto způsobem postupujeme od $i = N$ směrem zpět k $i = 1$.

Výsledné hodnoty

$$C_O = \text{cost}[1], \quad G = \text{gap}[1]$$

reprezentují optimální cenu a nejmenší dosažitelnou maximální mezeru.

Cena naivního řešení C_I se získá jednoduchým průchodem knih od začátku do konce: na polici přidáváme knihy, dokud se vejdou; pokud se další nevejde, polici uzavřeme a začneme novou. Výsledkem je součet výšek všech takto vytvořených polic.

Program tedy vypíše trojici C_N, C_O, G. Jeho časová složitost je $\mathcal{O}(N^2)$, tedy kvadratická v počtu knih.

Java kód

```
package alg;

import java.io.*;
import java.util.*;

public class Bookcase {

    public static int N, L;
    public static int[] width, height;
    public static int[] gShelfEnd, oShelfEnd;

    public static int naive() {
        int cost = 0;
        int w = 0, h = 0;
        for (int i = 0; i < N; i++) {
            if (w + width[i] > L) {
                cost += h;
                w = h = 0;
            }
            w += width[i];
            h = Math.max(h, height[i]);
        }
        cost += h;
        return cost;
    }

    public static int[] solve() {
        gShelfEnd = new int[N];
        oShelfEnd = new int[N];

        int[] cost = new int[N + 1];
        int[] gap = new int[N + 1];
        cost[N-1] = height[N-1];
        cost[N] = 0;
        gap[N-1] = L - width[N-1];
        gap[N] = 0;
        for (int i = N-2; i >= 0; i--) {
            // compute cost[i] and gap[i]
            int h = height[i];
            int w = width[i];
            int bestCost = h + cost[i+1];
            int bestGap = Math.max(gap[i+1], L-w);
            int gEnd = i;
            int oEnd = i;
            for (int j = i+1; j < N; j++) {
                if (w+width[j] > L)
```

```
                    break;
                else {
                    w += width[j];
                    h = Math.max(h, height[j]);
                    int hh = h + cost[j+1];
                    if (hh < bestCost) {
                        bestCost = hh;
                        bestGap = Math.max(gap[j+1], L-w);
                        gEnd = j;
                        oEnd = j;
                    } else if (hh == bestCost) {
                        int newGap = Math.max(gap[j+1], L-w);
                        if (newGap < bestGap) {
                            bestGap = newGap;
                            oEnd = j;
                        }
                    }
                }
            }
            cost[i] = bestCost;
            gap[i] = bestGap;
            gShelfEnd[i] = gEnd;
            oShelfEnd[i] = oEnd;
        }
        return new int[] {naive(), cost[0], gap[0]};
    }

    public static void loadData() throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        StringTokenizer tokenizer = new StringTokenizer(reader.readLine());
        N = Integer.valueOf(tokenizer.nextToken());
        L = Integer.valueOf(tokenizer.nextToken());
        height = new int[N];
        width = new int[N];
        for (int i = 0; i < N; i++) {
            tokenizer = new StringTokenizer(reader.readLine());
            height[i] = Integer.valueOf(tokenizer.nextToken());
            width[i] = Integer.valueOf(tokenizer.nextToken());
        }
    }

    public static void main(String[] args) throws IOException {
        loadData();
        int[] result = solve();
        System.out.print(result[0]);
        for (int i = 1; i < result.length; i++)
            System.out.print(" "+ result[i]);
        System.out.println();
    }

}
```

9 Horská výzva

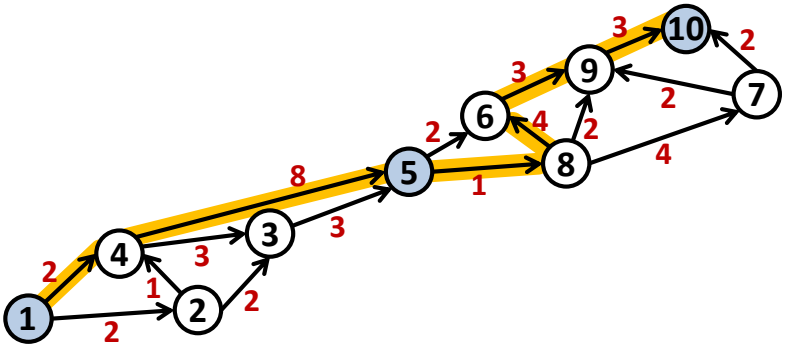
Zdatný turista Kvido se chystá na svou další túru. Má v úmyslu si ji co nejvíce užít, proto nad mapou velmi pečlivě plánuje. Za výchozí bod si zvolil rozcestník na úpatí hory a za cíl její vrchol. V mapě jsou vyznačené dílčí trasy od rozcestníku k rozcestníku a pro každou takovou trasu je uvedena očekávaná doba jejího průchodu. Kromě vyznačených tras je pohyb v uvažované oblasti zakázán. Jelikož Kvido rád stoupá vzhůru, požaduje, aby po každé zvolené dílčí trase dostal vždy od níže položeného rozcestníku k výše položenému rozcestníku. Označí si proto pro každou dílčí trasu směr možného postupu. Vzhledem k tomu, že se žádné dva rozcestníky nenacházejí ve stejné nadmořské výšce, přiřadí šipku všem trasám. Všiml si, že i po takovémto zorientování dílčích tras může ke každému rozcestníku dojít ze startu a zároveň může od každého rozcestníku dojít do cíle.

V místech některých rozcestníků leží horské chaty s možností občerstvení. Shodou okolností jsou tyto chaty právě u těch rozcestníků, kterým se Kvido při výšlapu ze startu na vrchol nemůže žádným způsobem vyhnout – po žádné orientované cestě je nemůže obejít. Tyto rozcestníky zahrnují výchozí i cílový rozcestník.

Kvido se rozhodne kvůli co největšímu zážitku **maximalizovat** dobu, kterou stráví na túře. A také myslí na svůj rituál spočívající v tom, že po projití dílčí trasy od rozcestníku k rozcestníku si vždy převrátí triko na opačnou stranu. Má zkušenost, že takto bude během výšlapu více v suchu. Vzhledem k tomu, že chce využít každou možnost občerstvení, kalkuluje současně s rozumnou podmínkou, aby na každou horskou chatu vstoupil s trikem obráceným na správnou stranu. Musí tudíž k rozcestníku před chatou dorazit s trikem naruby, které si následně dle svého zvyku převrátí (toto se samozřejmě nevztahuje na výchozí rozcestník). Jinými slovy, od výchozího bodu, kdy má triko oblečené správně, musí k libovolné chatě dojít přes sudý počet dílčích tras.

Úloha

Je dáno schéma dílčích tras, pro každou z nich pak očekávaná doba průchodu jakožto celočíselný násobek blíže neurčené časové jednotky. Rozcestníky s horskými chatami nejsou explicitně specifikovány. Určete maximální očekávanou dobu výstupu z výchozího rozcestníku na vrchol pro postup zvolený podle Kvidových kritérií. Stanovte tento údaj jako součet očekávaných časů průchodu přes zvolené dílčí trasy. Čas strávený nákupem občerstvení nebo jinými vedlejšími aktivitami nebereme v potaz.



Obrázek 10: Uzly a hrany reprezentují 10 rozcestníků a 16 dílčích tras. Výchozí, nejníže položený rozcestník má číslo 1, cílový, nejvýše položený rozcestník má číslo 10. Hrany jsou orientované od níže položeného rozcestníku k výše položenému rozcestníku. Červená čísla podél hran odpovídají očekávané době průchodu příslušnou dílčí trasou. Každý rozcestník je dosažitelný orientovanou cestou ze startu a z každého rozcestníku je podobně dosažitelný cíl. Horské chaty se nacházejí u světle modře zvýrazněných rozcestníků 1, 5 a 10. Všimněme si, že rozcestník 5 je specifický tím, že pro něj neexistuje žádná dílčí trasa, která by vedla od rozcestníku s nižší nadmořskou výškou, než má rozcestník 5, k rozcestníku s vyšší nadmořskou výškou, než má rozcestník 5. Pro každý jiný rozcestník než 1, 5 a 10 existuje orientovaná cesta ze startu na vrchol, která se tomuto rozcestníku vyhne. Optimální postup vyhovující Kvidovým požadavkům je zvýrazněn oranžově. Splňuje podmínku na sudý počet dílčích tras mezi rozcestníky 1 a 5 a také mezi rozcestníky 5 a 10. Celková očekávaná doba výstupu je pro tento postup rovna 21.

Vstup

Na prvním řádku jsou dvě celá čísla R a T oddělená mezerou. Číslo R je počet rozcestníků, T je počet dílčích tras. Rozcestníky jsou číslované od 1 do R. Výchozí, nejníže položený rozcestník má číslo 1, rozcestník na vrcholu má číslo R. Čísla ostatních rozcestníků jsou zvolena náhodně. Následuje T řádků, které v náhodném pořadí reprezentují všechny dílčí trasy. Každý z těchto řádků obsahuje tři celá čísla R_1, R_2, D oddělená mezerami. Příslušná trasa spojuje rozcestníky R_1 a R_2, přičemž rozcestník R_1 je vždy níže položený než rozcestník R_2. Číslo D je očekávaná doba průchodu trasou. Platí $1 \leq D \leq 45$.

Dále platí $2 \leq R \leq 2 \times 10^5$, $1 \leq T \leq 1.3 \times 10^6$.

Vstupní data zaručují existenci alespoň jedné cesty ze startu do cíle splňující Kvidovy požadavky.

Výstup

Jeden řádek se dvěma čísly O a H oddělenými mezerou, kde O je očekávaná doba výstupu ze startu na vrchol a H je počet horských chat, které Kvido navštíví, včetně horských chat ve výchozí a cílové pozici.

Příklad 1

Vstup

```
10 16
1 4 2
1 2 2
2 4 1
2 3 2
4 3 3
4 5 8
3 5 3
5 8 1
5 6 2
8 6 4
8 9 2
8 7 4
6 9 3
7 9 2
9 10 3
7 10 2
```

Výstup

```
21 3
```

Příklad 2

Vstup

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

Výstup

```
5 2
```

Data a řešení Příkladu 1 jsou vizualizována na Obrázku 10.

Rozbor řešení

Vzhledem k tomu, že nadmořské výšky rozcestníků jsou navzájem různé a cesty jen stoupají, tvoří rozcestníky a cesty orientovaný acyklický graf (DAG) s ohodnocenými hranami.

V tomto grafu hledáme maximální cestu ze startu do cíle, která má na každém úseku mezi horskými chatami sudou délku co do počtu hran. Právě tato vlastnost zaručuje, že Kvido vstoupí do každé horské chaty s trikem obráceným na správnou stranu.

Pro reprezentaci grafu zvolíme seznamy sousedů. Následně určíme topologické uspořádání grafu. K tomu můžeme využít dvě základní metody:

1. Průchod grafu do hloubky (DFS) – vrcholy se řadí podle času jejich uzavření, ale ve výsledném topologickém pořadí jsou uvedeny v opačném pořadí.

2. Postupné odebírání vrcholů se vstupním stupněm 0 – iterativní přístup, při kterém průběžně odstraňujeme vrcholy bez předchůdců, což zajišťuje správné uspořádání.

Předpokládejme, že vrcholy jsou číslované a zároveň seřazené podle zjištěného topologického uspořádání. Při jejich sekvenčním procházení můžeme určit, které z nich představují horské chaty. K tomu si v proměnné `maxReachable` uchováváme maximální číslo vrcholu dosažitelného po hraně vycházející z již prošlých vrcholů. Postupujeme následovně:

- Pro aktuální vrchol projdeme všechny jeho vycházející hrany a zjistíme, do jakého nejvyššího čísla vrcholu vedou.

- Pokud toto číslo přesáhne hodnotu v `maxReachable`, aktualizujeme ji.

- Jakmile přejdeme k dalšímu vrcholu, tento vrchol reprezentuje horskou chatu právě tehdy, pokud je jeho index roven `maxReachable`.

Pro nalezení maximální přípustné cesty aplikujeme dynamické programování. Vrcholy opět sekvenčně procházíme a pro každý vrchol v spočítáme dva údaje:

- `maxOdd`$[v]$ - maximální váhu cesty liché délky (v počtu hran), která začíná ve startovním vrcholu a končí ve v,

- `maxEven`$[v]$ - maximální váhu cesty sudé délky končící ve v.

Při výpočtu využíváme toho, že cestu sudé délky získáme prodloužením cesty liché délky o 1 hranu a naopak. Zároveň pro horské chaty rovnou nastavíme `maxOdd`$[v] := -\infty$, bez možnosti vylepšení, protože na horskou chatu nelze dorazit použitím lichého počtu tras.

Výsledkem algoritmu je počet zjištěných horských chat a hodnota `maxEven`$[t]$, kde t označuje cílový vrchol.

Jak topologické uspořádání, tak následné dynamické programování nad DAG vyžaduje čas úměrný velikosti grafu, tedy $\mathcal{O}(R+T)$.

Java kód

```
1  package alg;
2
3  import java.io.*;
4  import java.util.*;
5
6  public class MountainChallenge {
7
8      public static int N, M;
9      public static Node[] nodes;
10     public static int[] order;
11
12     public static class Node {
13
14         public int id;
15         public int orderNum = 0;
16         public int maxOdd = -1, maxEven = -1;
17         public int inDeg = 0;
18         public int outDeg = 0;
19         public List<int[]> edges = new ArrayList<>();
```

```
20
21          public Node(int id) {
22              this.id = id;
23          }
24
25          public void addEdge(int to, int len) {
26              nodes[to].inDeg++;
27              outDeg++;
28              edges.add(new int[] {to, len});
29          }
30      }
31
32      public static int[] solve() {
33          order = new int[N];
34          int index = 0, maxReachable = 0, innCount = 0;
35          LinkedList<Node> queue = new LinkedList<>();
36          queue.add(nodes[0]);
37          while (!queue.isEmpty()) {
38              Node node = queue.removeFirst();
39              node.orderNum = index;
40              order[index++] = node.id;
41              for (int[] edge : node.edges) {
42                  Node n = nodes[edge[0]];
43                  n.inDeg--;
44                  if (n.inDeg == 0)
45                      queue.addLast(n);
46              }
47          }
48          nodes[order[0]].maxEven = nodes[order[0]].maxOdd = 0;
49          for (index = 0; index < N; index++) {
50              Node node = nodes[order[index]];
51              boolean isInn = maxReachable == index;
52              if (isInn)
53                  innCount++;
54              for (int[] edge : node.edges) {
55                  Node n = nodes[edge[0]];
56                  if (node.maxEven >= 0 && node.maxEven + edge[1] > n.maxOdd)
57                      n.maxOdd = node.maxEven + edge[1];
58                  if (!isInn && node.maxOdd >= 0 && node.maxOdd + edge[1] > n.maxEven)
59                      n.maxEven = node.maxOdd + edge[1];
60                  maxReachable = Math.max(maxReachable, nodes[edge[0]].orderNum);
61              }
62          }
63          return new int[] {nodes[N-1].maxEven, innCount};
64      }
65
66      public static void loadData() throws IOException {
67          BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
68          StringTokenizer tokenizer = new StringTokenizer(reader.readLine());
69          N = Integer.valueOf(tokenizer.nextToken());
70          M = Integer.valueOf(tokenizer.nextToken());
71
72          nodes = new Node[N];
73          for (int i = 0; i < N; i++)
74              nodes[i] = new Node(i);
75
76          for (int i = 0; i < M; i++) {
77              tokenizer = new StringTokenizer(reader.readLine());
78              int from = Integer.valueOf(tokenizer.nextToken()) - 1;
79              int to = Integer.valueOf(tokenizer.nextToken()) - 1;
80              int len = Integer.valueOf(tokenizer.nextToken());
81              nodes[from].addEdge(to, len);
82          }
83      }
84
85      public static void main(String[] args) throws IOException {
```

```
        loadData();
        int[] result = solve();
        System.out.println(result[0] + " " + result[1]);
    }

}
```