## P-III-4 Mravenci

V této úloze máme spočítat zbytek, který dává jakési velké číslo po dělení číslem $M = 10^9 + 9$. Nejprve si proto připomeneme základy počítání se zbytky.

Mějme dvě celá čísla $A$ a $B$. Čísla můžeme jednoznačně zapsat ve tvaru $A = a_1M + a_0$ a $B = b_1M + b_0$, kde čísla $a_0, a_1, b_0, b_1$ jsou celá a $0 \leq a_0, b_0 < M$. Hodnoty $a_0$ a $b_0$ jsou zbytky, které dávají čísla $A$ a $B$ po dělení číslem $M$. Všimněte si, že $A \pm B = (a_1 \pm b_1)M + (a_0 \pm b_0)$ a $AB = (a_1b_1M + a_1b_0 + a_0b_1)M + a_0b_0$. Proto platí: $A \pm B$ dává po dělení číslem $M$ stejný zbytek jako $a_0 \pm b_0$ a $AB$ dává po dělení číslem $M$ stejný zbytek jako $a_0b_0$.

Uvedené pozorování nám v řešení úlohy umožní vyhnout se velkým číslům – budeme podle potřeby sčítat, odčítat a násobit a kdykoliv při tom nějaká průběžná hodnota překročí $M$, můžeme místo ní dále pracovat jenom se zbytkem po dělení $M$. Pro větší přehlednost nebudeme tuto úpravu v řešení nadále zapisovat. Když tedy například uvedeme v řešení „do $c$ přiřaď $a + b$“, myslíme tím „do $c$ přiřaď $((a + b) \bmod M)$.“

### Základní dynamické programování

Budeme určovat počet různých cest vedoucích z bodu $(0, 0)$ na všechna ostatní místa mřížky. Počet cest vedoucích do místa $(r, s)$ označíme $f(r, s)$.

Pokud $(r, s)$ leží vně naší mřížky, nelze se tam vůbec dostat a proto $f(r, s) = 0$. Podobně $f(r, s) = 0$ pro body, kde je umístěna překážka. Pro ostatní místa můžeme postupovat následovně: Když chceme jít na $(r, s)$, můžeme tam přijít z $(r-1, s)$ nebo z $(r, s-1)$. Cesty vedoucí přes tato místa jsou jistě navzájem odlišné, takže příslušné počty cest stačí jednoduše sečíst. Platí tedy $f(r, s) = f(r - 1, s) + f(r, s - 1)$.

Popsaným způsobem každou mapu vyřešíme s časovou složitostí $\mathcal{O}(RS)$, dostáváme tudíž řešení s celkovou časovou složitostí $\mathcal{O}(NRS)$, kde $R$ a $S$ jsou rozměry mřížky a $N$ počet mřížek na vstupu.

### Cesty bez překážek

Někdy nebývá špatné podívat se na úlohu z opačného konce. Překážky způsobují, že některé cesty nejsou použitelné. Proto nejprve spočítáme všechny cesty vedoucí na $(r, s)$ bez ohledu na překážky a následně odečteme nepoužitelné cesty.

Označme $c(r, s)$ počet cest z $(0, 0)$ do $(r, s)$, když neuvažujeme žádné překážky. Každá cesta, která urazí v jednom směru vzdálenost $r$ a ve druhém vzdálenost $s$, se jistě skládá přesně z $r+s$ kroků. Když chceme vybrat jednu konkrétní z takových cest, musíme zvolit, kterých $r$ z uvažovaných $r + s$ kroků povede směrem dolů. Každému možnému výběru odpovídá právě jedna cesta, a proto platí $c(r, s) = \binom{r+s}{r}$.

Později si ukážeme, jak lze tuto hodnotu rychle spočítat. Dodefinujme ještě $c(r, s) = 0$, jestliže $r < 0$ nebo $s < 0$.

**Inkluze a exkluze**

Představte si nyní, že máme právě jednu překážku, a to na souřadnicích $(r_1, s_1)$. Celkový počet cest vedoucích přes tuto překážku bude $c(r_1, s_1) \cdot c(R - r_1, S - s_1)$, jelikož máme $c(r_1, s_1)$ způsobů, jak se dostat z $(0, 0)$ k překážce, a následně $c(R - r_1, S - s_1)$ způsobů, jak pokračovat v cestě dále.

Počet všech cest, které přes tuto překážku *nevedou*, určíme tak, že od $c(R, S)$ odečteme počet cest, které přes překážku vedou.

Následuje jedno důležité *pozorování:* Máme-li libovolný počet překážek, můžeme je uspořádat podle hodnoty $r_i + s_i$. Jinými slovy, ve zbytku řešení můžeme předpokládat, že platí $r_1 + s_1 \leq r_2 + s_2 \leq \ldots$. Potom každá cesta, která prochází překážkou číslo $i$, může předtím procházet jen přes překážky s čísly menšími než $i$. Proč tomu tak je? Jednoduše proto, že každým krokem se o 1 zvýší součet obou souřadnic místa, na němž právě stojíme. Každé políčko, přes které jsme dosud šli, má tedy menší součet souřadnic než to, kde právě jsme.

Vraťme se k řešení původní úlohy. Co kdyby byly překážky právě dvě: na $(r_1, s_1)$ a na $(r_2, s_2)$? V souladu s uvedeným pozorováním předpokládáme, že $r_1+s_1 \leq r_2+s_2$.

Všech cest je $c(R, S)$. Od nich odečteme cesty, které vedou přes první překážku, těch je $c(r_1, s_1) \cdot c(R - r_1, S - s_1)$. Následně odečteme také cesty, které vedou přes druhou překážku, těch je $c(r_2, s_2) \cdot c(R - r_2, S - s_2)$. Ještě ale nemáme správný výsledek. Cesty, které vedou postupně přes obě překážky, jsme totiž odečetli dvakrát. Abychom dostali správný výsledek, musíme zjistit, kolik existuje takových cest, a tento počet k aktuálnímu výsledku zase zpátky přičíst. To je ale snadné: cest, které postupně procházejí přes $(r_1, s_1)$ a $(r_2, s_2)$, je $c(r_1, s_1) \cdot c(r_2 - r_1, s_2 - s_1) \cdot c(R - r_2, S - s_2)$. Díky tomu, jak jsme dodefinovali $c$ pro záporné vstupy, tento vztah funguje i v případech, že takové cesty neexistují.

Uvedené úvahy pro jednu a dvě překážky můžeme zobecnit i pro více překážek, čímž dostaneme tzv. *princip inkluze a exkluze.*

Nechť $p(X)$ je počet cest, které vedou přes každou překážku z množiny $X$ (a možná i přes jiné překážky). Pro libovolnou množinu $X$ dokážeme tento počet spočítat v čase úměrném $|X|$: stačí vynásobit počet cest z $(0, 0)$ k první překážce, počet cest od první překážky ke druhé, atd., až počet cest od poslední překážky na místo $(R, S)$.

Abychom dostali správný výsledek úlohy, potřebujeme vzít všechny cesty. Od nich odečteme cesty vedoucí přes libovolnou jednu překážku, přičteme cesty vedoucí přes dvojice překážek, zase odečteme cesty přes trojice překážek, atd. Stručně to můžeme zapsat následovně: Hledaný počet cest je roven součtu všech hodnot $(-1)^{|X|} p(X)$, kde sčítáme přes všechna $X \subseteq \{1, 2, \ldots, K\}$.

Implementací tohoto vztahu dostáváme řešení, které dokáže libovolnou mřížku s $K$ překážkami vyhodnotit v čase $\mathcal{O}(K \cdot 2^K)$ – za předpokladu, že má k dispozici všechna potřebná kombinační čísla.

**Vzorové řešení, první část**

Označme $z(i)$ počet způsobů, jak se lze dostat k překážce $i$ tak, abychom nešli přes žádnou překážku s číslem menším než $i$.

Určitě platí $z(1) = c(r_1, s_1)$ – cesta k první překážce jistě nemůže přecházet přes jiné překážky, proto každá možná cesta je dobrá.

Předpokládejme nyní, že už známe hodnoty $z(1)$ až $z(k-1)$. Jak určit hodnotu $z(k)$? Všechny cesty vedoucí k překážce $k$ můžeme rozdělit do následujících navzájem disjunktních množin:

1. Cesty, které vedou přes překážku 1.
2. Cesty, které nevedou přes překážku 1 a vedou přes překážku 2.
3. Cesty, které nevedou přes překážky 1 ani 2 a vedou přes překážku 3.

   ⋮

$k$. Cesty, které nevedou přes žádnou z překážek 1 až $k-1$.

Všech cest k překážce $k$ je celkem $c(r_k, s_k)$. Všimněte si cest $i$-tého z výše uvedených typů (pro nějaké $i < k$). Kolik jich je? Máme $z(i)$ způsobů, jak se lze dostat k $i$-té překážce, aniž bychom přešli přes některou z předcházejících, a $c(r_k - r_i, s_k - s_i)$ způsobů, jak se dostat odtamtud už libovolnou cestou ke $k$-té překážce. Proto platí:

$$z(k) = c(r_k, s_k) - \sum_{i<k} z(i) \cdot c(r_k - r_i, s_k - s_i).$$

Když už známe hodnoty $z(i)$, analogickou úvahou bychom mohli spočítat počet cest, které neprocházejí žádnou překážkou. Při implementaci si pomůžeme jednoduchým trikem: přidáme překážku číslo $K+1$ na souřadnice $(R, S)$. Potom jako $z(K+1)$ dostaneme přesně počet hledaných cest.

Toto řešení zpracuje jednu mřížku v čase $\mathcal{O}(K^2)$ – opět za předpokladu, že máme k dispozici všechna potřebná kombinační čísla.

**(Před)počítání kombinačních čísel**

Jednou možností je spočítat si všechna potřebná kombinační čísla pomocí vzorce $\binom{a+1}{b+1} = \binom{a}{b} + \binom{a}{b+1}$ nebo jinak řečeno pomocí Pascalova trojúhelníka. Kombinační čísla se nemění, spočítáme je jednou na začátku výpočtu programu a následně je můžeme využívat během celého výpočtu.

Nejjednodušší je uložit si je do dvojrozměrného pole. Při paměťovém limitu 64 MB se nám vejde do paměti přibližně 16 milionů celých čísel, což odpovídá zhruba tabulce $4\,000 \times 4\,000$.

Šikovnější řešení je založeno na pozorování, že většinu kombinačních čísel nikdy nevyužijeme. Lepší tedy bude nejprve načíst celý vstup a zjistit, která kombinační čísla potřebovat budeme. Následně budeme kombinační čísla počítat pomocí výše uvedeného vztahu a vždy, když narazíme na takové, které potřebujeme, si jeho hodnotu zapamatujeme. Pomocí tohoto triku bylo možné získat alespoň 13 bodů.

Jinou možností (která navíc postačovala i na testovací vstup 14) je počítání hodnot $\binom{a}{b}$ pomocí jejich prvočíselného rozkladu: pro libovolné prvočíslo $p$ snadno spočítáme, kolikrát dělí každé z čísel $a!$, $b!$ a $(a-b)!$.

**Dělení modulo $M$**

Kombinační čísla můžeme počítat i přímo podle vzorce $\binom{a}{b} = \frac{a!}{b!(a-b)!}$. Problém tohoto přístupu spočívá v dělení. Zatímco sčítat, odčítat a násobit modulo $M$ je snadné, s dělením to není tak jasné. Dělení při operacích se zbytky obecně nemusí fungovat. Například čísla 12 a 22 dávají po dělení číslem 10 stejný zbytek, ale $12/2 = 6$ a $22/2 = 11$ už stejný zbytek nedávají. Pokud počítáme modulo 10, pak například číslo 7 vůbec nedokážeme vydělit dvěma – tedy neexistuje takové $x$, aby $2x$ dávalo stejný zbytek jako 7.

V naší situaci jsme na tom ale dobře, jelikož číslo $M$ je prvočíslo. A v takovém případě umíme „dělit“? tzn. ke každému $a$ a každému $b$ (takovému, že $0 < b < M$) existuje právě jeden možný zbytek $x$ takový, že $bx \equiv a \pmod{M}$. Proč tomu tak je? Uvažujme hodnoty $0, b, 2b, \ldots, (M-1)b$. Žádné dvě z těchto hodnot nemohou po dělení $M$ dávat stejný zbytek, neboť potom by $M$ dělilo jejich rozdíl $(j-i)b$. To ale nemůže, jelikož oba činitelé jsou menší než $M$ a zároveň $M$ je prvočíslo. Speciálně tedy ke každému $b$ existuje právě jedno $x$ takové, že $bx \equiv 1 \pmod{M}$. Toto $x$ budeme nazývat inverzním prvkem k $b$ a budeme ho značit $b^{-1}$.

Nyní snadno zjistíme, že platí: jestliže $a/b$ je celé číslo, pak dává po dělení číslem $M$ stejný zbytek jako $a \cdot b^{-1}$. Když tedy potřebujeme znát hodnotu $\binom{a}{b}$ modulo $M$, zjistíme ji jako $\big(a! \cdot (b!)^{-1} \cdot ((a-b)!)^{-1}\big)$.

Pokud si spočítáme pro každé $n$ z rozsahu od 1 do 100 000 hodnoty $n!$ a $(n!)^{-1}$, dokážeme pak určit libovolné potřebné kombinační číslo v konstantním čase.

**Inverzní prvky**

Jakým způsobem najdeme inverzní prvek k danému číslu $b$? Můžeme použít například rozšířený Euklidův algoritmus, kde hledáme nějaké celočíselné řešení rovnice $bx + My = 1$. Jinou, jednodušší možností je použít malou Fermatovu větu. Ta říká, že když $M$ je prvočíslo, potom pro libovolné $b$ $(0 < b < M)$ platí $b^{M-1} \equiv 1 \pmod{M}$. No a $b^{M-1}$ můžeme přepsat jako $b \cdot b^{M-2}$, odkud již vidíme, že inverzním prvkem k $b$ je $b^{M-2} \bmod M$. Tuto hodnotu dokážeme spočítat šikovným umocňováním v čase $\mathcal{O}(\log M)$.

**Celé vzorové řešení**

Začneme tím, že pro každé $n$ spočítáme hodnotu $n! \bmod M$ a pomocí malé Fermatovy věty k ní určíme inverzní prvek. Následně postupně zpracováváme mřížky ze vstupu s využitím postupu založeného na počítání čísel $z(i)$. Toto řešení má časovou složitost $\mathcal{O}((R+S)\log M + NK^2)$.

```
#include <algorithm>
#include <vector>
#include <cstdio>
using namespace std;
```

```
long long modexp(long long number, long long power, long long modulus) {
  if (power==0) return 1LL % modulus;
  if (power==1) return number % modulus;
  long long tmp = modexp(number,power/2,modulus);
  tmp = (tmp*tmp) % modulus;
  if (power&1) tmp = (tmp*number) % modulus;
  return tmp;
}

bool distless(const pair<int,int> &A, const pair<int,int> &B) {
  return A.first + A.second < B.first + B.second;
}

int main() {
  int R, S, N, K, P=1000000009;
  scanf("%d%d%d", &R, &S, &N);

  vector<int> fact(R+S+2);
  fact[0]=fact[1]=1;
  for (int i=2; i<R+S+2; ++i) fact[i] = (1LL * fact[i-1] * i) % P;
  vector<int> inverse(R+S+2);
  for (int i=0; i<R+S+2; ++i) inverse[i] = modexp(fact[i], P-2, P);

  while (N--) {
    scanf("%d", &K);

    vector< pair<int,int> > prekazky;
    for (int k=0; k<K; ++k) {
      int x, y;
      scanf("%d%d",&x, &y);
      prekazky.push_back(make_pair(x, y));
    }
    prekazky.push_back(make_pair(R, S));
    sort(prekazky.begin(), prekazky.end(), distless);

    vector<long long> G(K+1);
    for (int k=0; k<=K; ++k) {
      int dx = prekazky[k].first, dy = prekazky[k].second;
      G[k] = (((1LL * fact[dx+dy] * inverse[dx]) % P) * inverse[dy]) % P;
    }
    for (int k=0; k<K; ++k)
      for (int l=k+1; l<=K; ++l) {
        int dx = prekazky[l].first - prekazky[k].first,
            dy = prekazky[l].second - prekazky[k].second;
        if (dx<0 || dy<0) continue;
        long long C = (((1LL * fact[dx+dy] * inverse[dx]) % P) * inverse[dy]) % P;
        G[l] -= G[k] * C;
        G[l] %= P;
        if (G[l] < 0) G[l] += P;
      }
    printf("%Ld\n", G[K]);
  }

  return 0;
}
```

### P-III-5 Hurikán

Je zřejmé, že kiribatské ostrovy a aktuálně stojící mosty mezi nimi tvoří neorientovaný graf. Když spadne některý z mostů, z grafu zmizí příslušná hrana.

Mezi vrcholy, které tvoří jednu komponentu souvislosti grafu, existuje spojení po mostech. Potřebujeme zabezpečit dostatek převozníků na dopravu mezi různými komponentami. Má-li graf $S$ komponent, nejmenší postačující počet převozníků je $S - 1$ (budou například jezdit mezi první komponentou a všemi ostatními). Proto nám stačí zjistit pro každý den, z kolika komponent souvislosti se skládá graf.

#### Přímočaré řešení

Počet komponent grafu lze určit prohledáváním do hloubky nebo do šířky. Začneme v prvním vrcholu a postupně obarvujeme všechny vrcholy, do nichž se dá z něho dostat. Pokud zůstaly nějaké neobarvené vrcholy, některý z nich si vybereme a začneme z něho znovu prohledávat další komponentu. Tento postup opakujeme, dokud neobarvíme celý graf. Počet komponent je určen tím, kolikrát jsme museli začít prohledávat.

Na popsaném principu je založeno jednoduché řešení úlohy: postupně pro každý den odstraníme z grafu hranu odpovídající tomu mostu, který právě spadl, a prohledáváním zjistíme aktuální počet komponent souvislosti. To znamená, že potřebujeme $(K + 1)$-krát prohledat celý graf. Časová složitost tohoto řešení je proto $\mathcal{O}(K \cdot (M + N))$.

#### Výpočet odzadu

Podívejme se nyní na úlohu od konce. Začneme s grafem, v němž chybí všech $K$ mostů s porušenou statikou. Budeme postupovat od posledního dne k prvnímu a postupně přidávat mosty do grafu. Zatím je toto řešení stejně dobré jako to předcházející, jenom dostáváme výsledky v opačném pořadí.

Přidáním hrany do grafu se mohou dvě komponenty spojit do jedné (pokud tato hrana vedla mezi vrcholy z dvou různých komponent), nebo se rozdělení na komponenty vůbec nezmění (pokud hrana vedla mezi dvěma vrcholy téže komponenty). Pro získání rychlejšího řešení budeme proto potřebovat datovou strukturu, která nám umožní efektivně zjišťovat, zda jsou dva vrcholy ve stejné komponentě, a také spojovat dvojice komponent.

#### Union-Find

Na chvíli teď zapomeneme, že se jedná o graf, a budeme řešit obecnější úlohu. Komponentu nazveme množinou, její vrcholy budou prvky této množiny. Množinu prvků si budeme reprezentovat stromem. Zavěsíme ho za kořen, který označíme jako *reprezentanta* množiny. Z ostatních prvků množiny vede vždy jedna hrana směrem nahoru, do *nadřízeného* prvku. Nadřízený prvek je blíže ke kořeni nebo je to dokonce kořen stromu.

Když vyjdeme z libovolného prvku množiny a budeme procházet v hierarchii stále výše (vždy do nadřízeného prvku), musíme skončit v reprezentantovi množiny. To znamená, že dva prvky patří do stejné množiny právě tehdy, když uvedeným postupem z obou dojdeme do téhož reprezentanta.

Dvě množiny sloučíme do jedné tak, že reprezentanta jedné z nich zavěsíme pod reprezentanta druhé množiny (nastavíme mu ho jako nadřízeného).

Spojováním množin mohou vznikat dlouhé řetězce prvků. V takovém případě bude mít hledání reprezentantů až lineární časovou složitost vzhledem k velikosti množiny. Ukážeme si dvě úpravy, které nám přinesou zrychlení.

Při slučování dvou množin zavěsíme vždy strom s menší hloubkou pod hlubší strom. To zajistí, že vznikající stromy budou mít hloubku řádově logaritmickou vzhledem k počtu prvků. Další možností je zvolit si vždy náhodně, který strom zavěsíme pod který. Potom budou také v průměrném případě vznikat stromy s malou hloubkou.

Druhým trikem je komprese cesty. Po nalezení reprezentanta ho nastavíme všem prvkům, jimiž jsme cestou nahoru prošli, jako jejich přímého nadřízeného.

Pomocí pokročilejších matematických metod lze dokázat, že průměrná časová složitost hledání reprezentanta s využitím obou uvedených zrychlení bude $\mathcal{O}(\alpha(n))$, kde $n$ je velikost množiny a $\alpha$ je inverzní Ackermannova funkce. Pro prakticky využitelná čísla nabývá její hodnota nejvýše 4, proto ji můžeme považovat za konstantu. Celková časová složitost tohoto algoritmu je proto $\mathcal{O}(N + M\alpha(N))$, případně $\mathcal{O}(N + M + K\alpha(N))$, když nejprve všechny stabilní mosty zpracujeme jedním prohledáváním.

V programu uvedeném na konci tohoto řešení používáme místo spojování podle hloubek stromů spojování náhodné – snáze se implementuje a očekávaná časová složitost je stejná.

Existuje i jiný způsob, jak lze napsat řešení, které získá plný počet bodů. Stejně jako v předchozím řešení budeme mosty zpracovávat od konce a zjišťovat počet komponent souvislosti grafu. To budeme dělat jednoduše tak, že si ke každému vrcholu budeme pamatovat číslo jeho komponenty a ke každé komponentě seznam jejích vrcholů. Vždy, když přidáváme hranu, ověříme, zda jsou oba její konce ve stejné komponentě. Pokud ano, nic se neděje, pokud ne, „přebarvíme“ celou menší komponentu – tzn. každému jejímu vrcholu změníme přiřazené číslo na číslo větší komponenty.

Všimněte si, že vždy, když nějaký vrchol přebarvíme, dostane se tím do komponenty alespoň dvojnásobné velikosti, než v jaké byl dosud. Proto každý vrchol přebarvíme nejvýše $\log_2 N$ krát, takže toto řešení má časovou složitost $\mathcal{O}(M + N \log N)$.

```
#include <cstdio>
#include <cstdlib>
using namespace std;

#define MAX 1234567

int N, M, K, components;
int E[MAX][2];
int boss[MAX], D[MAX], damaged[MAX], answer[MAX];

int sef(int x) {
    if (x == boss[x]) return x;
```

```
    else return boss[x] = sef(boss[x]);
}

void join(int x, int y) {
    x  = sef(x); y = sef(y);
    if (x == y) return;
    --components;
    if (rand() & 1) boss[x] = y; else boss[y] = x;
}

int main() {
    scanf("%d%d", &N, &M);
    components = N;
    for (int n=1; n<=N; ++n) boss[n]=n;
    for (int m=1; m<=M; ++m) scanf("%d%d", &E[m][0], &E[m][1]);
    scanf("%d", &K);
    for (int k=1; k<=K; ++k) { scanf("%d", &D[k]); damaged[D[k]] = 1; }
    for (int m=1; m<=M; ++m) if (!damaged[m]) join(E[m][0], E[m][1]);
    answer[0] = components-1;
    for (int k=K; k>=1; --k) {
        join(E[D[k]][0], E[D[k]][1]);
        answer[K+1-k] = components-1;
    }
    for (int k=K; k>=0; --k) printf("%d\n", answer[k]);
    return 0;
}
```