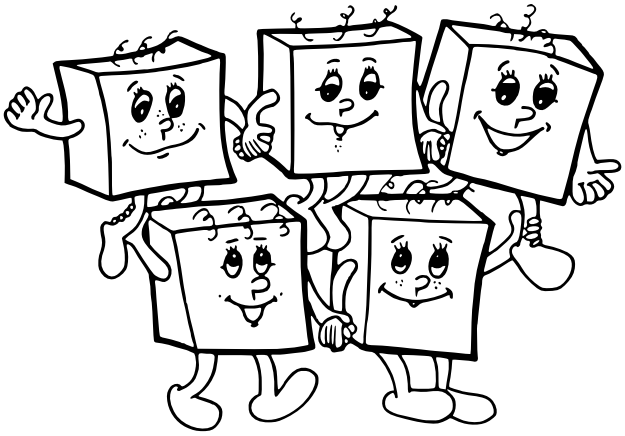
OLYMPIÁDA V INFORMATIKE

NA STREDNÝCH ŠKOLÁCH

http://oi.sk/



dvadsiaty piaty ročník

školský rok 2009/10

riešenia celoštátneho kola
kategória A

2. súťažný deň

A-III-4 Mravce idú

V tejto úlohe bolo cieľom spočítať zvyšok, ktorý dáva akési veľké číslo po delení číslom $M = 10^9 + 9$. Na úvod tohto riešenia preto spomenieme základy práce so zvyškami.

Majme dve celé čísla A a B. Vieme, že ich môžeme jednoznačne zapísať v tvare $A = a_1M+a_0$ a $B = b_1M+b_0$, kde čísla a_0, a_1, b_0, b_1 sú celé a $0 \leq a_0, b_0 < M$. Hodnoty a_0 a b_0 sú zvyšky, ktoré dávajú A a B po delení M. Všimnime si teraz, že $A \pm B = (a_1 \pm b_1)M + (a_0 \pm b_0)$ a $AB = (a_1b_1M + a_1b_0 + a_0b_1)M + a_0b_0$. Preto platí: $A \pm B$ dáva po delení M rovnaký zvyšok ako $a_0 \pm b_0$ a AB dáva po delení M rovnaký zvyšok ako a_0b_0.

Toto pozorovanie nám v našom riešení umožní vyhnúť sa veľkým číslam – sčítame, odčítame a násobíme, čo potrebujeme, a vždy, keď nejaká priebežná hodnota prekročí M, môžeme namiesto nej pracovať len so zvyškom, ktorý dáva po delení M.

Kvôli prehľadnosti budeme tento krok vo zvyšku riešenia vynechávať. Keď teda napr. uvedieme v riešení „do c priraď $a + b$“, myslíme tým „do c priraď $((a + b) \bmod M)$“.

Základné dynamické programovanie

Poďme si spočítať počet rôznych ciest z miesta $(0, 0)$ na všetky ostatné miesta mriežky. Počet ciest do miesta (r, s) označme $f(r, s)$.

Zjavne ak (r, s) leží mimo našej mriežky, tak sa tam nedá dostať, a teda $f(r, s) = 0$. Rovnako $f(r, s) = 0$ pre miesta, kde je prekážka. Pre ostatné miesta môžeme postupovať nasledovne: Všimnime si, že keď chceme ísť do (r, s), tak tam musíme prísť z $(r - 1, s)$ alebo z $(r, s - 1)$. A keďže cesty cez tieto miesta sú určite navzájom rôzne, tak počet týchto možností stačí spočítať. Teda platí $f(r, s) = f(r - 1, s) + f(r, s - 1)$.

Takýmto spôsobom každú mapu vyriešime s časovou zložitosťou $O(RS)$, dostávame teda riešenie s celkovou časovou zložitosťou $O(NRS)$.

Cesty bez prekážok

Niekedy nie je zlé sa na úlohu pozrieť z opačného konca. Totiž prekážky nám spôsobujú, že niektoré cesty budú nepoužiteľné. Preto najprv spočítame počet všetkých ciest ignorujúc prekážky a následne odpočítame zlé cesty.

Označme $c(r, s)$ počet ciest takých, že sa v jednom smere posunieme o r a v druhom o s. Inými slovami, $c(r, s)$ bude počet ciest z $(0, 0)$ do (r, s), pričom nemáme žiadne prekážky. Cesta, ktorá prejde v jednom smere vzdialenosť r a v druhom vzdialenosť s má určite presne $r + s$ krokov. Keď chceme vybrať konkrétnu z týchto ciest, musíme vybrať, ktorých r spomedzi týchto $r + s$ krokov povedie smerom dodola. Každému možnému výberu zodpovedá práve jedna cesta, a teda platí $c(r, s) = \binom{r+s}{r}$.

Ako túto hodnotu rýchlo spočítať si ukážeme neskôr. Dodefinujme ešte $c(r, s) = 0$, ak $r < 0$ alebo $s < 0$.

Inklúzia a exklúzia

Predstavme si teraz, že máme práve jednu prekážku, a to na súradniciach (r_1, s_1). Potom celkový počet ciest vedúcich cez prekážku bude $c(r_1, s_1) \cdot c(R - r_1, S - s_1)$, keďže máme $c(r_1, s_1)$ spôsobov ako sa dostať z $(0, 0)$ ku prekážke a následne $c(R - r_1, S - s_1)$ spôsobov ako pokračovať ďalej.

Počet všetkých ciest, ktoré cez túto prekážku *nevedú*, vieme teda zistiť tak, že od $c(R, S)$ odčítame počet ciest, ktoré cez prekážku vedú.

Teraz jedno dôležité pozorovanie: Keď máme ľubovoľný počet prekážok, môžeme ich usporiadať podľa hodnoty $r_i + s_i$. Inými slovami, môžeme vo zvyšku riešenia predpokladať, že platí $r_1 + s_1 \leq r_2 + s_2 \leq \cdots$. Potom bude platiť nasledujúce tvrdenie: každá cesta, ktorá prechádza prekážkou číslo i, môže predtým ísť len cez prekážky s číslom menším ako i. Prečo? Jednoducho preto, že každým krokom sa o 1 zvýši súčet súradníc miesta, na ktorom práve stojíme. A teda každé políčko, cez ktoré sme doteraz šli, má menší súčet súradníc ako to, kde práve sme.

Vráťme sa k riešeniu pôvodnej úlohy. Čo ak by boli prekážky práve dve: na (r_1, s_1) a na (r_2, s_2)? V súlade s vyššie uvedeným pozorovaním predpokladáme, že $r_1 + s_1 \leq r_2 + s_2$.

Všetkých ciest je $c(R, S)$. Od nich odčítame cesty, ktoré vedú cez prvú prekážku, tých je $c(r_1, s_1) \cdot c(R - r_1, S - s_1)$, a následne aj cesty, ktoré vedú cez druhú prekážku, tých je $c(r_2, s_2) \cdot c(R - r_2, S - s_2)$.

Ešte stále však nemáme správny výsledok. Cesty, ktoré vedú postupne cez obe prekážky, sme totiž odčítali dvakrát. Aby sme dostali správny výsledok, musíme zistiť, koľko je takýchto ciest, a tento počet k aktuálnemu výsledku pripočítať. No ale to je ľahké: ciest, ktoré postupne prechádzajú cez (r_1, s_1) a (r_2, s_2) je $c(r_1, s_1) \cdot c(r_2 - r_1, s_2 - s_1) \cdot c(R - r_2, S - s_2)$. Všimnite si, že vďaka tomu, ako sme dodefinovali c pre záporné vstupy, tento vzťah funguje aj v prípadoch, že takéto cesty neexistujú.

Vyššie uvedené úvahy pre jednu a dve prekážky vieme zovšeobecniť aj pre viac prekážok, čím dostaneme tzv. princíp inklúzie a exklúzie (alebo tiež „zapojenia a vypojenia“).[1]

Formálne môžeme toto riešenie sformulovať nasledovne: Nech $p(X)$ je počet ciest, ktoré idú cez každú prekážku v množine X (a možno aj cez iné prekážky). Pre ľubovoľnú množinu X vieme tento počet spočítať v čase úmernom $|X|$: stačí vynásobiť počet ciest z $(0, 0)$ k prvej prekážke, počet ciest od prvej prekážky ku druhej, atď., až počet ciest od poslednej prekážky na (R, S).

Aby sme dostali správny výsledok, potrebujeme zobrať všetky cesty. Od nich odčítame cesty vedúce cez ľubovoľnú jednu prekážku, pripočítame cesty cez dvojice prekážok, zase odpočítame cesty cez trojice prekážok, atď. Stručne to môžeme zapísať nasledovne: hľadaný počet ciest je rovný sume všetkých hodnôt $(-1)^{|X|} p(X)$, kde sčitujeme cez všetky $X \subseteq \{1, 2, \ldots, K\}$.

Implementáciou tohto vzťahu dostávame riešenie, ktoré vie ľubovoľnú mriežku s K prekážkami vyhodnotiť v čase $O(K \cdot 2^K)$ – za predpokladu, že má k dispozícii všetky potrebné kombinačné čísla.

Vzorové riešenie, prvá časť

Označme $z(i)$ počet spôsobov, ako sa dostať ku prekážke i tak, aby sme nešli cez žiadnu prekážku s číslom menším ako i.

[1] `http://sk.wikipedia.org/wiki/Princíp_zapojenia_a_vypojenia`

Určite platí $z(1) = c(r_1, s_1)$ – totiž cesta na prvú prekážku nemá ako prechádzať cez iné prekážky, preto každá možná cesta je dobrá.

Predpokladajme teraz, že už poznáme hodnoty $z(1)$ až $z(k-1)$. Ako určiť hodnotu $z(k)$? Všetky cesty ku prekážke k vieme rozdeliť do nasledujúcich navzájom disjunktných množín:

1. Cesty, ktoré vedú cez prekážku 1.
2. Cesty, ktoré nevedú cez prekážku 1 a vedú cez prekážku 2.
3. Cesty, ktoré nevedú cez prekážky 1 ani 2 a vedú cez prekážku 3.

   ...

k. Cesty, ktoré nevedú cez žiadnu z prekážok 1 až $k-1$.

Všetkých ciest ku prekážke k dokopy je $c(r_k, s_k)$. Všimnime si teraz cesty i-teho z vyššie uvedených typov (pre nejaké $i < k$). Koľko ich je? Máme $z(i)$ spôsobov, ako sa dostať k i-tej prekážke tak, aby sme netrafili žiadnu z predchádajúcich, a $c(r_k - r_i, s_k - s_i)$ spôsobov, ako sa odtiaľ, už ľubovoľnou cestou, dostať ku k-tej prekážke. Preto platí:

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

Keď už poznáme hodnoty $z(i)$, analogickou úvahou by sme vedeli spočítať počet ciest, ktoré neprechádzajú žiadnou prekážkou. Pri implementácii si ale môžeme pomôcť jednoduchým trikom: pridáme prekážku číslo $K+1$ na súradniciach (R, S). Potom ako $z(K+1)$ dostaneme presne počet hľadaných ciest.

Toto riešenie spracuje jednu mriežku v čase $O(K^2)$ – opäť za predpokladu, že máme k dispozícii všetky potrebné kombinačné čísla.

(Pred)počítanie kombinačných čísel

Jednou možnou cestou je predpočítať si všetky kombinačné čísla pomocou vzorca $\binom{a+1}{b+1} = \binom{a}{b} + \binom{a}{b+1}$ alebo ináč povedané Pascalovho trojuholníka. Kľúčové je uvedomiť si, že kombinačné čísla sa nemenia. Stačí ich predpočítať raz, na začiatku behu programu, a následne ich využívať počas behu.

Najjednoduchšie riešenie je uložiť si ich jednoducho v dvojrozmernom poli. Pri pamäťovom limite 64 MB sa nám zmestí do pamäte približne 16 miliónov celých čísel, čo zodpovedá zhruba tabuľke 4000×4000.

Šikovnejšie riešenie je založené na pozorovaní, že väčšinu kombinačných čísel nikdy nevyužijeme. Lepšie je teda na začiatku načítať celý vstup a zistiť, ktoré kombinačné čísla potrebovať budeme. Následne budeme kombinačné čísla počítať pomocou vyššie uvedeného vzťahu a vždy, keď narazíme na také, ktoré potrebujeme, si jeho hodnotu zapamätáme. Pomocou tohto triku sa určite dalo získať aspoň 13 bodov.

Inou možnou cestou (ktorá okrem iného stačila aj na testovací vstup 14) je počítanie hodnôt $\binom{a}{b}$ cez ich prvočíselný rozklad: pre ľubovoľné prvočíslo p ľahko spočítame, koľkokrát delí každé z čísel $a!$, $b!$ a $(a-b)!$.

Delenie modulo M

Kombinačné čísla vieme počítať aj priamo, podľa vzorca $\binom{a}{b} = \frac{a!}{b!(a-b)!}$. Problém s týmto prístupom je ale v delení. Zatiaľ čo sčítať, odčítať a násobiť modulo M je ľahké, s delením do vôbec nie je také jasné.

V prvom rade, delenie pri operáciách so zvyškami vo všeobecnosti nemusí fungovať. Napríklad čísla 12 a 22 dávajú po delení 10 rovnaký zvyšok, ale $12/2 = 6$ a $22/2 = 11$ už rovnaký zvyšok nedávajú. No a ak rátame modulo 10, tak napríklad číslo 7 vôbec nevieme vydeliť dvomi – teda neexistuje také x, aby $2x$ dávalo rovnaký zvyšok ako 7.

V našej situácii sme však na tom dobre, lebo číslo M, modulo ktoré rátame, je prvočíslo. A v takejto situácii vieme „deliť“ – teda ku každému a a každému b (takému, že $0 < b < M$) existuje práve jeden možný zvyšok x taký, že $bx \equiv a \pmod{M}$.

(Prečo je tomu tak? Uvažujme hodnoty $0, b, 2b, \ldots, (M-1)b$. Žiadne dve z týchto hodnôt nemôžu po delení M dávať rovnaký zvyšok, lebo potom by M delilo ich rozdiel $(j-i)b$. To ale nemôže, lebo oba činitele sú menšie ako M a zároveň M je prvočíslo.)

Špeciálne teda ku každému b existuje práve jedno x také, že $bx \equiv 1 \pmod{M}$. Takéto x budeme volať inverzným prvkom k b a budeme ho značiť b^{-1}.

Teraz ľahko nahliadneme, že platí: ak a/b je celé číslo, tak dáva po delení M rovnaký zvyšok ako $a \cdot b^{-1}$. Ak teda potrebujeme vedieť hodnotu $\binom{a}{b}$ modulo M, zistíme ju ako $\left(a! \cdot (b!)^{-1} \cdot ((a-b)!)^{-1}\right)$.

Ak teda predpočítame pre každé n z rozsahu od 1 po 100 000 hodnoty $n!$ a $(n!)^{-1}$, budeme vedieť ľubovoľné potrebné kombinačné číslo zistiť v konštantnom čase.

Inverzné prvky

Ako nájsť inverzný prvok k danému číslu b? Toto číslo vieme efektívne zistiť napríklad rozšíreným Euklidovým algoritmom, kde hľadáme nejaké celočíselné riešenie rovnice $bx + My = 1$.

Inou, jednoduchšou možnosťou je použiť malú Fermatovu vetu. Tá hovorí, že ak M je prvočíslo, tak pre ľubovoľné b $(0 < b < M)$ platí $b^{M-1} \equiv 1 \pmod{M}$. No a b^{M-1} môžeme prepísať ako $b \cdot b^{M-2}$, odkiaľ vidíme, že inverzným prvkom k b je $b^{M-2} \bmod M$. Túto hodnotu vieme šikovným umocňovaním vypočítať v čase $O(\log M)$.

Celé vzorové riešenie

Začneme tým, že pre každé n predpočítame hodnotu $n! \bmod M$ a následne k nej pomocou malej Fermatovej vety zistíme aj inverzný prvok. Následne postupne spracúvame mriežky zo vstupu použitím postupu s počítaním čísel $z(i)$. Takéto riešenie má časovú zložitosť $O((R+S)\log M + NK^2)$.

Listing programu:

```
// riesenie pouzivajuce inverzne prvky a dynamicke programovanie
#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]);
  }
}
```

A-III-5 Hurikán

Ľahko vidno, že kiribatské ostrovy a aktuálne stojace mosty medzi nimi predstavujú neorientovaný graf – keď spadne niektorý z mostov, z grafu zmizne príslušná hrana.

Medzi vrcholmi, ktoré sú v jednom komponente súvislosti grafu, existuje spojenie po mostoch. Zostáva zabezpečiť dostatok prievozníkov na dopravu medzi rôznymi komponentmi. Keď má graf S komponentov, najmenší postačujúci počet prievozníkov je $S - 1$ (napríklad budú premávať medzi prvým komponentom a všetkými ostatnými). Preto nám stačí zistiť pre každý deň, z koľkých komponentov sa skladá graf.

Priamočiare riešenie

Počet komponentov grafu sa dá zistiť prehľadávaním do hĺbky alebo do šírky. Začneme v prvom vrchole a ofarbujeme všetky tie vrcholy, do ktorých sa dá z neho dostať. Ak ešte ostali nejaké neofarbené vrcholy, niektorý z nich si vyberieme a začneme z neho znova prehľadávať. Toto opakujeme, kým neofarbíme celý graf. Počet komponentov je rovný práve počtu, koľkokrát sme museli začínať prehľadávať.

Na tomto je založené nasledujúce riešenie: postupne pre každý deň od začiatku odstránime z grafu hranu mosta, ktorý práve spadol, a prehľadávaním zisťujeme aktuálny počet komponentov. To znamená, že potrebujeme práve $(K + 1)$-krát prehľadať celý graf. Časová zložitosť toho riešenie je preto $O(K \cdot (M + N))$.

Prístup odzadu

Pozrime sa na úlohu od konca. Začneme s grafom, v ktorom chýba všetkých K mostov s porušenou statikou. Budeme postupovať od posledného dňa k prvému a postupne pridávať tieto mosty do grafu. Zatiaľ je toto riešenie rovnako dobré ako predchádzajúce, akurát dostávame výsledky v opačnom poradí.

Ďalšou zmenou je, že tentoraz nám do grafu hrany pribúdajú. Pridaním hrany sa môžu dva komponenty spojiť do jedného (ak táto hrana viedla medzi vrcholmi z dvoch rôznych komponentov) alebo sa rozdelenie na komponenty vôbec nezmení (ak hrana viedla medzi vrcholmi z toho istého komponentu).

Na rýchlejšie riešenie budeme preto potrebovať dátovú štruktúru, ktorá nám umožní efektívne zisťovať, či sú dva vrcholy v rovnakom komponente, a tiež zlučovať dvojicu komponentov.

Union-find

Na chvíľu zabudnime, že ide o graf a riešme všeobecnejšiu úlohu. Komponent nazvime množinou; vrcholy v ňom budú prvkami tejto množiny.

Množinu prvkov si reprezentujeme stromom. Zavesíme ho za koreň, ktorý označme ako *reprezentanta* množiny. Z ostatných prvkov vedie vždy jedna hrana smerom hore, do *nadriadeného* prvku. Nadriadený prvok je bližšie ku koreňu alebo je to dokonca koreň.

Keď začneme v ľubovoľnom prvku množiny a budeme prechádzať v hierarchii čoraz vyššie (stále do nadriadeného prvku), musíme skončiť v reprezentantovi množiny. To znamená, že dva prvky sú v rovnakej množine práve vtedy, keď z oboch dôjdeme do toho istého reprezentanta.

Dve množiny zlúčime do jednej tak, že reprezentanta jednej z nich zavesíme pod reprezentanta druhej (nastavíme mu ho ako nadriadeného).

Spájaním množín môžu vznikať dlhé reťaze prvkov. Vtedy bude mať hľadanie reprezentantov až lineárnu časovú zložitosť od veľkosti množiny. Ukážeme si dve úpravy, ktoré prinesú zrýchlenie.

Pri zlučovaní dvoch množín zavesíme vždy strom s menšou hĺbkou pod hlbší strom. To zabezpečí, že vzniknuté stromy budú mať hĺbku rádovo logaritmickú od počtu prvkov. Ďalšou možnosťou je zvoliť si náhodne, ktorý strom zavesíme pod ktorý, vďaka čomu budú v očakávanom prípade tiež vznikať stromy s malou hĺbkou.

Druhým trikom je kompresia cesty. Po nájdení reprezentanta ho nastavíme všetkým prvkom, ktorými sme na ceste nahor prešli, ako priameho nadriadeného.

Pomocou pokročilých matematických metód sa dá dokázať, že priemerná časová zložitosť hľadania reprezentanta s využitím oboch zrýchlení bude $O(\alpha(n))$, kde n je veľkosť množiny a α je inverzná Ackermannova funkcia. Pre prakticky využiteľné čísla je jej hodnota najviac 4, preto ju môžeme pokladať za konštantu. Celková časová zložitosť tohto algoritmu je teda $O(N + M\alpha(N))$, prípadne $O(N + M + K\alpha(N))$ ak najskôr všetky stabilné mosty spracujeme jedným prehľadávaním.

V listingu programu na konci tohto riešenia používame namiesto spájania podľa hĺbok spájanie náhodné – ľahšie sa implementuje a očakávaná časová zložitosť je rovnaká.

Iný spôsob ako napísať riešenie, ktoré by malo získať plný počet bodov: Rovnako ako v predchádzajúcom riešení budeme mosty spracúvať od konca a zisťovať počet komponentov súvislosti. To budeme robiť jednoducho tak, že ku každému vrcholu si budeme pamätať číslo jeho komponentu a ku každému komponentu zoznam jeho vrcholov. Vždy, keď pridávame hranu, pozrieme sa, či sú jej konce v tom istom komponente. Ak áno, nič sa nedeje, ak nie, „prefarbíme“ celý menší komponent – teda každému vrcholu v ňom zmeníme jeho číslo na číslo väčšieho komponentu.

Všimnite si, že vždy, keď vrchol prefarbíme, dostane sa tým do komponentu aspoň dvakrát takej veľkosti ako dovtedy. Preto každý vrchol prefarbíme nanajvýš $\log_2 N$ krát, a teda má toto riešenie časovú zložitosť $O(M + N \log N)$.

Listing programu:

```
#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]);
}
```

SLOVENSKÁ KOMISIA OLYMPIÁDY V INFORMATIKE

DVADSIATY PIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Vydala IUVENTA s finančnou podporou Ministerstva školstva SR
Náklad: 40 výtlačkov
Zodpovedný redaktor: Michal Forišek
Sadzba programom LaTeX