по един „въпрос", който се състои от номерата **X** и **Y** на агентите, за които е зададен въпросът. Входните данни са коректни и непротиворечиви.

Изход: За всеки въпрос от входа на отделен ред на стандартния изход програмата трябва да се изведе отговора като число: 0, ако **X** и **Y** са приятели, 1 – ако са врагове, 2 – ако информацията не е достатъчна за да се даде един от предните два отговора.

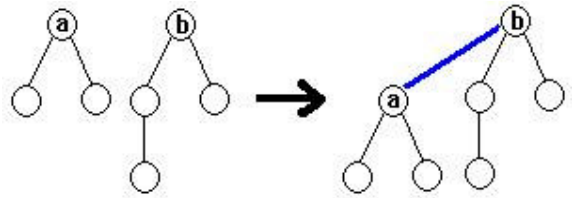
ПРИМЕР:

Вход:	**Изход:**
4 3 0 1 1 1 2 0 2 3 1 1 0 3	0

Решение:

Това решение ползва една структура от данни представяща разбиване на множество, която позволява лесно да се извършва операцията обединение на две групи от разбиването – наричат я "disjoint sets" или "union find".

Всяка група се представя като дърво от елементи (пазим си само бащата на всеки възел), а коренът му се явява „официалния представител", или „големия брат", или „вожда" на всички под него. Когато обединяваме групи, просто закачаме вожда на едната за вожда на другата – така че да станат баща и син.



Можем да намерим вожда на произволен елемент като тръгнем нагоре по дървото до корена. Два елемента са в една и съща група от разбиването точно когато имат един и същ вожд.

Смисълът който влагаме е следния: разбиваме множеството от агенти на групи, в рамките на всяка от които сме сигурни че агентите са приятели. Това представяне е удачно, защото приятелството (~) е релация на еквивалентност:

- Аз съм приятел със себе си (`a~a`)
- За двама приятели чувствата са взаимни (`a~b => b~a`)
- Приятелите на моите приятели са и мои приятели (`a~b & b~c => a~c`)

Трябва ни и начин да представяме знанието си че две групи приятели са врагове помежду си. Враждата (`*`) при този „двуполюсен модел" (има само две чужди държави) не е нищо друго освен антоним на приятелството:

- Ако не си ми приятел, значи си ми враг, и обратно (`a~b <=> !(a*b)`)

От изброените по-горе принципи, могат да се изведат интересни следствия:

- За двама врагове чувствата са взаимни (`a*b => b*a`)
- Враговете на моите приятелите са и мои врагове (`a~b & b*c => a*c`)
- Враговете на моите врагове са ми приятели (`a*b & b*c => a~c`)

Доказват се лесно с допускане на противното.

И така, оказва се достатъчно да си пазим най-много по един враг за група приятели. Няма смисъл да пазим по два врага, защото ще е ясно че те са приятели помежду си и веднага можем да ги обединим.

За конкретната реализация ползваме два масива:

`friend[x]` – Бащата на `x` в дървото (гората) на приятелството.

`enemy[x]` – Ще го ползваме само ако `x` е вожд – `enemy[x]` сочи към вожда на вражеската група приятели. Ако не знаем враг за групата на `x`, тогава `enemy[x]` ще бъде `-1`. Винаги ще поддържаме масива такъв че ако `enemy[x] == y`, то и `enemy[y] == x`.

Ще прилагаме последователно информацията от дадените телефонни разговори, т.е. ще прилагаме операциите: „направи `x` и `y` приятели", и „направи `x` и `y` врагове". Да означим с `getLeader(x)` функцията която ни връща вожда на `x`, а за всяка стъпка нека:

```
lx = getLeader(x);
ly = getLeader(y);
ex = enemy[lx];
ey = enemy[ly];
```

Разглеждаме следните случаи:

- „направи X и Y приятели"
  - ако `lx == ly`, те вече са си приятели, не правим нищо.
  - ако `enemy[lx] == ly`, нещо не е наред – стигнали сме до противоречие. Това не трябва да се случва според условието на задачата.
  - иначе, сприятеляваме `lx` и `ly`, определяйки едното от тях за вожд и закачайки другото за вожда (нека за определеност закачим `ly` за `lx`), и след това:
    - ако `ex != -1` и `ey != -1`, сприятеляваме `ex` и `ey`, а този от тях, който е нов вожд, го правим взаимен враг с `lx`.
    - ако точно едно от `ex` и `ey` е различно от `-1`, правим го взаимен враг с `lx`.
    - ако `ex == -1` и `ey == -1`, всичко е ok, не правим нищо.
- „направи X и Y врагове"
  - ако `enemy[lx] == ly`, те вече са си врагове, не правим нищо.
  - ако `lx == ly`, имаме противоречие.
  - иначе, правим `lx` и `ly` взаимни врагове, като преди това: ако `ex != -1`, сприятеляваме го с `ly`; и ако `ey != -1`, сприятеляваме го с `lx`.

И накрая, отговаряме на въпросите за отношенията произволни x и y:

- ако `getLeader(x) == getLeader(y)`, те са приятели.
- ако `enemy[getLeader(x)] == getLeader(y)`, те са врагове.
- иначе – не е ясно.

```
#include <stdio.h>
#include <assert.h>
#include <memory.h>

#define MAX_N 1024
```

```
int n;
int fri[MAX_N];      // "friend" is a reserved word in C++
int enemy[MAX_N];


int getLeader(int x) {
   int leader = x;
   while (fri[leader] != -1) {
      leader = fri[leader];
   }
   if (x != leader) {
      fri[x] = leader; // "Path compression" optimization
   }
   return leader;
}

void main() {
   scanf("%d", &n);
   memset(fri, -1, sizeof(fri));
   memset(enemy, -1, sizeof(enemy));
   int m;
   scanf("%d", &m);
   for (int i = 0; i < m; i++) {
      int a, b, r;
      scanf("%d%d%d", &a, &b, &r);
      a = getLeader(a);
      b = getLeader(b);
      int ea = enemy[a];
      int eb = enemy[b];
      if (r == 0) {
         // Must make them friends, unless they already are
         assert(ea != b);
         if (a != b) {
            // The groups of a and b merge, and b becomes the leader
            fri[a] = b;
            if (ea != -1) {
               if (eb == -1) {
                  // Since b is the new leader, he adopts the enemies of a
                  enemy[b] = ea;
                  enemy[ea] = b;
               } else {
                  // If a and b both have enemies, those enemies become friends
                  fri[ea] = eb;
               }
            }
         }
      } else {
         // Must make them enemies, unless they already are
         assert(a != b);
         if (ea != b) {
            if (ea != -1) {
```

```
            // The enemies of a become friends of b
            fri[ea] = b;
          }
          if (eb != -1) {
            // The enemies of b become friends of a
            fri[eb] = a;
          }
          enemy[b] = a;
          enemy[a] = b;
        }
      }
    }
    int q;
    scanf("%d", &q);
    for (int i = 0; i < q; i++) {
      int x, y;
      scanf("%d%d", &x, &y);
      x = getLeader(x);
      y = getLeader(y);
      if (x == y) {
        printf("0\n");
      } else if (enemy[x] == y) {
        printf("1\n");
      } else {
        printf("2\n");
      }
    }
}
```

Задача B3. ПРАЗНИК

Разглеждаме числовата права като път. В някои от целочислените точки по пътя може да има къщи (в една точка може да има много къщи), във всяка от които могат да живеят много хора. Така се е образувал град. Хората могат да се придвижват по този път, но трябва да плащат за това. Цената за да пропътува един човек единица разстояние по пътя е 1 лев. Тъй като това е доста скъпо, хората не пътуват често и надалеч. Градската управа се нуждае непрекъснато от пари, но за да не изглежда прекалено нагло, решила да ги събира като пътни такси. Затова периодически организира градски празници, в които участието е задължително. Всеки празник се прави в определена точка с целочислени координати. Жителите на града сами трябва да си осигурят превоза. Преди всеки празник управата внимателно трябва да избере мястото, за да може да си осигури предварително намислената сума. Напишете програма **FEST**, която да може да отговаря на въпроса: "Къде да се организира празника така, че парите които ще се съберат да са максимално близко до дадено число S?". Ще считаме, че в началото градът е празен.

Вход: Програмата трябва да чете данни от стандартния вход. На първия ред са зададени две числа **M N**, където **M** е броят на къщите, а **N** – броят на въпросите. Всеки от следващите **M** ред ще има вида **A B**, където **A** е координатата на къщата, а **B** – броят на хората, които живеят в нея. Следващите **N** реда са въпроси, като всеки въпрос съдържа по едно число **S -** сумата, която управата иска да събере. Максималният брой редове на входа е 200 000. Координатите **A** ще са в интервала [0,10 000 000], числата **B** – в интервала [0,1000], а сумата **S** – в интервала [0,10^15].