Двудольные графы

3165. Двухцветность
4002. Долой списывание!
9006. Пройти через реку
10346. Великое возрождение (серебро)
10994. Монтеско против Копулето
11343. Шеф и двудольный граф

3165. Двухцветность

В 1976 году “Теорема четырёх красок” была доказана с помощью компьютера. Эта теорема утверждает, что каждая карта может быть окрашена с использованием только четырех цветов таким образом, что ни одна область не будет окрашена тем же цветом, что и ее соседние.

Вам предлагается решить подобную задачу, но попроще. Необходимо выяснить, является ли заданный связный граф двухцветным. То есть можно ли его вершинам назначить цвета (имеется всего два цвета) таким образом, чтобы никакие две смежные вершины не имели одинаковый цвет. Для упрощения задачи можно предположить, что:

- граф не имеет петель.
- граф неориентированный. То есть если вершина a связана с вершиной b, то и b связана с a.
- граф сильно связный. То есть всегда существует как минимум один путь из любой вершины в любою другую.

Вход. Состоит из нескольких тестов. Каждый тест начинается со строки, содержащей количество вершин n ($0 \le n \le 1000$). Вторая строка содержит количество рёбер l ($1 \le l \le 250000$). После этого идёт l строк, каждая из которых содержит два числа, указывающие на ребро между двумя вершинами, которые оно соединяет. Вершины в графе будут помечены числом a ($1 \le a \le n$). Последний тест содержит $n = 0$ и не обрабатывается.

Выход. Выяснить, можно ли сделать входной граф двухцветным и вывести соответствующее сообщение, как показано в примере.

Пример входа	**Пример выхода**
3 3 1 2	NOT BICOLOURABLE. BICOLOURABLE.

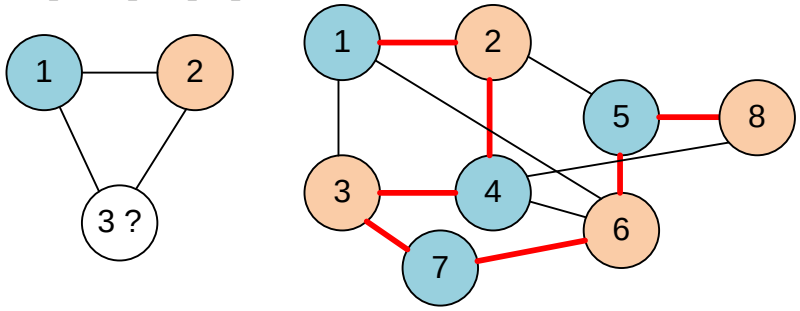
```
2 3
3 1
8
12
1 2
2 4
3 4
3 1
3 7
7 6
4 6
1 6
2 5
5 6
4 8
5 8
0
```

Анализ алгоритма

Для решения задачи воспользуемся поиском в глубину. Изначально вершины не просмотрены, пометим их все цветом 0. По мере прохождения по вершинам будем красить их в два цвета: 1 и 2. Если текущая вершина имеет цвет i (i = 1, 2), то следующую вершину, в которую мы попадаем при поиске в глубину, красим в цвет 3 – i. При этом проверяем, чтобы две соседние вершины не были покрашены в одинаковый цвет.

Пример

Приведенные в примере графы имеют вид:



Реализация алгоритма

Матрицу смежности графа храним в массиве g размера MAX = 1001 (нумерация вершин идет от 1 до $n \le 1000$). Массив *used* содержит информацию о цвете вершины. Глобальная переменная *Error* отвечает за правильность раскраски: как только найдутся две соседние вершины, покрашенные одинаково, переменная примет значение 1.

```
#define MAX 1000
int g[MAX][MAX], used[MAX], Error;
```

Функция ***dfs*** выполняет поиск в глубину. Ей передаются два параметра: текущая вершина *v* и ее цвет *color*. Если уже встретился случай невозможности требуемой раскраски (*Error* = 1), то следует выйти из функции.

```
void dfs(int v,int color)
{
```

Если уже встретился случай невозможности требуемой раскраски (*Error* = 1), то следует выйти из функции.

```
  if (Error) return;
```

Помечаем вершину *v* цветом *color*.

```
  used[v] = color;
```

Ищем непросмотренную вершину *i*, в которую можно пойти, продолжив поиск в глубину. При этом если соседняя вершина *i* уже была просмотрена, проверяем условие окрашенности вершин *v* и *i* в разные цвета. Если указанные вершины покрашены в одинаковый цвет, устанавливаем *Error* = 1.

```
  for(int i = 1; i <= n; i++)
    if (g[v][i] == 1)
      if (used[i] == 0) dfs(i,3-color); else
      if (used[v] == used[i]) Error = 1;
}
```

Основной цикл программы. Читаем число вершин *n* и количество ребер *l* графа. Обнуляем матрицу смежности и массив *used*.

```
while(scanf("%d %d",&n,&l), n)
{
   memset(g,0,sizeof(g));
   memset(used,0,sizeof(used));
```

Читаем данные графа.

```
   for(i = 0; i < l; i++)
   {
     scanf("%d %d",&a,&b);
     g[a][b] = g[b][a] = 1;
   }
```

Обнуляем значение переменной *Error*, запускаем поиск в глубину с первой вершины, покрасив ее изначально в цвет 1.

```
   Error = 0; dfs(1,1);
```

В зависимости от значения переменной *Error* выводим результат.

```
   if (Error) printf("NOT BICOLOURABLE.\n");
         else printf("BICOLOURABLE.\n");
}
```

4002. Долой списывание!

Во время контрольной работы профессор Флойд заметил, что некоторые студенты обмениваются записками. Сначала он хотел поставить им всем двойки, но в тот день профессор был добрым, а потому решил разделить студентов на две группы: списывающих и дающих списывать, и поставить двойки только первым.

У профессора записаны все пары студентов, обменявшихся записками. Требуется определить, сможет ли он разделить студентов на две группы так, чтобы любой обмен записками осуществлялся от студента одной группы студенту другой группы.

Вход. В первой строке находятся два числа n и m – количество студентов и количество пар студентов, обменивающихся записками ($1 \le n \le 100$, $0 \le m \le (n * (n - 1)) / 2$). Далее в m строках расположены описания пар студентов: два различных числа, соответствующие номерам студентов, обменивающимися записками (нумерация студентов идёт с 1). Каждая пара студентов перечислена не более одного раза.

Выход. Вывести ответ на задачу профессора Флойда. Если можно разделить студентов на две группы, выведите “YES”, иначе выведите “NO”.

Пример входа	**Пример выхода**
3 2 1 2 2 3	YES

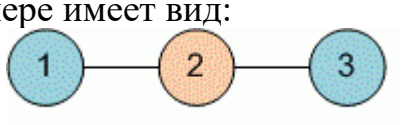
Анализ алгоритма

В задаче требуется выяснить, является ли заданный граф двухцветным. То есть можно ли его вершины покрасить двумя цветами так чтобы любое ребро соединяло вершины разных цветов.

Двухцветность в графе проверяется при помощи поиска в глубину.

Пример

Граф приведенный в примере имеет вид:



Реализация алгоритма

Объявим матрицу смежности g и массив использованных вершин used.

```
#define MAX 110
int g[MAX][MAX], used[MAX];
```

Поиск в глубину из вершины *v*. Вершину *v* красим цветом *color*.

```
void dfs(int v, int color)
{
```

Если граф не двуцветный (*Error* = 1), то продолжать поиск в глубину нет смысла.

```
  if (Error) return;
```

Присваиваем цвет *color* вершине *v*.

```
  used[v] = color;
```

Перебираем вершины *i*, ищем куда можно пойти из *v* в *i*.

```
  for (int i = 1; i <= n; i++)
```

Если существует ребро из *v* в *i*.

```
    if (g[v][i])
```

Если вершина *i* еще не просмотрена, то продолжаем из нее поиск в глубину.

```
      if (!used[i]) dfs(i, 3 - color); else
```

Если из *v* в *i* существует обратное ребро, а вершины *v* и *i* покрашены в одинаковый цвет, то граф не двухцветный. Устанавливаем *Error* = 1.

```
      if (used[v] == used[i]) Error = 1;
}
```

Основная часть программы. Читаем список ребер графа, строим матрицу смежности.

```
scanf("%d %d", &n, &m);
memset(g, 0, sizeof(m));
memset(used, 0, sizeof(used));
for (i = 0; i < m; i++)
{
  scanf("%d %d", &a, &b);
  g[a][b] = g[b][a] = 1;
}
```

Запускаем поиск в глубину, корень красим цветом 1. Граф может быть несвязный. *Error* = 0 означает что граф двухцветный, *Error* = 1 означает что нет.

```
Error = 0;
for (i = 1; i <= n; i++)
  if (!used[i]) dfs(i, 1);
```

В зависимости от значения переменной *Error* выводим ответ.

```
if (Error) printf("NO\n"); else printf("YES\n");
```

9006. Пройти через реку

Барыш и Мурад работают надсмотрщиками в отделе надсмотра за животными Бакинского зоопарка. Очень часто им приходится решать задачи сложнее задачи крестьянина. Например, однажды, используя только две лодки они должны были перевести всех животных зоопарка через реку Кура. Представьте себя на их месте.

В зоопарке имеются n животных. Если оставить некоторых животных вместе без присмотра, то один из них может съесть другого. Для каждого животного известно каких других животных он может съесть. Оба надсмотрщика мастера своего дела и могут справиться даже с тигром! Любое животное не сможет съесть другое животное, если рядом находится надсмотрщик.

У надсмотрщиков имеются две лодки. По правилам безопасности, нельзя брать в лодку двух или более животных. В каждой из лодок одновременно могут находиться самое большее один надсмотрщик и одно животное. В самом начале все животные и надсмотрщики находятся по левую сторону реки. Им всем надо перейти на правую сторону. (Очевидно, что у реки имеются только две стороны)

Надсмотрщики могут переплывать через реку в любом направлении. Лодки не могут двигаться без надсмотрщиков. В любой момент если на одной из сторон останутся животные без присмотра которые могут съесть друг друга, то произойдет несчастный случай. Естественно, это не допустимо. Все животные должны перейти на другую сторону реки целыми и невредимыми.

От вас требуется выяснить, возможно ли это.

Вход. В первой строке дано одно целое число n ($1 \le n \le 200$) – количество животных в зоопарке. Животные пронумерованы различными числами от 1 до n.

В каждой из последующих n строк, для каждого животного, дан список животных которые могут съесть это животное. В i-той строке для животного с номером i дается сначала неотрицательное целое число k_i – количество животных, которые могут съесть это животное, далее следуют k_i различных чисел – номера этих животных. Все эти числа находятся в промежутке от 1 до n и отличны от i (никакое животное не может съесть самого себя).

Сумма всех k_i не больше 1500.

Выход. Если возможно перевести всех животных на другую сторону реки, выведите ":)", в противном случае выведите ":(".

Пример входа 1

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

Пример выхода 1

```
:)
```

Пример входа 2

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

Пример выхода 2

```
:(
```

Анализ алгоритма

Каждому животному поставим в соответствие вершину графа. Если одно животное может съесть другое, проведем между ними ребро.

Пока один из надсмотрщиков остается на первом берегу с животными, второй начинает перевозить на второй берег тех, кто друг друга не ест. Предположим, что наступает момент, когда очередную такую перевозку совершить невозможно. Тогда два надсмотрщика берут с первого берега по животному себе в лодку и перевозят на второй берег. При этом должно выполняться условие:

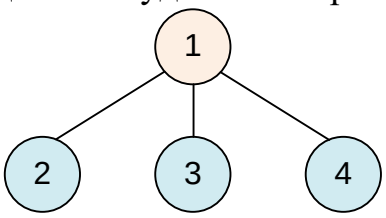
- Ни на первом, ни на втором берегу животные не должны есть друг друга. То есть после удаления двух вершин (двое животных что в лодках надсмотрщиков) граф должен быть двудольным.

Далее после перевоза двух животных на второй берег, один из надсмотрщиков остается там беречь животных, а второй начинает перевозить оставшихся на первом берегу животных.

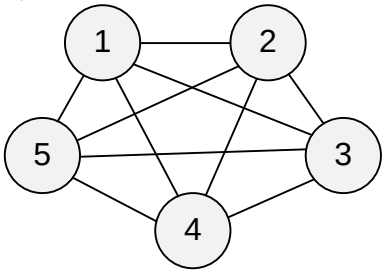
Таким образом животных можно перевезти на второй берег реки, если найдутся такие две вершины графа (двое животных), что удалив их, граф станет двудольным. Осталось перебрать все возможные пары вершин и проверить граф на двудольность.

Пример

В первом примере граф даже без удаления вершин уже является двудольным.



Во втором примере граф является полным и невозможно удалить две вершины так чтобы он стал двудольным.



Реализация алгоритма

Объявим матрицу смежности g и массив использованных вершин used.

```
#define MAX 202
int g[MAX][MAX];
int used[MAX];
```

Функция ***dfs*** совершает поиск в глубину из вершины *v*. Вершину *v* красим цветом *color*.

```
void dfs(int v, int color = 1)
{
```

Если граф не двудольный (*flag* = 1), то нет смысла продолжать поиск в глубину.

```
  if (flag == 1) return;
```

Красим вершину *v* цветом *color*.

```
  used[v] = color;
```

Перебираем вершины, куда можно пойти из *v*.

```
  for (int i = 1; i <= n; i++)
    if (g[v][i] == 1)
    {
```

Двигаемся по ребру (*v*, *i*). Если вершина *i* не пройдена, то запускаем из нее поиск в глубину. Красим вершину *i* цветом 3 – *color*.

```
      if (used[i] == 0) dfs(i, 3 - color); else
```

Если цвет вершин *v* и *i* совпадает, значит ребро (*v*, *i*) обратное и граф не двудольный. Устанавливаем *flag* = 1.

```
      if (used[v] == used[i]) flag = 1;
    }
}
```

Основная часть программы. Читаем входные данные. Строим граф.

```
scanf("%d", &n);
for (i = 1; i <= n; i++)
{
  scanf("%d", &k);
  while (k--)
  {
    scanf("%d", &j);
    g[i][j] = g[j][i] = 1;
  }
}
```

Если животных не более 4, то их всегда можно перевезти.

```
if (n <= 4)
{
  puts(":)");
  return 0;
}
```

Перебираем пару животных (*i*, *j*).

```
for (i = 1; i <= n; i++)
for (j = i + 1; j <= n; j++)
{
  memset(used, 0, sizeof(used));
```

Удаляем животных с номерами *i* и *j* и не рассматриваем их при поиске в глубину.

```
  used[i] = used[j] = 3;
```

Изначально положим *flag* = 0 считая граф двудольным.

```
  flag = 0;
```

Граф не обязательно связный. Запускаем поиск в глубину на несвязном графе.

```
  for (k = 1; k <= n; k++)
    if (used[k] == 0)
    {
      dfs(k);
```

Если после обработки очередной компоненты связности компонента не является двудольной, то нет смысла далее продолжать поиск в глубину.

```
      if (flag == 1) break;
    }
```

Если граф двудольный, то перевозка животных возможна.

```
  if (flag == 0)
  {
    puts(":)");
    return 0;
  }
}
```

Перевезти животных невозможно.

```
puts(":(");
```

10346. Великое возрождение (серебро)

Из-за продолжительной засухи пастбища фермера Джона остались без травы. Однако с приближением сезона дождей пришло время “возделывать растительность”. В сарае фермера Джона имеется два ведра, в каждом из которых находятся семена разной травы. Он хочет посадить траву на каждом из своих n пастбищ. Каждое пастбище должно быть засеяно ровно одним видом травы.

Фермер Джон, занимающийся молочным животноводством, хочет быть уверенным в том, что он удовлетворит особые диетические потребности своих m коров. У каждой из его m коров имеются два любимых пастбища. У некоторых коров имеется диетическое ограничение: они должны постоянно есть только один вид травы – поэтому фермер Джон хочет, чтобы на двух любимых полях любой такой коровы был посажен один и тот же тип травы. У других коров совсем другие диетические ограничения, требующие от них есть разные виды травы. Что касается этих коров, фермер Джон, конечно же, хочет, чтобы на двух их любимых полях были разные типы травы.

Помогите фермеру Джону определить количество различных способов посадки травы на его n пастбищах.

Вход. Первая строка содержит числа n ($2 \le n \le 10^5$) и m ($1 \le m \le 10^5$). Каждая из следующих m строк содержит символ ‘S’ или ‘D’, за которым следуют два целых числа в диапазоне 1 … n, описывающий пару пастбищ, являющиеся любимыми для одной из коров фермера Джона. Символ ‘S’ означает, что корове на ее двух любимых пастбищах нужна трава одного типа. Символ ‘D’ означает, что корове на пастбищах нужны разные типы трав.

Выход. Выведите количество способов, которыми фермер Джон может посадить траву на n пастбищах. Запишите ответ в двоичном формате.

Пример входа	**Пример выхода**
3 2 S 1 2 D 3 2	10

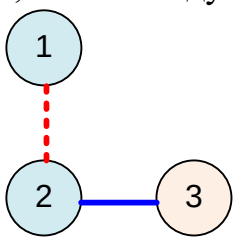
Анализ алгоритма

Построим два графа из n вершин. Ребра первого графа g_1 будут соответствовать пожеланиям коров с символом ‘S’, второго графа g_2 – пожеланиям коров с символом ‘D’. Разобьём неориентированный граф на компоненты связности. Каждая компонента связности должна представлять собой двудольный граф, где ребра из g_1 соединяют вершины из одной компоненты, а ребра из g_2 соединяют вершины из разных компонент. Это свойство компоненты проверим, раскрасив вершины двумя цветами. Если некоторая компонента не удовлетворяет указанному свойству, то ответ равен 0.

Каждую компоненту связности можно раскрасить двумя цветами двумя способами (левую долю цветом 1, правую – цветом 2, или наоборот). Следовательно *cnt* компонент можно раскрасить 2^{cnt} способами. Количество возможных раскрасок равно числу способов, которыми фермер Джон может посадить траву на n пастбищах.

Пример

Граф, приведенный в условии, имеет следующий вид.



Он двудольный, его можно покрасить двумя цветами двумя способами.

Реализация алгоритма

Объявим графы g_1 и g_2, а также массив использованных вершин used.

```
vector<vector<int>> g1, g2;
vector<int> used;
```

Функция ***dfs*** совершает поиск в глубину из вершины *v*. Вершина *v* красится цветом *color*. Функция ***dfs*** проверяет, является ли текущая компонента связности двудольной, причем ребра из g_1 должны соединять вершины из одной компоненты, а ребра из g_2 должны соединять вершины из разных компонент. Если это условие не выполняется, установим *flag* = 1.

```
void dfs(int v, int color)
{
```

Присваиваем цвет вершине *v*.

```
  used[v] = color;
  for (int i = 0; i < g1[v].size(); i++)
  {
```

В графе g_1 имеется ребро (*v*, *to*). Вершины *v* и *to* должны иметь одинаковый цвет.

```
    int to = g1[v][i]; // same color
```

Если вершины *v* и *to* имеют разный цвет, текущая компонента не удовлетворяет условию.

```
    if (used[to] == 3 - color) flag = 1;
    if (used[to] == 0) dfs(to, color);
  }

  for (int i = 0; i < g2[v].size(); i++)
  {
```

В графе g_2 имеется ребро (*v*, *to*). Вершины *v* и *to* должны иметь разный цвет.

```
    int to = g2[v][i]; // different color
```

Если вершины *v* и *to* имеют одинаковый цвет, текущая компонента не удовлетворяет условию.

```
    if (used[to] == color) flag = 1;
    if (used[to] == 0) dfs(to, 3 - color);
  }
}
```

Основная часть программы. Читаем входные данные.

```
cin >> n >> m;
g1.resize(n + 1);
g2.resize(n + 1);
```

Читаем информацию про диетические ограничения коров. Строим два графа g_1 и g_2.

```
for (i = 0; i < m; i++)
{
  cin >> ch >> a >> b;
  if (ch == 'S') // same
  {
    g1[a].push_back(b);
    g1[b].push_back(a);
  }
  else // different
  {
    g2[a].push_back(b);
    g2[b].push_back(a);
  }
}
```

Запускаем поиск в глубину на несвязном графе. Подсчитываем количество компонент связности *cnt*. Изначально положим *flag* = 0, считая что все компоненты двудольные и удовлетворяют указанному свойству.

```
used.resize(n + 1);
flag = cnt = 0;
```

```
for (i = 1; i <= n; i++)
  if (used[i] == 0)
  {
```

Вершину *i* красим цветом 1.

```
    dfs(i, 1);
    cnt++;
  }
```

Если *flag* = 1, то ответ 0.

```
if (flag == 1)
  cout << "0" << endl;
else
{
```

Выводим ответ 2^{cnt} в двоичном коде: одна единица и *cnt* нулей.

```
  cout << "1";
  for (i = 0; i < cnt; i++)
    cout << "0";
  cout << endl;
}
```

10994. Монтеско против Копулето

Ромео и Джульетта наконец-то решили пожениться. Но приготовить свадебную вечеринку будет непросто, так как хорошо известно, что их семьи – Монтеско и Капулето – являются кровавыми врагами. В этой задаче Вам следует решать, кого пригласить, а кого не пригласить, чтобы предотвратить кровопролитие.

У нас есть список *n* людей, которых можно пригласить на вечеринку или нет. Для каждого человека *i* известен список его врагов: e_1, e_2, . . . , e_p. Отношения “враги” обладают следующими свойствами:

Антитранзитивность. Если *a* враг *b*, а *b* враг *c*, то *a* друг *c*. Кроме того, враги друзей *a* – его враги, а друзья друзей *a* – его друзья.

Симметричность. Если *a* является врагом *b*, то *b* является врагом *a* (хотя это может быть и не указано в его списке врагов).

Один человек примет приглашение на вечеринку, если и только если он приглашен, приглашены все его друзья и никто из его врагов не приглашен. Вы должны найти максимальное количество людей, которых можно пригласить, чтобы все они приняли приглашение.

Например, если *n* = 5, и мы знаем, что: 1 – враг 3, 2 – враг 1 и 4 – враг 5, тогда мы можем пригласить максимум 3 человек. Этими людьми могут быть 2, 3 и 4, но для этой задачи нам нужно только количество приглашенных людей.

Вход. Первая строка содержит количество тестов m. Далее следует пустая строка. Пустая строка также используется для разделения тестов. Первая строка каждого теста содержит количество людей n ($n \leq 200$). Для каждого из этих n людей имеется строка со списком его врагов. Первая строка содержит список врагов человека 1, вторая строка содержит список врагов человека 2 и так далее. Каждый список врагов начинается с целого числа p (количество известных врагов этого человека), за которым следуют p целых чисел (p врагов этого человека). Так, например, если враги человека 5 и 7, то список его врагов выглядит так: 2 5 7.

Выход. Для каждого теста выведите одну строку, содержащей целое число – максимальное количество людей, которых можно пригласить так, чтобы все они приняли приглашение.

Пример входа	**Пример выхода**
3 5 1 3 1 1 0 1 5 0 8 2 4 5 2 1 3 0 0 0 1 3 0 1 5 3 2 2 3 1 3 1 1	3 5 0

Анализ алгоритма

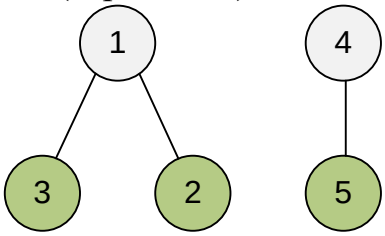
Выделим в неориентированном графе компоненты связности.

- Если компонента связности является *двудольной*, то наибольшее количество людей, которое можно из нее пригласить, равно числу вершин в большей двудольной компоненте.

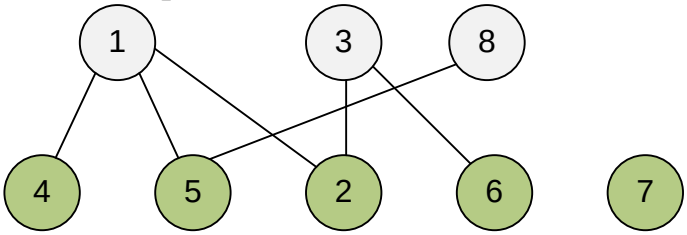
- Если компонента связности не является двудольной, то людей пригласить из нее невозможно.

Пример

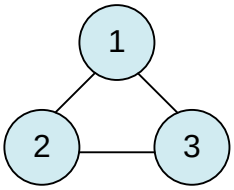
В первом тесте имеются две компоненты связности, обе из которых двудлольные. В первой компоненте наибольшее число вершин в одной из долей – 2 (вершины 2 и 3), во второй – 1 (вершина 5). Таким образом ответ равен 3.



Во втором тесте можно пригласить 5 людей.



В третьем тесте имеется одна компонента связности, которая не является двудольной.



Реализация алгоритма

Объявим матрицу смежности графа g и массив использованных вершин used.

```
#define MAX 201
int g[MAX][MAX], used[MAX];
```

Функция ***dfs*** запускает поиск в глубину из вершины *v*. Совершаем раскраску вершин двумя цветами – 1 и 2 так чтобы каждое ребро соединяло вершины разного цвета.

- Вершину *v* красим цветом *color*.
- В переменной *a* подсчитываем количество вершин, имеющих цвет 1.
- Подсчитываем общее количество вершин в текущей компоненте связности в переменной *cnt*.

```
void dfs(int v, int &a, int &cnt, int color, int &ok)
{
  used[v] = color;
  cnt++;
  if (color == 1) a++;
  for (int i = 1; i <= n; i++)
    if (g[v][i] == 1)
```

Если вершина *i* еще не посещена, то запускаем из нее поиск в глубину. Красим ее цветом 3 – *color*.

```
      if (used[i] == 0) dfs(i, a, cnt, 3 - color, ok);
      else
```

Если граф раскрасить двумя цветами невозможно, то он не двудольный. Установим *ok* = 0.

```
        if (used[i] == used[v]) ok = 0;
}
```

Основная часть программы. Читаем входные данные. Строим граф.

```
scanf("%d", &tests);
while (tests--)
{
  memset(g, 0, sizeof(g));
  scanf("%d", &n);
  for (i = 1; i <= n; i++)
  {
    scanf("%d", &k);
    for (j = 1; j <= k; j++)
    {
      scanf("%d", &to);
```

Добавляем в граф неориентированное ребро (*i*, *to*).

```
      g[i][to] = g[to][i] = 1;
    }
  }
```

Запускаем поиск в глубину на несвязном графе.

```
  memset(used, 0, sizeof(used));
```

В переменной *res* подсчитываем наибольшее возможное количество выбранных вершин (приглашенных людей).

```
  res = 0;
```

Перебираем вершины графа.

```
  for (i = 1; i <= n; i++)
  {
```

Если мы еще не были в вершине *i*, то запускаем из нее поиск в глубину.

```
    if (used[i] == 0)
    {
      int a = 0, cnt = 0, ok = 1;
```

Изначально положим *ok* = 1, считая что компонента связности двудольная. Вершину *i* красим цветом 1.

```
      dfs(i, a, cnt, 1, ok);
```

Если компонента связности двудольная ($ok = 1$), то размеры ее долей равны a и $cnt - a$. Из двудольной компоненты мы выбираем долю с наибольшим количеством вершин, а именно $\max(a, cnt - a)$.

```
      res += ok * max(a, cnt - a);
    }
  }
```

Выводим ответ для текущего теста.

```
  printf("%d\n", res);
}
```

11343. Шеф и двудольный граф

Шеф заинтересовался изучением двудольных графов. Сегодня он хочет построить двудольный граф с n вершинами в каждой из двух частей и с общим числом ребер, равным m. Вершины слева пронумерованы от 1 до n. Вершины справа тоже пронумерованы от 1 до n. Он также хочет, чтобы степень каждой вершины была больше или равна d и меньше или равна D, то есть для всех v: $d \le \deg(v) \le D$.

По четырем целым числам n, m, d, D Вы должны помочь Шефу построить некоторый двудольный граф, удовлетворяющий описанному свойству. Если такого графа не существует, выведите -1.

Вход. Первая строка содержит количество тестов t.

Одна строка каждого набора входных данных содержит четыре целых числа n ($1 \le n \le 100$), m ($0 \le m \le n * n$), d, D ($1 \le d \le D \le n$).

Выход. Для каждого набора входных данных выведите -1, если не существует двудольного графа, удовлетворяющего заданному свойству. В противном случае выведите m строк, каждая из которых должна содержать два целых числа u и v, обозначающие наличие ребра между вершиной u в левой части и вершиной v в правой части. Если существует несколько возможных ответов, выведите любой. Обратите внимание, что двудольный граф не должен иметь кратных ребер.

Пример входа	**Пример выхода**
2 2 3 1 2 2 3 1 1	1 1 2 2 1 2 -1

Анализ алгоритма

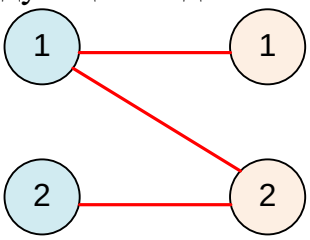
В левой и правой долях графа располагается n вершин. Поскольку степень каждой вершины больше или равна d и меньше или равна D, то число ребер m должно быть как минимум $n * d$ ($n * d \le m$) и максимум n * D ($n * D \ge m$).

Граф будем строить конструктивно следующим образом.

1. Пусть *cur* = 0.
2. Для каждого i ($1 \le i \le n$) соединим вершину i левой доли с вершиной i + *cur* правой доли. Если окажется i + *cur* > n, то в правой доле используем вершину i + *cur* – n. Как только будет построено m ребер, завершаем алгоритм.
3. *cur* = *cur* + 1 и переходим к пункту 2.

Пример

Граф из примера имеет следующий вид:



Реализация алгоритма

Читаем количество тестов *tests*.

```
scanf("%d", &tests);

while (tests--)
{
```

Читаем входные данные очередного теста.

```
  scanf("%d %d %d %d", &n, &m, &d, &D);
```

Если граф построить нельзя из-за невыполнения необходимых неравенств, то выводим -1.

```
  if (n * d > m || n * D < m)
  {
    printf("-1\n");
    continue;
  }
```

Установим *cur* = 0.

```
  cur = 0;
  while (m > 0)
  {
```

Пока не будет построено в точности *m* ребер, строим ребра по указанному алгоритму.

```
    for (i = 1; i <= n; i++)
    {
      j = (i + cur) % n;
      if (j == 0) j = n;
```

Выводим ребро (*i*, *j*).

```
      printf("%d %d\n", i, j);
      m--;
```

Как только будут построены все *m* ребер, завершаем программу.

```
      if (m == 0) break;
    }
```

Увеличим значение переменной *cur* на 1.

```
    cur++;
  }
}
```