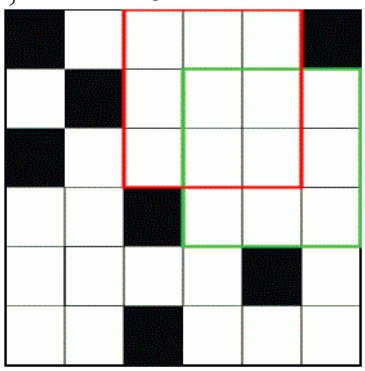
Февраль 25, 2022

Задача A. Квадрат
Задача B. Починка забора
Задача C. Мстители
Задача D. Транзитивное замыкание
Задача E. Такси
Задача F. Мистер Найн и его любовь к манго
Задача G. Предок
Задача H. Подсчет треугольников
Задача I. Разрезанное число
Задача J. Генерация подмножеств

7757. Квадрат

У Джан-Джи имеется кусок металла, из которого он хочет вырезать квадрат. Кусок состоит из сетки n на n, которую Джан-Джи может разрезать только по границам сетки. Каждая ячейка сетки либо исправная либо дефектная, при этом Джан-Джи хочет вырезать наибольший квадрат, не содержащий дефектных клеток. После определения максимального размера квадрата Джан-Джи хочет узнать сколькими различными способами он может вырезать его из имеющегося куска. После чего Джан-Джи следует вывести произведение максимального размера на количество способов расположения наибольшего квадрата.

Рассмотрим кусок материала размера 6 на 6. Черные ячейки дефектные. Наибольший квадрат, который может вырезать Джан-Джи, имеет размер 3 на 3, причем имеется два варианта его вырезать – красный и зеленый квадраты. Джан-Джи выведет произведение 3 и 2, то есть 6.



Вам следует найти размер наибольшего квадрата, который можно вырезать из куска материала, а также посчитать, сколькими различными вариантами это можно сделать. После чего вывести их произведение.

Вход. Первая строка содержит размер материала n ($1 \le n \le 1000$). Каждая из следующих n строк содержит n целых чисел. 1 означает что участок сетки целый, а 0 означает что участок сетки поврежден.

Выход. Вывести одно число – произведение размера наибольшего квадрата в материале на количество возможных его расположений в материале.

Пример входа 1

```
6
1 0 1 0 1 0
0 1 0 1 0 1
1 0 1 0 1 0
0 1 0 1 0 1
1 0 1 0 1 0
0 1 0 1 0 1
```

Пример выхода 1

```
18
```

Пример входа 2

```
6
0 1 1 1 1 0
1 0 1 1 1 1
0 1 1 1 1 1
1 1 0 1 1 1
1 1 1 1 0 1
1 1 0 1 1 1
```

Пример выхода 2

```
6
```

Заведем массив dp, где dp[i][j] хранит размер наибольшего квадрата, который можно вырезать из прямоугольника (0, 0) – (i, j) при условии, что этому наибольшему квадрату принадлежит ячейка (i, j).

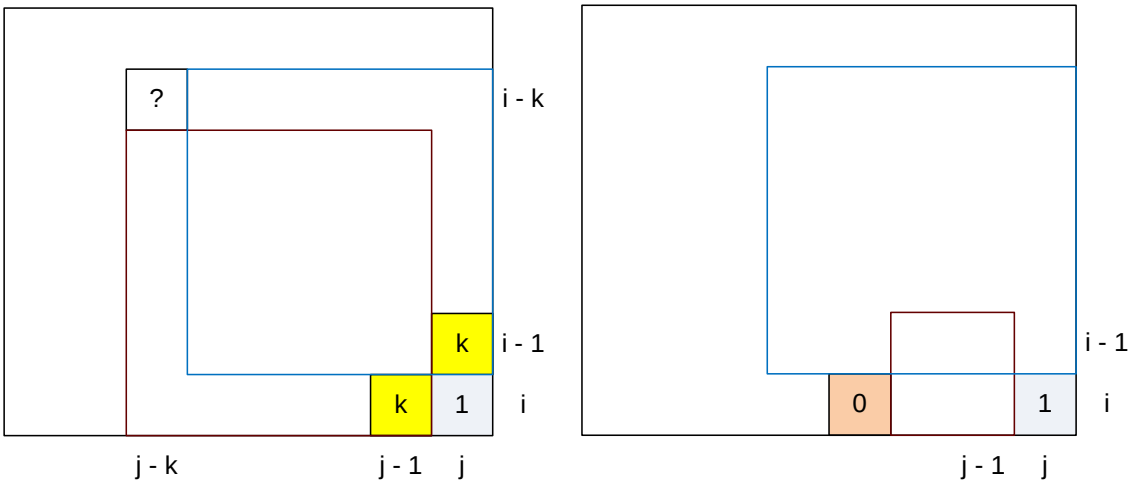
Пусть массив m содержит входной участок.

Если m[i][j] = 0 (участок сетки поврежден), то dp[i][j] = 0.

Пусть m[i][j] = 1. Рассмотрим два случая:

1. m[i – 1][j] = m[i][j – 1] = k. Тогда dp[i][j] зависит от значения ячейки m[i – k][j – k]:

- если m[i – k][j – k] = 1, то весь квадрат (i – k, j – k) – (i, j) будет содержать только единицы и dp[i][j] = k + 1.
- если m[i – k][j – k] = 0, то dp[i][j] = k.



2. $m[i-1][j] \neq m[i][j-1]$. Тогда $dp[i][j] = \min(dp[i-1][j], dp[i][j-1]) + 1$.

Суммируя выше сказанное можно заметить, что

$$dp[i][j] = \min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1$$

Пример. Рассмотрим второй тест. Заполним для него массив dp.

0	1	1	1	1	0
1	0	1	1	1	1
0	1	1	1	1	1
1	1	0	1	1	1
1	1	1	1	0	1
1	1	0	1	1	1

0	1	1	1	1	0
1	0	1	2	2	1
0	1	1	2	3	2
1	1	0	1	2	3
1	2	1	1	0	1
1	2	0	1	1	1

Имеется 2 наибольших квадрата размером 3 * 3. Произведение размера наибольшего квадрата на количество его расположений равно 3 * 2 = 6.

Реализация алгоритма

Объявим рабочий массив dp.

```
#define MAX 1010
int dp[MAX][MAX];
```

Читаем значение *n*. Переменная *size* хранит размер наибольшего квадрата, *cnt* – количество раз, которое он встречается в куске. Инициализируем массив dp нулями.

```
scanf("%d", &n);
memset(dp,0,sizeof(dp));
size = cnt = 0;
```

Пересчитываем массив dp по возрастанию строк, ячейки в каждой строке – по возрастанию столбцов.

```
for(i = 1; i <= n; i++)
for(j = 1; j <= n; j++)
{
  scanf("%d",&val);
```

Пусть все значения массива dp до клетки (*i*, *j*) уже посчитаны. Читаем очередное значение *val* = m[*i*][*j*] (входную матрицу в памяти не держим, читаем и обрабатываем ее на лету). Поскольку изначально мы обнулили массив dp, то при *val* = 0 значение dp[*i*][*j*] будет оставаться равным 0.

```
  if (val == 1)
  {
    dp[i][j] = min(min(dp[i][j-1],dp[i-1][j]),dp[i-1][j-1]) + 1;
```

Пересчитываем значения *size* и *cnt* для очередного квадрата размером dp[*i*][*j*].

```
    if (dp[i][j] == size) cnt++;
    if (dp[i][j] > size) {size = dp[i][j]; cnt = 1;}
  }
}
```

Выводим ответ.

```
printf("%lld\n",1LL * size * cnt);
```

1529. Починка забора

Вам необходимо починить старый забор. Забор состоит из набора досок, некоторые из которых выломаны. Доски пронумерованы слева направо в возрастающем порядке. Починка всех досок от *i*-ой до *j*-ой включительно, где *j* больше или равно *i*, стоит $\sqrt{j-i+1}$. Для уменьшения общей стоимости ремонта иногда выгодно ремонтировать даже целые доски. Необходимо найти минимальную стоимость ремонта всего забора.

Вам задана информация о заборе. Сломанные доски обозначаются символами 'X', а целые символами '.'. Найти наименьшую стоимость починки всего забора.

Вход. Каждая строка является отдельным тестом, описывающей забор. Она содержит только символы 'X' и '.'. Длина каждого забора не более 2500 символов.

Выход. Для каждого теста в отдельной строке вывести наименьшую стоимость починки всего забора с 4 десятичными цифрами.

Пример входа	**Пример выхода**
X.X...X.X	3.0000
X.X.....X	2.7321
X.............XX.X.......X...X..	5.0000

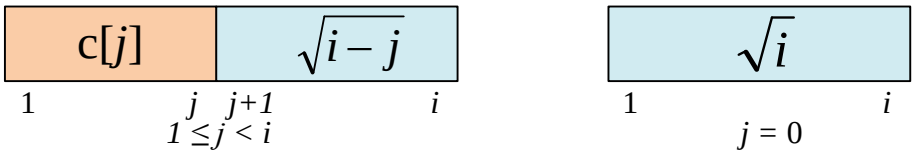
Пусть boards – строка, содержащая информацию о заборе. Заведем массив с длины 2502, в котором c[i] будет хранить минимальную стоимость, за которую можно починить забор от первой позиции до i - ой, причем c[0] положим равным 0. Тогда дешевле всего починить первые i секции (от первой до i - ой) забора следующим образом:

1. Если i - ая секция цела (boards[i] = ‘.’), то достаточно починить только первые i – 1 секций: c[i] = c[i – 1];

2. Если i - ая секция сломана (boards[i] = ‘X’), то чиним забор от первой до j - ой секции ($0 \le j < i$), потратив c[j] денег, а затем чиним доски от (j + 1) - ой до i - ой секции за $\sqrt{i-j}$ денег. При этом среди всех возможных j следует выбрать такое, для которого сумма c[j] + $\sqrt{i-j}$ наименьшая. То есть

$$c[i] = \min_{0 \le j \le i-1} \left(c[j] + \sqrt{i-j} \right)$$

При j = 0 весь забор от первой до i - ой секции следует починить при помощи одной новой доски, заплатив $\sqrt{i}$ денег.



Положив c[0] = 0, последовательно вычисляем c[1], c[2] , …, c[n], где n – длина забора. Ответом задачи будет значение c[n].

Реализация алгоритма

Объявим строку *boards*, в которой будем хранить состояние забора. В ячейках c[i] храним минимальную стоимость, за которую можно починить забор от первой до i – ой позиции. Значение c[0] положим равным 0.

```
#define MAX 2502
string boards;
double c[MAX];
```

Читаем состояние забора, обрабатываем тесты последовательно.

```
while (cin >> boards)
{
```

Сделаем так, чтобы забор начинался с позиции 1.

```
  boards = " " + boards;
```

Положим c[0] = 0.

```
  c[0] = 0;
```

Вычисляем минимальную стоимость починки части забора от 1 до *i*, результат заносим в c[*i*].

```
  for (i = 1; i < boards.length(); i++)
  {
    c[i] = INF;
    if (boards[i] == '.')
      c[i] = c[i - 1];
    else
      for (j = 0; j < i; j++)
        c[i] = min(c[i], c[j] + sqrt(1.0 * i - j));
  }
```

Выводим ответ.

```
  printf("%.4lf\n", c[boards.length() - 1]);
}
```

10648. Мстители

Нурлашко, Нурбакыт и Жора – члены последнего клана ниндзя, сражающегося против злого правления императора Рена. После сокрушительного поражения в открытом бою они решили разделить свою армию на три лагеря и начать партизанскую войну.

Нелепые реформы одного императора Рэна позволяют проезжать дороги между городами только в одном направлении. Также он выбрал разрешенные направления дорог таким образом, чтобы невозможно было начать движение и вернуться в один и тот же город после прохождения нескольких дорог.

Прямо сейчас клан решает, где разместить свои лагеря. Армия императора Рэна совершает регулярные набеги, осматривая какой-то путь. Если армия сокрушит все три лагеря во время рейда, клан не сможет перегруппироваться и проиграет войну. Помогите клану выбрать три города таким образом, чтобы не было пути, проходящего через все три города.

Вход. Первая строка содержит два числа n, m ($1 \le n, m \le 10^6$) – количество городов и дорог в империи. Следующие m строк содержат пары чисел v_i, u_i ($1 \le v_i, u_i \le n$), описывающие направленную дорогу от v_i к u_i.

Выход. Выведите три числа – индексы городов, в которых клан должен разместить свои лагеря. Если нет таких трех городов, то выведите -1. Если существует несколько ответов, то выведите любой из них.

Пример входа 1

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

Пример выхода 1

```
-1
```

Пример входа 2

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

Пример выхода 2

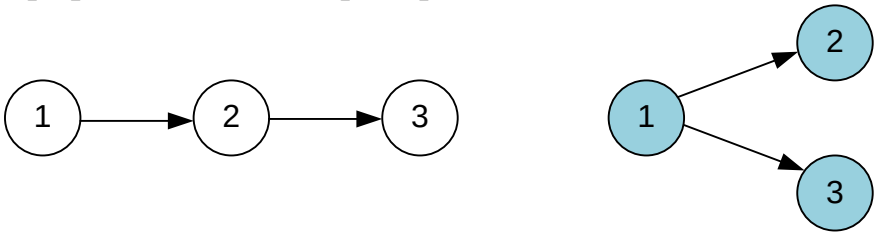
```
2 3 1
```

В графе требуется найти три любые вершины, не лежащие на одном пути.

Запустим алгоритм топологической сортировки Кана. Найдем две вершины, которые будут одновременно находиться в очереди. В этом случае не существует пути, на котором лежат эти две вершины. Добавив к ним любую третью вершину, получим ответ.

Если при реализации алгоритма Кана очередь всегда содержит не более одной вершины, то все вершины графа лежат на одном пути.

Пример. Графы из тестовых примеров имеют вид:



В первом примере все три вершины лежат на одном пути. Во втором примере три вершины не лежат на одном пути.

Реализация алгоритма

Объявим структуры данных. Входной граф содержится в списке смежности g. Входящие степени вершин храним в массиве *InDegree*.

```
vector<vector<int> > g;
vector<int> InDegree;
deque<int> q;
```

Читаем входные данные.

```
scanf("%d %d", &n, &m);
g.resize(n + 1);
InDegree.resize(n + 1);
for (i = 0; i < m; i++)
{
  scanf("%d %d", &a, &b);
  g[a].push_back(b);
}
```

Проходим по всем ребрам графа. Вычисляем входящие степени всех вершин. Для каждого ребра (*i*, *to*) увеличим InDegree[*to*] на 1.

```
for (i = 1; i < g.size(); i++)
for (j = 0; j < g[i].size(); j++)
{
  to = g[i][j];
  InDegree[to]++;
}
```

Все вершины, входящие степени которых равны нулю, заносим в очередь *q*.

```
for (i = 1; i < InDegree.size(); i++)
  if (!InDegree[i]) q.push_back(i);
```

Искомые три вершины сохраним в *x*, *y*, *z*.

```
x = y = -1;
```

Продолжаем работу алгоритма, пока очередь *q* не пуста.

```
while (!q.empty())
{
```

Если очередь содержит более одной вершины, то ответ найден.

```
  if (q.size() > 1)
  {
    x = q[0];
    y = q[1];
    break;
  }
```

Извлекаем вершину *v* из очереди (она будет текущей в топологическом порядке).

```
  v = q.front(); q.pop_front();
```

Удаляем из графа ребра (*v*, *to*). Для каждого такого ребра уменьшаем входящую степень вершины *to*. Если степень вершины *to* станет нулевой, то заносим ее в очередь.

```
  for (i = 0; i < g[v].size(); i++)
  {
    to = g[v][i];
    InDegree[to]--;
    if (!InDegree[to]) q.push_back(to);
  }
}
```

Алгоритм Кана завершен. Если значение *x* все еще равно -1, то решения не существует.

```
if (x == -1)
  printf("-1\n");
else
{
```

Решение существует. В качестве z выбираем минимальную вершину, не равную x и y.

```
  z = 1;
  while (z == x || z == y) z++;
    printf("%d %d %d\n", x, y, z);
}
```

10157. Транзитивное замыкание

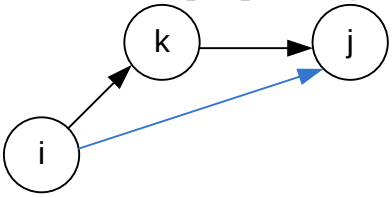
Найдите транзитивное замыкание ориентированного графа.

Вход. Ориентированный граф задан списком ребер. Первая строка содержит количество вершин n ($1 \le n \le 100$). Каждая из следующих строк содержит две вершины a и b ($1 \le a, b \le n$) описывающих ориентированное ребро от a к b.

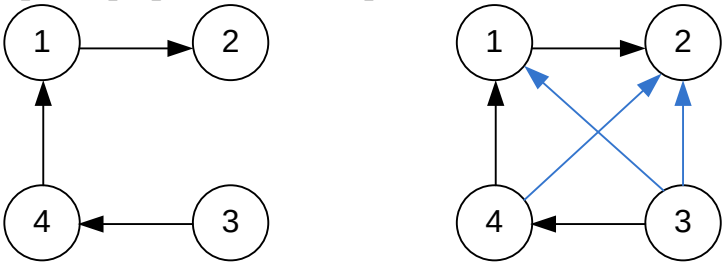
Выход. Выведите матрицу смежности транзитивного замыкания ориентированного графа.

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

В задаче следует найти транзитивное замыкание графа. Если граф содержит ребра $i \to k$ и $k \to j$, то следует добавить ребро $i \to j$.



Пример. Рассмотрим граф слева. Его транзитивное замыкание дано справа.



В транзитивном замыкании добавятся ребра 3 → 1 (существует путь 3 → 4 → 1), 3 → 2 (путь 3 → 4 → 1 → 2) и 4 → 2 (путь 4 → 1 → 2).

Реализация алгоритма

Объявим матрицу смежности g.

```
#define MAX 101
bool g[MAX][MAX];
```

Читаем входные данные. Строим граф.

```
scanf("%d", &n);
while (scanf("%d %d", &a, &b) == 2)
  g[a][b] = true;
```

Запускаем алгоритм транзитивного замыкания. Если существуют ребра $i \to k$ и $k \to j$, то следует провести ребро $i \to j$.

```
for (k = 1; k <= n; k++)
for (i = 1; i <= n; i++)
for (j = 1; j <= n; j++)
  if (g[i][k] && g[k][j]) g[i][j] = true;
```

Выводим матрицу смежности транзитивного замыкания графа.

```
for (i = 1; i <= n; i++)
{
  for (j = 1; j <= n; j++)
    printf("%d ", g[i][j]);
  printf("\n");
}
```

9608. Такси

Ивана и Сергея пригласили к участию в областной олимпиаде по информатике. Поскольку они проживают в разных населенных пунктах, Иван предложил воспользоваться службой такси, чтобы добраться до места проведения олимпиады.

Такси выезжает из пункта A, где проживает Иван, и направляется в пункт B, забирая Сергея, и дальше едет к месту проведения олимпиады (в пункт C).

Зная длины дорог между населенными пунктами области и стоимость проезда одного километра на такси, посчитайте на сколько больше денег потратил на поездку Иван, заезжая за Сергеем, чем следуя сразу к месту проведения олимпиады.

Известно, что такси всегда выбирает самый короткий из существующих маршрутов.

Вход. В первой строке записано пять чисел – четыре целых и одно действительное:

- n – количество населенных пунктов,
- A – населенный пункт, где проживает Иван,
- B – населенный пункт, где проживает Сергей,
- C – населенный пункт, где проходит олимпиада,
- d – стоимость проезда 1 километра на такси.

Следующая строка содержит количество дорог m. Каждая из следующих m строк описывает дорогу – два целых числа (номера населенных пунктов, которые соединяет дорога) и действительное число – длина соответствующей дороги в километрах.

Все целые числа натуральные и не больше 100.

Выход. Выведите действительное число – сумму, на которую пришлось заплатить больше, заезжая за Сергеем или -1, если из пункта A не существует маршрута до населенного пункта, где проживает Сергей, или из пункта A не существует маршрута к месту проведения олимпиады.

Пример входа	**Пример выхода**
5 1 2 3 4.25 5 1 4 2.5 4 2 3 4 5 3 3 5 2 3 4 8	25.5

Во взвешенном графе при помощи алгоритма Дейкстры ищем:

- кратчайший путь $dist_{ab}$ от A до B;
- кратчайший путь $dist_{ac}$ от A до C;
- кратчайший путь $dist_{bc}$ от B до C;

Расстояние от Ивана до места проведения олимпиады равно $dist_{ac}$.

Расстояние от Ивана до места проведения олимпиады с заездом за Сергеем равно $dist_{ab} + dist_{bc}$.

Ответ на задачу равен $(dist_{ab} + dist_{bc} - dist_{ac}) * d$.

Реализация алгоритма

Граф храним в весовой матрице g. Объявим массив кратчайших расстояний *dist* и массив вершин *used*, расстояние до которых уже посчитано.

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

Функция ***Dijkstra*** возвращает длину кратчайшего пути от *s* до *f*.

```
double Dijkstra(int s, int f)
{
  memset(used, 0, sizeof(used));
  for (i = 1; i <= n; i++) dist[i] = 1e18;
  dist[s] = 0;

  for (i = 1; i < n; i++)
  {
    double min = 1e18;
    int w = -1;
    for (j = 1; j <= n; j++)
      if (used[j] == 0 && dist[j] < min) { min = dist[j]; w = j; }

    if (min == 1e18) break;

    for (j = 1; j <= n; j++)
      if (used[j] == 0) dist[j] = minimum(dist[j], dist[w] + g[w][j]);

    used[w] = 1;
  }
  return dist[f];
}
```

Основная часть программы. Читаем входные данные.

```
scanf("%d %d %d %d %lf", &n, &a, &b, &c, &d);
scanf("%d", &m);
for (i = 1; i <= n; i++)
for (j = 1; j <= n; j++) g[i][j] = 1e18;

for (i = 0; i < m; i++)
{
  scanf("%d %d %lf", &s, &f, &w);
  g[s][f] = g[f][s] = w;
}
```

Вычисляем значения $dist_{ab}$, $dist_{bc}$, $dist_{ac}$.

```
double d1 = Dijkstra(a, b);
double d2 = Dijkstra(b, c);
double ds = Dijkstra(a, c);
```

Если одно из значений равно плюс бесконечности, то соответствующего пути не существует. Выводим -1.

```
if (d1 == 1e18 || d2 == 1e18 || ds == 1e18) printf("-1\n");
```

Иначе выводим ответ $(dist_{ab} + dist_{bc} - dist_{ac}) * d$.

```
else printf("%lf\n", (d1 + d2 - ds) * d);
```

9655. Мистер Найн и его любовь к манго

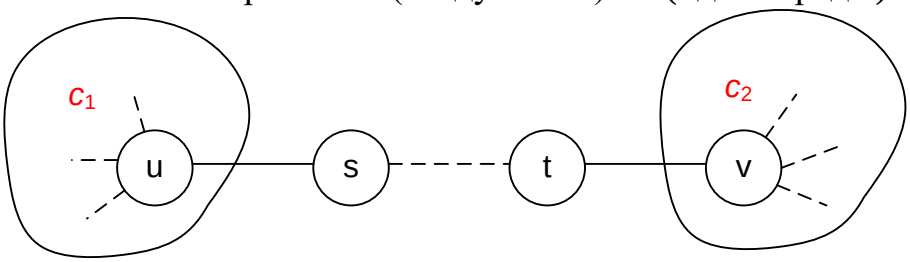
Мистер Найн во время перерывов в середине семестра случайно бродил по вселенной Параллель и неожиданно наткнулся на манговое дерево. Он любит манго, поэтому решил его сорвать. Но внезапно появилась фея и дала ему решить сложную задачу. Дерево содержит n узлов. Заданы также два узла u и v. Фея спрашивает, сколько существует таких пар узлов в дереве, что кратчайший путь между ними не содержит узла v после узла u (например, $u \to a \to b \to v$ также не допускается, где a и b – два разных узла). Если мистер Найн сможет определить правильное количество пар узлов, то получит все манго. Однако он не в состоянии это сделать и нуждается в Вашей помощи.

Вход. Первая строка содержит числа n ($1 \le n \le 300005$), u и v. Каждая из следующих $n - 1$ строк содержит два целых числа x и y означающих присутствие ребра между вершинами x и y ($1 \le x, y \le n$).

Выход. Выведите общее количество пар вершин.

Пример входа	**Пример выхода**
3 1 3 1 2 2 3	5

Поскольку на вход дается дерево, то существует единственный путь между u и v. Пусть этот путь имеет вид: $u \to s \to \ldots \to t \to v$. Заполним массив *parent*, чтобы можно было найти вершины s (следует за u) и t (идет перед v).

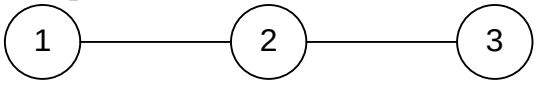


Пусть c_1 – количество вершин в поддереве с корнем u, при условии удаления ребра (u, s). Пусть c_2 – количество вершин в поддереве с корнем v, при условии удаления ребра (t, v). Количество путей, в которых после u идет v, равно $c_1 * c_2$.

Общее число путей на графе равно $n * (n - 1)$, где n – количество вершин. Граф неориентированный, путь из a в b и из b в a считаем разными. Количество искомых пар вершин равно

$$n * (n - 1) - c_1 * c_2$$

Пример. Граф из примера имеет вид:



Существует 5 возможных пар:

- (1, 2) : путь 1 → 2
- (2, 3) : путь 2 → 3
- (3, 2) : путь 3 → 2
- (2, 1) : путь 2 → 1
- (3, 1) : путь 3 → 2 → 1

Найн не может выбрать пару (1, 3), так как кратчайшим путем между ними будет 1 → 2 → 3 и он не приемлем, так как содержит 3 после 1, что не допустимо.

Реализация алгоритма

Объявим список смежности графа *g* и рабочие массивы.

```
vector<vector<int> > g;
vector<int> used, parent;
```

Функция ***dfs*** реализует поиск в глубину. Строим массив предков *parent*.

```
void dfs(int v)
{
  used[v] = 1;
  for (int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i];
    if (used[to] == 0)
    {
      parent[to] = v;
      dfs(to);
    }
  }
}
```

Функция ***dfs1*** реализует поиск в глубину из вершины *v*. В переменной c_1 подсчитываем количество вершин в поддереве с корнем *v*, при условии что переход в вершину *s* запрещен.

```
void dfs1(int v)
{
  used[v] = 1;
  c1++;
  for (int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i];
    if (to == s) continue;
    if (used[to] == 0) dfs1(to);
  }
}
```

Функция ***dfs2*** реализует поиск в глубину из вершины *v*. В переменной c_2 подсчитываем количество вершин в поддереве с корнем *v*, при условии что переход в вершину *t* запрещен.

```
void dfs2(int v)
{
  used[v] = 1;
  c2++;
  for (int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i];
    if (to == t) continue;
    if (used[to] == 0) dfs2(to);
  }
}
```

Основная часть программы. Читаем входные данные. Строим граф.

```
scanf("%d %d %d", &n, &start, &finish);
g.resize(n + 1);
used.resize(n + 1);
parent.resize(n + 1);
for (i = 0; i < n - 1; i++)
{
  scanf("%d %d", &a, &b);
  g[a].push_back(b);
  g[b].push_back(a);
}
```

Запускаем поиск в глубину из вершины *start*.

```
dfs(start);
```

Используя массив предков *parent*, находим вершины *s* и *t*:

$$start \to s \to \ldots \to t \to finish$$

```
t = parent[finish];
s = finish;
while (parent[s] != start) s = parent[s];
```

При помощи поиска в глубину вычисляем значения c_1 и c_2.

```
c1 = 0;
used.clear(); used.resize(n + 1);
dfs1(start);

c2 = 0;
used.clear(); used.resize(n + 1);
dfs2(finish);
```

Выводим ответ.

```
res = 1LL * n * (n - 1) - c1 * c2;
printf("%lld\n", res);
```

1941. Предок

Напишите программу, которая для двух вершин дерева определяет, является ли одна из них предком другой.

Вход. Первая строка содержит количество вершин в дереве n ($1 \le n \le 10^5$). Во второй строке находится n чисел, i-ое из которых определяет номер непосредственного родителя вершины с номером i. Если это число равно нулю, то вершина является корнем дерева.

В третьей строке находится количество запросов m ($1 \le m \le 10^5$). Каждая из следующих m строк содержит два различных числа a и b ($1 \le a, b \le n$).

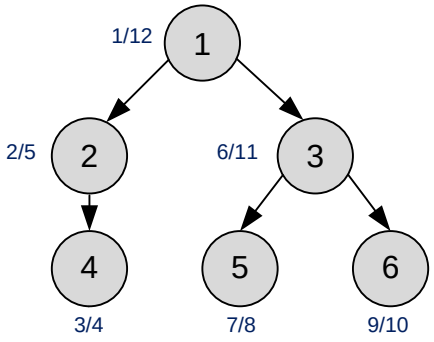
Выход. Для каждого из m запросов выведите в отдельной строке число 1, если вершина a является одним из предков вершины b, и 0 в противном случае.

Пример входа	**Пример выхода**
6 0 1 1 2 3 3 5 4 1 1 4 3 6 2 6 6 5	0 1 1 0 0

Дерево будем хранить в виде ориентированного графа, в котором ребра идут от предков к сыновьям (для экономии памяти). Реализуем на нем поиск в глубину, расставив для каждой вершины v метки d[v] и f[v]. Вершина a является одним из предков вершины b, если d[a] < d[b] < f[b] < f[a]. Это означает, что интервал [d[b]; f[b]] должен включаться в интервал [d[a]; f[a]].

Но поскольку для каждой вершины b всегда выполняется неравенство d[b] < f[b], то для каждой входной пары вершин (запроса) a и b достаточно проверить неравенства d[a] < d[b] и f[b] < f[a].

Пример. Граф, приведенный в примере, с расстановками меток имеет следующий вид:



Например, 1 является предком 5, так как d[1] < d[5] и f[5] < f[1]. Действительно, интервал [7; 8] включается в [1; 12].

Реализация алгоритма

Поскольку количество вершин в графе велико, будем хранить граф в виде списка смежности *g*. Массив *used* используется для отмечания уже пройденных вершин. Метки времени входа и выхода из вершины будем хранить в массивах *d* и *f*.

```
vector<vector<int> > g;
vector<int> used, d, f;
```

Запускаем процедуру поиска в глубину ***dfs***. Расставляем метки d[*v*] и f[*v*].

```
void dfs(int v)
{
  used[v] = 1; d[v] = time++;
  for(int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i];
    if (!used[to]) dfs(to);
  }
  f[v] = time++;
}
```

Функция ***is_Parent*** возвращает 1, если *a* является предком *b*, и 0 иначе. Проверяем выполнение двух неравенств d[*a*] < d[*b*] и f[*b*] < f[*a*].

```
int is_Parent(int a, int b)
{
  return (d[a] < d[b]) && (f[b] < f[a]);
}
```

Основная часть программы. Читаем входной граф. Если вершина *a* является родителем вершины *i*, то добавим в граф ориентированное ребро (*a*, *i*) (для экономии памяти можно воспользоваться ориентированным графом). Вершины графа нумеруются числами от 1 до *n*. Если *a* = 0, то вершина *i* является корнем, в этом случае никакого ребра добавлять не надо. В переменной *root* ищем номер вершины – корня дерева.

```
scanf("%d",&n);
g.resize(n+1); used.resize(n+1);
d.resize(n+1); f.resize(n+1);
for(i = 1; i <= n; i++)
{
  scanf("%d",&a);
  if(!a) root = i; else g[a].push_back(i);
}
```

Запускаем поиск в глубину из корня дерева *root*.

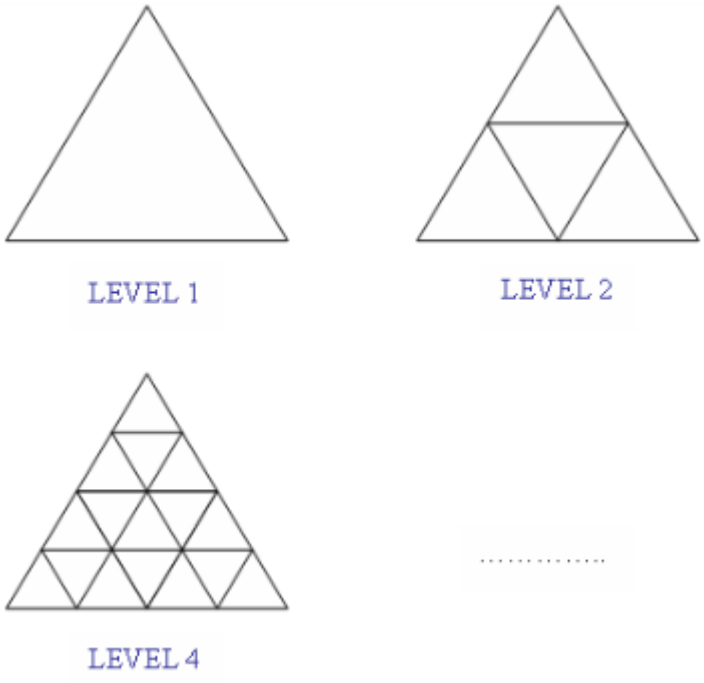
```
dfs(root);
```

Читаем запросы и выводим ответ на каждый из них за константное время.

```
scanf("%d",&m);
for(i = 0; i < m; i++)
{
  scanf("%d %d",&a,&b);
  printf("%d\n",is_Parent(a,b));
}
```

4556. Подсчет треугольников

Определим УРОВЕНЬ треугольника следующим изображением:



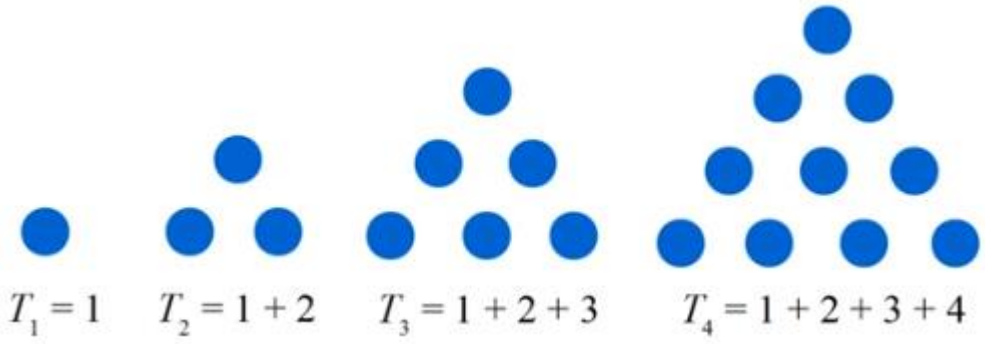
Вам следует подсчитать количество всех возможных треугольников в самом большом (на уровне n).

Вход. Первая строка содержит количество тестов t ($t \le 10000$). Каждая строка содержит одно целое число n ($1 \le n \le 10^6$) – уровень треугольника.

Выход. Для каждого теста вывести в отдельной строке количество треугольников в наибольшем (на уровне n). Все ответы помещаются в целочисленный 64-битовый тип.

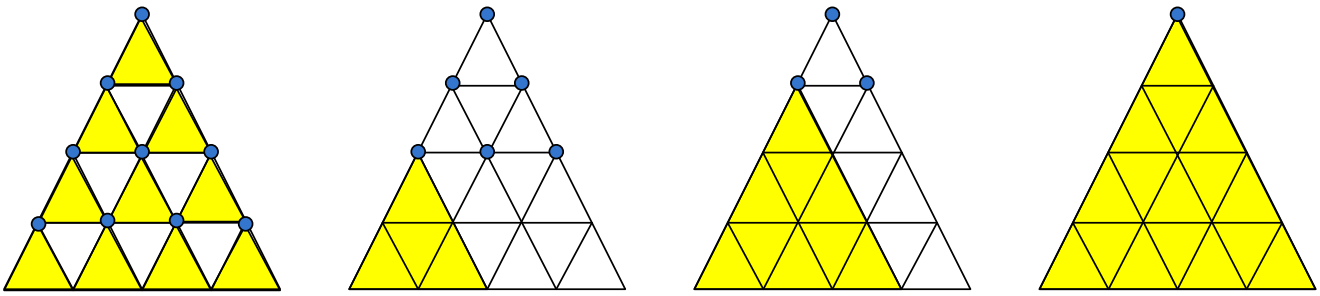
Пример входа	**Пример выхода**
3 1 2 3	1 5 13

Рассмотрим ряд треугольников:



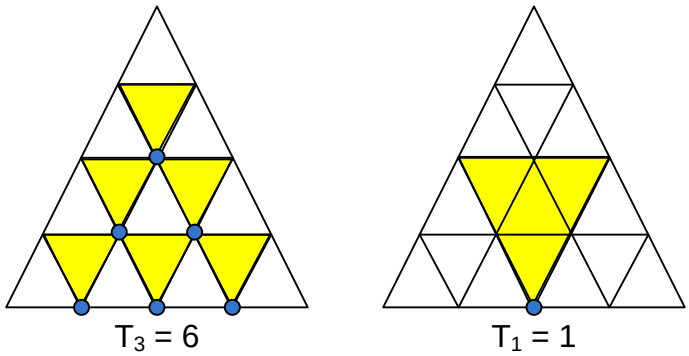
Обозначим через T_n количество точек в нем (назовем T_n треугольными числами). По сумме арифметической прогрессии $T_n = n\,(n + 1) / 2$.

Подсчитаем сначала количество треугольников S_n^1 с вершиной вверх. Каждая жирная точка на исходном треугольнике является вершиной некоторого меньшего треугольника.



Количество треугольников со стороной 1 равно T_n.
Количество треугольников со стороной 2 равно T_{n-1}.
…
Количество треугольников со стороной n равно T_1.
Итого $S_n^1 = T_n + T_{n-1} + T_{n-2} + \ldots + T_2 + T_1$ или $S_n^1 = T_n + S_{n-1}^1$.

Теперь подсчитаем количество треугольников S_n^2 с вершиной вниз.



Количество треугольников со стороной 1 равно T_{n-1}.
Количество треугольников со стороной 2 равно T_{n-3}.
Количество треугольников со стороной 3 равно T_{n-5}.

…

Количество треугольников со стороной n равно 0.

Для четного n имеем: $S_n^2 = T_{n-1} + T_{n-3} + \ldots + T_3 + T_1$.

Для нечетного n имеем: $S_n^2 = T_{n-1} + T_{n-3} + \ldots + T_4 + T_2$.

Или $S_n^2 = T_{n-1} + S_{n-2}^2$.

Реализация алгоритма

Объявим рабочий массив.

```
#define MAX 1000010
long long s[MAX][2];
```

Вычисление значения $T_n = n\,(n + 1) / 2$.

```
long long T(long long n)
{
  return n * (n + 1) / 2;
}
```

Основная часть программы. Инициализируем и затем вычисляем значения S_n^1 и S_n^2.

```
s[1][0] = 1; s[2][0] = 3;
s[1][1] = 0; s[2][1] = 1;

for(i = 2; i < MAX; i++)
{
  s[i][0] = s[i-1][0] + T(i);
  s[i][1] = s[i-2][1] + T(i-1);
}
```

Читаем входные данные и выводим ответ.

```
scanf("%d",&tests);
while(tests--)
{
  scanf("%lld",&n);
  printf("%lld\n",s[n][0] + s[n][1]);
}
```

50. Разрезанное число

Василий на бумажке в виде полоски написал число, кратное d. Его младший брат Дмитрий разрезал число на k частей. Василий решил восстановить написанное число, но столкнулся с проблемой. Он помнил только число d, а чисел, кратных d, можно сложить несколько.

Сколько чисел, кратных числу d, может составить Василий, если составляя исходное число, он использует все части.

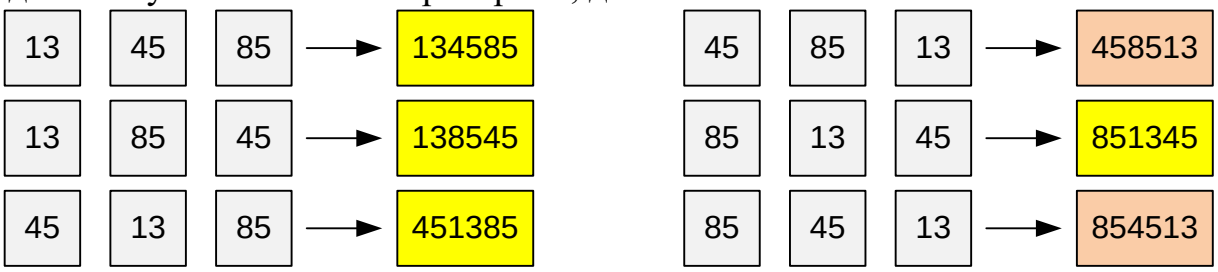
Вход. В первой строке записано два числа d и k ($1 \le k < 9$, $1 \le d \le 100$). В следующих k строках находятся части числа. Количество цифр в разрезанных частях не превышает 10.

Выход. Выведите количество разных чисел, кратных d.

Пример входа	**Пример выхода**
5 3 13 85 45	4

Занесем числа всех частей в массив. Рассмотрим все возможные склейки имеющиеся частей. Такой перебор возможен, так как $k < 9$ и разных склеек будет не более 9!. Для каждого полученного склейкой числа проверяем, делится ли оно на d.

Пример. Рассмотрим все возможные склейки трех частей 13, 85 и 45. Для каждого полученного числа проверяем, делится ли оно на $d = 5$.



Реализация алгоритма

Функция ***f*** возвращает значение 10^k, где k – количество цифр в числе n.

```
long long f(long long n)
{
  long long res = 1;
  while (n > 0)
  {
    res *= 10;
    n /= 10;
  }
  return res;
}
```

Основная часть программы. Числа, записанные в частях бумажки, занесем в массив v.

```
scanf("%d %d", &d, &k);
for (c = i = 0; i < k; i++)
{
  scanf("%lld", &value);
  v.push_back(value);
}
```

Отсортируем числа, записанные на частях.

```
sort(v.begin(), v.end());
```

Перебираем все возможные перестановки частей при помощи ***next_permutation***. В переменной *value* вычисляем остаток от деления склеенного числа на *d*.

```
do
{
  for (i = value = 0; i < k; i++)
```

К числу *value* справа приписываем v[*i*].

```
    value = (value * f(v[i]) + v[i]) % d;
```

Если полученное склеенное число *value* делится на *d*, то увеличиваем счетчик *c* на 1.

```
  if (value % d == 0) c++;
} while (next_permutation(v.begin(), v.end()));
```

Выводим ответ.

```
printf("%d\n", c);
```

4106. Генерация подмножеств

Задано множество *s* мощности *n*, содержащее все элементы из интервала [1..*n*]. Необходимо сгенерировать все его подмножества.

Вход. Единственное число *n* ($1 \le n \le 8$).

Выход. В каждой строчке необходимо вывести одно из подмножеств исходного множества. Подмножество записывается перечислением своих элементов в порядке возрастания. Элементы подмножества должны быть записаны слитно, то есть без пробелов. Каждое подмножество должно встречаться не более одного раза. Подмножества также нужно перечислять в возрастающем порядке (см. пример). Пустое подмножество выводить не нужно.

Пример входа	**Пример выхода**
3	1 2 3 12 13 23 123

В задаче необходимо сгенерировать все подмножества заданного множества. Для этого переберем (в цикле по i) все числа от 1 до $2^n - 1$. Число i представляем в двоичном виде и рассматриваем его последние n бит (возможно с нулями впереди). Такому двоичному представлению соответствует подмножество: если на его k-ом месте находится единица, то в подмножество включается число k ($1 \le k \le n$). Например, если $n = 3$ и $i = 2$, то его битовое представление 010 и ему соответствует множество {2}.

Пример. Рассмотрим генерацию подмножеств для $n = 3$.

	3	2	1	
0	0	0	0	{ }
1	0	0	1	{ 1 }
2	0	1	0	{ 2 }
3	0	1	1	{ 2, 1 }

	3	2	1	
4	1	0	0	{ 3 }
5	1	0	1	{ 3, 1 }
6	1	1	0	{ 3, 2 }
7	1	1	1	{ 3, 2, 1 }

Реализация алгоритма

Подмножества будем генерировать в виде чисел в массиве v. Например, подмножество {1, 2, 3} будем представлять числом 123.

```
#define MAX 8
int v[1<<MAX];
```

Читаем значение n. Генерируем в цикле по i все $2^n - 1$ подмножеств (кроме пустого).

```
scanf("%d",&n);
for(i = 1; i < (1 << n); i++)
{
  for(val = j = 0; j < n; j++)
    if (i & (1 << j)) val = val * 10 + j + 1;
  v[i] = val;
}
```

Отсортируем подмножества по возрастанию чисел, их представляющих.

```
sort(v,v + (1<<n));
```

Выводим подмножества в требуемом порядке.

```
for(i = 1; i < 1 << n; i++)
  printf("%d\n",v[i]);
```