Задачи на деревья

2157. Сумма
2923. Дерево
3384. Диспетчер
3983. Манкунианец и цветное дерево
9655. Мистер Найн и его любовь к манго
10653. Сумма XOR
10654. Уникальный цвет
10655. Вирусное дерево 2
10656. Дерево коротких расстояний
10667. Интервалы на дереве
10684. Улучшение дорог
11002. Покраска дерева
11058. Погоня за бабочкой
11153. Кефа и парк

2157. Сумма

Родители подарили Роману неориентированный связный взвешенный граф с n вершинами и $n - 1$ ребрами. Роман хочет найти суммарную длину всех путей в графе. Длина пути равна сумме длин ребер в нем. Роман считает, что путь из u в v такой же как и из v в u, поэтому он не различает их.

Вход. Первая строка содержит количество вершин в графе n ($2 \le n \le 10^5$). Следующие $n - 1$ строк описывают ребра. Каждая строка содержит три целых числа: номера вершин, соединенных ребром (вершины пронумерованы числами от 1 до n), и вес ребра.

Выход. Вывести сумму длин всех путей, вычисленную по модулю 10^9.

Пример входа 1

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

Пример выхода 1

```
8
```

Пример входа 2

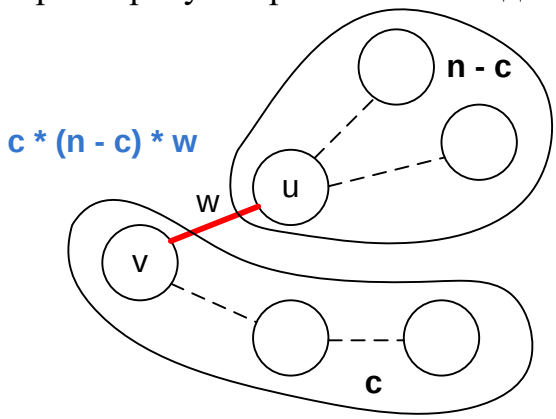
```
6
1 2 5
1 3 1
2 4 2
2 5 4
2 6 3
```

Пример выхода 2

```
90
```

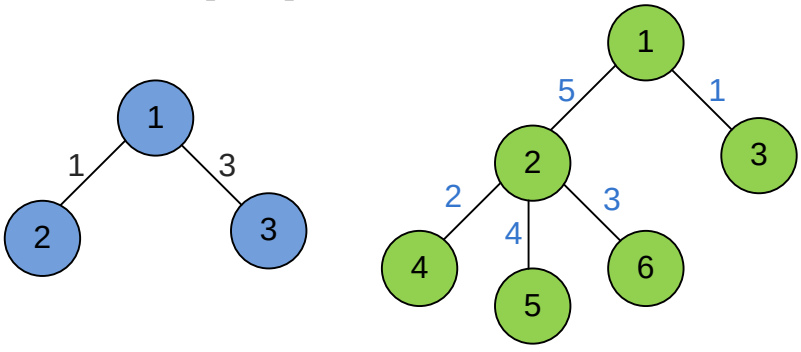
В первом примере все пути на дереве: 1 – 2, 1 – 3, 2 – 1 – 3, сумма их длин равна 1 + 3 + 4 = 8.

Запустим поиск в глубину из некоторой вершины дерева. Рассмотрим ребро (u, v) дерева с весом w. Пусть количество вершин в поддереве с корнем v равно c. Тогда в дереве с одной стороны ребра находится c вершин, а с другой n – c вершин. Ребро (u, v) входит в c * (n – c) различных путей. Поэтому вклад этого ребра в сумму длин всех путей составляет c * (n – c) * w. Остается перебрать поиском в глубину все ребра и просуммировать их вклад в общую сумму длин.

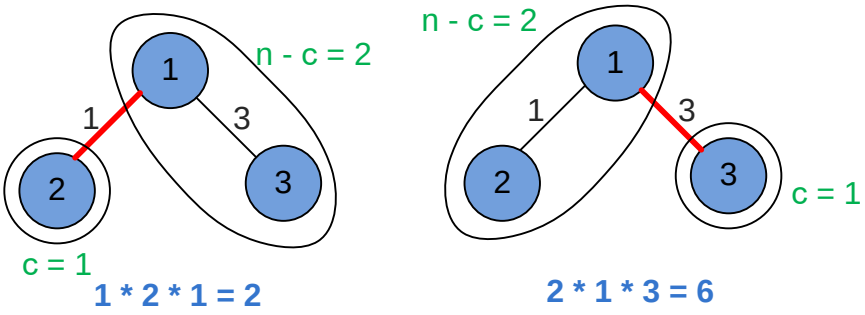


Пример

Графы, приведенные в примерах, имеют вид:



Рассмотрим вклады ребер в общую сумму длин всех путей в первом примере.



Ребро (1, 2) весом 1 принадлежит двум путям:

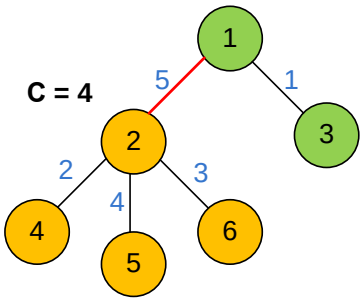
- 1 – 2;
- 2 – 1 – 3;

Его вклад в общую сумму составляет 1 * 2 * 1 = 2.
Ребро (1, 3) весом 3 принадлежит двум путям:

- 1 – 3;
- 2 – 1 – 3;

Его вклад в общую сумму составляет 2 * 1 * 3 = 6.
Сумма длин всех путей равна 2 + 6 = 8.

Во втором примере рассмотрим вклад ребра (1, 2) весом 5 в общую сумму длин всех путей.



Количество вершин в поддереве с корнем 2 равно $c = 4$ (включая саму вершину 2). Тогда с другой стороны ребра (1, 2) находится $n - c = 6 - 4 = 2$ вершины. Следовательно количество разных путей, содержащих ребро (1, 2), равно $c * (n - c) = 4 * 2 = 8$. Действительно, такими путями будут

1 – 2, 1 – 2 – 4, 1 – 2 – 5, 1 – 2 – 6,
3 – 1 – 2, 3 – 1 – 2 – 4, 3 – 1 – 2 – 5, 3 – 1 – 2 – 6

Вклад ребра (1, 2) весом 5 в сумму длин всех путей составляет $c * (n - c) * w = 4 * 2 * 5 = 40$.

Реализация алгоритма

Входной взвешенный граф храним в списке смежности g.

```
#define MOD 1000000000;
vector<vector<pair<int,int> > > g;
```

Функция ***dfs*** реализует поиск в глубину из вершины *v* возвращает количество вершин в поддереве с корнем *v* (включая саму вершину *v*). Подсчет этих вершин происходит в переменной *cnt*. Изначально положим *cnt* = 1, считая что в поддерево включена сама вершина *v*.

```
int dfs(int v, int p = -1)
{
  int cnt = 1, c;
  for(int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i].first;
    int w = g[v][i].second;
```

Рассмотрим ребро (*v*, *to*) с весом *w*. Вычисляем количество вершин *c* в поддереве с корнем *to*. Таким образом в дереве с одной стороны ребра находится *c* вершин, а с другой *n* – *c* вершин. Ребро (*v*, *to*) входит в *c* * (*n* – *c*) различных путей. Поэтому вклад этого ребра в сумму длин всех путей составляет *c* * (*n* – *c*) * *w*.

```
    if (to != p)
    {
      c = dfs(to, v);
      res = (res + 1LL * c * (n - c) * w) % MOD;
      cnt += c;
    }
  }
  return cnt;
}
```

Основная часть программы. Читаем взвешенный граф в список смежности *g*.

```
scanf("%d",&n);
g.resize(n+1);
for(i = 1; i < n; i++)
{
  scanf("%d %d %d",&u,&v,&d);
  g[u].push_back(make_pair(v,d));
  g[v].push_back(make_pair(u,d));
}
```

Запускаем поиск в глубину из вершины 1. Вершины графа нумеруются с 1 до *n*.

```
dfs(1);
```

Выводим ответ.

```
printf("%lld\n",res);
```

2923. Дерево

Задано подвешенное дерево, содержащее n ($1 \le n \le 10^6$) вершин. Каждая вершина покрашена в один из n цветов. Требуется для каждой вершины v вычислить количество различных цветов, встречающихся в поддереве с корнем v.

Вход. В первой строке задано число n. Следующие n строк описывают вершины по одной в строке. Описание очередной вершины i имеет вид p_i c_i, где p_i – номер родителя вершины i, а c_i – цвет вершины i ($1 \le c_i \le n$). Для корня дерева $p_i = 0$.

Выход. Выведите n чисел, обозначающих количества различных цветов в поддеревьях с корнями в вершинах 1, 2, ..., n.

Пример входа

```
5
2 1
3 2
0 3
3 3
2 1
```

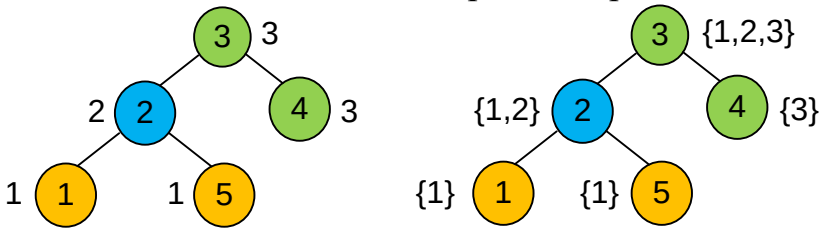
Пример выхода

```
1 2 3 1 1
```

Запустим поиск в глубину из корня дерева. Для каждой вершины i храним множество s_i, в котором будем накапливать цвета вершин в ее поддеревьях. Если j – сын вершины i при поиске в глубину, то s_j должно включаться в s_i. Количество различных цветов в поддереве с корнем i равно размеру множества s_i.

Пример

Слева возле каждой вершины приведен ее цвет. Справа возле каждой вершины приведено множество цветов в поддереве с корнем в этой вершине.



Реализация алгоритма

В ячейке color[i] храним цвет i-ой вершины. Во множестве s[i] будем накапливать цвета в поддереве с корнем i. Ориентированное дерево храним в списке смежности графа g. В res[i] храним количество различных цветов в поддереве с корнем i.

```
#define MAX 1000010
int color[MAX], res[MAX];
set<int> s[MAX];
vector<vector<int> > g;
```

Поиск в глубину из вершины *v*. Изначально заносим в s[*v*] цвет вершины *v*. Для каждого ребра дерева (*v*, *to*) добавляем множество s[*to*] в s[*v*]. Количество различных цветов в поддереве с корнем *v* равно размеру множества s[*v*], заносим его в res[*v*].

```
void dfs(int v)
{
  int i, to;
  s[v].insert(color[v]);
  for(i = 0; i < g[v].size(); i++)
  {
    to = g[v][i];
    dfs(to);
```

Если размер множества s[*v*] меньше размера множества s[*to*], то меняем их местами. Далее содержимое меньшего множества s[*to*] добавляем во множество s[*v*].

```
    if (s[v].size() < s[to].size()) s[v].swap(s[to]);
    s[v].insert(s[to].begin(), s[to].end());
```

Очищаем множество s[*to*] – оно нам больше не пригодится.

```
    s[to].clear();
  }
  res[v] = s[v].size();
}
```

Основная часть программы. Читаем входные данные.

```
scanf("%d",&n);
g.resize(n+1);
for(i = 1; i <= n; i++)
{
  scanf("%d %d",&p,&c);
  g[p].push_back(i);
  color[i] = c;
}
```

Запускаем поиск в глубину из корня дерева – нулевой вершины.

```
dfs(0);
```

Выводим ответ.

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

3384. Диспетчер

В клане ниндзя, ниндзя отправляют к клиенту, и потом они получают вознаграждение в соответствии с их работой.

В клане ниндзя имеется один ниндзя, именуемый Мастером. Каждый ниндзя кроме Мастера имеет ровно одного босса. Чтобы гарантировать конфиденциальность и поощрить лидерство, все инструкции по заданиям всегда передаются боссом своим подчинённым. Другими методами запрещается передавать инструкции.

Вы хотите собрать некоторое количество ниндзя и отправить их к клиенту. Вы должны заплатить каждому из отправленных ниндзя. Для каждого ниндзя зафиксирована оплата. Суммарная плата должна уложится в бюджет. Кроме того, чтобы передавать инструкции отправленным ниндзя, Вы должны выбрать одного ниндзя как менеджера, который сможет посылать инструкции всем им. Ниндзя, который не был выбран, может передавать сообщения. Менеджер не обязательно должен быть отправлен к клиенту. Если менеджер не отправлен, ему не нужно платить.

Вы хотите максимизировать степень удовлетворённости клиента, оставаясь в рамках бюджета. Степень удовлетворённости вычисляется как произведение общего количества отправленных ниндзя и уровня лидерства менеджера. Для каждого ниндзя обозначен его уровень лидерства.

Напишите программу, которая зная для каждого ниндзя его босса b_i, размер оплаты c_i, уровень лидерства l_i ($1 \le i \le n$), и размер бюджета m, выведет максимально возможное значение уровня удовлетворённости клиента, при условии, что менеджер и отправленные ниндзя выбраны так, что все условия соблюдены.

Вход. Первая строка содержит количество ниндзя n ($1 \le n \le 10^5$) и бюджет m ($1 \le m \le 10^9$). Следующие n строк описывают босса, зарплату и уровень лидерства каждого ниндзя. (i + 1) - ая строка содержит три целых числа b_i, c_i, l_i ($0 \le b_i < i$, $1 \le c_i \le m$, $1 \le l_i \le 10^9$), обозначающих босса i-го ниндзя b_i, зарплату i-го ниндзя c_i, и его лидерский уровень l_i. i-ый ниндзя является Мастером, если $b_i = 0$. Так как всегда соблюдается неравенство $b_i < i$, то для каждого ниндзя номер его босса всегда меньше номера его самого.

Выход. Выведите максимальное возможное значение уровня удовлетворённости клиента.

Пример входа

```
5 4
0 3 3
1 3 5
2 2 2
1 2 4
2 3 1
```

Пример выхода

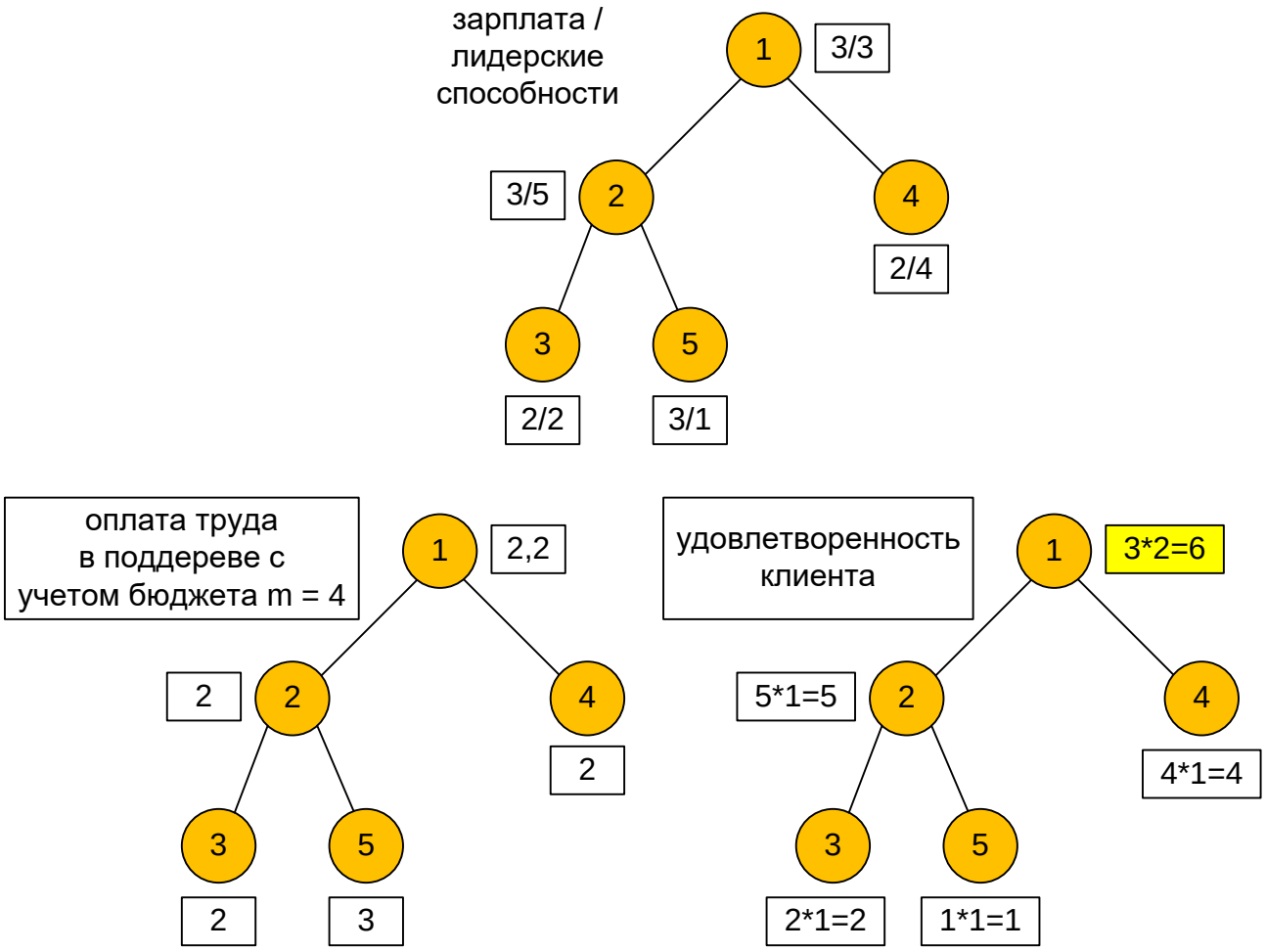
```
6
```

Запустим поиск в глубину из вершины 0. Для каждой вершины дерева v построим мультимножество зарплат, находящихся в поддереве с корнем v. Будем, например, хранить такое мультимножество в очереди с приоритетами. Одновременно будем поддерживать значение суммы всех этих зарплат.

Пусть ниндзя v выбран в качестве менеджера. Выгодно отправлять к клиенту тех ниндзя, которые требуют меньшую зарплату. Удаляем из мультимножества наибольшие зарплаты до тех пор, пока сумма оставшихся зарплат не станет меньше или равной бюджету m. Вычисляем удовлетворенность клиента.

Перебираем всех ниндзя в качестве возможных менеджеров, ищем максимальное значение удовлетворенности клиента.

Пример



Рассмотрим вершину номер 2. При объединении ее зарплаты с зарплатами ее сыновей получим {2, 3, 3}. Бюджет равен $m = 4$. Следует произвести оплату тем кто требует меньше всего и помещается в бюджет. Поэтому множество зарплата станет {2}.

Удовлетворенность клиента считаем в каждой вершине как произведение лидерских качеств ниндзя (менеджера) на количество элементов в оплате труда.

Пусть ниндзя 1 – менеджер. Тогда он отправит к клиенту ниндзя с номерами 3 и 4, заплатив им 2 + 2 = 4, таким образом вложившись в бюджет m = 4. Удовлетворенность клиента равна лидерским способностям менеджера 1, умноженная на количество отправленных к клиентам ниндзя, то есть 3 * 2 = 6.

Реализация алгоритма

Граф храним в g. Объявим массив очередей с приоритетами pq: pq[v] хранит мультимножество зарплат, находящихся в поддереве с корнем v. Значение sum[v] хранит сумму чисел в мультимножестве pq[v].

```
#define MAX 100010
priority_queue<int> pq[MAX];
vector<vector<int> > g;
int cost[MAX], leader[MAX];
long long sum[MAX];
```

Поиск в глубину из вершины v. Строим мультимножество pq[v].

```
void dfs(int v)
{
  int i, to;
  pq[v].push(cost[v]);
  sum[v] += cost[v];

  for (i = 0; i < g[v].size(); i++)
  {
    to = g[v][i];
    dfs(to);
```

Вершина *to* является сыном v. Занесем все элементы pq[*to*] в pq[v]. Размер очереди, куда будут переноситься числа, должен быть больше.

```
    if (pq[v].size() < pq[to].size()) swap(pq[v], pq[to]);
```

Переносим числа в pq[v], одновременно очищая в pq[*to*] для экономии памяти.

```
    while (pq[to].size() > 0)
    {
      pq[v].push(pq[to].top());
      pq[to].pop();
    }
```

Добавляем к sum[v] сумму чисел мультимножества потомка sum[*to*].

```
    sum[v] += sum[to];
```

На вершине очереди находятся большие числа (у нас ***max*** - куча). Удаляем их пока сумма чисел в оставшемся мультимножестве не будет превышать бюджет m.

```
    while (sum[v] > m)
    {
      sum[v] -= pq[v].top();
```

```
      pq[v].pop();
    }
  }
  res = max(res, leader[v] * (long long)pq[v].size());
}
```

Основная часть программы. Читаем входной граф.

```
scanf("%d %d",&n,&m);
g.resize(n+1);
for (i = 1; i <= n; i++)

  scanf("%d %d %d",&parent,&cost[i],&leader[i]);
  g[parent].push_back(i);
}
```

Запускаем поиск в глубину из нулевой вершины.

```
dfs(0);
```

Выводим результат.

```
printf("%lld\n", res);
```

3983. Манкунианец и цветное дерево

После напряженной недели на работе, жители Манчестера и Ливерпуля решили пойти в поход на выходные. Когда они проходили по лесу, они наткнулись на уникальное дерево, состоящее из n вершин. Вершины пронумерованы числами от 1 до n.

Каждой вершине дерева поставлен в соответствие цвет (из c возможных цветов). Чтобы побороть скуку, они решили проверить вместе свои навыки рассуждения. Корнем дерева является вершина 1. Для каждой вершины они решили найти ближайшего предка, цвет которого совпадает с цветом вершины.

Вход. Первая строка содержит два целых числа n и c ($1 \le n, c \le 10^5$) – количество вершин в дереве и количество возможных цветов.

Вторая строка содержит $n - 1$ число. i-ое число указывает на отца $i + 1$ - ой вершины.

Третья строка содержит n целых чисел, задающих цвета вершин. Значения цветов лежат в промежутке от 1 до c включительно.

Выход. В одной строке выведите n чисел. i-ым числом является вершина, являющаяся ближайшим предком i-ой вершины, имеющей такой же цвет. Если такого предка для вершины не существует, то вывести -1.

Пример входа

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

Пример выхода

```
-1 -1 -1 1 3
```

Рассмотрим более простой вариант задачи. Пусть все вершины дерева покрашены в один цвет. Заведем стек, который сначала пустой. Запустим поиск в глубину из корня дерева. При входе в вершину v кладем в стек значение v, а при завершении обработки вершины v удаляем верхушку стека (это как раз будет вершина v). Когда поиск в глубину завершится, стек окажется пустым.

Вопрос: что будет на вершине стека при входе в вершину v и еще до того как мы положим v в стек?

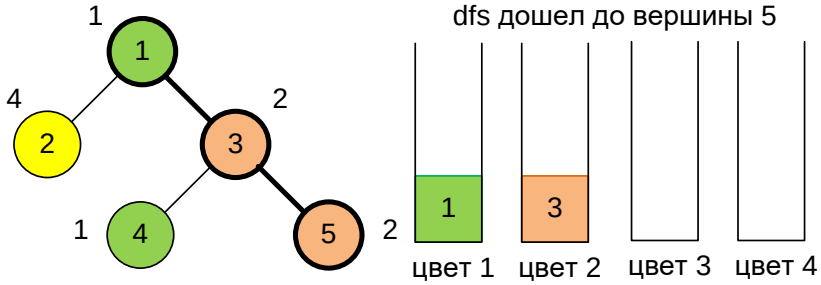
Теперь перейдем к решению нашей задачи. Заведем c стеков, по одному для каждого цвета (например, вектор стеков). Изначально все стеки пустые. Запустим поиск в глубину из корня – вершины 1. Обработка вершины v цвета *color* состоит из следующих шагов:

- Если стек s[*color*] не пуст, то на его вершине находится номер вершины, являющейся ближайшим предком v, имеющей такой же цвет как и v. Если стек пуст, то требуемой вершины не существует, ответом для вершины v будет -1.
- Заносим вершину v в стек s[*color*].
- Запускаем поиск в глубину со всех сыновей вершины v.
- Удаляем вершину из стека s[*color*].

Когда мы находимся при поиске в глубину в вершине v, в стеках хранится информация о цветах всех вершин, расположенных на единственном пути от корня до v. То есть стек s[*color*] содержит номера вершин на пути от корня до v, которые имеют цвет *color*. При этом вершины в стек заносятся в порядке их посещения поиском в глубину.

Пример

Когда поиск в глубину дойдет до вершины 5, из $c = 4$ стеков два будут пустыми (соответствующие цветам 3 и 4). Стек номер 1 (первый цвет) содержит вершину 1, Стек номер 2 (второй цвет) содержит вершину 3. Вершина номер 5 имеет цвет 2, следовательно ближайшим предком с тем же цветом будет вершина, находящаяся на вершине стека 2. Таким предком для вершины 5 будет вершина 3.



Реализация алгоритма
Объявим рабочие массивы.

```
vector<int> col, res;
vector<vector<int> > g;
vector<stack<int> > s;
```

Поиск в глубину с вершины *v*.

```
void dfs(int v)
{
  int color = col[v];
  if(s[color].empty())
    res[v] = -1;
  else
    res[v] = s[color].top();

  s[color].push(v);

  for(int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i];
    dfs(to);
  }
  s[color].pop();
}
```

Основная часть программы. Читаем дерево.

```
scanf("%d %d",&n,&c);
g.resize(n+1);
for(i = 2; i <= n; i++)
{
  scanf("%d",&val);
  g[val].push_back(i);
}
```

Читаем цвета вершин дерева.

```
col.resize(n+1);
for(i = 1; i <= n; i++)
  scanf("%d",&col[i]);
```

Запускаем поиск в глубину из вершины 1.

```
s.resize(c+1);
res.resize(n+1);
dfs(1);
```

Выводим ответ.

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

9655. Мистер Найн и его любовь к манго

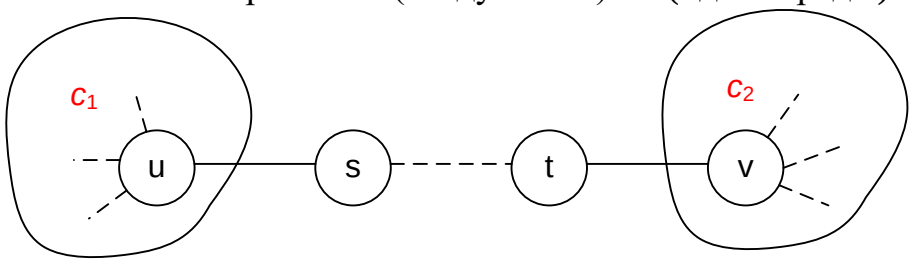
Мистер Найн во время перерывов в середине семестра случайно бродил по вселенной Параллель и неожиданно наткнулся на манговое дерево. Он любит манго, поэтому решил его сорвать. Но внезапно появилась фея и дала ему решить сложную задачу. Дерево содержит n узлов. Заданы также два узла u и v. Фея спрашивает, 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).



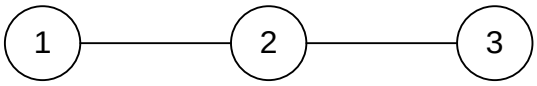
Пусть c_1 – количество вершин в поддереве с корнем u, при условии удаления ребра (u, s). Пусть c_2 – количество вершин в поддереве с корнем v, при условии удаления ребра (t, v). Количество путей, в которых после u идет v, равно $c_1 * c_2$.

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);
```

10653. Сумма XOR

Задано дерево с n вершинами. Ребра дерева имеют вес только 0 или 1. Найдем XOR сумму между всеми парами вершин. Вычислите сумму всех XOR сумм.

Вход. Первая строка содержит количество вершин в графе n ($2 \le n \le 10^5$). Следующие $n-1$ строк описывают ребра. Каждая строка содержит три целых числа: номера вершин, соединенных ребром (вершины пронумерованы числами от 1 до n), и вес ребра (0 или 1).

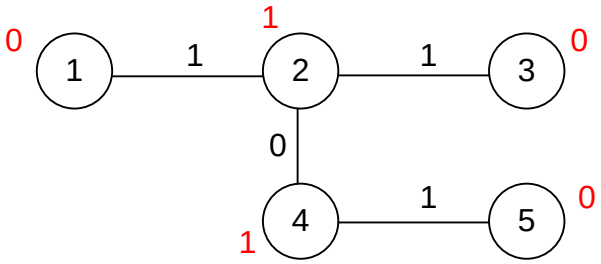
Выход. Выведите сумму XOR сумм между всеми парами вершин.

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

При помощи поиска в глубину вычислим XOR сумму между вершиной 1 и всеми остальными вершинами. XOR сумму между вершинами 1 и v занесем в x[v]. Пусть *ones* – количество единиц, а *zeroes* – количество нулей в массиве x (*ones* + *zeroes* = n). Тогда ответом на задачу будет число *ones* * *zeroes*.

Пример

Рассмотрим граф, приведенный в примере.



Возле каждой вершины v записана XOR сумма x[v] между 1 и v. Если x[v] = x[u] для некоторых вершин v и u, то XOR сумма между ними равна 0, таким образом совершая вклад 0 в общую сумму. XOR сумма для каждой пары вершин (v, u), для которой x[v] ≠ x[u], дает вклад 1 в общую сумму.

Следовательно ответ равен количеству пар вершин (v, u), для которых x[v] ≠ x[u]. Это число равно *ones* * *zeroes* = 2 * 3 = 6. Парами вершин, дающими 1 в общую сумму, будут: (1, 2), (1, 4) , (2, 3), (2, 5) , (3, 4) , (4, 5).

Реализация алгоритма

Входной граф храним в списке смежности *g*. Объявим массив *x*.

```
vector<vector<pair<int, int> > > g;
vector<int> x;
```

Функция ***dfs*** реализует поиск в глубину, который вычисляет XOR сумму x[*v*] между вершинами 1 и *v*. Текущая XOR сумма между 1 и *v* равна *cur_xor*. Предком вершины *v* является *p*.

```
void dfs(int v, int cur_xor, int p = -1)
{
  x[v] = cur_xor;
  for (int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i].first;
    int w = g[v][i].second;
    if (to != p) dfs(to, cur_xor ^ w, v);
  }
}
```

Читаем входной граф.

```
scanf("%d", &n);
g.resize(n + 1);
x.resize(n + 1);

for (i = 1; i < n; i++)
{
  scanf("%d %d %d", &u, &v, &d);
  g[u].push_back(make_pair(v, d));
  g[v].push_back(make_pair(u, d));
}
```

Запускаем поиск в глубину из вершины 1.

```
dfs(1, 0, -1);
```

Вычисляем количество нулей *zeroes* и единиц *ones* в массиве *x*.

```
ones = 0;
for (i = 1; i <= n; i++)
  if (x[i] == 1) ones++;
zeroes = n - ones;
```

Выводим ответ.

```
printf("%lld\n", 1LL * ones * zeroes);
```

10654. Уникальный цвет

Дано дерево с n вершинами, пронумерованными от 1 до n. i - ое ребро соединяет вершину a_i и вершину b_i. Вершина i окрашена в цвет c_i (в этой задаче цвета представлены целыми числами).

Вершина x считается ***хорошей***, если кратчайший путь от вершины 1 до вершины x не содержит вершину, окрашенную в тот же цвет, что и вершина x, кроме самой вершины x.

Найдите все хорошие вершины.

Вход. Первая строка содержит количество вершин n ($2 \le n \le 10^5$). Вторая строка содержит цвета $c_1, c_2, ..., c_n$ ($1 \le c_i \le 10^5$). Каждая из следующих $n - 1$ строк содержит два целых числа a_i и b_i ($1 \le a_i, b_i \le n$).

Выход. Выведите все хорошие вершины в виде целых чисел в порядке возрастания. Каждое число следует выводить в отдельной строке.

Пример входа 1

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

Пример выхода 1

```
1
2
3
4
6
```

Пример входа 2

```
10
3 1 4 1 5 9 2 6 5 3
1 2
2 3
3 4
4 5
5 6
6 7
7 8
8 9
9 10
```

Пример выхода 2

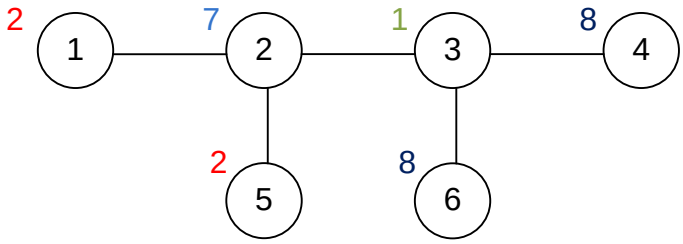
```
1
2
3
5
6
7
8
```

Запустим поиск в глубину из вершины 1. При входе в вершину v цвета color[v] увеличим значение used[color[v]] на 1. Значение used[color[v]] содержит количество раз, которое вершина цвета color[v] встретилась на пути от 1 до v (включая саму вершину v). Если цвет color[v] на пути встретился только один раз, то вершина v хорошая, заносим ее в результирующее множество.

При выходе из вершины v значение used[color[v]] следует уменьшить на 1.

Пример

Граф из первого примера имеет вид:



Вершина 5 не является хорошей, так как на пути 1 – 2 – 5 вершины 5 и 1 имеют одинаковый цвет.

Вершина 6 является хорошей, так как на пути 1 – 2 – 3 – 6 цвета вершин отличны от цвета вершины 6.

Реализация алгоритма

Входной граф храним в списке смежности *g*. Объявим рабочие массивы.

```
vector<int> used, color;
vector<vector<int>> g;
set<int> st;
```

Функция ***dfs*** реализует поиск в глубину. Переменная *par* является предком *v*.

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

Вершина *v* имеет цвет color[*v*]. Отмечаем, что на пути из вершины 1 встретилась вершина цвета color[*v*].

```
  used[color[v]]++;
```

Значение used[color[*v*]] содержит количество раз, которое вершина цвета color[*v*] встретилась на пути от 1 до *v* (включая саму вершину *v*). Если цвет color[*v*] на пути встретился только один раз, то вершина *v* хорошая, заносим ее в результирующее множество *st*.

```
  if (used[color[v]] == 1) st.insert(v);

  for (int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i];
    if (to == par) continue;
    dfs(to, v);
  }
```

При выходе из вершины *v* уменьшаем значение used[color[*v*]] на 1.

```
  used[color[v]]--;
}
```

Основная часть программы. Читаем входные данные.

```
scanf("%d", &n);
color.resize(n + 1);
for (i = 1; i <= n; i++)
  scanf("%d", &color[i]);

used.resize(100001);
g.resize(n + 1);
for (i = 1; i < n; i++)
{
  scanf("%d %d", &x, &y);
  g[x].push_back(y);
  g[y].push_back(x);
}
```

Запускаем поиск в глубину из вершины 1.

```
dfs(1, 1);
```

Выводим хорошие вершины.

```
for (int val : st)
  printf("%d\n", val);
```

10655. Вирусное дерево 2

Вам дано дерево с n вершинами и $n - 1$ ребрами. Вершины пронумерованы от 1 до n, i-ое ребро соединяет вершины a_i и b_i.

У Вас имеется k цветов. Для каждой вершины в дереве Вы выбираете один из k цветов для ее покраски, чтобы выполнялось следующее условие:

- Если расстояние между двумя разными вершинами x и y меньше или равно двум, то x и y имеют разные цвета.

Сколько существует способов раскрасить дерево? Найдите ответ по модулю $10^9 + 7$.

Вход. Первая строка содержит два числа n и k. Каждая из следующих $n - 1$ строк содержит два целых числа a_i и b_i.

Выход. Выведите количество способов раскрасить дерево по модулю $10^9 + 7$.

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

Пример входа 2

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

Пример выхода 2

```
48
```

РЕШЕНИЕ

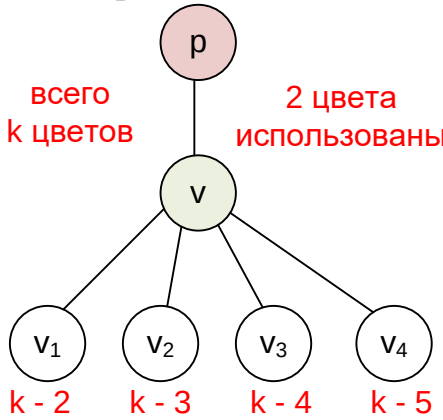
графы – поиск в глубину

Анализ алгоритма

Начнем раскраску дерева с вершины 1. Ее можно покрасить k цветами.

Пусть мы находимся в вершине v. Если она не имеет предка ($v = 1$), то сыновей можно раскрасить $k – 1$ цветами. Если v имеет предка, то сыновей можно раскрасить $k – 2$ цветами. Поскольку расстояние между сыновьями вершины v равно 2, то их нельзя раскрашивать одинаковыми цветами.

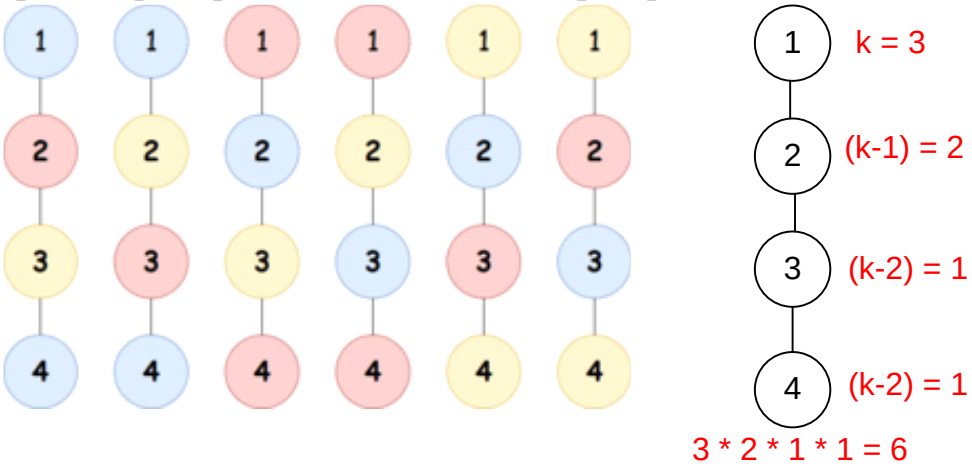
Пусть для определенности вершина v имеет предка. Тогда первого сына v можно покрасить $k – 2$ цветами, второго сына $k – 3$ цветами и так далее.



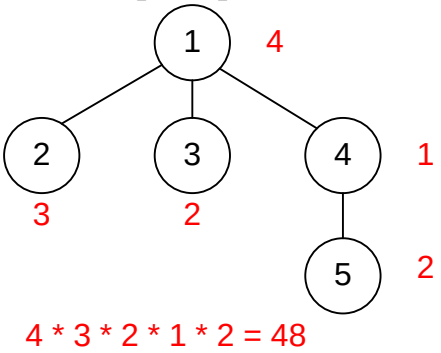
Остается перемножить количества цветов, которыми можно раскрасить каждую вершину.

Пример

Для первого примера имеется 6 способов раскраски:



Рассмотрим второй тест. Возле каждой вершины укажем количество цветов, которыми ее можно раскрасить. Например, вершину 5 можно раскрасить 2 цветами, так как ее цвет не должен совпадать с цветами вершин 1 и 4. Общее количество способов раскрасить дерево равно 4 * 3 * 2 * 1 * 2 = 48.



Реализация алгоритма

Входной граф храним в списке смежности *g*. Объявим константу – модуль MOD.

```
#define MOD 1000000007
vector<vector<int>> g;
```

Функция ***dfs*** возвращает количество способов раскрасить дерево с корнем в вершине *v* по модулю MOD = $10^9 + 7$.

```
int dfs(int v, int col, int p = -1)
{
  long long res = col;
```

Мы находимся в вершине *v*. В переменной *used* отмечаем количество цветов, которыми нельзя красить сыновей вершины *v*. Изначально установим *used* = 1, так как сыны вершины *v* не могут иметь цвет вершины *v*. Если вершина *v* имеет предка ($p \neq -1$), то сыны вершины *v* также не могут иметь цвет предка *v*.

```
  int used = 1;
  if (p >= 0) used++;
```

Перебираем сыновей *to* вершины *v*.

```
  for (int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i];
    if (to == p) continue;
```

Вершина *to* может быть покрашена *k* – *used* цветами. С каждым сыном значение *used* будет увеличиваться на 1.

```
    res = (res * dfs(to, k - used, v)) % MOD;
    used++;
  }
  return res;
}
```

Основная часть программы. Читаем входные данные.

```
scanf("%d %d", &n, &k);
g.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 1. Вершину номер 1 можно покрасить k цветами.

```
res = dfs(1, k);
```

Выводим ответ.

```
printf("%lld\n", res);
```

10656. Дерево коротких расстояний

Задано неориентированное дерево, состоящее из n вершин. Неориентированное дерево – это связный граф с $n – 1$ ребром.

Ваша задача – добавить минимальное количество ребер таким образом, что длина кратчайшего пути из вершины 1 до любой другой вершины не превышала 2. Заметьте, что вы не можете добавлять петли и кратные ребра.

Вход. Первая строка содержит одно целое число n ($2 \le n \le 2 * 10^5$) – количество вершин в дереве.

Следующие $n – 1$ строк описывают ребра: ребро i задано как пара вершин u_i, v_i ($1 \le u_i, v_i \le n$). Гарантируется, что заданный граф является деревом. Гарантируется, что среди заданных ребер петли и кратные ребра отсутствуют.

Выход. Выведите одно целое число – минимальное количество ребер, которое необходимо добавить, чтобы длина кратчайшего пути из вершины 1 до любой другой вершины не превышает 2. Заметьте, что вы не можете добавлять петли и кратные ребра.

Пример входа 1

```
7
1 2
2 3
2 4
4 5
4 6
5 7
```

Пример выхода 1

```
2
```

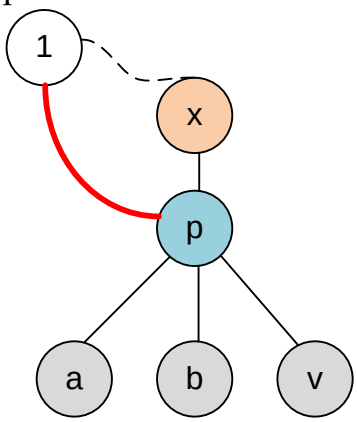
Пример входа 2

```
7
1 2
1 3
2 4
2 5
3 6
1 7
```

Пример выхода 2

```
0
```

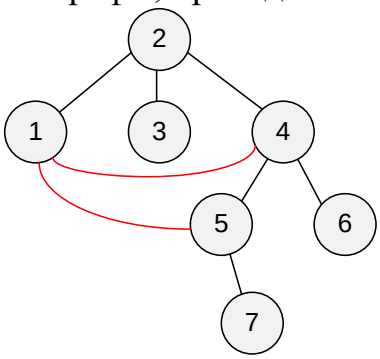
Ребра будем добавлять из вершины 1. Действительно, если добавлено ребро (u, v), то заменив его на (1, v), мы не ухудшим результат. Если расстояние от 1 до вершины v больше двух, то есть смысл провести ребро из 1 в предка v. Таким образом мы сократим до двух расстояние от вершины 1 до всех братьев вершины v (вершины назовем братьями если они имеют общего непосредственного предка), а также до предка v – вершины x.

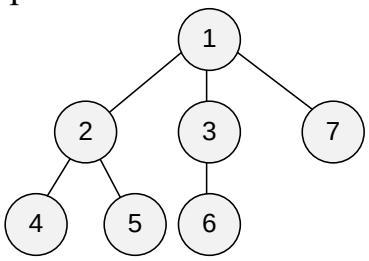


Пусть массив m содержит вершины, расстояние до которых от вершины 1 больше 2. Вершины в массив m будем заносить в порядке выхода из них при поиске в глубину. Далее воспользуемся жадным подходом. Перебираем вершины из m слева направо и для каждой вершины v (если расстояние до нее еще больше двух с учетом уже построенных ребер) проводим ребро из 1 в ее предка p. Далее все вершины, соседние с p, отмечаем таковыми, расстояние до которых уже не больше 2.

Пример

Рассмотрим графы, приведенные в примере.





В первом примере ответ равен 2. В массив m вершины будут занесены в порядке {7, 5, 6}. Первой на рассмотрении будет вершина *v* = 7. Проводим ребро в ее предка: (1, 5). Следующей будет вершина *v* = 6. Проводим ребро в ее предка: (1, 4).

Во втором примере ответ 0, так как все вершины находятся на расстоянии не более 2 от вершины 1.

Реализация алгоритма

Входной граф храним в списке смежности *g*. Объявим рабочие массивы.

```
vector<vector<int> > g;
vector<bool> used;
vector<int> m, rev;
```

Функция ***dfs*** реализует поиск в глубину. Значение *dist* равно расстоянию от вершины 1 до *v*, *parent* – предок *v*.

```
void dfs(int v, int dist = 0, int parent = 0)
{
  used[v] = true;
```

Для каждой вершины *v* сохраним значение ее предка в rev[*v*].

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

В массиве m сохраним номера вершин дерева, расстояние до которых от 1 больше двух.

```
  if (dist > 2) m.push_back(v);
}
```

Основная часть программы. Читаем входные данные.

```
scanf("%d", &n);
g.resize(n + 1);
for (i = 1; i < n; i++)
{
  scanf("%d %d", &u, &v);
  g[u].push_back(v);
  g[v].push_back(u);
}
```

Запускаем поиск в глубину из вершины 1.

```
used.resize(n + 1); rev.resize(n + 1);
dfs(1);
used.clear(); used.resize(n + 1);
```

В переменной *ans* вычисляем результат.

```
int ans = 0;
```

Перебираем вершины, находящиеся на расстоянии больше 2 от вершины 1 (все они находятся в массиве m). В массиве used будем отмечать вершины, расстояние до которых от 1 уже стало равным не больше два.

```
for (i = 0; i < m.size(); i++)
{
  v = m[i];
  if (!used[v])
  {
```

Проводим ребро от 1 в rev[*v*]. Расстояние от 1 до сыновей вершины rev[*v*] становится не больше 2.

```
    ans++;
    used[rev[v]] = true;
    for (j = 0; j < g[rev[v]].size(); j++)
    {
      u = g[rev[v]][j];
      used[u] = true;
    }
  }
}
```

Выводим ответ.

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

10667. Интервалы на дереве

Задано дерево из n вершин и $n - 1$ ребер, соответственно пронумерованных 1, 2, ..., n и 1, 2, ..., $n - 1$. Ребро i соединяет вершины u_i и v_i.

Для чисел L, R ($1 \le L \le R \le n$) объявим функцию f(L, R) следующим образом:

- Пусть S – множество вершин с номерами от L до R. Функция f (L, R) представляет собой количество компонент связности в подграфе, образованном только из множества вершин S и ребер, оба конца которых принадлежат S.

Вычислите

$$\sum_{L=1}^{n}\sum_{R=L}^{n} f(L,R)$$

Вход. Первая строка содержит число n ($1 \le n \le 2 * 10^5$). Каждая из следующих $n - 1$ строк содержит две вершины (u_i, v_i) ($1 \le u_i, v_i \le n$) – ребро в графе.

Выход. Выведите значение суммы.

Пример входа	**Пример выхода**
3 1 3 2 3	7

Пусть V – множество вершин в дереве, E – множество ребер. Тогда для дерева имеет место формула: V = E + 联通分量个数

$$|V| = |E| + \text{количество компонент связности}$$

Тогда количество компонент связности f(L, R) можно вычислить как |V| – |E|, где

- V = nodes(L, R) – множество вершин {L, …, R};
- E = edges(L, R) – множество ребер с концами на множестве {L, …, R};

Искомую сумму $res = \sum_{L=1}^{n}\sum_{R=L}^{n} f(L,R)$ можно вычислить с помощью следующего алгоритма: 答案可以通过下面代码计算

```
res = 0;
for (L = 1; L ≤ n; L++)
for (R = L; R ≤ n; R ++)
   res += nodes(L, R) - edges(L, R)
```

Обозначим через $S(n) = 1 + 2 + \ldots + n$. 以1为起点的数量等于 S(n)

Пусть L = 1. Тогда в качестве R можно выбрать вершины 1, 2, …, *n*.

$$\sum_{i=1}^{n} nodes(1,i) = \text{nodes}(1, 1) + \text{nodes}(1, 2) + \ldots + \text{nodes}(1, n) =$$

$$1 + 2 + \ldots + n = S(n)$$

Пусть L = 2. Тогда в качестве R можно выбрать вершины 2, …, *n*.

$$\sum_{i=2}^{n} nodes(2,i) = \text{nodes}(2, 2) + \text{nodes}(2, 3) + \ldots + \text{nodes}(2, n) =$$

$$1 + 2 + \ldots + n - 1 = S(n - 1)$$ 以2为起点的数量等于 S(n-1)

Пусть L = *k*. Тогда в качестве R можно выбрать вершины *k*, …, *n*.

$$\sum_{i=k}^{n} nodes(k,i) = \text{nodes}(k, k) + \text{nodes}(k, k + 1) + \ldots + \text{nodes}(k, n) =$$

$$1 + 2 + \ldots + n - k + 1 = S(n - k + 1)$$

Таким образом

$$\sum_{L=1}^{n}\sum_{R=L}^{n} nodes(L,R) = S(n) + S(n - 1) + \ldots + S(1)$$

所以答案中总的顶点数等于上式

Указанная сумма равна сумме всех чисел в следующей таблице:

1	2	3	...	n - 1	n
1	2	3	...	n - 1	
...					
1	2	3			
1	2				
1					
1 * n	2*(n-1)	3*(n-2)	...	(n – 1)*2	n * 1

В таблице присутствует n единиц, $(n-1)$ двоек, $(n-2)$ троек и так далее. Сумму можно переписать в виде

$$\sum_{i=1}^{n} i*(n-i+1) = 1*n + 2*(n-1) + 3*(n-2) + \ldots + (n-1)*2 + n*1$$

Известно, что

- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$,
- $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

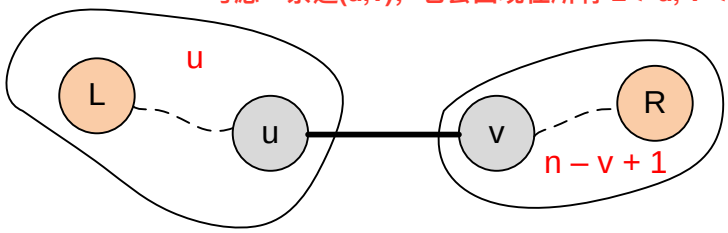
Тогда

$$\sum_{L=1}^{n}\sum_{R=L}^{n} nodes(L,R) = \sum_{i=1}^{n} i*(n-i+1) = \sum_{i=1}^{n} i*(n+1) - \sum_{i=1}^{n} i^2 =$$

$$\frac{n(n+1)^2}{2} - \frac{n(n+1)(2n+1)}{6} = \frac{n(n+1)}{6}(3n+3-2n-1) = \frac{n(n+1)(n+2)}{6}$$

Рассмотрим ребро дерева (u, v). Оно будет входить во все множества edges(L, R), где $L \le u$ и $v \le R$.

考虑一条边(u,v)，它会出现在所有 L<=u, v <= R 的区间中



Значение L можно выбрать u способами, а значение R можно выбрать $(n - v + 1)$ способами. Следовательно количество множеств edges(L, R), которым принадлежит ребро (u, v), равно $u * (n - v + 1)$.

其中满足条件的L有u种，满足条件的R有(n-v-1)种，所以(u,v)对总边数的贡献为u*(n-v+1)

Пример

Для приведенного примера имеются шесть возможных пар (L, R):

- Для L = 1, R = 1, S = {1}, имеется 1 связная компонента.
- Для L = 1, R = 2, S = {1, 2}, имеется 2 связные компоненты.
- Для L = 1, R = 3, S = {1, 2, 3}, имеется 1 связная компонента, так как S содержит оба конца каждого из ребер 1, 2.
- Для L = 2, R = 2, S = {2}, имеется 1 связная компонента.
- Для L = 2, R = 3, S = {2, 3}, имеется 1 связная компонента, так как S содержит оба конца ребра 2.
- Для L = 3, R = 3, S = {3}, имеется 1 связная компонента.

Сумма всех значений равна 7.

Вычислим ответ по формуле:

$$\sum_{L=1}^{3}\sum_{R=L}^{3} nodes(L,R) = \frac{3(3+1)(3+2)}{6} = \frac{3*4*5}{6} = 10$$

- edges(1, 3) = 1;
- edges(2, 3) = 2 * 1 = 2;

Следовательно ответ равен 10 – 1 – 2 = 7.

Реализация алгоритма

Читаем входное значение *n*.

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

Инициализируем *res* значением $\sum_{L=1}^{n}\sum_{R=L}^{n} nodes(L,R) = \frac{n(n+1)(n+2)}{6}$.

```
res = 1LL * n * (n + 1) * (n + 2) / 6;
```

Перебираем ребра (*u*, *v*). Установим $u \le v$. Для каждого ребра вычтем из общей суммы значение $u * (n - v + 1)$.

```
for (i = 0; i < n - 1;i++)
{
  scanf("%d %d", &u, &v);
  if (u > v)
  {
    temp = u; u = v; v = temp;
  }
  res -= 1LL * u * (n - v + 1);
}
```

Выводим ответ.

```
printf("%lld\n", res);
```

10684. Улучшение дорог

В стране есть n городов и $n - 1$ двусторонняя дорога, причем из каждого города можно добраться до любого другого города, двигаясь только по дорогам. Города пронумерованы целыми числами от 1 до n включительно.

Все дороги изначально плохие, однако правительство хочет улучшить состояние некоторых дорог. Будем считать, что граждане довольны улучшением дорог, если на пути от столицы, расположенной в городе 1, до любого города не более одной плохой дороги на пути.

Определите количество способов улучшить качество некоторых дорог так, чтобы удовлетворить требованиям граждан. Поскольку ответ будет большим, выведите его по модулю 1 000 000 007 ($10^9 + 7$).

Вход. В первой строке задано одно целое число n ($2 \le n \le 2 * 10^5$) – количество городов в стране. В следующей строке задано $n - 1$ целое положительное число $p_2, p_3, p_4, ..., p_n$ ($1 \le p_i \le i - 1$) – описание дорог страны. Число p_i означает, что в стране имеется дорога, соединяющая город p_i и город i.

Выход. Выведите искомое количество способов улучшить качество дорог по модулю 1 000 000 007 ($10^9 + 7$).

Пример входа 1

```
3
1 1
```

Пример выхода 1

```
4
```

如果道路（v，to）不好，那么顶点为to的子树中的所有道路都应该是好的。

·如果道路（v，to）得到改善，则顶点为f（to）的子树的道路质量得到改善的方法数量。

Пример входа 2

```
6
1 2 2 1 5
```

Пример выхода 2

```
15
```

设f（v）为顶点为v的子树改善道路质量的方法数。

Пусть f(v) – количество способов улучшить качество дорог для поддерева с вершиной v. Пусть to – сын v. 考虑v的子节点to

- Если дорогу (v, to) оставить плохой, то все дороги в поддереве с вершиной to должны быть хорошими.
- Если дорогу (v, to) улучшить, то количество способов улучшить качество дорог для поддерева с вершиной to равно f(to).

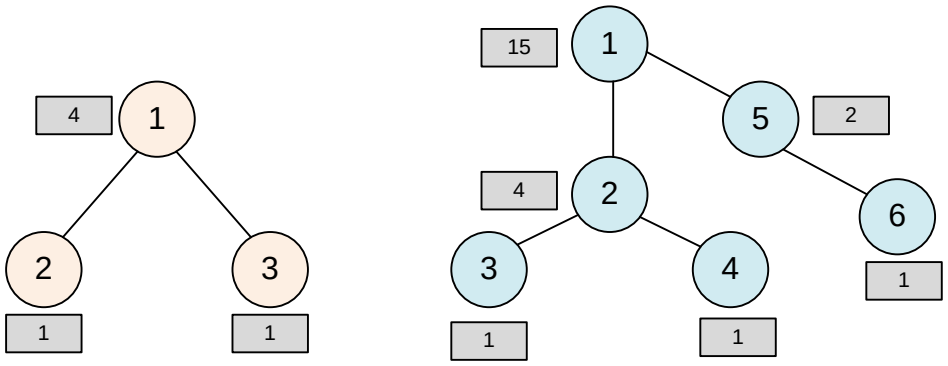
Пусть $to_1, to_2, \ldots, to_k$ – сыновья v. Тогда

$$f(v) = (f(to_1) + 1) * (f(to_2) + 1) * \ldots * (f(to_k) + 1)$$

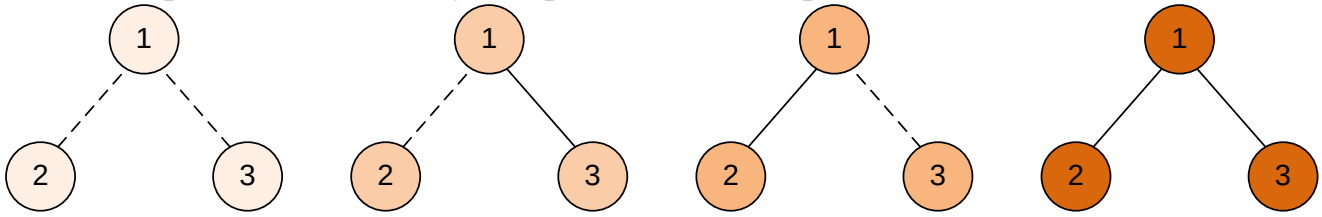
Если v лист (вершина, не имеющая сыновей), то положим f(v) = 1.

Пример

Рассмотрим заданные в условии примеры. Возле каждой вершины v запишем значение f(v).



В первом примере имеются 4 способа улучшить качество дорог (на пути от города 1 до любого другого города имеется не более одной плохой дороги). Плохая дорога обозначена пунктирной линией, хорошая – сплошной.



Реализация алгоритма

Вычисления проводим по модулю MOD = $10^9 + 7$.

```
#define MOD 1000000007
```

Функция ***dfs*** вычисляет значение f(v) – количество способов улучшить качество дорог для поддерева с вершиной v. Предком v является вершина p.

```
long long dfs(int v, int p = -1)
{
  long long s = 1;
  for (int i = 0; i < g[v].size(); i++)
  {
```

Если to_1, to_2, …, to_k – сыновья v, то

$$f(v) = (f(to_1) + 1) * (f(to_2) + 1) * \ldots * (f(to_k) + 1)$$

```
    int to = g[v][i];
    if (to == p) continue;
    long long sub = dfs(to, v);
    s = (s * (sub + 1)) % MOD;
  }
  return s;
}
```

Читаем входные данные, строим дерево.

```
scanf("%d", &n);
g.resize(n + 1);
```

```
for (i = 2; i <= n; i++)
{
  scanf("%d", &p);
  g[i].push_back(p);
  g[p].push_back(i);
}
```

Вычисляем и выводим результат.

```
res = dfs(1);
printf("%lld\n", res);
```

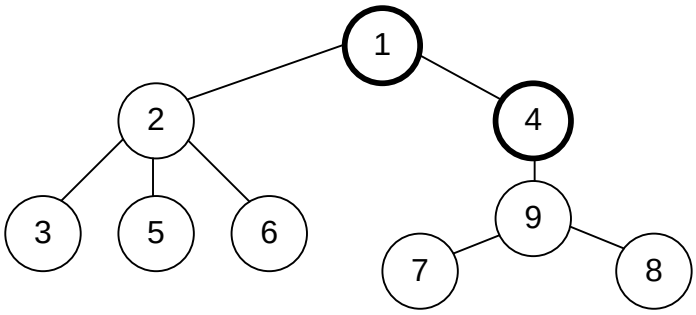
11002. Покраска дерева

Вам задано дерево (неориентированный связный ацикличный граф), состоящее из n вершин. Вы играете в игру на этом дереве.

Изначально все вершины белые. На первом ходу игры Вы выбираете одну вершину и красите ее в черный. Затем на каждом ходу Вы выбираете белую вершину, смежную (соединенную ребром) с любой черной вершиной и красите ее в черный.

Каждый раз, когда вы выбираете вершину (даже во время первого хода), Вы получаете количество очков, равное размеру компоненты связности, состоящей только из белых вершин, содержащей выбранную вершину. Игра заканчивается, когда все вершины покрашены в черный цвет.

Рассмотрим следующий пример:



Вершины 1 и 4 уже покрашены в черный цвет. Если выберете вершину 2, то получите 4 очка за компоненту связности, состоящую из вершин 2, 3, 5 и 6. Если выберете вершину 9, то получите 3 очка за компоненту связности, состоящую из вершин 7, 8 и 9.

Ваша задача – максимизировать количество очков, которое Вы получите.

Вход. Первая строка содержит одно целое число n ($2 \le n \le 2 * 10^5$) – количество вершин в дереве.

Каждая из следующих $n - 1$ строк описывает ребро дерева. Ребро i описывается двумя целыми числами u_i и v_i ($1 \le u_i, v_i \le n, u_i \ne v_i$), номерами вершин, которые оно соединяет.

Гарантируется, что заданные ребра образуют дерево.

Выход. Выведите одно целое число – максимальное количество очков, которое вы получите, если будете играть оптимально.

Пример входа 1

```
9
1 2
2 3
2 5
2 6
1 4
4 9
9 7
9 8
```

Пример выхода 1

```
36
```

Пример входа 2

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

Пример выхода 2

```
14
```

Количество полученных очков однозначно определяется первой выбранной вершиной. Остается перебрать в качестве первой все возможные вершины, вычислить полученное количество очков и вывести максимальное среди них значение.

weight[v] 以v为根的子树顶点数

Запустим поиск в глубину из вершины 1. Вычислим weight[v] – размер поддерева (количество вершин) с корнем в вершине v. Пусть to_1, to_2, …, to_k – сыновья v. Тогда

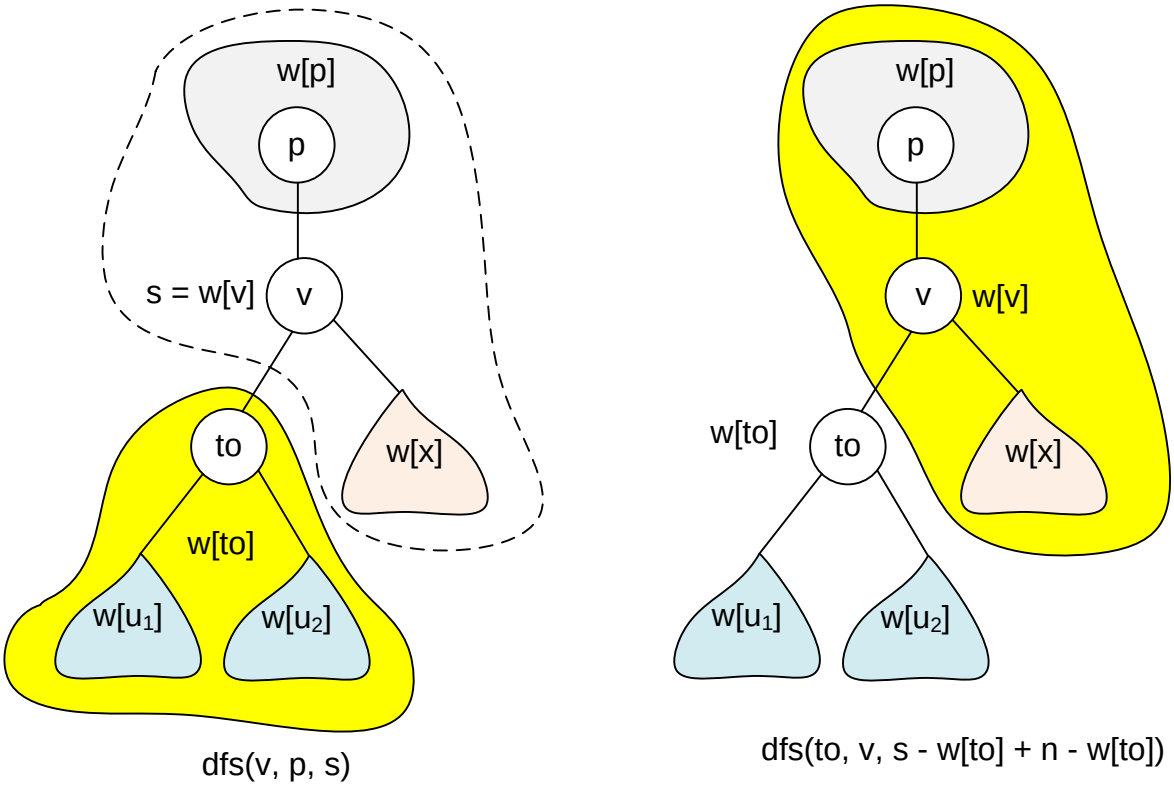
$$\text{weight}[v] = 1 + \text{weight}[to_1] + \text{weight}[to_2] + \ldots + \text{weight}[to_k]$$

Второй поиск в глубину будет проходить по всем вершинам и пересчитывать максимальное количество очков, которое можно получить если стартовать из этой вершины.

Рассмотрим функцию ***dfs2***. Мы находимся в вершине v, предком которой является p. Если первым ходом игры выбрать вершину v, то количество полученных очков составляет s.

```
void dfs2(int v, int p, long long s)
```

Сделаем переход по ребру (v, to) и покажем как вычислить ответ, если игру изначально стартовать из вершины to. Фактически произведем переподвешивание дерева с вершины v в вершину to.

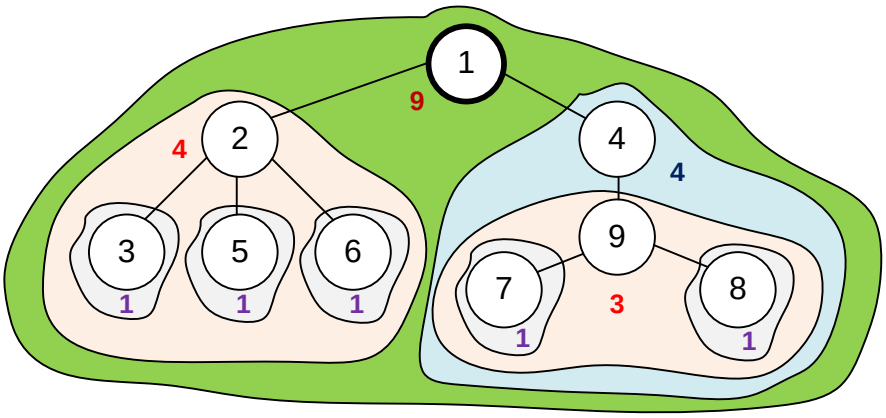


Слева приведено дерево с корнем в *v*. Вершина *to* – один из ее сыновей, *p* – предок. Справа приведено дерево с корнем в *to*.

Для вычисления weight[*to*] правого дерева необходимо из значения *s* в левом дереве вычесть weight[*to*] левого дерева и прибавить количество вершин в области, обозначенной пунктиром. Оно равно *n* – weight[*to*].

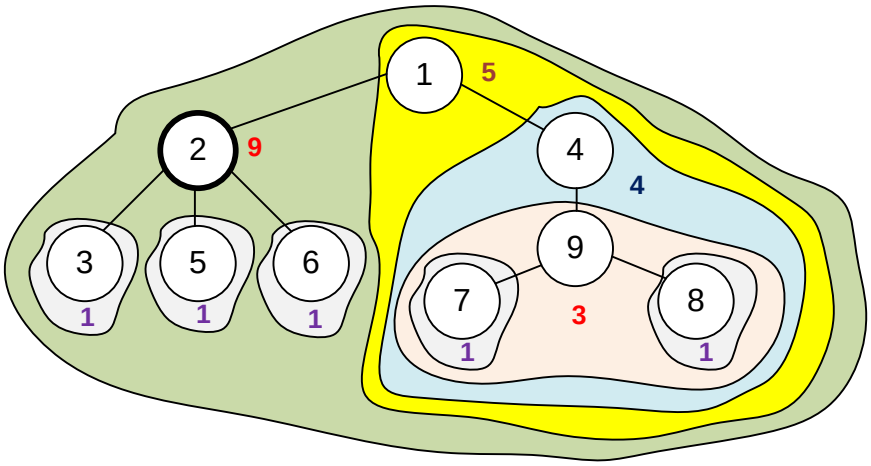
Пример

Пусть первой выбранной вершиной будет 1. Тогда количество полученных очков составит 25.



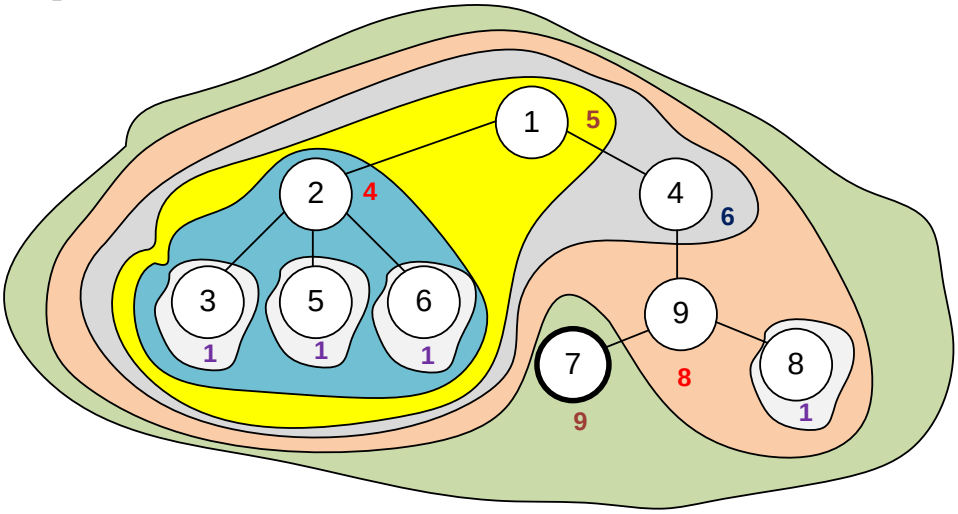
9 + 4 + 4 + 1 + 1 + 1 + 3 + 1 + 1 = 25

Если первой выбранной вершиной будет 2, то количество полученных очков равно 26. Переход от вершины 1 к вершине 2 произведем следующим образом. Вычтем из *s* = 25 значение weight[2] = 4 и прибавим (9 – weight[4]) = 5. Получим 25 – 4 + 5 = 26, значение *s* для дерева с корнем в вершине 2.



9 + 1 + 1 + 1 + 5 + 4 + 3 + 1 + 1 = 26

Максимальное количество очков – 36, будет получено если начать игру с вершины номер 7.



9 + 8 + 1 + 6 + 5 + 4 + 1 + 1 + 1 = 36

Реализация алгоритма

Объявим список смежности графа *g*.

```
vector<vector<int>> g;
vector<int> weight;
```

Функция ***dfs1*** запускает поиск в глубину из вершины *v*, предком которой является *p*. Вычисляем weight[*v*] – размер поддерева (количество вершин) с корнем в вершине *v*.

```
void dfs1(int v, int p)
{
  weight[v] = 1;
  for (int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i];
    if (to == p) continue;
    dfs1(to, v);
```

```
    weight[v] += weight[to];
  }
}
```

Функция ***dfs2*** запускает поиск в глубину из вершины *v*, предком которой является *p*. Если первым ходом игры выбрать вершину *v*, то количество полученных очков составит *s*.

```
void dfs2(int v, int p, long long s)
{
  res = max(res, s);
  for (int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i];
    if (to == p) continue;
      dfs2(to, v, s - weight[to] + n - weight[to]);
  }
}
```

Основная часть программы. Читаем входные данные. Строим граф.

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

Запускаем поиск в глубину из вершины 1.

```
dfs1(1, 0);
```

В переменной *s* вычислим сумму всех чисел в массиве weight. Оно равно количеству полученных очков в случае, если первой в игре выбрать вершину номер 1.

```
s = 0;
for (i = 1; i <= n; i++)
  s += weight[i];
```

Запускаем второй поиск в глубину из вершины 1.

```
dfs2(1, 0, s);
```

Выводим ответ.

```
printf("%lld\n", res);
```

11058. Погоня за бабочкой

Ясными летними днями Нюша любит ловить бабочек на свежем воздухе. Сегодня ей попалась хитрая бабочка: она залетела в лабиринт и хотела скрыться в нем от Нюши.

Лабиринт состоит из n комнат, пронумерованных от 1 до n, некоторые из которых соединены между собой коридорами. Известно, что между любыми двумя комнатами существует единственный путь из коридоров. Иными словами, лабиринт представляет собой дерево, и количество коридоров равно $n - 1$.

Вход в лабиринт расположен в комнате с номером 1. Будем называть листом любую комнату лабиринта, которая соединена коридором ровно с одной другой комнатой и не совпадает при этом с корнем. В каждом из листов располагается выход из лабиринта. Бабочка летит от входа по направлению к одному из листов. Она летит с постоянной скоростью и не разворачивается. Все коридоры имеют одинаковую длину, и за одну минуту бабочка пролетает один коридор, перемещаясь в соседнюю комнату.

Для поимки бабочки Нюша решила позвать несколько своих друзей. Исходно каждый из друзей может расположиться в любой из комнат, содержащих выход. В тот момент, когда бабочка начнет лететь от входа в лабиринт к одному из выходов, каждый из друзей может начать двигаться из своей комнаты по направлению ко входу. Друзья двигаются с такой же скоростью, что и бабочка. Если кто-то из друзей оказался в одной точке (в комнате или в середине одного из коридоров) с бабочкой, то бабочка считается пойманной. Если же бабочка долетит до вершины с выходом, не встретив никого из друзей по пути, она благополучно выпорхнет из лабиринта и улетит на свободу.

Помогите Нюше определить, какое минимальное число друзей понадобится для того, чтобы гарантированно поймать бабочку, вне зависимости от того, к какому выходу она полетит.

Вход. Первая строка содержит целое число n ($2 \le n \le 200000$) – количество комнат в лабиринте.

В следующих $n - 1$ строках содержатся описания коридоров, соединяющих комнаты. Каждая из этих строк содержит два целых числа u и v ($1 \le u, v \le n, u \ne v$) – номера комнат, которые соединяет коридор. Гарантируется, что структура коридоров представляет собой дерево.

Выход. Выведите одно целое число – минимальное количество друзей, необходимое для того, чтобы гарантированно поймать бабочку.

Пример входа 1

```
3
1 2
1 3
```

Пример выхода 1

```
2
```

Пример входа 2

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

Пример выхода 2

```
1
```

Запустим поиск в глубину из вершины 1. Для каждой вершины вычислим два значения: 从顶点1开始深度搜索。对于每个顶点，我们计算两个值

- d[v] – расстояние от вершины 1 до v; 从顶点1到顶点V的距离
- h[v] – расстояние от вершины v до ближайшего листа в поддереве с корнем v; 从顶点v到根v子树中最近的叶节点的距离

Если p – родитель v, то d[v] = d[p] + 1.

Если to_1, to_2, …, to_k – сыновья v, то

$$h[v] = 1 + \min(h[to_1], h[to_2], \ldots, h[to_k])$$

res[v]: 以v为根的子树需要的最少朋友数量，确保蝴蝶飞不出子树v

如果 h[v]<=d[v],则res[v]=1,只要把一个朋友放在最小深度的叶子节点上，他就可以在蝴蝶从到达v之前到达顶点V

Запустим второй обход в глубину вершин дерева. В нем будем вычислять ответ res[v] для вершины v: минимальное количество друзей, которое следует поставить в некоторые из листьев этого поддерева, чтобы гарантированно поймать бабочку при условии, что она обязательно полетит в это поддерево.

Если h[v] ≤ d[v], то res[v] = 1. Достаточно поставить одного друга в лист с минимальной глубиной и до вершины v он доберется не позже чем бабочка из корня. Иначе если to_1, to_2, …, to_k – сыновья v, то

$$res[v] = res[to_1] + res[to_2] + \ldots + res[to_k]$$

Если мы не словим бабочку в вершине v, то следует быть готовым ловить ее в любом из поддеревьев сыновей вершины v.

Ответом на задачу будет число res[1].

否则，res[v]等于上式，最终答案为res[1]

Реализация алгоритма

Объявим константу бесконечность и рабочие массивы.

```
#define INF 2000000000
vector<vector<int> > g;
vector<int> d, h, res;
```

Функция ***dfs*** (v, p, cnt) совершает обход в глубину из вершины v. Предком вершины v является p. Расстояние от вершины 1 до v равно cnt. Для каждой вершины v вычисляем значения d[v] и h[v].

```
int dfs(int v, int p = -1, int cnt = 0)
{
```

Расстояние от вершины 1 до v равно cnt, заносим его в d[v].

```
  d[v] = cnt;
  int height = INF;
  for (int i = 0; i < g[v].size(); i++)
  {
```

Перебираем сыновей вершины v. Рассматриваем ребро $v \to to$. Если to совпадает с предком v ($to = p$), то переходим к следующему сыну.

```
    int to = g[v][i];
    if (to == p) continue;
```

В переменной *height* вычисляем $\min(h[to_1], h[to_2], \ldots, h[to_k])$, где $to_1, to_2, \ldots, to_k$ – сыновья v.

```
    int temp = dfs(to, v, cnt + 1);
    if (temp < height) height = temp;
  }
```

Если *height* = INF, то v является листом и следует установить $h[v] = 0$. Иначе возвращаем $h[v] = 1 + \min(h[to_1], h[to_2], \ldots, h[to_k])$.

```
  return h[v] = (height == INF) ? 0 : 1 + height;
}
```

Функция ***dfs2*** (v, p) совершает обход в глубину из вершины v. Для каждой вершины v вычисляем $res[v]$.

```
int dfs2(int v, int p = -1)
{
  int s = 0;
  for (int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i];
    if (to == p) continue;
    dfs2(to, v);
```

Если $to_1, to_2, \ldots, to_k$ – сыновья v, то $res[v] = res[to_1] + res[to_2] + \ldots + res[to_k]$.

```
    s += res[to];
  }
```

Если $h[v] \le d[v]$, то достаточно одного друга и $res[v] = 1$.

```
  return res[v] = (h[v] <= d[v]) ? 1 : s;
}
```

Основная часть программы. Читаем входные данные. Строим граф.

```
scanf("%d", &n);
g.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);
}
```

Инициализируем массивы.

```
d.resize(n + 1);
h.resize(n + 1);
res.resize(n + 1);
```

Запускаем поиски в глубину. Первый поиск для каждой вершины v вычисляет значения d[v] и h[v].

```
dfs(1);
dfs2(1);
```

Выводим ответ – наименьшее количество друзей, которое требуется для поимки бабочки.

```
printf("%lld\n", res[1]);
```

11153. Кефа и парк

Кефа решил отпраздновать свой первый крупный заработок походом в ресторан.

Он живет возле необычного парка. Парк представляет из себя подвешенное дерево из n вершин с корнем в вершине 1. В вершине 1 также находится дом Кефы. К сожалению для нашего героя, в парке также находятся коты. Кефа уже выяснил номера вершин, в которых находятся коты.

В листовых вершинах парка находятся рестораны. Кефа хочет выбрать ресторан, в который он пойдет, но, к сожалению, он очень боится котов, поэтому он ни за что не пойдёт в ресторан, на пути к которому от его дома найдётся более m подряд идущих вершин с котами.

Ваша задача – помочь Кефе посчитать количество ресторанов, в которые он может сходить.

Вход. В первой строке записаны два целых числа n и m ($2 \le n \le 10^5$, $1 \le m \le n$) – количество вершин дерева и максимальное количество подряд идущих вершин с котами, которое способен перенести Кефа.

Во второй строке содержится n целых чисел a_1, a_2, ..., a_n, где каждое a_i либо равняется 0 (тогда в вершине i нет кота), либо равняется 1 (тогда в вершине i есть кот).

В следующих n – 1 строках записаны ребра дерева в формате x_i y_i ($1 \le x_i, y_i \le n$, $x_i \ne y_i$), где x_i и y_i – вершины дерева, соединенные очередным ребром.

Гарантируется, что данный набор рёбер задаёт дерево.

Выход. Выведите количество различных листьев дерева, на пути от дома Кефы до которых не больше m подряд идущих вершин с котами.

Пример входа 1

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

Пример выхода 1

```
2
```

Пример входа 2

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

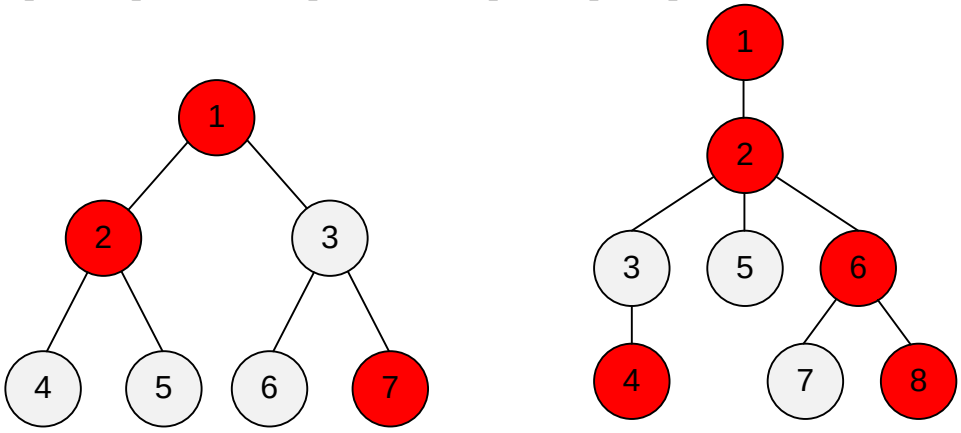
Пример выхода 2

```
2
```

Запустим поиск в глубину из вершины 1, где находится дом Кефы. В одном из параметров функции ***dfs*** будем хранить количество последовательно пройденных вершин с котами. Если в вершине v это число превысит m, то поиск в глубину из вершины v не производим. Подсчитываем количество посещенных листов. Вершина является листом, если ее степень равна 1 и при этом она не является корнем (то есть вершиной 1).

Пример

Рассмотрим деревья из первого и второго примеров.



В первом примере Кефа может пройти подряд только через 1 вершину с котом. Он сможет попасть в рестораны, расположенные в вершинах 6 и 7.

Во втором примере Кефа может пройти подряд только через 2 вершины с котами. Он сможет попасть в рестораны, расположенные в вершинах 4 и 5.

Реализация алгоритма

Расположение котов храним в массиве cats. Дерево храним в списке смежности g.

```
vector<int> cats;
vector<vector<int> > g;
```

Функция ***dfs*** совершает поиск в глубину из вершины *v*. Предком *v* является вершина *prev*. Число последовательных вершин с котами, которые пройдены непосредственно до вершины *v* включительно, равно *cnt*.

```
int dfs(int v, int prev, int cnt)
{
```

Если последовательно уже пройдено более *m* котов, то поиск в глубину из *v* не продолжаем.

```
  if (cnt > m) return 0;
```

Если мы в листе, то добавляем один ресторан к ответу, в который мы можем попасть.

```
  if (g[v].size() == 1 && prev != -1) return 1;
```

В переменной *res* подсчитываем число ресторанов, в которое можно добраться из *v*.

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

Рассматриваем ребро (*v*, *to*). Если *v* – предок *to*, то не идем туда.

```
    int to = g[v][i];
    if (to == prev) continue;
```

Если в вершине *v* кота нет, то установим *c* = 0. Иначе присвоим *c* = *cnt* + 1. Переменная *c* хранит число посещенных последовательных котов вплоть до вершины *to* включительно.

```
    int c = (cats[to] == 0) ? 0 : cnt + 1;
```

Запускаем поиск в глубину из вершины *to*.

```
    res += dfs(to, v, c);
  }
  return res;
}
```

Основная часть программы. Читаем входные данные.

```
scanf("%d %d", &n, &m);
cats.resize(n + 1);
```

```
for (i = 1; i <= n; i++)
  scanf("%d", &cats[i]);
```

Строим дерево.

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

Функция ***dfs***, вызванная из вершины 1, возвращает ответ на задачу. Предка у вершины 1 нет, поэтому второй параметр положим равным -1. Стартовав из вершины 1, мы посетили cats[1] котов.

```
printf("%d\n", dfs(1, -1, cats[1]));
```