Март 1, 2022

7526. Сервер

Вы отвечаете за сервер, на котором необходимо выполнить несколько задач по принципу первый пришел – первый выполнен. Каждый день для выполнения этих задач Вы можете выделить на сервере более t минут. Зная время выполнения каждой задачи, Вы хотите определить, сколько задач будут выполнены сегодня.

Рассмотрим следующий пример. Пусть t = 180, время выполнения задач равны 45, 30, 55, 20, 80 и 20 минут (именно в таком порядке). Только четыре задания могут быть выполнены. На выполнение первых четырех задач следует потратить 150 минут. Пять заданий выполнить нельзя, так как тогда потребуется 230 минут, что больше 180. Несмотря на то что еще останется время на выполнение шестой задачи (на которую требуется 20 минут), после четвертой задачи нельзя выполнить шестую, так как пятая еще не совершена.

Вход. Первая строка содержит два целых числа n ($1 \le n \le 50$) и t ($1 \le t \le 500$), где n – количество задач. Следующая строка содержит n натуральных чисел, не больших 100, указывающих на время выполнения каждой задачи.

Выход. Вывести количество задач, которое может быть выполнено за t минут по принципу первый пришел – первый выполнен.

Пример входа	**Пример выхода**
6 180 45 30 55 20 80 20	4

Вычисляем префиксные суммы, пока они не станут большими *t*. Количество чисел в последней префиксной сумме, не большей *t*, равно искомому выполненному числу задач.

Реализация алгоритма

Читаем входные данные.

```
scanf("%d %d",&n,&t);
res = s = 0;
```

Обрабатываем задачи на сервере последовательно.

```
for(i = 0; i < n; i++)
{
  scanf("%d",&val);
```

Считаем сумму чисел.

```
  s += val;
```

Как только сумма *s* превзойдет *t* – останавливаемся обрабатывать задачи. Количество слагаемых *res* в ней и является ответом.

```
  if (s > t) break;
  res++;
}
```

Выводим ответ.

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

2669. Поворот

Дан массив $n \times m$. Требуется повернуть его по часовой стрелке на 90 градусов.

Вход. В первой строке даны натуральные числа n и m ($1 \le n, m \le 50$). На следующих n строках записано по m неотрицательных чисел, не превышающих 10^9 – сам массив.

Выход. Выведите перевернутый массив в формате входных данных.

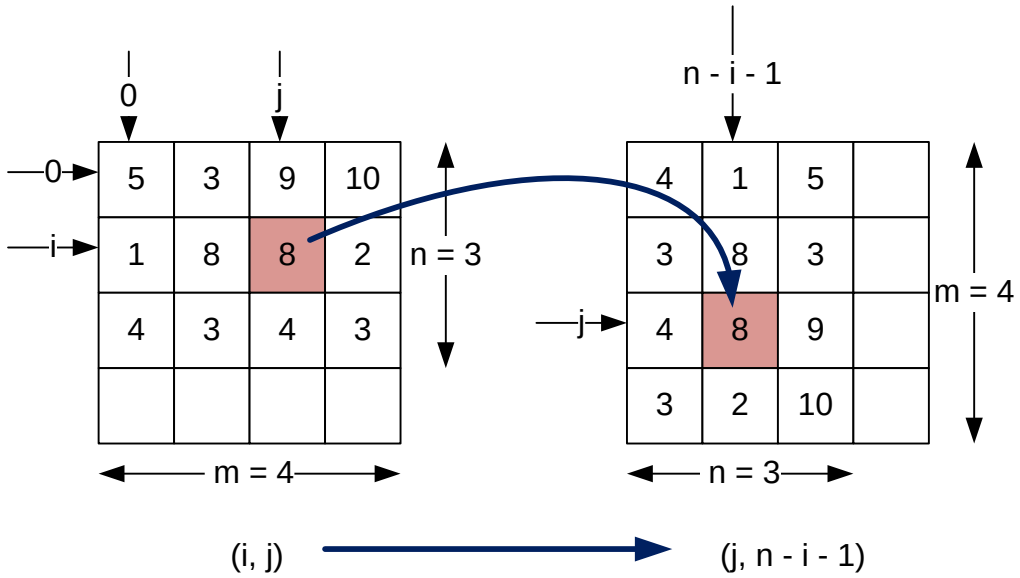
Пример входа

```
3 4
1 2 3 4
5 6 7 8
9 10 11 12
```

Пример выхода

```
4 3
9 5 1
10 6 2
11 7 3
12 8 4
```

При повороте на 90 градусов по часовой стрелке элемент матрицы a с координатами (i, j) переходит в элемент матрицы b с координатами $(j, n - i - 1)$. Если размер матрицы a составляет $n \times m$, то размер матрицы b будет $m \times n$. Индексация обеих матриц начинается с 0.



Реализация алгоритма

Объявим входной массив a и результирующий перевернутый b.

```
#define MAX 55
int a[MAX][MAX], b[MAX][MAX];
```

Читаем входной массив.

```
scanf("%d %d",&n,&m);
for(i = 0; i < n; i++)
for(j = 0; j < m; j++)
  scanf("%d",&a[i][j]);
```

Переворачиваем массив a в b.

```
for(i = 0; i < n; i++)
for(j = 0; j < m; j++)
  b[j][n-i-1] = a[i][j];
```

Выводим перевернутый массив b.

```
printf("%d %d\n",m,n);
for(i = 0; i < m; i++)
{
  for(j = 0; j < n; j++)
    printf("%d ",b[i][j]);
  printf("\n");
}
```

7326. Спальные вагоны

Поезд состоит из спальных вагонов, обозначенных буквой *k*, и сидячих вагонов, обозначенных буквой *p*. Найдите наибольшее количество спальных вагонов, следующих друг за другом в поезде.

Вход. В одной строке содержится последовательность букв *k* и *p*, длина которой от 1 до 1000 символов.

Выход. Вывести одно число – наибольшее количество следующих друг за другом спальных вагонов.

Пример входа	**Пример выхода**
kpkkp	2

В заданной строке необходимо найти самую длинную подстроку, состоящую только из букв *k*.

Объявим переменную *temp* – счетчик подряд идущих букв *k*. Если текущей является буква *k*, то увеличиваем *temp* на 1. Иначе сбрасываем *temp* в 0. Максимальное достижимое значение *temp* равно ответу, подсчитываем его в переменной *res*.

Пример. Рассмотрим пример из условия задачи.

s	k	p	k	k	p
temp	1	0	1	2	0
res	1	1	1	2	2

Реализация алгоритма

Входную строку храним в массиве s.

```
char s[1010];
```

Читаем входную строку *s*.

```
gets(s);
```

В переменной *temp* подсчитываем количество идущих подряд букв *k*. Ответ сохраняем в переменной *res*.

```
res = temp = 0;
for (i = 0; i < strlen(s); i++)
{
```

Если текущей буквой является *k*, то увеличим *temp* на 1. Иначе ряд из последовательных букв *k* обрывается, устанавливаем *temp* равным 0.

```
  if (s[i] == 'k') temp++; else temp = 0;
```

Ответом является максимальное значение среди всех возможных значений *temp*.

```
  if (temp > res) res = temp;
}
```

Выводим ответ.

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

8318. Удалите цифры

Дано натуральное число *n*. Удалить в этом числе все цифры 3 и 9, оставив порядок остальных цифр прежним.

Например, число 539013 преобразуется в 501.

Вход. Одно натуральное число *n* ($1 \le n \le 10^{18}$).

Выход. Удалите из числа *n* все тройки и девятки, сохранив относительный порядок остальных цифр.

Пример входа	**Пример выхода**
539013	501

Входное число *n* читаем в переменную *s* типа *string*. В строке *s* следует удалить цифры 3 и 9. Рассмотрим два подхода решения задачи:

1. Проходим по строке и удаляем символы ‘3’ и ‘9’ при помощи метода *erase*;
2. Объявим пустую строку *res*. Проходим по входной строке *s* и приписываем в конец *res* каждый символ s[*i*], не равный ‘3’ и ‘9’.

Реализация алгоритма

Объявим символьный массив *s*.

```
char s[20];
int i, j;
```

Читаем входную строку.

```
gets(s);
```

Устанавливаем указатели *i* и *j* на начало строки.

```
j = 0;
```

```
for(i = 0; i < strlen(s); i++)
```

Каждый символ s[*j*], не равный ‘3’ и ‘9’, копируем s[*j*].

```
  if (s[i] != '3' && s[i] != '9') s[j++] = s[i];
```

Ставим 0 байт в конце результирующей строки.

```
s[j] = 0;
```

Выводим ответ.

```
puts(s);
```

1642. Домашнее задание

Коля по-прежнему пытается пройти тест по теории чисел. Лектор настолько доведен до отчаяния знаниями Коли, что она дает ему каждый раз одну и ту же задачу.

В задаче следует проверить, делится ли $n!$ на n^2.

Вход. Одно число n ($1 \le n \le 10^9$).

Выход. Вывести "YES", если $n!$ делится на n^2. Иначе вывести "NO".

Пример входа	**Пример выхода**
3	NO

Поскольку $n! = 1 * 2 * \ldots * n$, то $n!$ делится на n. Если n простое, то произведение $n! / n = 1 * 2 * \ldots * (n - 1)$ не делится на n. Следовательно при простом n значение $n!$ не делится на n^2.

Если n не простое, то его можно представить в виде произведения двух чисел (не обязательно простых): например $n = a * b$. Тогда $n! = (a * b)!$ содержит множители a, b, $a * b$ и $n!$ делится на n^2.

Отдельно рассмотрим два случая:

- при $n = 1$ (не простое и не составное) значение $n!$ делится на n^2;
- при $n = 4$ (составное) значение $n!$ не делится на n^2.

Пример. Если $n = 5$ (простое), то $5! = 1 * 2 * 3 * 4 * 5$ не делится на 5^2.

Если $n = 15$ (составное), то $15!$ в своем разложении содержит множители 3, 5 и 15, и следовательно делится на 15^2.

Если $n = 4$ (составное), то $4! = 1 * 2 * 3* 4$ делится на 4, но не делится на 4^2.

Реализация алгоритма

Функция ***IsPrime*** проверяет, является ли число n простым. Если n простое, то функция возвращает 1. Иначе функция возвращает 0.

```
int IsPrime(int n)
{
  for(int i = 2; i <= sqrt(1.0*n); i++)
    if (n % i == 0) return 0;
  return 1;
}
```

Основная часть программы. Читаем входное значение *n*.

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

Выводим ответ в зависимости от значения *n*.

```
if (n == 4) puts("NO"); else
if (n == 1 || !IsPrime(n)) puts("YES");
                      else puts("NO");
```

6033. Пробег

Один раз в год Джо и его друзья хотят посетить местную ярмарку в Эрлангене, называемую Бергкирхвайх. В этом году они хотят совершить Kastenlauf (box run). Они стартуют от дома Джо, имея одну коробку (Kasten) пива с двадцатью бутылками. Чтобы их не мучила жажда, они пьют одну бутылку пива каждые 50 метров.

Поскольку путь от дома Джо к Бергкирхвайх довольно долгий, они должны потребить больше пива, чем взяли изначально. К счастью на пути имеются магазины, торгующие пивом. Когда они посещают магазин, то могут отказаться от своих пустых бутылок и купить новые, однако общее количество полных бутылок должно быть не больше двадцати (ребята слишком ленивы, чтобы нести более одной полной коробки).

Вам заданы координаты магазинов, дома Джо и расположения Бергкирхвайх. Напишите программу, которая определит, смогут ли Джо и его друзья счастливо добраться до Бергкирхвайх, или же пиво закончится на их пути.

Вход. Первая строка содержит количество тесов t ($t \le 50$). Каждая строка начинается количеством магазинов n ($0 \le n \le 100$), торгующих пивом. Следующие $n + 2$ строки содержат (именно в таком порядке) местоположение дома Джо, магазинов и Бергкирхвайх. Местоположение задается двумя целочисленными координатами x и y (в метрах, $-32768 \le x, y \le 32767$). Поскольку Эрланген представляет собой город в виде прямоугольной сетки, то расстояние между двумя точками равно сумме разностей координат (Манхетенская метрика).

Выход. Для каждого теста вывести в отдельной строке либо "happy" (если Джо и его друзья благополучно достигнут Бергкирхвайха), или "sad" (если у них на пути закончится пиво).

Пример входа

```
2
2
0 0
1000 0
1000 1000
2000 1000
2
0 0
1000 0
2000 1000
2000 2000
```

Пример выхода

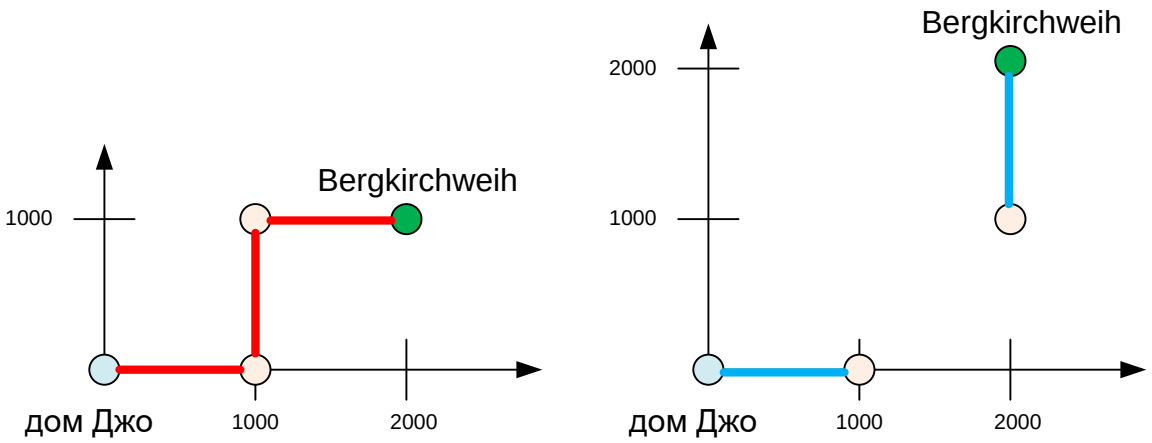
```
happy
sad
```

Поскольку количество бутылок, которое ребята могут нести с собой, равно 20, а через каждые 50 метров следует выпить одну бутылку, то перемещаться они смогут только между магазинами, расстояние между которыми не более 20 * 50 = 1000 метров.

Построим граф, вершинами которого будут дом Джо, магазины и Бергкирхвайх. Ребро между вершинами присутствует если только расстояние между ними не более 1000 метров. То есть неориентированные ребра указывают возможные пути передвижения Джо и его друзей.

Запускаем поиск в глубину из дома Джо. Если из нее можно достичь Бергкирхвайх, то друзья будут счастливы, иначе – нет.

Пример. Для первого и второго примеров граф будет выглядеть следующим образом:



В первом примере Джо с друзями могут достичь ярмарку Бергкирхвайх. Во втором – нет. Передвигаться можно только между теми вершинами графа, расстояние между которыми не больше 1000 метров.

Реализация алгоритма

Координаты магазинов храним в (x[i], y[i]). Объявим граф передвижения g.

```
#define MAX 110
int x[MAX], y[MAX];
```

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

Поиск в глубину из вершины *v*.

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

Основная часть программы. Читаем входные данные.

```
scanf("%d",&tests);
while(tests--)
{
  scanf("%d",&n);
```

Поскольку входные данные содержат несколько тестов, то очищаем массивы.

```
  g.clear(); g.resize(n+2);
  used.clear(); used.resize(n+2);
```

Читаем координаты дома Джо (x[0], y[0]), магазинов и Бергкирхвайх (x[n – 1], y[n – 1]).

```
  for(i = 0; i < n + 2; i++)
    scanf("%d %d",&x[i],&y[i]);
```

Строим граф возможных передвижений. Из вершины (x[i], y[i]) можно перейти в вершину (x[j], y[j]) если только расстояние между ними не более 1000 метров. Граф неориентированный. Метрика манхетенская.

```
  for(i = 0; i < n + 2; i++)
  for(j = i + 1; j < n + 2; j++)
    if (abs(x[i] - x[j]) + abs(y[i] - y[j]) <= 1000)
    {
      g[i].push_back(j);
      g[j].push_back(i);
    }
```

Стартуем из дома Джо. Запускаем поиск в глубину из вершины 0.

```
  dfs(0);
```

Если Бергкирхвайх достижим, то выводим “happy”, иначе выводим “sad”.

```
  if (used[n + 1]) puts("happy");
  else puts("sad");
}
```

5115. Хоккей на Урале

Для популяризации хоккея и повышения мастерства хоккейных команд Урала был организован Всеуральский турнир. Для участия в турнире были приглашены n хоккейных команд из городов Урала.

После первых двух туров, в каждом из которых каждая команда провела по одному матчу, оказалось, что команд слишком много. Организаторами турнира было решено допустить к дальнейшему участию только k команд, никакие две из которых не встречались в рамках первых двух туров.

Требуется написать программу, которая находит набор из k команд, удовлетворяющий условиям, либо выводит сообщение о том, что это сделать невозможно. В случае существования нескольких подходящих наборов необходимо найти любой из них.

Вход. В первой строке содержится число n ($2 \le n \le 10^5$, n чётное). Следующие n строк содержат описания всех прошедших матчей. Описание каждого матча состоит из двух натуральных чисел, не превышающих n – номеров команд, игравших в матче. Первые n / 2 из них соответствуют матчам первого тура, оставшиеся – матчам второго тура. Последняя строка содержит одно число k ($2 \le k \le n$).

Гарантируется, что каждая команда сыграла ровно два матча: один в первом туре и один во втором.

Выход. Вывести либо единственное число 0, если решения не существует, либо k различных чисел – номера отобранных команд.

Пример входа 1

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

Пример выхода 1

```
1 4 3
```

Пример входа 2

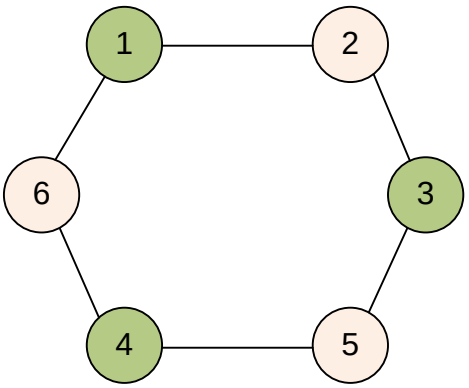
```
4
1 2
3 4
2 1
4 3
3
```

Пример выхода 2

```
0
```

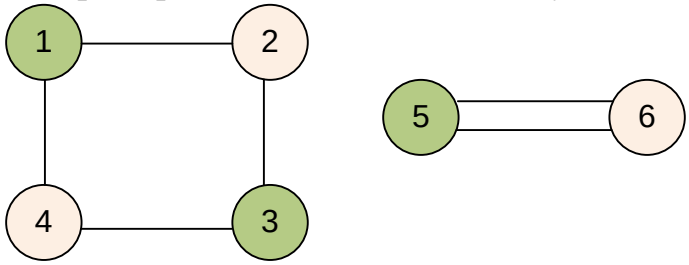
Каждой хоккейной команде поставим в соответствие вершину графа. Сыгранные игры – ребра графа. Каждая команда провела в точности две игры. Степень каждой вершины графа равна 2, значит граф представляет собой набор простых циклов. В каждом цикле можно выбрать вершины (номера команд) через одну. Они нам подходят, так как они еще не играли между собой. Таким образом всегда можно выбрать не более n / 2 вершин. Следовательно решение существует только при $k \le n / 2$.

Пример. После сыгранных двух туров граф имеет вид одного цикла из 6 вершин.



Отобранными могут быть команды, расположенные в цикле через один – например 1, 3, 4 или 2, 5, 6.

После двух туров, например, может сложиться следующая стуация:



Граф представляет собой два цикла. Тогда отобранными могут быть команды 1, 3, 5 или 2, 4, 6.

Реализация алгоритма

Объявим список смежности графа g и рабочие массивы.

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

Поиск в глубину из вершины v. В массиве res запоминаем порядок обхода вершин.

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

```
 }
}
```

Основная часть программы. Читаем входные данные. Строим граф.

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

Граф может быть несвязным. Запускаем поиск в глубину на несвязном графе.

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

В зависимости от значения k выводим ответ. Решение существует только при $k \le n / 2$.

```
if (k <= n / 2)
{
  for (i = 0; i < k; i++)
    printf("%d ", res[2 * i]);
  printf("\n");
}
else
  puts("0");
```

9767. Понк Уоршалл

Слушание рок-музыки меняет вашу ядерную ДНК. Этот поразительный и невероятный факт был недавно опубликован в Rock Nature Weekly, одном из ведущих научных журналов на планете. Частью исследования было взятие образцов ДНК у добровольцев как до, так и после сезона рок-концертов. Образцы были обработаны, и из них были выделены различные гены. Для каждого человека каждый ген был выделен дважды: вариант до рок-сезона и вариант после сезона. Эти два варианта были парными, и во многих случаях было обнаружено, что один вариант представляет собой некоторую перестановку другого варианта в паре.

Следующим шагом в исследовании является определение того, как происходят перестановки. Преобладающая гипотеза предполагает, что перестановка состоит из последовательности транспозиций, так называемых обменов. Обмен – это событие (его химия еще полностью не изучена), при котором ровно два азотистых основания в гене меняются местами. Никакие

другие азотистые основания гена обменом не затрагиваются. Положения двух замененных нуклеотидных оснований могут быть совершенно произвольными.

Чтобы предсказать и наблюдать движение молекул в процессе перестановки, исследователям необходимо знать теоретическое минимальное количество перестановок, которые могут привести к конкретной перестановке азотистых оснований в гене. Напоминаем, что ген ядерной ДНК представляет собой последовательность азотистых оснований цитозина, гуанина, аденина и тимина, которые кодируются как C, G, A и T соответственно.

Вход. Содержит две строки. Каждая строка содержит n заглавных букв “A”, “C”, “G” или “T” ($1 \le n \le 10^6$). Две строки представляют собой одну пару некоторых версий гена. Первая строка задает ген до рок-сезона, вторая строка задает тот же ген от того же человека после рок-сезона. Число вхождений каждого азотистого основания одинаково в обеих строках.

Выход. Выведите минимальное количество обменов, преобразующих первую версию гена во вторую.

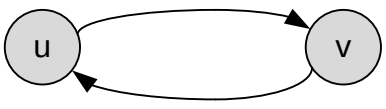
Пример входа 1	**Пример выхода 1**
CGATA ATAGC	2

Пример входа 2	**Пример выхода 2**
CTAGAGTCTA TACCGTATAG	7

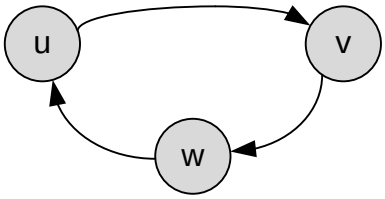
Построим граф из 4 вершин: A, C, T, G. Пусть *dna*1 и *dna*2 – входные строки. Нам следует при помощи обменов преобразовать *dna*1[*i*] в *dna*2[*i*]. Проведем в графе мультиребра: для каждого i ($0 \le i < n$) проведем ориентированное ребро (*dna*1[*i*], *dna*2[*i*]).

Пусть ecnt[*u*][*v*] содержит количество ориентированных ребер $u \to v$. Далее посчитаем количество циклов длины 2, 3 и 4 графе. Поскольку граф состоит из 4 вершин, то он может содержать цикл максимум из 4 вершин.

Количество циклов длины 2 между вершинами u и v равно $r = \min(\text{ecnt}[u][v], \text{ecnt}[v][u])$. Удалим эти циклы из графа. Для этого следует вычесть r из ecnt[*u*][*v*] и ecnt[*v*][*u*]. Для каждого цикла длины 2 достаточно совершить 1 обмен.

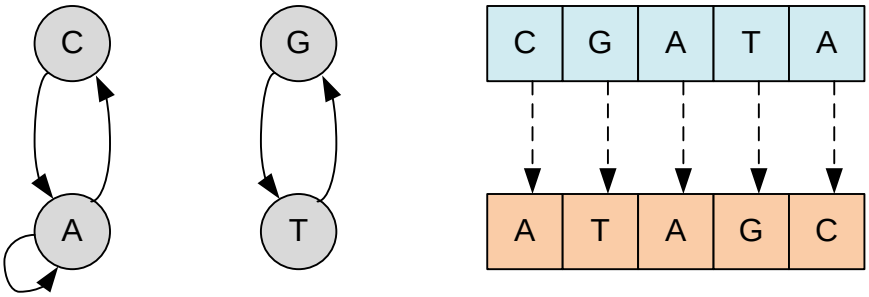


Для нахождения циклов длины 3 следует перебрать тройки вершин (u, v, w). Их количество равно $r = \min(\text{ecnt}[u][v], \text{ecnt}[v][w], \text{ecnt}[w][u])$. Снова удаляем эти циклы из графа, вычитая r из ecnt[*u*][*v*], ecnt[*v*][*w*], ecnt[*w*][*u*]. Для каждого цикла длины 3 достаточно совершить 2 обмена.



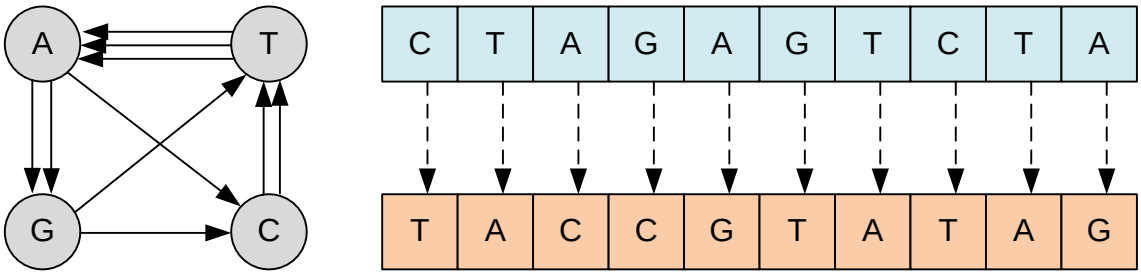
После удаления циклов длины 2 и 3 в графе останутся только циклы длины 4. Подсчитаем сумму s всех оставшихся значений ecnt[u][v] по всем парам (u, v). Полученная сумма s будет кратна 4. Для каждого цикла длины 4 следует совершить 3 обмена.

Пример. Граф из первого примера имеет следующий вид. Он соответствует преобразованию, приведенному справа.



Граф содержит два цикла длины 2. Следовательно для преобразования первой строки во вторую достаточно 2 обмена (по одному обмену на каждый цикл).

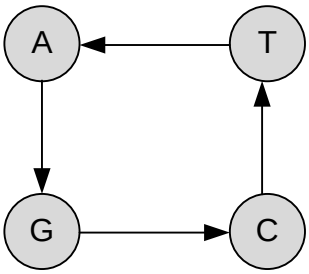
Построим граф для второго примера.



Граф, например, содержит два цикла длины 3:

- A → G → T;
- A → C → T;

Для уничтожения каждого из них следует провести два обмена. Удалив эти два цикла, получим граф:



Остался цикл длины 4, на который следует потратить 3 обмена.

Реализация алгоритма

Объявим матрицу смежности графа *ecnt* из 4 вершин. Граф допускает мультиребра. Значение ecnt[*u*][*v*] содержит количество ориентированных ребер *u* → *v*. При помощи отображения letterid каждой букве поставим в соответствие номер вершины графа.

```
vector<vector<int>> ecnt(4, vector<int>(4));
map<char, int> letterid{ {'A', 0}, {'C', 1}, {'G', 2}, {'T', 3} };
```

Читаем входные строки.

```
cin >> dna1 >> dna2;
```

Строим граф. Из *dna*1[*i*] в *dna*2[*i*] проводим ориентированное ребро.

```
len = dna1.length();
for (i = 0; i < len; ++i)
{
  l1 = letterid[dna1[i]];
  l2 = letterid[dna2[i]];
  ecnt[l1][l2]++;
}
```

В переменной *result* подсчиываем количество обменов.

```
result = 0;
```

Подсчитываем количество циклов длины 2 и удаляем их из матрицы смежности.

```
for (i = 0; i < 4; i++)
for (j = i + 1; j < 4; j++)
{
  mn = min(ecnt[i][j], ecnt[j][i]);
  result += mn;
  ecnt[i][j] -= mn;
  ecnt[j][i] -= mn;
}
```

Подсчитываем количество циклов длины 3 и удаляем их из матрицы смежности.

```
for (i = 0; i < 4; i++)
for (j = 0; j < 4; j++)
for (k = 0; k < 4; k++)
{
  if (i == j || i == k || j == k) continue;
  mn = min(min(ecnt[i][j], ecnt[j][k]), ecnt[k][i]);
  result += 2 * mn;

  ecnt[i][j] -= mn;
  ecnt[j][k] -= mn;
  ecnt[k][i] -= mn;
}
```

В переменной *rest* подсчитываем оставшиеся ребра. Все они принадлежат циклам длины 4.

```
rest = 0;
for (i = 0; i < 4; ++i)
for (j = 0; j < 4; ++j)
  if (i != j) rest += ecnt[i][j];
```

rest ребер образуют *rest* / 4 циклов длины 4. На каждый цикл требуется 3 обмена.

```
result += 3 * rest / 4;
```

Выводим ответ.

```
cout << result << endl;
```

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| + 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:

```
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 + … + n.

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

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

$$1 + 2 + \ldots + n = \text{S}(n)$$

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

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

$$1 + 2 + \ldots + n - 1 = \text{S}(n - 1)$$

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

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

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

Таким образом

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

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

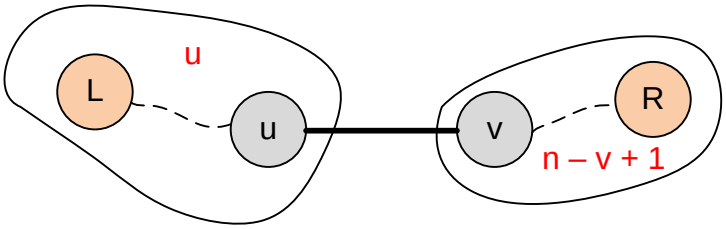
- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$,
- $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

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 $v \le R$.



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 $u * (n - v + 1)$.

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);
```

4516. Деревья в саду

Центральный сад страны Олимпия настолько большой, что один садовник не в силах его обслуживать. Было принято решение разделить сад на две части. Определенные деревья будут отнесены к первой части, а оставшиеся – ко второй. Одна из частей сада может остаться пустой.

Между каждой парой деревьев в саду протоптаны тропинки. Когда садовники идут от дерева к дереву, они обязательно идут по тропинке соединяющей непосредственно эти два дерева. Длина тропинки одинакова при перемещении в обе стороны.

Для упрощения работы садовников, разделение решили проводить так, чтобы длина самой длинной тропинки между парой деревьев, принадлежащих одной и той же части, была минимальна.

Напишите программу, которая по информации о длинах тропинок между всеми парами деревьев находит длину самой длинной тропинки между деревьями из одной части сада, при оптимальном разделении сада на части.

Вход. Первая строка содержит целое число n ($2 \le n \le 1000$) – количество деревьев в саду. Каждая i-ая из последующих ($n - 1$) -ой строк содержит $n - i$ чисел, которые последовательно представляют длины тропинок между i-ым деревом и деревьями с $i + 1$-го до n-го. Все числа целые, неотрицательные, и не превышают 10^6.

Выход. Вывести одно целое число – минимальную для всех возможных разбиений сада длину самой длинной тропинки между деревьями в одной из частей сада.

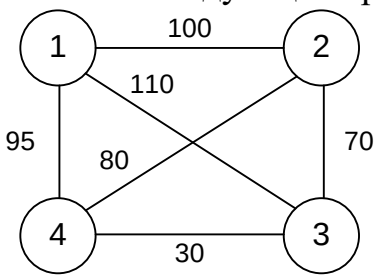
Пример входа	**Пример выхода**
3 1 5 1	1

В задаче требуется поставить в соответствие каждому дереву одного садовника, который будет его обслуживать.

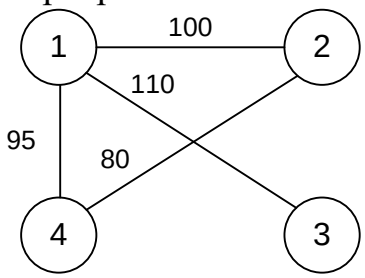
Рассмотрим следующую подзадачу: можно ли разбить деревья сада между двумя садовниками таким образом, чтобы не существовало тропинок (по которым каждый из садовников будет ходит в отдельности) длины больше x. Если мы научимся отвечать на этот вопрос, то дальше ищем минимальное значение x (ответ к задаче), при котором существует такое разбиение, бинарным поиском.

Построим граф, в котором присутствуют только ребра длины больше x. Если такой граф является двудольным, то каждое множество деревьев будет обслуживать один из садовников. И ни один из садовников не будет иметь в своем распоряжении тропинки длиной больше x.

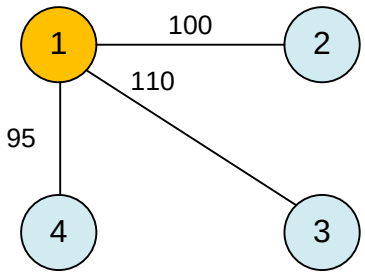
Пример. Пусть изначально имеется следующий граф:



Попробуем разбить деревья сада между двумя садовниками таким образом, чтобы не существовало тропинок длины строго больше $x = 70$. Для этого следует проверить граф, содержащий лишь ребра величины больше 70, на двудольность:



Такой граф не является двудольным. Рассмотрим граф, для которого $x = 80$. Такой граф будет двудольным:



Можно заметить, что при $x < 80$ в графе из ребер с весами большими x, всегда будет цикл нечетной длины (1 – 2 – 4). Поэтому ответом будет значение 80.

Реализация алгоритма

Объявим массивы для работы с графом. Граф храним в виде весовой матрицы.

```
.
#define MAX 1010
int g[MAX][MAX], used[MAX];
```

Поиск в глубину из вершины *v*. Красим вершину *v* цветом *Color*. Проверяем, является ли граф двудольным, если разрешено проходить лишь по ребрам, веса которых больше *Value*. То есть можно ли разбить граф на две доли так, чтобы все ребра с весом больше *Value* соединяли вершины из разных долей.

```
void dfs(int v, int Color, int Value)
{
  if (Error) return;
  used[v] = Color;
```

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

Функция CanDivide возвращает 1, если возможно разделить сад так, чтобы по тропинкам длины *Value* не ходили садовники. Поиск в глубину запускаем как на несвязном графе, так как реально он будет вестись только по ребрам, вес которых больше *Value*.

```
int CanDivide(int Value)
{
  memset(used,0,sizeof(used));
  Error = 0;
  for(int i = 1; i <= n; i++)
  {
    if (!used[i]) dfs(i,1,Value);
    if (Error) return 0;
  }
  return 1;
}
```

Основная часть программы. Читаем входные данные. Ищем длину минимального *mind* и максимального *maxd* ребра.

```
scanf("%d", &n);
mind = 2e9; maxd = 0;
memset(g,0,sizeof(g));
for(i = 1; i < n; i++)
for(j = i + 1; j <= n; j++)
{
  scanf("%d", &g[i][j]);
  g[j][i] = g[i][j];
  if (g[i][j] > maxd) maxd = g[i][j];
  if (g[i][j] < mind) mind = g[i][j];
}
```

Если имеется два дерева ($n = 2$), то ответ 0. Этот случай следует обработать отдельно.

```
if (n == 2) maxd = 0; else
```

Иначе запускаем бинарный поиск на промежутке [*mind*, *maxd*].

```
while(mind < maxd)
{
  int Middle = (mind + maxd) / 2;
  if (CanDivide(Middle)) maxd = Middle;
  else mind = Middle + 1;
}
```

Выводим ответ.

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