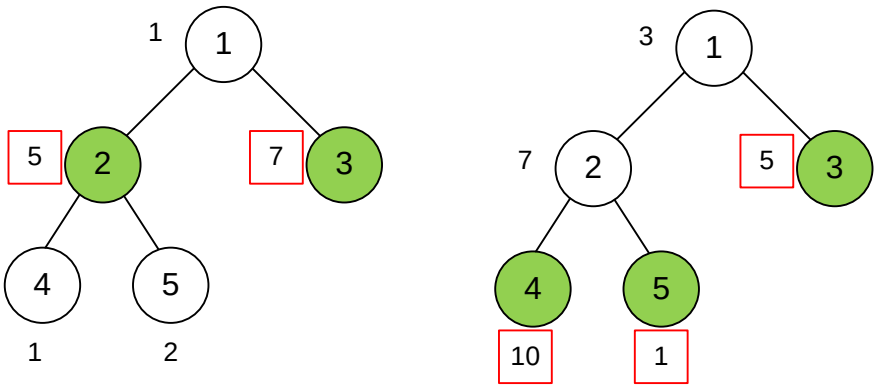
Март 16, 2022

Задача A. Максимальная сумма на дереве
Задача B. Дерево
Задача C. Суперпалиндромы
Задача D. Палиндромы
Задача E. Забор для травы
Задача F. Проблема Лонги
Задача G. Экстремум Эйлера
Задача H. Гистограмма
Задача I. Sigma-функция на отрезке
Задача J. Триомино

973. Максимальная сумма на дереве

Имеется дерево из n вершин, где вершина номер i ($1 \le i \le n$) имеет c_i монет. Вам следует выбрать такое подмножество вершин, в котором никакие две из них не являются соседними (то есть вершины соединенные ребром), а сумма монет в выбранных вершинах наибольшая.



Вход. Первая строка содержит количество вершин n ($1 \le n \le 10^5$) в дереве. Каждая из следующих $n - 1$ строк содержит два числа u и v ($1 \le u, v \le n$), задающих ребро в дереве. Последняя строка содержит n целых неотрицательных чисел c_1, ... c_n – количество монет в вершинах дерева.

Выход. Вывести максимально возможную сумму монет в выбранном подмножестве вершин дерева.

Пример входа 1

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

Пример выхода 1

```
12
```

Пример входа 2

```
5
1 2
1 3
2 4
2 5
3 7 5 10 1
```

Пример выхода 2

```
16
```

Пусть v – вершина дерева. Обозначим через:

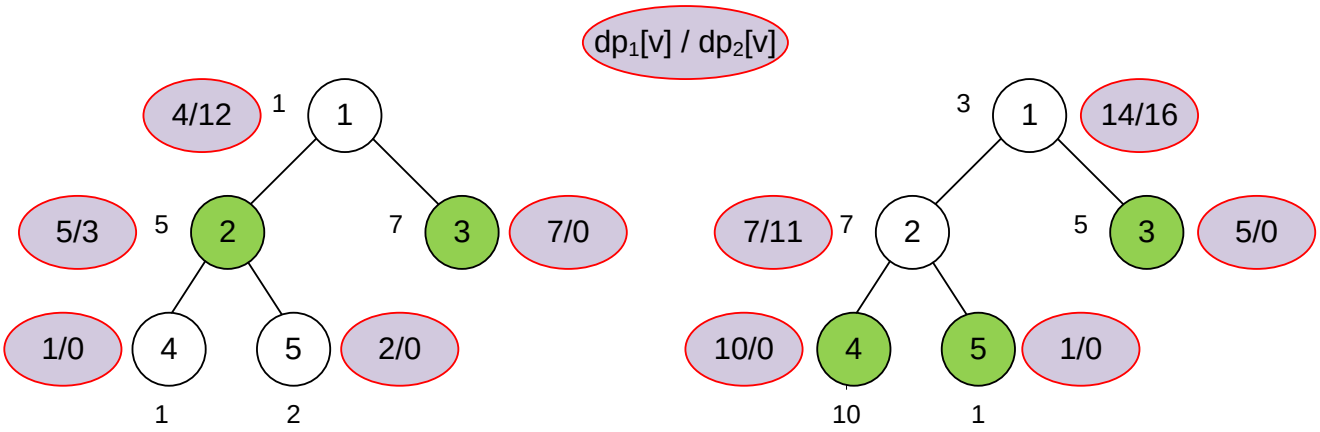
- $dp_1(v)$ наибольшую сумму монет, которую можно собрать с поддерева с корнем v при условии, что монеты в вершине v мы берем.
- $dp_2(v)$ наибольшую сумму монет, которую можно собрать с поддерева с корнем v при условии, что монеты в вершине v мы не берем.

Тогда ответом задачи будет $\max(dp_1(v), dp_2(v))$, если взять первую вершину за корень дерева. Определим рекурсивно введенные функции.

- Пусть монеты в вершине v мы берем. Тогда монеты ни с одного сына вершины v брать нельзя: $dp_1(v) = c_v + \sum_{i=1}^{k} dp_2(to_i)$, где to_1, …, to_k – сыновья вершины v.
- Пусть монеты в вершине v мы не берем. Тогда монеты в сыновьях вершины v можно брать, а можно и не брать. Из этих двух вариантов следует выбрать тот, в котором сумма монет максимальна: $dp_2(v) = \sum_{i=1}^{k} \max(dp_1(to_i), dp_2(to_i))$, где to_1, …, to_k – сыновья вершины v.

Пример

Расставим метки $dp_1(v)$ / $dp_2(v)$ на вершинах деревьев из примеров.



Для первого примера имеем:

- $dp_1(1) = c_1 + dp_2(2) + dp_2(3) = 1 + 3 + 0 = 4$;
- $dp_2(1) = \max(dp_1(2), dp_2(2)) + \max(dp_1(3), dp_2(3)) = 5 + 7 = 12$;

- $dp_1(2) = c_2 + dp_2(4) + dp_2(5) = 5 + 0 + 0 = 5$;
- $dp_2(2) = \max(dp_1(4), dp_2(4)) + \max(dp_1(5), dp_2(5)) = 1 + 2 = 3$;

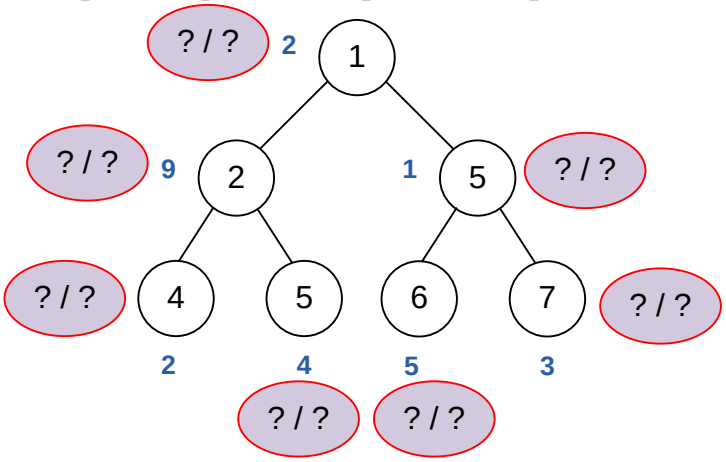
Для второго примера имеем:

- $dp_1(1) = c_1 + dp_2(2) + dp_2(3) = 3 + 11 + 0 = 14$;
- $dp_2(1) = \max(dp_1(2), dp_2(2)) + \max(dp_1(3), dp_2(3)) = 11 + 5 = 16$;

- $dp_1(2) = c_2 + dp_2(4) + dp_2(5) = 7 + 0 + 0 = 7$;
- $dp_2(2) = \max(dp_1(4), dp_2(4)) + \max(dp_1(5), dp_2(5)) = 10 + 1 = 11$;

Упражнение

Расставьте метки $dp_1(v)$ / $dp_2(v)$ на вершинах дерева.



Реализация алгоритма

Объявим рабочие массивы.

```
vector<vector<int> > g;
vector<int> dp1, dp2, cost;
```

Функция ***dfs*** реализует поиск в глубину – динамику на дереве. Вычисляем значения dp_1 и dp_2 во всех вершинах дерева.

```
void dfs(int v, int parent = -1)
{
  dp1[v] = cost[v];
  dp2[v] = 0;

  for(int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i];
    if(to == parent) continue;
    dfs(to, v);
    dp1[v] += dp2[to];
    dp2[v] += max(dp1[to], dp2[to]);
  }
}
```

Основная часть программы. Читаем дерево и массив монет с.

```
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);
}

dp1.resize(n+1); dp2.resize(n+1);
cost.resize(n+1);
for(i = 1; i <= n; i++)
  scanf("%d",&cost[i]);
```

Объявляем вершину 1 корнем. Запускаем из нее поиск в глубину.

```
dfs(1);
```

Вычисляем и выводим ответ.

```
ans = max(dp1[1], dp2[1]);
printf("%d\n",ans);
```

3849. Дерево

Дерево называется взвешенным, если каждому его ребру приписано одно число – длина ребра. Все длины положительны. Для каждой вершины необходимо найти наибольшее возможное расстояние до любой из вершин дерева.

Вход. Содержит описание дерева. Первая строка содержит количество его вершин n ($2 \le n \le 50000$). Каждая из следующих $(n - 1)$ строк содержит описание ребер дерева. Каждое ребро задается тремя положительными целыми числами.

Первые два числа – номера вершин, которые соединяет ребро, от 1 до n, третье число – длина ребра. Общая длина всех ребер не превосходит $2^{31} - 1$. Гарантируется, что входное дерево корректно.

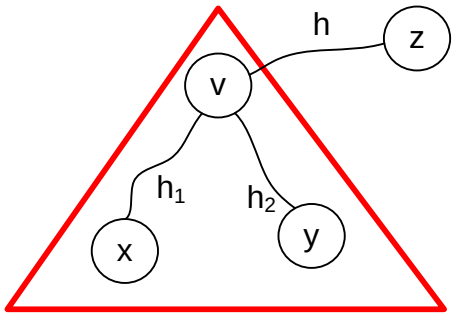
Выход. Содержит в точности n строк: k-ая строка содержит расстояние от вершины k ($k = 1..n$) до самой дальней вершины.

Пример входа	**Пример выхода**
6 1 5 3 2 6 3 6 1 1 1 3 5 4 6 4	5 9 10 10 8 6

Пусть g[i][j] содержит вес ребра между вершинами i и j. Реализуем функцию ***dfs*1**, которая для каждой вершины v найдет максимальное расстояние depth[v] от v до листа в поддереве с корнем v. Если u_1, …, u_k – сыновья v, и значения depth[u_i] уже посчитаны, то

$$\text{depth}[v] = \max\,(g[v][u_i] + \text{depth}[u_i])$$

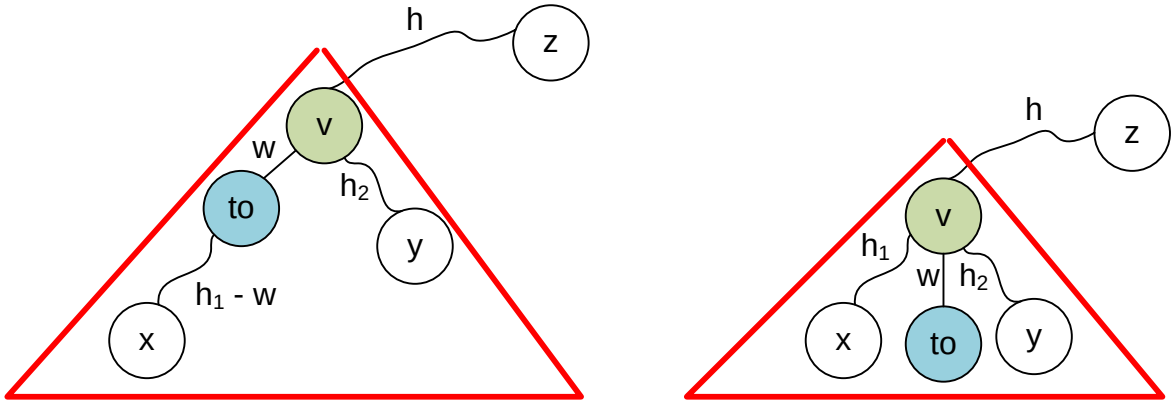
Второй поиск в глубину ***dfs*2**(v, *prev*, h) для каждой вершины v будет вычислять ответ res[v]. Второй параметр *prev* – предок вершины v. Значение h равно наибольшей возможной длине от v до вершины, лежащей вне поддерева с корнем v. Сначала найдем наибольшее h_1 и второе наибольшее h_2 расстояние от v до листа в поддереве с корнем в v ($h_1 \geq h_2$).



Отметим, что $h_1 = \max\limits_{to}(g[v][to] + depth[to])$, где максимум берется по всем сыновьям to вершины v. Соответственно h_2 является вторым максимум указанной суммы. Заметим также, что значения h_1 и depth[v] совпадают.

Наибольшее возможное расстояние res[v] от v до любой вершины дерева равно

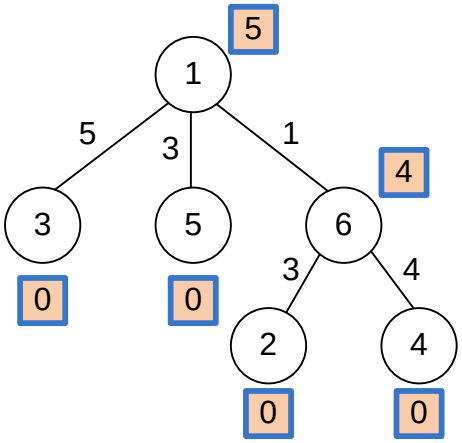
$$\max(h, \text{depth}[v])$$



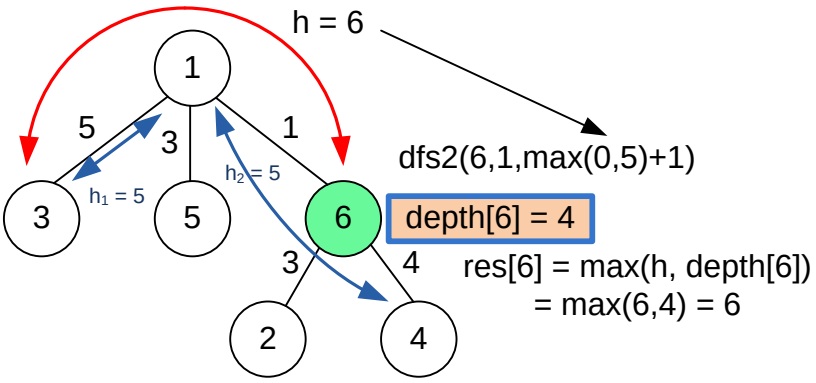
Пусть *to* – сын вершины *v*, лежащий на пути длины h_1 из *v* в самый дальний лист (картинка слева). Тогда при входе в *to* самым длинным расстоянием от *to* до вершины вне поддерева с корнем в *to* будет $\max(h, h_2) + w$, где $w = g[v][to]$. Поэтому состоится рекурсивный вызов dfs2(*to*, *v*, $\max(h, h_2) + w$).

Пусть сын *to* вершины *v* не лежит на самом длинном пути h_1 (картинка справа). Тогда при входе в *to* самым длинным расстоянием от *to* до вершины вне поддерева с корнем в *to* будет $\max(h, h_1) + w$, совершаем рекурсивный вызов dfs2(*to*, *v*, $\max(h, h_1) + w$).

Пример. Первым обходом в глубину возле каждой вершины *v* вычислим значение depth[*v*].



Второй обход в глубину. Для вершины $v = 1$ значение $h = 0$, наибольшими расстояниями до листов будут $h_1 = 5$ и $h_2 = 5$. При входе в вершину 6 обход в глубину будет вызван с параметрами dfs2(6, 1, max(0, 5) + 1) или dfs2(6, 1, 6), то есть в вершине 6 значение $h = 6$. При первом обходе в глубину было вычислено depth[6] = 4. Следовательно res[6] = max(*h*, depth[6]) = max(6, 4) = 6.



Реализация алгоритма

Взвешенное дерево храним в списке смежностей g. Объявим рабочие массивы.

```
vector<vector<pair<int,int> > > g;
vector<int> depth, res;
```

Максимальное расстояние от вершины *v* до листа в поддереве с корнем *v* вычисляем в depth[*v*] при помощи функции ***dfs*1**.

```
void dfs1(int v, int prev = -1)
{
  for(int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i].first;
    int w = g[v][i].second;
    if(to != prev)
    {
      dfs1(to, v);
      depth[v] = max(depth[v], depth[to] + w);
    }
  }
}
```

Второй обход в глубину ***dfs*2** для каждой вершины *v* находит наибольшее возможное расстояние до любой из вершин дерева res[*v*].

```
void dfs2(int v, int prev = -1, int h = 0)
{
  res[v] = max(h, depth[v]);
```

Вычисляем наибольшее h_1 и второе наибольшее h_2 расстояние от *v* до листа в поддереве с корнем в *v*.

```
  int h1 = 0, h2 = 0;
  for(int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i].first;
    int w = g[v][i].second;
```

```
    if (to != prev)
    {
      int h = depth[to] + w;
      if (h > h1) h2 = h1, h1 = h; else
      if (h > h2) h2 = h;
    }
  }

  for(int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i].first;
    int w = g[v][i].second;
    if (to == prev) continue;

    if(h1 == depth[to] + w)
      dfs2(to,v,max(h,h2) + w);
    else
      dfs2(to,v,max(h,h1) + w);
  }
}
```

Основная часть программы. Читаем входное дерево.

```
scanf("%d",&n);
g.resize(n+1);
depth.resize(n+1); res.resize(n+1);
for(i = 0; i < n - 1; i++)
{
  scanf("%d %d %d",&u,&v,&dist);
  g[u].push_back(make_pair(v,dist));
  g[v].push_back(make_pair(u,dist));
}
```

Совершаем два обхода в глубину.

```
dfs1(1);
dfs2(1);
```

Выводим ответ.

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

470. Суперпалиндромы

Назовём палиндромом строку длиной более одного символа, которая одинаково читается как справа налево, так и слева направо. Назовём суперпалиндромом строку, которая может быть представлена как конкатенация одного или более палиндромов. Дана строка S. Необходимо найти количество подстрок в S, которые являются суперпалиндромами.

Вход. Строка S содержит последовательность от 1 до 1000 строчных латинских букв без пробелов.

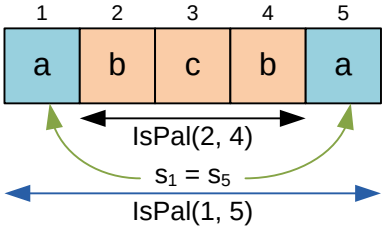
Выход. Выведите одно число – количество подстрок S, являющихся суперпалиндромами.

Пример входа	**Пример выхода**
abacdc	3

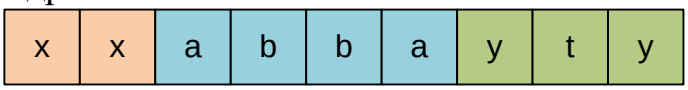
Пусть s – входная строка. Подстрока $s_i \ldots s_j$ является палиндромом, если:

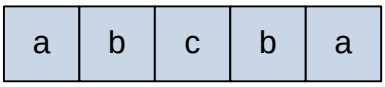
- $i \geq j$ (подстрока пустая или состоит из одного символа);
- $s_i = s_j$ и $s_{i+1} \ldots s_{j-1}$ является палиндромом;

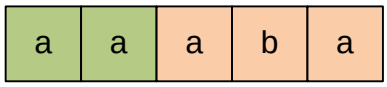
Пусть функция IsPal(i, j) возвращает 1, если $s_i \ldots s_j$ является палиндромом, и 0 иначе. Значения IsPal(i, j) запоминаем в ячейках pal[i][j].



Строка является суперпалиндромом, если она может быть представлена как конкатенация одного или более палиндромов. Например, следующие строки являются суперпалиндромами:

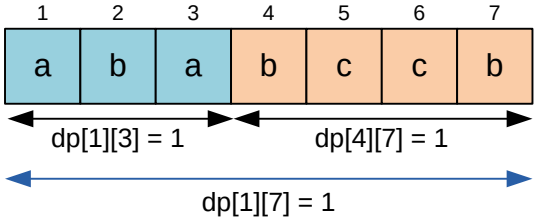






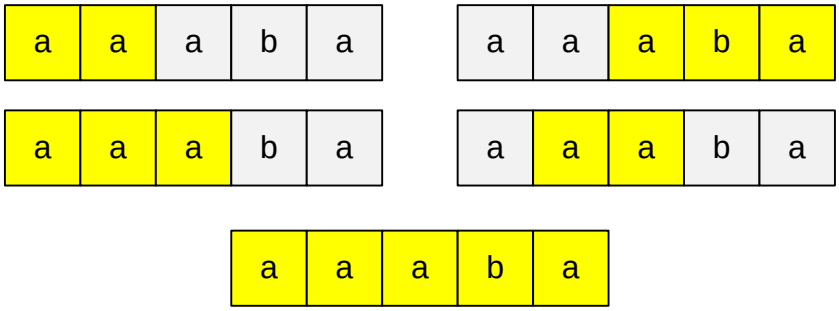
Пусть dp[i][j] = 1, если подстрока $s_i \ldots s_j$ ($i < j$) является суперпалиндромом и dp[i][j] = 0 иначе. Переберем все пары (i, j) для $0 \leq i < j < n$ и если подстрока $s_i \ldots s_j$ является палиндромом, то она является и суперпалиндромом, установим dp[i][j] = 1. Отметим, что dp[i][j] = 0 при $i \geq j$. Слово из одной буквы палиндромом не считается, поэтому как частный случай имеем dp[i][i] = 0.

Для каждой пары (i, j) переберем все возможные значения k ($i < k < j - 1$) и если $s_i \ldots s_k$ и $s_{k+1} \ldots s_j$ являются суперпалиндромами (они состоят из более чем одного символа вследствие ограничения на k), то и $s_i \ldots s_j$ является суперпалиндромом.



Остается подсчитать количество пар (i, j), для которых $i < j$ и dp[i][j] = 1. Это число и есть ответ.

Пример. Для строки *aaaba* имеется 5 подстрок, которые являются суперпалиндромами.



Реализация алгоритма

Объявим рабочие массивы.

```
#define MAX 1010
char s[MAX];
int dp[MAX][MAX];
int pal[MAX][MAX];
```

Реализуем рекурсивную функцию ***IsPal***(*i*, *j*), которая возвращает 1, если $s_i \ldots s_j$ является палиндромом. Иначе функция возвращает 0. Подстрока $s_i \ldots s_j$ есть палиндром, если $s_i = s_j$ и $s_{i+1} \ldots s_{j-1}$ является палиндромом. Значения IsPal(*i*, *j*) запоминаем в ячейках pal[*i*][*j*].

```
int IsPal(int i, int j)
{
  if (i >= j) return pal[i][j] = 1;
  if (pal[i][j] != -1) return pal[i][j];
  return pal[i][j] = (s[i] == s[j]) && IsPal(i+1,j-1);
}
```

Основная часть программы. Читаем входную строку.

```
gets(s); n = strlen(s);
memset(dp,0,sizeof(dp));
memset(pal,-1,sizeof(pal));
```

Переберем все пары (*i*, *i* + *len*) по возрастанию длин интервалов.

```
for(len = 1; len < n; len++)
for(i = 0; i + len < n; i++)
{
  j = i + len;
```

Подстрока $s_i \ldots s_j$ содержит более одного символа. Если она палиндром, то она и суперпалиндром.

```
  if (IsPal(i,j))
  {
    dp[i][j] = 1;
    continue;
  }
```

Если $s_i \ldots s_k$ и $s_{k+1} \ldots s_j$ суперпалиндромы, то $s_i \ldots s_j$ является суперпалиндромом.

```
  for(k = i + 1; k < j - 1; k++)
    if(dp[i][k] && dp[k + 1][j])
    {
      dp[i][j] = 1;
      break;
    }
}
```

Подсчитываем количество суперпалиндромов.

```
res = 0;
for(i = 0; i < n; i++)
for(j = i+1; j < n; j++)
  res += dp[i][j];
```

Выводим ответ.

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

Реализация алгоритма – рекурсивная
Объявим входную строку s и рабочие массивы.

```
#define MAX 1010
char s[MAX];
int dp[MAX][MAX], pal[MAX][MAX];
```

Реализуем рекурсивную функцию ***IsPal***(i, j), которая возвращает 1, если $s_i \ldots s_j$ является палиндромом. Иначе функция возвращает 0. Подстрока $s_i \ \ldots \ s_j$ есть палиндром, если $s_i = s_j$ и $s_{i+1} \ldots s_{j-1}$ является палиндромом. Значения IsPal(i, j) запоминаем в ячейках pal[i][j].

```
int IsPal(int i, int j)
{
  if (i >= j) return pal[i][j] = 1;
  if (pal[i][j] != -1) return pal[i][j];
  return pal[i][j] = (s[i] == s[j]) && IsPal(i+1,j-1);
}
```

Функция ***f***(i, j) возвращает 1, если $s_i \ldots s_j$ является суперпалиндромом.

```
int f(int i, int j)
{
```

Суперпалиндром должен содержать более одного символа.

```
  if (i == j) return dp[i][j] = 0;
```

Если значение ***f***(i, j) уже вычислено ранее, то возвращаем его значение.

```
  if (dp[i][j] != -1) return dp[i][j];
```

Если подстрока $s_i \ldots s_j$ ($i < j$) является палиндромом, то она является и суперпалиндромом.

```
  if (IsPal(i,j)) return dp[i][j] = 1;
```

Если $s_i \ldots s_k$ ($i < k$) палиндром, а $s_{k+1} \ldots s_j$ ($k + 1 < j$) суперпалиндром, то $s_i \ldots s_j$ является суперпалиндромом.

```
  for(int k = i + 1; k < j - 1; k++)
    if(IsPal(i,k) && f(k + 1,j)) return dp[i][j] = 1;
```

Если ни одно из выше описанных условий не выполняется, то $s_i \ldots s_j$ не является суперпалиндромом.

```
  return dp[i][j] = 0;
}
```

Основная часть программы. Читаем входную строку *s*. Инициализируем массивы dp и pal.

```
gets(s); n = strlen(s);
memset(dp,-1,sizeof(dp));
memset(pal,-1,sizeof(pal));
```

В переменой *res* подсчитываем количество суперпалиндромов.

```
res = 0;
for(i = 0; i < n; i++)
for(j = i + 1; j < n; j++)
  res += f(i,j);
```

Выводим ответ.

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

873. Палиндромы

Непустая строка, содержащая некоторое слово, называется **палиндромом**, если это слово одинаково читается как слева направо, так и справа налево.

Имеется слово *s*, состоящее из *n* прописных букв латинского алфавита. Вычёркиванием из этого слова некоторого набора символов можно получить палиндром. Найти количество способов вычёркивания из данного слова некоторого (возможно, пустого) набора символов таких, что полученная в результате строка являлась палиндромом. Способы, различающиеся порядком вычёркивания символов, считаются одинаковыми.

Вход. Одно слово *s* длины *n* ($1 \le n \le 60$).

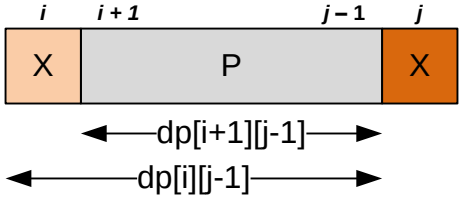
Выход. Вывести одно целое число – количество способов вычёркивания.

Пример входа	**Пример выхода**
BAOBAB	22

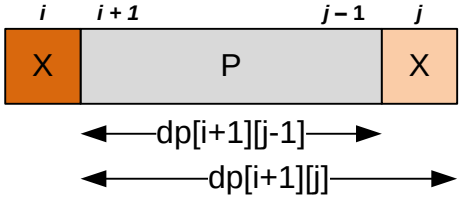
Пусть dp[i][j] хранит количество палиндромов, которое можно получить из подстроки $s_i \ldots s_j$ удалением букв. Тогда dp[i][i] = 1, так как слово из одного символа является палиндромом.

Пусть $s_i = s_j$ = X, подстрока $s_i \ldots s_j$ имеет вид XPX. Здесь через P обозначена подстрока $s_{i+1} \ldots s_{j-1}$. Разобьем палиндромы строки XPX на непересекающиеся группы:

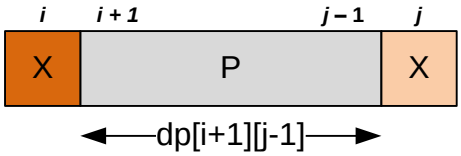
- включающие левый символ X и не включающие правый символ X. Их число равно количеству палиндромов в строке XP минус количество палиндромов в P, то есть dp[i][j – 1] – dp[i + 1][j – 1];



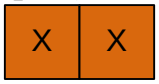
- включающие правый символ X и не включающие левый символ X. Их число равно количеству палиндромов в строке PX минус количество палиндромов в P, то есть dp[i + 1][j] – dp[i + 1][j – 1];



- палиндромы строки P. Их количество равно dp[i + 1][j – 1]. Однако с каждым палиндромом Q строки P мы можем построить и палиндром XQX. То есть палиндромов типа XQX также будет dp[i + 1][j – 1].



- палиндром XX (один палиндром).



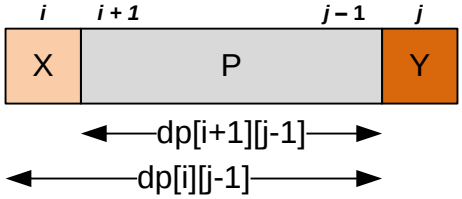
Общее количество палиндромов для случая $s_i = s_j$ равно

(dp[i][j – 1] – dp[i + 1][j – 1]) +
(dp[i + 1][j] – dp[i + 1][j – 1]) +
2 * dp[i + 1][j – 1] +
1

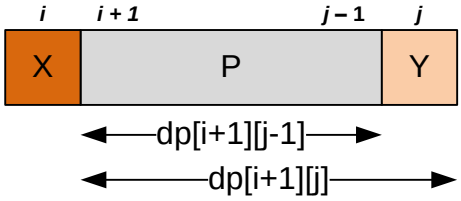
= dp[i][j – 1] + dp[i + 1][j] + 1.

Пусть $s_i \neq s_j$, подстрока $s_i \ldots s_j$ имеет вид XPY (s_i = X, s_j = Y). Разобьем палиндромы строки XPY на непересекающиеся группы:

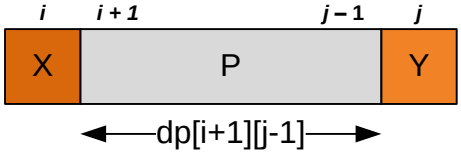
- включающие символ X и не включающие символ Y. Их число равно количеству палиндромов в строке XP минус количество палиндромов в P, то есть dp[i][j – 1] – dp[i + 1][j – 1];



- включающие символ Y и не включающие символ X. Их число равно количеству палиндромов в строке PY минус количество палиндромов в P, то есть dp[i + 1][j] – dp[i + 1][j – 1];



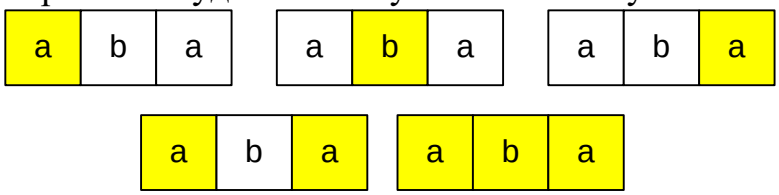
- палиндромы строки P. Их количество равно dp[i + 1][j – 1].



Общее количество палиндромов для случая $s_i \neq s_j$ равно

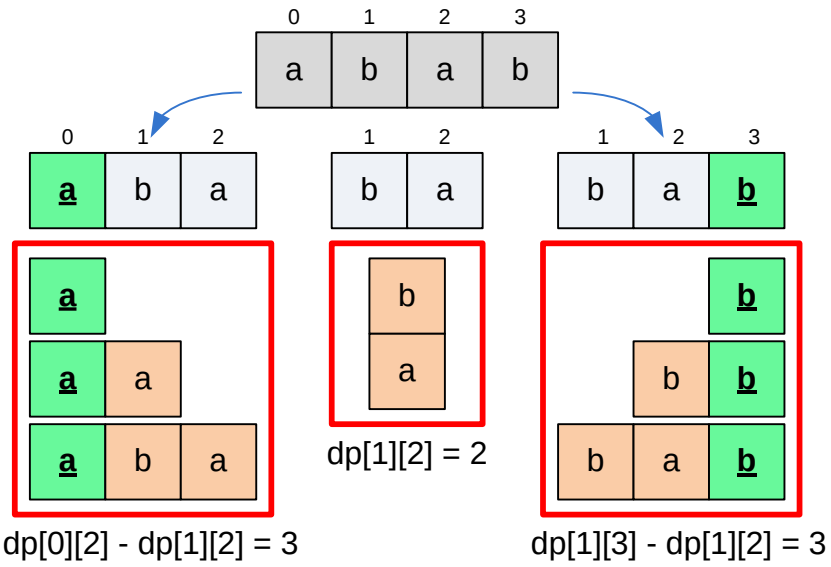
(dp[i][j – 1] – dp[i + 1][j – 1]) +
(dp[i + 1][j] – dp[i + 1][j – 1]) +
dp[i + 1][j – 1]
= dp[i][j – 1] + dp[i + 1][j] – dp[i + 1][j – 1].

Пример. Из строки *aba* удалением букв можно получить 5 палиндромов.



Рассмотрим строку *abab* = $s_0 s_1 s_2 s_3$. Поскольку $s_0 \neq s_3$, то подстрока $s_0 \ldots s_3$ имеет вид XPY. Отсюда dp[0][3] =

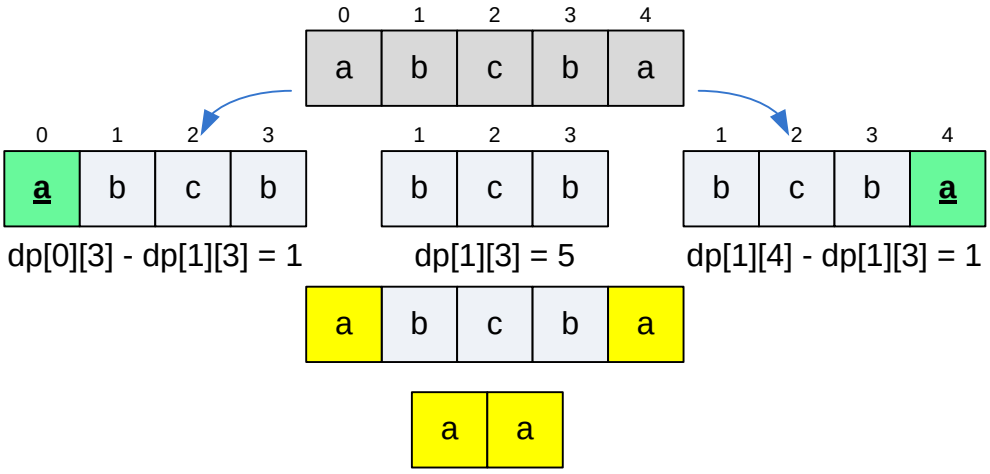
(dp[0][2] – dp[1][2]) +
(dp[1][3] – dp[1][2]) +
dp[1][2] =
= dp[0][2] + dp[1][3] – dp[1][2] = 5 + 5 – 2 = 8.



Рассмотрим строку *abcba* = $s_0s_1s_2s_3s_4$. Поскольку $s_0 = s_4$, то подстрока $s_0 \ldots s_4$ имеет вид XPX. Отсюда dp[0][4] =

(dp[0][3] – dp[1][3]) +
(dp[1][4] – dp[1][3]) +
2 * dp[1][3] +
1 =

= dp[0][3] + dp[1][4] + 1 = 6 + 6 + 1 = 13.



Реализация алгоритма

В массиве *s* храним входную строку. Объявим массив dp.

```
#define MAX 65
char s[MAX];
long long dp[MAX][MAX];
```

Читаем входную строку. Заполняем dp[*i*][*i*] = 1.

```
gets(s); n = strlen(s);
memset(dp,0,sizeof(dp));
for(i = 0; i < n; i++) dp[i][i] = 1;
```

Перебираем длины подстрок *len* и позиции их начала *i*.

```
for(len = 1; len < n; len++)
for(i = 0; i + len < n; i++)
{
  j = i + len;
```

Для каждой такой подстроки $s_i \ldots s_j$ вычисляем значение dp[*i*][*j*] – количество палиндромов, которое можно получить из нее удалением символов. Поскольку подстроки $s_i \ldots s_j$ перебираются в порядке возрастания их длин, то значения dp на всех подотрезках меньшей длины уже вычислены.

```
  if (s[i] == s[j])
    dp[i][j] = dp[i+1][j] + dp[i][j-1] + 1;
  else
    dp[i][j] = dp[i+1][j] + dp[i][j-1] - dp[i+1][j-1];
}
```

Выводим ответ.

```
printf("%lld\n",dp[0][n-1]);
```

Реализация алгоритма – рекурсия

```
#include <stdio.h>
#include <string.h>
#define MAX 61
```

В массиве *s* храним входную строку. Объявим массив dp.

```
char s[MAX];
long long dp[MAX][MAX];
int i, j, len, n;

long long f(int i, int j)
{
```

Если *i* > *j*, то палиндромов не существует.

```
  if (i > j) return 0;
```

Слово из одного символа является палиндромом, установим dp[*i*][*i*] = 1.

```
  if (i == j) return dp[i][j] = 1;
```

Если значение dp[*i*][*j*] уже вычислено, то возвращаем его.

```
  if (dp[i][j] != -1) return dp[i][j];
```

Вычисляем значение dp[*i*][*j*] в зависимости от того, одинаковы или не одинаковы символы s_i и s_j.

```
  if (s[i] == s[j])
    dp[i][j] = f(i + 1, j) + f(i, j - 1) + 1;
  else
    dp[i][j] = f(i + 1, j) + f(i, j - 1) - f(i + 1, j - 1);
  return dp[i][j];
}

int main(void)
{
```

Основная часть программы. Читаем входную строку *s*. Инициализируем массив dp.

```
  gets(s); n = strlen(s);
  memset(dp, -1, sizeof(dp));
```

Выводим ответ.

```
  printf("%lld\n", f(0, n - 1));
  return 0;
}
```

1526. Забор для травы

Возле вилы Пемберлей в южном районе Байтерляндии находится большое пастбище. Миссис Дарси беспокоится за свои нежные растения, которые могут быть вытоптаны незнакомцами. Поэтому она решила на пастбище окружить треугольным забором некоторые участки земли.

В подвале у миссис Дарси есть несколько оград для забора. Для ограничения одной треугольной области она может использовать только три ограды. То есть каждая сторона образованного треугольника представляет собой только одну имеющуюся в наличии ограду. Ограды достаточно красивы, поэтому она решила не склеивать несколько оград для получения одной стороны, а также не разрезать одну ограду на несколько меньших. Задача миссис Дарси - ограничить забором как можно большую площадь пастбища.

Вход. Каждая строка является отдельным тестом. Первое число строки содержит количество оград n ($n \le 16$), имеющихся у миссис Дарси. Следующие n целых чисел в промежутке от 1 до 100 описывают длины этих оград.

Выход. Для каждого теста в отдельной строке вывести максимальную площадь, которую можно оградить имеющимися кусками забора. Ответ следует выводить с 4 десятичными знаками.

Пример входа	**Пример выхода**
7 3 4 5 6 7 8 9	36.7544
4 1 2 4 8	0.0000
4 7 4 4 4	6.9282

Площадь треугольника со сторонами *a*, *b*, *c* ищем в функции area(*a*, *b*, *c*) по формуле Герона:

$$S = \sqrt{p(p-a)(p-b)(p-c)}, \text{ где } p = (a+b+c)/2$$

Пусть P – некоторое подмножество имеющихся кусков забора. Функция FindSol(*mask*) будет находить максимальную площадь, которую можно оградить при помощи них треугольными формами. Переменная *mask* содержит маску подмножества: она является 16-битовым числом, *i*-ый бит которого равен 1, если подмножество P содержит кусок fences[*i*], и 0 иначе. Ответом на задачу в таком случае будет значение FindSol($2^n - 1$), где *n* – количество чисел в массиве fences.

Значения всевозможных масок *mask* лежит в промежутке от 0 до $2^{16} - 1$ (по условию имеются не более 16 кусков забора). В ячейке best[*mask*] храним максимальную площадь, которую можно оградить подмножеством кусков изгороди, описывающим маской *mask*.

Для вычисления FindSol(*mask*) следует перебрать все возможные тройки кусков забора *i*, *j*, *k*, присутствующих в *mask*, проверить для них неравенство треугольника (можно ли из них составить треугольник) и найти сумму area(fences[*i*], fences[*j*], fences[*k*]) + FindSol(*mask* \ {*i*, *j*, *k*}). Перебираем все тройки (*i*, *j*, *k*) и находим такую, для которой указанная сумма максимальна. Ее значение присваиваем ячейке best[*mask*].

$$\text{FindSol}(mask) = \max_{\substack{(i,j,k) \\ 0 \le i < j < k < n}} \left(\text{area}(\text{fences}[i], \text{fences}[j], \text{fences}[k]) + \text{FindSol}(mask \setminus \{i, j, k\})\right)$$

Если изначально длины кусков забора отсортировать, то при проверке неравенства треугольника (стороны которого равны fences[*i*], fences[*j*], fences[*k*]), достаточно проверить только одно условие:

$$\text{fences}[i] + \text{fences}[j] > \text{fences}[k],$$

так как из неравенства fences[*i*] ≤ fences[*j*] ≤ fences[*k*] всегда сдедует, что fences[*i*] + fences[*k*] > fences[*j*] и fences[*j*] + fences[*k*] > fences[*i*].

Реализация алгоритма

Объявим глобальные переменные.

```
#define MAX 16
double best[1<<MAX];
int f[MAX+1];
```

Функция area вычисляет площадь треугольника по формуле Герона.

```
double area(int a, int b, int c)
{
  double p = (a + b + c) / 2.0;
  return sqrt(p * (p-a) * (p-b) * (p-c));
}
```

Функция FindSol возвращает максимальную площадь, которую можно оградить при помощи оград, входящих в маску *mask*, треугольными формами.

```
double FindSol(int mask)
{
  if (best[mask] >= 0.0) return best[mask];
  best[mask] = 0;
  for(int i = 0; i < n - 2; i++) if (mask & (1 << i))
  for(int j = i + 1; j < n - 1; j++) if (mask & (1 << j))
  for(int k = j + 1; k < n; k++) if (mask & (1 << k))
    if (f[i] + f[j] > f[k]) best[mask] = max(best[mask],
        area(f[i],f[j],f[k]) +
        FindSol(mask ^ (1 << i) ^ (1 << j) ^ (1 << k)));
  return best[mask];
}
```

Основная часть программы. После считывания длин оград сортируем их по неубыванию.

```
while(scanf("%d",&n) == 1)
{
  memset(f,0,sizeof(f));
  for(i = 0; i < n; i++) scanf("%d",&f[i]);
  sort(f,f+n);
  for(i = 0; i < (1 << MAX); i++) best[i] = -1.0;
  printf("%.4lf\n",FindSol((1 << n) - 1));
}
```

2302. Орехи для орехов

Райан и Ларри решили, что их орехи не слишком приятны на вкус. Однако имеется несколько орехов, расположенных в некоторых местах острова, и они очень вкусные! Но поскольку ребята ленивые и жадные, то они хотят знать кратчайший путь, по которому можно собрать все орехи. Можете ли Вы им помочь?

Вход. Первая строка каждого теста содержит размеры прямоугольного острова x и y (x, $y \le 20$). Далее следуют x строк по y символов, описывающие карту местности. Она состоит из символов ‘.’, ‘#’ и ‘L’. Изначально Ларри и Райан находятся в ‘L’, орехи обозначаются символами ‘#’. Ребята за один шаг могут попасть в любую из восьми соседних клеток. Количество клеток, в которых могут располагаться орехи, не более 15. Клетка с символом ‘L’ на карте всего одна.

Выход. Для каждой карты в отдельной строке вывести наименьшее количество шагов, за которое стартовав с клетки ‘L’, можно собрать все орехи, и вернуться в ‘L’.

Пример входа

```
5 5
L....
#....
#....
.....
#....
8 10
L.........
..........
.......#..
..........
#....#....
.........#
..........
....#.....
```

Пример выхода

```
8
23
```

Это классическая задача коммивояжера. Следует найти цикл минимальной длины, проходящий по всем вершинам графа. Или то же самое что найти гамильтонов цикл минимальной длины. Задача является NP полной и требует экспоненциального времени для ее решения относительно числа вершин в графе.

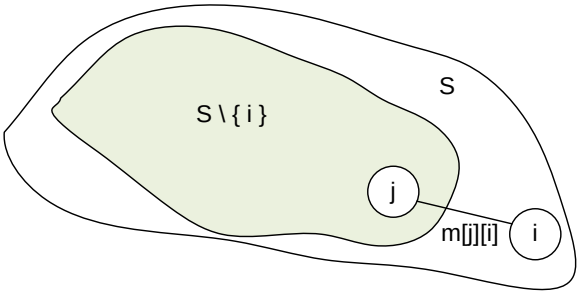
Пусть n – число вершин в графе. Каждому ореху и начальному местоположению Лари поставим в соответствие вершину графа. Из условия задачи следует, что $n \leq 16$. Все вершины графа будут попарно соединены (решается евклидова задача коммивояжера). Длина ребра, соединяющего вершины которым соответствуют координаты (a, b) и (c, d), равна $\max(|a - c|, |b - d|)$. Матрицу смежности построенного графа храним в двумерном массиве m.

Можно сгенерировать при помощи функции ***next_permutation*** все возможные перестановки чисел от 1 до n, каждой перестановке будет соответствовать гамильтонов цикл. Ищем минимальное значение среди всех длин таких циклов. Приведенный алгоритм требует $n!$ времени, что для $n = 16$ слишком много (следует перебрать 16! = 20922789888000 вариантов).

Используя метод динамического программирования, можно решить задачу с оценкой $O(2^n)$ по используемой памяти и $O(n^2 * 2^n)$ по времени работы алгоритма. При $n = 16$ следует перебрать 16777216 вариантов, что реально по времени.

Для непустого подмножества $S \subseteq \{1, 2, \ldots, n\}$ и каждой вершины $i \in S$ определим $dp(S, i)$ как длину кратчайшего пути, начинающегося в первой (начальной) вершине, проходящего по всем вершинам из множества $S \setminus \{i\}$ в произвольном порядке и оканчивающегося в вершине i. Тогда имеют место равенства:

$$dp(\{1, i\}, i) = m[1][i],$$
$$dp(S, i) = \min_{j \in S \setminus \{i\}} (dp(S \setminus \{i\}, j) + m[j, i])$$

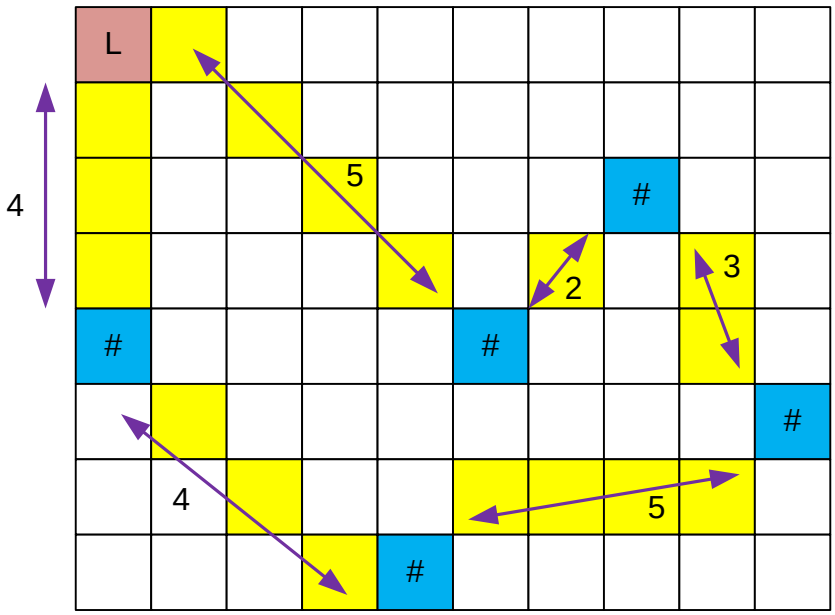


Значения dp(S, *i*) пересчитываем динамически, запоминая их в массиве dp. Гамильтонов цикл минимальной длины равен

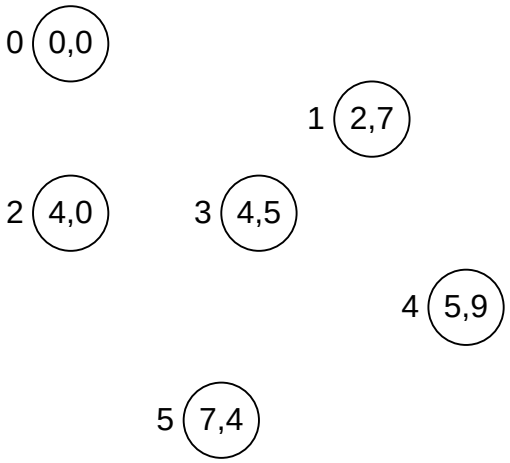
$$\min_{2 \le j \le n} (\mathrm{dp}(\{1, 2, ..., n\}, j) + \mathrm{m}[j, 1])$$

Пример. В первом тесте достаточно пройти вниз острова (4 шага), собрав по пути все орехи, и подняться наверх к исходному месту (еще 4 шага).

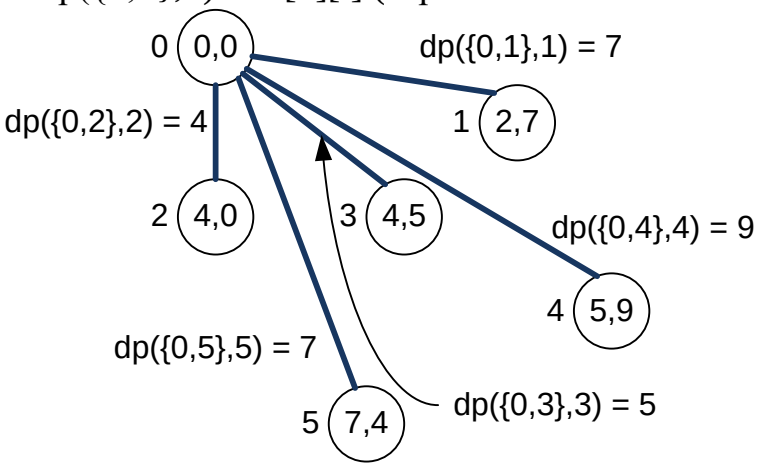
Рассмотрим второй тест. Пять орехов и начальное положение Ларри образуют граф из 6 вершин. На рисунке изображен гамильтонов путь длины 23.



Рассмотрим процесс вычислений более подробно. В каждой вершине указана (*x*, *y*) координата ореха, которому она соответствует. Возле каждой вершины указан ее номер.

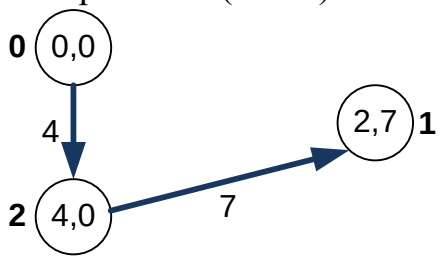


Инициализируем dp({0, *i*}, *i*) = m[0][*i*] (вершина 0 – местоположение Лари):

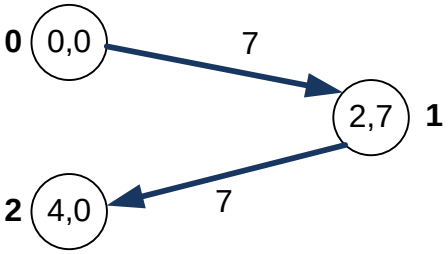


Вершина 0 в маске соответствует младшему биту. Равенство dp({0, *i*}, *i*) = m[0][*i*] эквивалентно dp(2^i + 1, *i*) = m[0][*i*], так как множеству {0, *i*} соответствует маска 2^i + 1.

Рассмотрим гамильтоновы пути по первым трем вершинам, если путь завершается в вершине 1 (слева) или в вершине 2 (справа).



dp({0,1,2},1) = dp(7,1) = 11

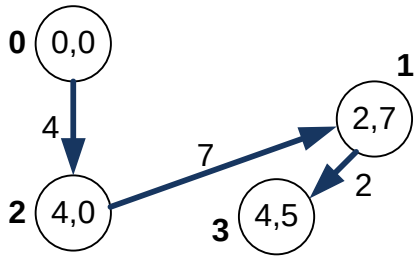


dp({0,1,2},2) = dp(7,2) =14

Найдем минимальную длину гамильтонова пути по первым четырем вершинам, который завершается в вершине 3:

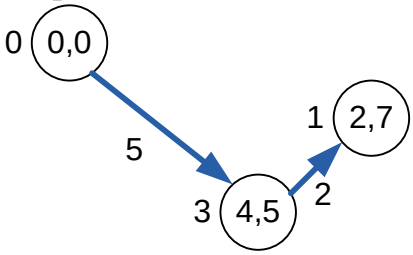
dp({0, 1, 2, 3}, 3) = min(dp({0, 1, 2}, 1) + m[1][3], dp({0, 1, 2}, 2) + m[2][3]) =
min(11 + 2, 14 + 5) = min(13, 19) = 13

Минимум достигается на слагаемом dp({0, 1, 2}, 1) + m[1][3], следовательно наилучшим путем будет

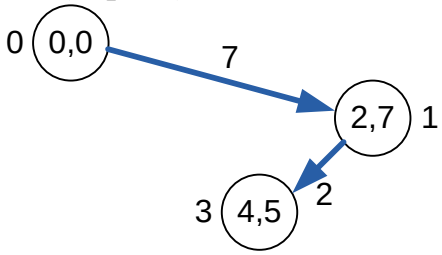


dp({0, 1, 2, 3}, 3) = 13

Рассмотрим гамильтоновы пути по вершинам {0, 1, 3}, если путь завершается в вершине 1 (слева) или в вершине 3 (справа).



dp({0, 1, 3}, 1) = 7

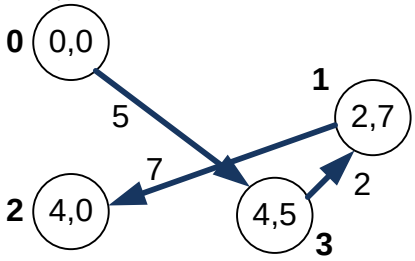


dp({0, 1, 3}, 3) = 9

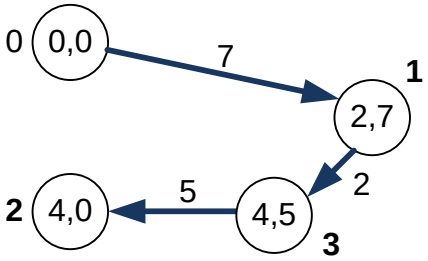
Найдем минимальную длину гамильтонова пути по первым четырем вершинам, который завершается в вершине 2:

dp({0, 1, 2, 3}, 2) = min(dp({0, 1, 3}, 1) + m[1][2], dp({0, 1, 3}, 3) + m[3][2]) = min(7 + 7, 9 + 5) = min(14, 14) = 14

Минимум достигается на обоих слагаемых, следовательно имеется два гамильтоновых пути одной длины:

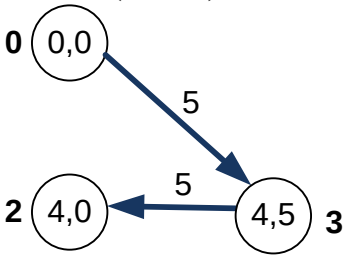


dp({0, 1, 2, 3}, 2) = 14

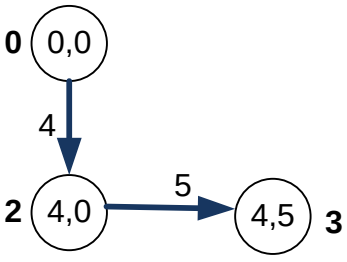


dp({0, 1, 2, 3}, 2) = 14

Рассмотрим гамильтоновы пути по вершинам {0, 2, 3}, если путь завершается в вершине 2 (слева) или в вершине 3 (справа).



dp({0, 2, 3}, 2) = 10



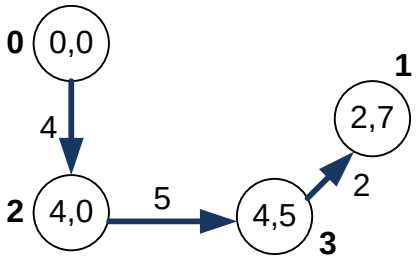
dp({0, 2, 3}, 3) = 9

Найдем минимальную длину гамильтонова пути по первым четырем вершинам, который завершается в вершине 1:

dp({0, 1, 2, 3}, 1) = min(dp({0, 2, 3}, 2) + m[2][1], dp({0, 2, 3}, 3) + m[3][1]) =

min(10 + 7, 9 + 2) = min(14, 11) = 11

Минимум достигается на слагаемом dp({0, 2, 3}, 3) + m[3][1], следовательно наилучшим путем будет



dp({0, 1, 2, 3}, 1) = 11

Найдем минимальную длину гамильтонова пути по первым пяти вершинам, который завершается в вершине 4:

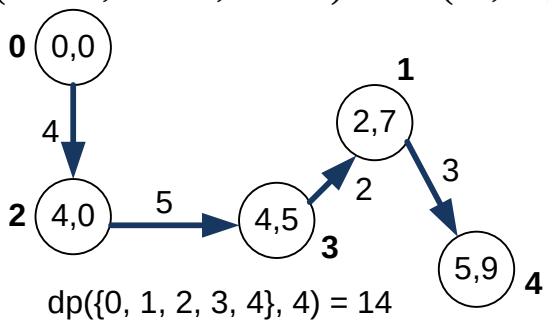
dp({0, 1, 2, 3, 4}, 4) = min(dp({0, 1, 2, 3}, 1) + m[1][4],

dp({0, 1, 2, 3}, 2) + m[2][4],

dp({0, 1, 2, 3}, 3) + m[3][4])

=

min(11 + 3, 14 + 9, 13 + 4) = min(14, 23, 17) = 14



dp({0, 1, 2, 3, 4}, 4) = 14

Реализация алгоритма

Определим переменную INF, условно равную бесконечности, максимально возможное число вершин в графе MAX и массив dp, в котором будем хранить динамически пересчитываемые значения dp(S, *i*). Каждое подмножество S будем хранить в виде числа, в котором *i* - ый бит равен 1, если вершина с номером *i* в нем присутствует, и нулю иначе. Например, при *n* = 5 подмножество {1, 4, 5} кодируется числом $11001_2 = 25$.

```
#define INF 100000000
#define MAX 16
int dp[1<<MAX][MAX+1];
```

Карту острова храним в символьном массиве mas, координаты орехов и начального положения Лари – в массивах *x* и *y*, матрицу смежности графа – в массиве *m*.

```
int x[21], y[21], m[21][21];
char mas[21][21];
```

Функция ***solve*** вычисляет значение dp(S, *last*), где множество S кодируется целым числом *mask*. При этом S \ {*last*} равно *mask* ^ (1<< *last*). Далее перебираем все *i*, *i* ∈ S \ {*last*} и ищем минимальное значение *res* среди dp(S \ {*last*}, *i*) + m[*i*][*last*]. Переменная *res* указывает на ячейку dp[*mask*][*last*], так что с изменением *res* изменяется и само значение dp[*mask*][*last*].

```
int solve(int mask, int last)
{
  int &res = dp[mask][last];
  if(res == INF)
  {
    mask ^= (1 << last);
    for(int i = 1; i < nuts; ++i)
      if(mask & (1 << i)) res = min(res,solve(mask,i) + m[i][last]);
  }
  return res;
}
```

Основная часть программы. Читаем данные острова в массив mas, заносим координаты орехов в массивы x и y. Начальное местоположение Лари сохраняем в (x[0], y[0]).

```
while(scanf("%d %d\n",&xx,&yy) == 2)
{
  for(i = 0; i < xx; i++) gets(mas[i]);

  nuts = 1;
  for(i = 0; i < xx; i++)
  for(j = 0; j < yy; j++)
    if (mas[i][j] == 'L')
    {
      x[0] = i; y[0] = j;
    } else
    if (mas[i][j] == '#')
    {
      x[nuts] = i; y[nuts++] = j;
    }
```

Число вершин графа, равное количеству орехов на острове плюс один (начальное состояние Лари), хранится в переменной *nuts*. Если орехов на острове нет, то выводим 0.

```
  if (nuts == 1)
  {
    printf("0\n"); continue;
  }
```

Строим матрицу смежности графа m. Вычисляем длины ребер.

```
  memset(m,0x3F,sizeof(m));
```

```
for(i = 0; i < nuts - 1; i++)
for(j = i + 1; j < nuts; j++)
  m[i][j] = m[j][i] = max(abs(x[i] - x[j]), abs(y[i] - y[j]));
```

Изначально значения dp(S, *i*) неизвестны, положим их равными плюс бесконечности. Множеству {0} соответствует маска 1, положим dp({0}, 0) = 0 (минимальный путь из нулевой вершины в нее же без посещения других вершин равен нулю). В переменной *total* храним искомую минимальную длину цикла.

```
memset(dp,0x3F,sizeof(dp));
dp[1][0] = 0; total = INF;
```

Положим dp({0, *i*}, *i*) = m[0][*i*] (вершина 0 – местоположение Лари).

```
for(i = 1; i < nuts; i++) dp[1 | (1 << i)][i] = m[0][i];
```

Вычисляем гамильтонов цикл минимальной длины. Значение $2^{nuts} - 1$ соответствует множеству {0, 1, 2, ..., *nuts* – 1}. Вычисляем минимум среди всех значений dp({0, 1, 2, ..., *nuts* – 1}, *i*) + m[*i*][0], $1 \le i < nuts$.

```
for(i = 1; i < nuts; i++)
  total = min(total, solve((1 << nuts) - 1,i) + m[i][0]);
```

Выводим искомую минимальную длину цикла.

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

1128. Проблема Лонги

Лонги хорошо разбирается в математике, он любит задумываться над трудными математическими задачами, которые могут быть решены при помощи некоторых изящных алгоритмов. И вот такая задачка возникла:

Дано целое число n ($1 < n < 2^{31}$), Вы должны вычислить $\sum \gcd(i, n)$ для всех $1 \le i \le n$.

“О, я знаю, я знаю!” – воскликнул Лонги! А знаете ли Вы? Пожалуйста, решите её.

Вход. Каждая строка содержит одно число n.

Выход. Для каждого значения n следует вывести в отдельной строке сумму $\sum \gcd(i, n)$ для всех $1 \le i \le n$.

Пример входа	**Пример выхода**
2 6 12	3 15 40

Теорема. Если функция $f(n)$ мультипликативна, то сумматорная функция $S_f(n) = \sum_{d|n} f(d)$ также мультипликативна.

► Пусть $x, y \in N$, причем x и y взаимно просты. Пусть $x_1, x_2, \ldots, x_k$ – все делители x. Пусть $y_1, y_2, \ldots, y_m$ – все делители y. Тогда НОД(x_i, y_j) = 1, а все возможные произведения $x_i y_j$ задают все делители xy. Тогда

$$S_f(x) * S_f(y) = \sum_{i=1}^{k} f(x_i) * \sum_{j=1}^{m} f(y_j) = \sum_{i,j} f(x_i) f(y_j) = \sum_{i,j} f(x_i y_j) = S_f(xy)$$

Следствие. Рассмотрим функцию f(n) = НОД(n, c), где c – константа. Если x и y взаимно просты, то f($x * y$) = НОД($x * y$, c) = НОД(x, c) * НОД(y, c) = f(x) * f(y). Следовательно функция f(n) = НОД(n, c) мультипликативна.

Пусть $g(n) = \sum_{i=1}^{n} НОД(i, n)$. Тогда

$$g(p_1^{a1} p_2^{a2} \ldots p_k^{ak}) = g(p_1^{a1}) * g(p_2^{a2}) * \ldots * g(p_k^{ak})$$

Теорема. Для любого простого p и натурального a имеет место соотношение:

$$g(p^a) = (a + 1)p^a - ap^{a-1}$$

► При $a = 1$ имеем:

$$g(p) = НОД(1, p) + НОД(2, p) + \ldots + НОД(p, p) = (p - 1) + p = 2p - 1$$

Аналогично при $a = 2$:

$g(p^2)$ =

НОД(1,p²)+	НОД(2,p²)+	. . .	НОД(p,p²)+
НОД(p+1,p²)+	НОД(p+2,p²)+	. . .	НОД(2p,p²)+
НОД(2p+1,p²)+	НОД(2p+2,p²)+	. . .	НОД(3p,p²)+
. . .			
НОД((p-1)p+1,p²)+	НОД((p-1)p+2,p²)+	. . .	НОД(p²,p²)

=

$$= (1 + 1 + \ldots + 1 + p) +$$
$$(1 + 1 + \ldots + 1 + p) +$$
$$\ldots$$
$$(1 + 1 + \ldots + 1 + p^2) =$$

$$= (p - 1 + p) * (p - 1) + (p - 1 + p^2) =$$
$$(2p - 1) * (p - 1) + (p^2 + p - 1) =$$
$$2p^2 - 2p - p + 1 + (p^2 + p - 1) =$$
$$= 3p^2 - 2p$$

Лемма. Если d – делитель n, то существует в точности $\varphi(n/d)$ таких чисел i что НОД(i, n) = d.

► Очевидно что i должно делиться на d, пусть $i = dj$. Тогда

$$НОД(i, n) = НОД(dj, n) = d * НОД(j, n / d)$$

Если последнее выражение равно d, то $НОД(j, n / d) = 1$. Количество таких j что $НОД(j, n / d) = 1$ равно $\varphi(n/d)$.

Пример. Количество таких чисел i что $НОД(i, 24) = 3$ равно $\varphi(8) = 4$.

$НОД(j, 8) = 1$ при $j \in \{1, 3, 5, 7\}$, следовательно $НОД(i, 24) = 3$ при $i \in \{3, 9, 15, 21\}$ (у нас $i = 3j$).

Теорема.

$$g(n) = \sum_{i=1}^{n} НОД(i,n) = n\sum_{d|n} \frac{\varphi(d)}{d}$$

► Согласно выше приведенной лемме количество пар (i, n), для которых $НОД(i, n) = e$, имеется в точности $\varphi(n/e)$. Положив $n / e = d$, получим:

$$g(n) = \sum_{e|n} e\varphi\left(\frac{n}{e}\right) = \sum_{d|n} \frac{n}{d}\varphi(d) = n\sum_{d|n} \frac{\varphi(d)}{d}$$

Пример. Пусть $n = 6$.

i	1	2	3	4	5	6
НОД(i,6)	1	2	3	2	1	6

Тогда $g(6) = \sum_{i=1}^{6} НОД(i,6) =$

$= НОД(1, 6) + НОД(2, 6) + НОД(3, 6) + НОД(4, 6) + НОД(5, 6) + НОД(6, 6) =$

$= 1 + 2 + 3 + 2 + 1 + 6 = 15$

В то же время $g(6) = g(2) * g(3) =$

$(НОД(1, 2) + НОД(2, 2)) * (НОД(1, 3) + НОД(2, 3) + НОД(3, 3)) =$

$(1 + 2) * (1 + 1 + 3) = 3 * 5 = 15$

Вычислим $g(6)$ по формуле $g(n) = n\sum_{d|n} \frac{\varphi(d)}{d}$:

$$g(6) = 6\sum_{d|6} \frac{\varphi(d)}{d} = 6 \cdot \left(\frac{\varphi(1)}{1} + \frac{\varphi(2)}{2} + \frac{\varphi(3)}{3} + \frac{\varphi(6)}{6}\right) =$$

$$= 6\varphi(1) + 3\varphi(2) + 2\varphi(3) + \varphi(6) = 6 + 3 + 4 + 2 = 15$$

Вычислим $g(6)$ исходя из мультипликативности функции $f(x) = НОД(x, n)$:

$g(6) = g(2) * g(3) = (2*2 – 1) * (2*3 – 1) = 3 * 5 = 15$

Пример. Пусть $n = 12$. Тогда $g(12) = \sum_{i=1}^{12} НОД(i,12) =$

$$1 + 2 + 3 + 4 + 1 + 6 + 1 + 4 + 3 + 2 + 1 + 12 = 40$$

i	1	2	3	4	5	6	7	8	9	10	11	12
НОД(i,12)	1	2	3	4	1	6	1	4	3	2	1	12

В то же время g(12) = g(4) * g(3) =

$$(НОД(1, 4) + НОД(2, 4) + НОД(3, 4) + НОД(4, 4)) *$$
$$* (НОД(1, 3) + НОД(2, 3) + НОД(3, 3)) =$$
$$(1 + 2 + 1 + 4) * (1 + 1 + 3) = 8 * 5 = 40$$

Вычислим g(12) по формуле $g(n) = n\sum_{d|n} \frac{\varphi(d)}{d}$:

$$g(12) = 12\sum_{d|12} \frac{\varphi(d)}{d} = 12 \cdot \left(\frac{\varphi(1)}{1} + \frac{\varphi(2)}{2} + \frac{\varphi(3)}{3} + + \frac{\varphi(4)}{4} + \frac{\varphi(6)}{6} + \frac{\varphi(12)}{12} \right) =$$
$$= 12\varphi(1) + 6\varphi(2) + 4\varphi(3) + 3\varphi(4) + 2\varphi(6) + \varphi(12) =$$
$$= 12 + 6 + 8 + 6 + 4 + 4 = 40$$

Делителями числа 12 являются: 1, 2, 3, 4, 6, 12. Количество таких i что НОД(i, 12) = d равно $\varphi(12/d)$. Например НОД(i, 12) = 3 имеет место для $\varphi(12/3) = \varphi(4) = 2$ различных i, а именно для i = 3, 9.

Вычислим g(12) исходя из мультипликативности функции f(x) = НОД(x, n):

$$g(12) = g(2^2) * g(3) = (3 * 2^2 – 2 * 2) * (2*3 – 1) = 8 * 5 = 40$$

Реализация алгоритма

Функция ***euler*** вычисляет функцию Эйлера.

```
long long euler(long long n)
{
  long long i, result = n;
  for (i = 2; i * i <= n;i++)
  {
    if (n % i == 0) result -= result / i;
    while (n % i == 0) n /= i;
  }
  if (n > 1) result -= result / n;
  return result;
}
```

Основная часть программы. Читаем значение n. Вычислим значение g(n) по формуле $\sum_{e|n} e\varphi\left(\frac{n}{e}\right)$. Все делители n ищем среди чисел от 1 до $\lfloor\sqrt{n}\rfloor$. Если i – делитель n, то n / i также будет делителем n. Поэтому с каждым найденным делителем $i \le \lfloor\sqrt{n}\rfloor$ к результату *res* следует прибавить $i\varphi\left(\frac{n}{i}\right)+\frac{n}{i}\varphi(i)$. Если n является полным квадратом, а $i = sq = \lfloor\sqrt{n}\rfloor$, то $i\varphi\left(\frac{n}{i}\right)=\frac{n}{i}\varphi(i)$ и в сумму *res* добавится два одинаковых слагаемых. Поэтому одно из них вычтем из *res* еще на этапе инициализации этой переменной.

```
while(scanf("%lld",&n) == 1)
{
  sq = (long long)sqrt(1.0*n);
  res = (sq * sq == n) ? -sq * euler(sq) : 0;
  for(i = 1; i <= sq; i++)
    if(n % i == 0) res = res + i * euler(n/i) + (n / i) * euler(i);
  printf("%lld\n",res);
}
```

Реализация с учетом мультипликативности

```
#include <stdio.h>
#include <math.h>

long long i, n, res, a, p;

int main(void)
{
  while(scanf("%lld",&n) == 1)
  {
    res = 1;
    for(i = 2; i * i <= n; i++)
    {
      if(n % i == 0)
      {
        a = 0; p = 1;
        while(n % i == 0)
        {
          a++;
          p *= i;
          n /= i;
        }
        res *= (a + 1) * p - a * p / i;
      }
    }
    if (n > 1) res *= (2*n - 1);
    printf("%lld\n",res);
  }
  return 0;
}
```

4107. Экстремум Эйлера

По заданному значению n следует найти значение H, которое задается следующим кодом:

```
H = 0;
for (i = 1; i <= n; i++) {
    for (j = 1; j <= n; j++) {
        H = H + totient(i) * totient(j);
    }
}
```

Функция Эйлера $\varphi(n)$ или totient(n) является арифметической функцией, равной количеству натуральных чисел, меньших или равных n, взаимно простых с n. То есть если n натуральное число, то $\varphi(n)$ количество таких k в промежутке $1 \le k \le n$ что НОД(n, k) = 1.

Вход. Первая строка содержит количество тестов t ($0 < t \le 10^6$). Каждая из следующих t строк содержит одно число n ($0 < n \le 10^4$).

Выход. Для каждого теста вывести в отдельной строке значение H для соответствующего значения n.

Пример входа	**Пример выхода**
2 3 10	16 1024

Перепишем указанную сумму H следующим образом:
$\varphi(1) * \varphi(1) + \varphi(1) * \varphi(2) + \dots \varphi(1) * \varphi(n) +$
$\varphi(2) * \varphi(1) + \varphi(2) * \varphi(2) + \dots \varphi(2) * \varphi(n) +$
. . .
$\varphi(n) * \varphi(1) + \varphi(n) * \varphi(2) + \dots \varphi(n) * \varphi(n) =$

$\varphi(1) * (\varphi(1) + \varphi(2) + \dots \varphi(n)) +$
$\varphi(2) * (\varphi(1) + \varphi(2) + \dots \varphi(n)) +$
. . .
$\varphi(n) * (\varphi(1) + \varphi(2) + \dots \varphi(n)) =$

$$= (\varphi(1) + \varphi(2) + \dots \varphi(n))^2$$

Реализуем решето, которое вычислит все значения функции Эйлера от 1 до 10^4 и занесет их в массив fi. Заполним массив частичных сумм sum[i] = $\varphi(1) + \varphi(2) + \dots \varphi(i)$. Далее для каждого входного значения n выведем sum[n] * sum[n].

Пример. Рассмотрим состояние массива значений функции Эйлера fi и массива частичных сумм sum:

i	1	2	3	4	5	6	7	8	9	10
φ(*i*)	1	1	2	2	4	2	6	4	6	4
sum(*i*)	1	2	4	6	10	12	18	22	28	32

Для $n = 10$ ответ равен

$$(\varphi(1) + \varphi(2) + \dots \varphi(10))^2 = \text{sum}[10]^2 = 32^2 = 1024$$

Реализация алгоритма

Объявим рабочие массивы fi и sum, где

- fi[*i*] = φ(*i*);
- sum[*i*] = φ(1) + φ(2) + … φ(*i*)

```
#define MAX 10001
long long fi[MAX], sum[MAX];
```

Функция ***FillEuler*** заполняет массив fi[*i*] значениями функции Эйлера: fi[*i*] = φ(*i*), $1 \le i <$ MAX.

```
void FillEuler(void)
{
  int i, j;
  for (i = 0; i < MAX; i++) fi[i] = i;
  for (i = 2; i < MAX; i++)
    if (fi[i] == i)
      for (j = i; j < MAX; j += i)
        fi[j] -= fi[j] / i;
```

Вычисление частичных сумм sum[*i*].

```
  sum[0] = 0; sum[1] = 1;
  for(i = 2; i < MAX; i++)
    sum[i] = sum[i-1] + fi[i];
}
```

Основная часть программы. Заполняем массивы fi и sum.

```
FillEuler();
```

Обрабатываем *tests* тестов.

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

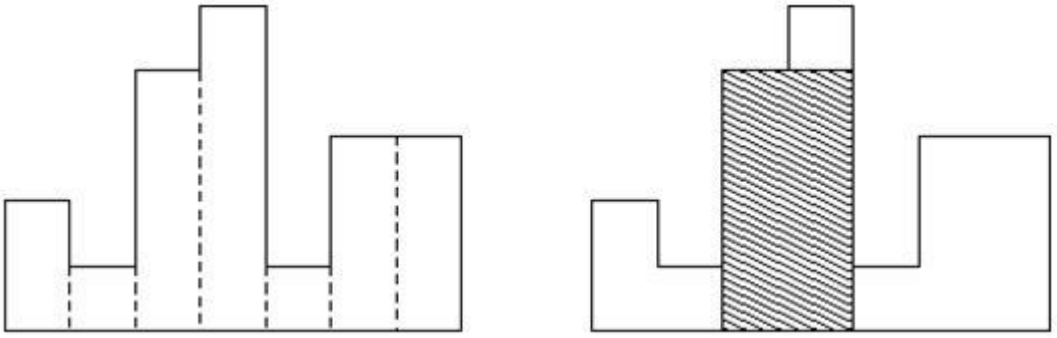
Для каждого входного значения *n* выводим

$$\text{sum}[n] * \text{sum}[n] = (\varphi(1) + \varphi(2) + \dots \varphi(n))^2$$

```
  res = sum[n] * sum[n];
  printf("%lld\n",res);
}
```

383. Гистограмма

Гистограмма является многоугольником, сформированным из последовательности прямоугольников, выровненных на общей базовой линии. Прямоугольники имеют равную ширину, но могут иметь различные высоты. Например, фигура слева показывает гистограмму, которая состоит из прямоугольников с высотами 2, 1, 4, 5, 1, 3, 3. Все прямоугольники на этом рисунке имеют ширину, равную 1.



Обычно гистограммы используются для представления дискретных распределений, например, частоты символов в текстах. Отметьте, что порядок прямоугольников очень важен. Вычислите область самого большого прямоугольника в гистограмме, который также находится на общей базовой линии. На рисунке справа заштрихованная фигура является самым большим выровненным прямоугольником на изображенной гистограмме.

Вход. В первой строке записано число n ($0 \le n \le 10^6$) – количество прямоугольников гистограммы. Затем следует n целых чисел h_1, ..., h_n ($0 \le h_i \le 10^9$). Эти числа обозначают высоты прямоугольников гистограммы слева направо. Ширина каждого прямоугольника равна 1.

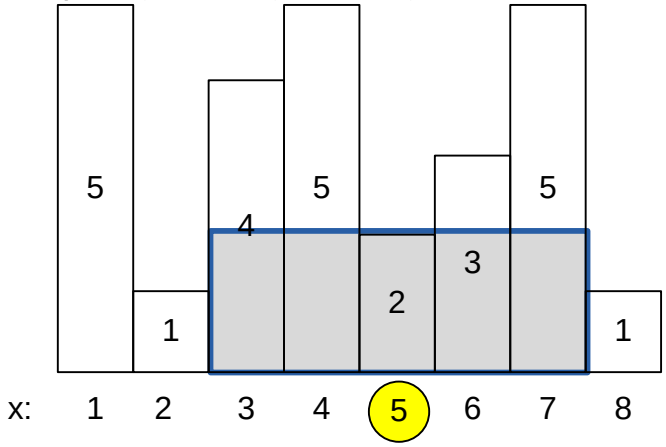
Выход. Вывести площадь самого большого прямоугольника в гистограмме. Помните, что этот прямоугольник должен быть на общей базовой линии.

Пример входа	**Пример выхода**
7 2 1 4 5 1 3 3	8

Реализация за $O(n^2)$. Для каждого i-го прямоугольника ширины 1 постараемся раздвинуть его границы влево *left* и вправо *right* пока это возможно (то есть высоты всех прямоугольников от *left*-го до *right*-го не менее h_i, $1 \le left \le i \le right \le n$). Получим максимально возможный прямоугольник, вписанный в

гистограмму, который упирается в верх i-го прямоугольника. Среди всех таких прямоугольников ищем максимум. Решение дает Time Limit.

Впишем максимальный прямоугольник, упирающийся в верх 5-го прямоугольника. Его границами будут [*left*; *right*] = [3; 7], высота равна 2. Площадь равна (*right* – *left* + 1) * h_5 = (7 – 3 + 1) * 2 = 5 * 2 = 10.



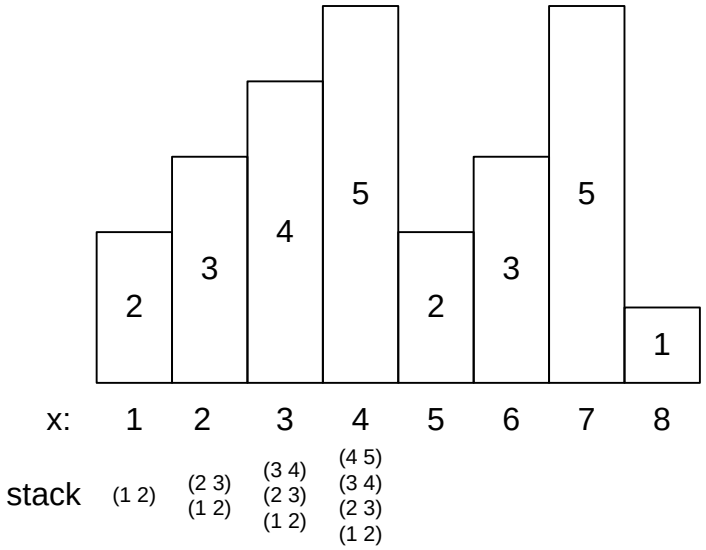
Реализация за O(n). Каждый прямоугольник характеризуется абсциссой i и высотой h_i. Заведем стек из пар (i, h_i) – характеристик прямоугольников. Введем в рассмотрение два дополнительных прямоугольника с высотами h_0 = -1, h_{n+1} = 0. Занесем нулевой прямоугольник с высотой -1 в стек (пару (0, -1) кладем в стек). Такие высоты выбраны для того чтобы:

- нулевой прямоугольник никогда не был извлечен из стека;
- обработка последнего прямоугольника с высотой 0 вытолкнет из стека все имеющиеся там прямоугольники кроме нулевого;

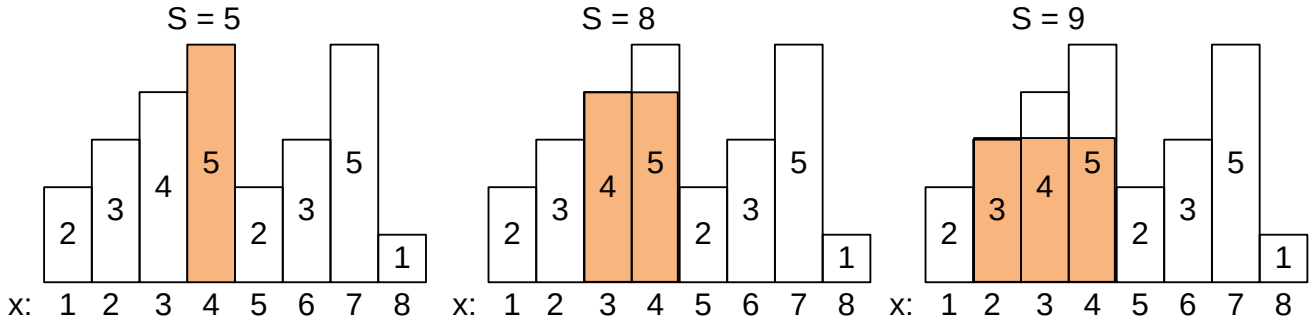
Пусть текущим является i-ый прямоугольник абсциссой i и высотой h_i. Тогда:

- Если его высота больше высоты прямоугольника на вершине стека, то заносим i-ый прямоугольник в стек.
- Пока высота текущего прямоугольника (i, h_i) меньше или равна высоте прямоугольника на вершине стека (x, h_x) (то есть $h_i \le h_x$), то извлекаем прямоугольник из стека и вычисляем площадь максимального вписанного в гистограмму прямоугольника. Подозрительным на максимальный будет прямоугольник со сторонами $i - x$ (он начинается в абсциссе x и заканчивается в абсциссе i – 1) и h_x. Пусть последний вытолкнутый из стека прямоугольник ширины 1 имеет характеристики (j, h_j) ($h_j \ge h_i$). Тогда в стек следует положить пару (j, h_i).

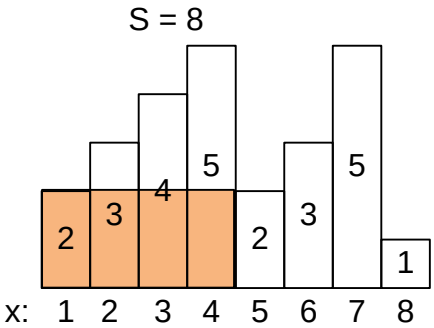
Пример



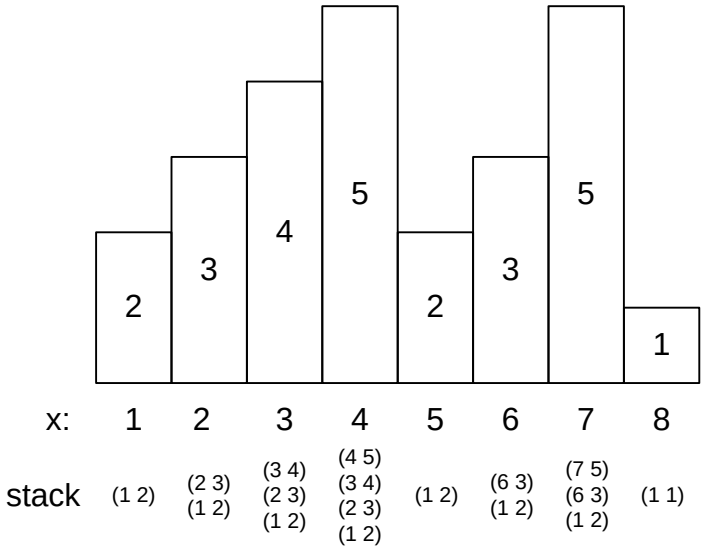
Пусть мы дошли до четвертого прямоугольника включительно. Поскольку до него высоты прямоугольников шли по возрастающей, то они добавлялись в стек. Следующий пятый прямоугольник имеет высоту 2. Последовательно извлекаем прямоугольники из стека, высоты которых строго больше текущего и пересчитываем площади получившихся максимальных прямоугольников:



Пятый прямоугольник имеет высоту 2, извлекаем из стека первый прямоугольник, пересчитываем площадь:



В стеке остался только нулевой прямоугольник с высотой -1. Последний вытолкнутый из стека прямоугольник имеет характеристики $(j, h_j) = (1, 2)$. Текущим рассматриваемым является прямоугольник номер 5 с высотой 2, то есть $(i, h_i) = (5, 2)$. Следовательно в стек заносим прямоугольник с параметрами $(j, h_i) = (1, 2)$. В дальнейшем состояние стека будет следующим:



Реализация алгоритма

В структуре *Node* будем хранить абсциссу *x* и высоту прямоугольника *Height* в этой абсциссе. Объявим стек s из этих структур.

```
struct Node
{
  int x;
  int Height;
  Node(int x, int Height) : x(x), Height(Height) {};
};

stack<Node> s;
```

Читаем входные данные. Считаем высоты нулевого и (n + 1)-го прямоугольника равными соответственно -1 и 0. Занесем в стек нулевой прямоугольник. Поскольку его высота равна -1, то он никогда не будет вытолкнут из стека.

```
scanf("%d",&n);
s.push(Node(0,-1));
```

Последовательно обрабатываем прямоугольники. Площадь максимального покрывающего гистограмму прямоугольника подсчитываем в переменной *res*.

```
res = 0;
for(i = 1; i <= n + 1; i++)
{
```

Читаем высоту *i*-го прямоугольника. Его абсцисса *x* равна *i*. Если $i = n + 1$, то его высота равна нулю: в конце необходимо вытолкнуть все прямоугольники из стека кроме первого с высотой -1 и пересчитать искомую площадь.

```
  if (i <= n) scanf("%d",&h); else h = 0;
  int x = i;
```

```
  while (h <= s.top().Height)
  {
    x = s.top().x; hPrev = s.top().Height; s.pop();
    area = 1LL * hPrev * (i - x);
    if (area > res) res = area;
  }
  s.push(Node(x,h));
}
```

Выводим площадь наибольшего прямоугольника.

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

2888. Sigma-функция на отрезке

Вычислить

$$S(L,R)=\sum_{i=L}^{R}\sigma(i)$$

где σ(n) – сумма натуральных делителей числа n.

Вход. Последовательность из не более чем 10^5 запросов. Каждый запрос задан в отдельной строке и содержит два числа l, r ($1 \le l \le r \le 5 \cdot 10^6$).

Выход. Для каждого запроса вывести в отдельной строке одно число – σ(n).

Пример входа	**Пример выхода**
3 10	83

Запустим решето Эратосфена, которое строит массив lp: lp[i] содержит наименьший делитель числа i.

Если $n = p_1^{a_1} p_2^{a_2} \ldots p_k^{a_k}$, то сумма σ($n$) всех делителей числа n (включая собственный делитель n) равна

$$\sigma(n)=\frac{p_1^{a_1+1}-1}{p_1-1}\cdot\frac{p_2^{a_2+1}-1}{p_2-1}\cdot\ldots\cdot\frac{p_k^{a_k+1}-1}{p_k-1}$$

Функция σ(n) мультипликативная: если a и b взаимно просты, то σ(a*b) = σ(a) * σ(b).

Пусть d = lp[i] – наименьший делитель числа i. Очевидно, что он простой. Пусть k – такая максимальная степень, для которой $i = d^k * x$. Тогда σ(i) = σ(d^k) * σ(x), или σ(i) = $\frac{d^{k+1}-1}{d-1}$ * σ(x).

Пример

$\sigma(9) = 1 + 3 + 9 = 13$, $\sigma(4) = 1 + 2 + 4 = 7$.

Тогда $\sigma(36) = \sigma(2^2 * 3^2) = \sigma(2^2) * \sigma(3^2) = (1 + 2 + 4) * (1 + 3 + 9) = 7 * 13 = 91$.

Реализация алгоритма

В массиве primes будем сохранять простые числа, lp[i] содержит наименьший делитель числа i.

```
#define MAX 5000010
#define LMAX 500010
int lp[MAX], primes[LMAX];
long long s[MAX];
```

Вычисление содержимого массива lp.

```
void LinearEratosfen(void)
{
  int i, j, x;
  long long d;
  s[1] = 1; cnt = 0;
  for (i = 2; i < MAX; ++i)
  {
    if (lp[i] == 0)
    {
      lp[i] = i;
      primes[cnt++] = i;
    }
    for (j = 0; j < cnt && primes[j] <= lp[i] && i * primes[j] < MAX;
         j++)
      lp[i * primes[j]] = primes[j];
```

Если lp[i] = i, то i простое и $\sigma(i) = i + 1$.

```
    if (lp[i] == i) s[i] = i + 1;
    else
    {
      x = i; d = lp[i];
```

Пусть $i = lp[i]^k * x$. Тогда $\sigma(i) = \frac{lp[i]^{k+1} - 1}{lp[i] - 1} * \sigma(x)$.

```
      while(lp[x] == lp[i])
      {
        x /= lp[i]; d *= lp[i];
      }
      s[i] = s[x] * (d - 1) / (lp[i] - 1);
    }
  }
}
```

Основная часть программы. Строим массив суммы делителей s(n) и в нем же вычисляем сумму на его префиксах.

```
LinearEratosfen();
```

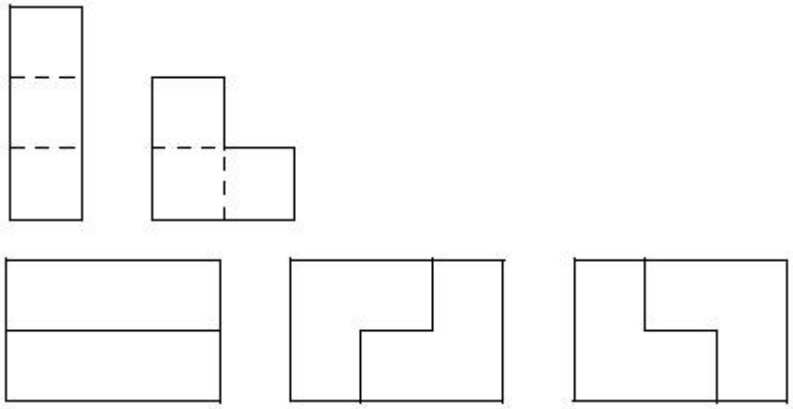
```
for(i = 1; i < MAX; i++)
  s[i] += s[i - 1];
```

$\sum_{n=l}^{r} \sigma(n) = s(r) - s(l-1)$. Читаем и выводим ответ.

```
while(scanf("%d %d", &l, &r) == 2)
  printf("%lld\n", s[r] - s[l - 1]);
```

236. Триомино

Сколькими способами можно замостить прямоугольник $2 \times n$ триоминошками? Триомино – это геометрическая фигура, составленная из трех квадратов, соединяющихся между собой вдоль полного ребра. Есть только две возможных триоминошки:



Например, замостить прямоугольник 2×3 можно только тремя различными способами. Поскольку ответ может быть достаточно большим, искомое количество способов следует вычислять по модулю 10^6.

Вход. Первая строка содержит количество тестов t ($1 \le t \le 100$). Каждая из следующих t строк содержит значение n ($0 < n < 10^9$).

Выход. Для каждого теста в отдельной строке выведите количество способов, которыми можно замостить прямоугольник $2 \times n$. Результат следует выводить по модулю 10^6.

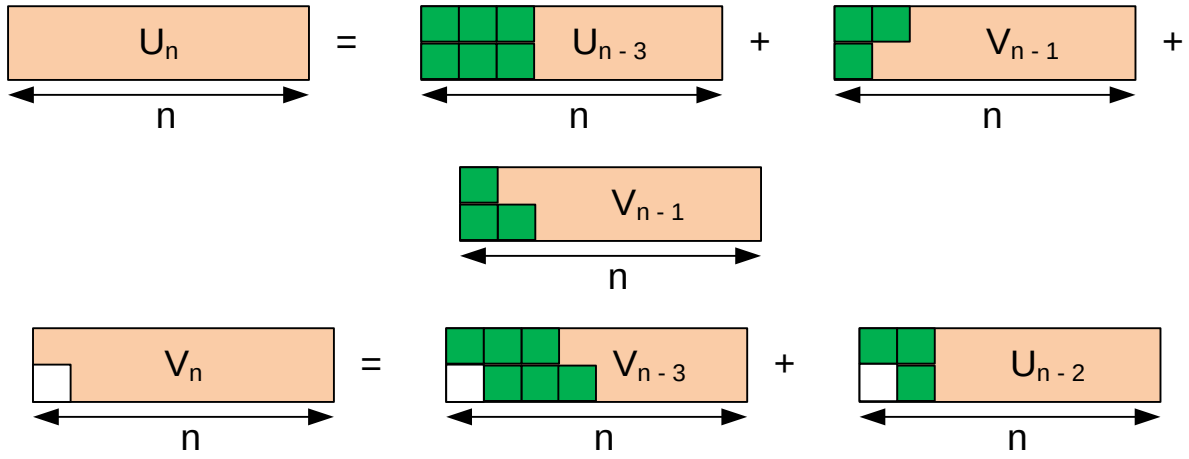
Пример входа	**Пример выхода**
3 3 4 6	3 0 11

Обозначим через U_n количество способов замостить прямоугольник $2 \times n$ с помощью триомино. Обозначим через V_n количество способов замостить прямоугольник $2 \times n$ без угла 1×1 с помощью триомино.

Имеем следующие начальные соотношения:

$$U_1 = 0,\ U_2 = 0,\ U_3 = 3$$
$$V_1 = 0,\ V_2 = 1,\ V_3 = 0$$

Рассмотрим возможные способы замощения прямоугольников U_n и V_n.



Получим следующие рекуррентные соотношения:

$$\begin{cases} U_n = U_{n-3} + 2V_{n-1} \\ V_n = V_{n-3} + U_{n-2} \end{cases}$$

Исходя из начальных условий, найдем U_0 и V_0:

$$\begin{cases} U_3 = U_0 + 2V_2 \\ V_3 = V_0 + U_1 \end{cases}, \begin{cases} 3 = U_0 + 2 \\ 0 = V_0 + 0 \end{cases}, \begin{cases} U_0 = 1 \\ V_0 = 0 \end{cases}$$

Составим таблицу значений для U_n и V_n:

n	0	1	2	3	4	5	6	7	8	9	10	11	12
U_n	1	0	0	3	0	0	11	0	0	41	0	0	153
V_n	0	0	1	0	0	4	0	0	15	0	0	56	0

Сдвинем вторую строку на две позиции влево. Тогда столбцы для индексов, не кратных 3, будут содержать только нули. Удалим их. Перенумеруем новые столбцы. Таблица примет вид:

n	0	1	2	3	4
U_n	1	3	11	41	153
V_n	1	4	15	56	209

Рекуррентные соотношения перепишутся в виде:

$$\begin{cases} U_n = U_{n-1} + 2V_{n-1} \\ V_n = V_{n-1} + U_n \end{cases}, \begin{cases} U_0 = 1 \\ V_0 = 1 \end{cases}$$

Или то же самое что

$$\begin{cases} U_n = U_{n-1} + 2V_{n-1} \\ V_n = U_{n-1} + 3V_{n-1} \end{cases}, \begin{cases} U_0 = 1 \\ V_0 = 1 \end{cases}$$

Рассмотрим матрицу A = $\begin{pmatrix} 1 & 1 \\ 2 & 3 \end{pmatrix}$. $A^2 = \begin{pmatrix} 3 & 4 \\ 8 & 11 \end{pmatrix}$, $A^3 = \begin{pmatrix} 11 & 15 \\ 30 & 41 \end{pmatrix}$. Можно заметить, что первая строка матрицы A^n содержит значения U_{n-1} и V_{n-1}. Это действительно так, потому что

$$\begin{pmatrix} U_{n-1} & V_{n-1} \\ x & y \end{pmatrix} * \begin{pmatrix} 1 & 1 \\ 2 & 3 \end{pmatrix} = \begin{pmatrix} U_{n-1} + 2V_{n-1} & U_{n-1} + 3V_{n-1} \\ x + 2y & x + 3y \end{pmatrix} = \begin{pmatrix} U_n & V_n \\ x + 2y & x + 3y \end{pmatrix}$$

Для решения задачи (нахождения значения U_n) достаточно вычислить A^{n+1} и вывести ее элемент в левом верхнем углу. Возведение в степень совершаем за время $O(\log_2 n)$, так как $n < 10^9$. Все вычисления проводим по модулю 10^6.

Реализация алгоритма

Объявим модуль MOD, по которому будем проводить вычисления. Объявим класс двумерной матрицы Matrix.

```
class Matrix
{
public:
  long long a, b, c, d;
  Matrix(long long a = 1, long long b = 0,
         long long c = 0, long long d = 1)
  {
    this->a = a; this->b = b;
    this->c = c; this->d = d;
  }
```

Перегрузим оператор умножения матриц.

```
  Matrix operator* (const Matrix &x)
  {
    Matrix res;
    res.a = (a * x.a + b * x.c) % MOD;
    res.b = (a * x.b + b * x.d) % MOD;
    res.c = (c * x.a + d * x.c) % MOD;
    res.d = (c * x.b + d * x.d) % MOD;
    return res;
  }
```

Перегрузим оператор возведения матрицы в степень *n*.

```
  Matrix operator^ (int n)
  {
    Matrix res, x(*this);
```

```
    while(n > 0)
    {
      if (n & 1) res = res * x;
      n >>= 1;
      x = x * x;
    }
    return res;
  }
};
```

Возводим матрицу $\begin{pmatrix} 1 & 1 \\ 2 & 3 \end{pmatrix}$ в степень n.

```
long long Solve(long long n)
{
  Matrix res(1,1,2,3);
  res = res^n;
  return res.a;
}
```

Основная часть программы. Читаем входные данные. Если значение n не делится на 3, то ответ 0. Иначе делим n на 3 и возводим матрицу $\begin{pmatrix} 1 & 1 \\ 2 & 3 \end{pmatrix}$ в степень $n + 1$.

```
scanf("%d",&tests);
while(tests--)
{
  scanf("%d",&n);
  if (n % 3)
  {
    printf("0\n"); continue;
  }
  n /= 3;
  res = Solve(n+1);
  printf("%d\n",res);
}
```