Февраль 11, 2022

10443. Карабахские села

В Карабахе есть много сел. Некоторые села имеют одинаковые названия. Например, можно встретить много сел под названием “Абдуррахманлы”. Вам дается список названий сел Карабаха и далее следует ответить на запросы. В каждом запросе дается название одного села. Определите количество сел в Карабахе с таким названием.

Для простоты названия сел были заменены на числа.

Вход. В первой строке заданы два числа n ($1 \le n \le 10^5$) и q ($1 \le q \le 10^4$) – соответственно количество сел в Карабахе и количество запросов.

Во второй строке заданы n целых чисел a_i ($0 \le a_i, x \le 10^9$) – названия сел. В каждой из последующих q строк дается одно целое число x – название села, количество которых следует определить.

Выход. На каждый запрос в отдельной строке необходимо вывести одно число – количество сел в Карабахе с данным названием.

Пример входа	Пример выхода
9 4 2 8 1 5 2 8 4 1 8 1 8 3 2	2 3 0 2

Рассмотрим структуру map<int, int> m, где m[*x*] содержит количество сел с названием *x*. Прочитаем села в Карабахе, построим отображение m. Далее для каждого села *x* выведем число раз m[*x*], которое оно встречается.

Реализация алгоритма

Объявим структуру данных m.

```
map<int, int> m;
```

Читаем входные данные. Заносим села Карабаха в отображение m.

```
scanf("%d %d", &n, &q);
for (i = 0; i < n; i++)
{
  scanf("%d", &x);
  m[x]++;
}
```

Обрабатываем *q* запросов. Для каждого села *x* выведем число раз m[*x*], которое оно встречается.

```
for (i = 0; i < q; i++)
{
  scanf("%d", &x);
  printf("%d\n", m[x]);
}
```

1403. Арифметика Балу

Это случилось в то время, когда медведь Балу обучал Маугли Закону Джунглей. Большой и важный бурый медведь радовался способностям ученика, потому что волчата обычно выучивают из Закона Джунглей только то, что нужно их Стае и племени. Но Маугли, как детенышу человека, нужно было знать гораздо больше.

На занятиях по арифметике Балу придумал следующую игру. Надо было из числа 1 получить число *n*, при этом разрешалось текущее число либо умножить на три, либо к текущему числу прибавить 4. За каждое умножение Балу давал пять тумаков, а за каждое сложение 2 тумака. Например,

$$1 \xrightarrow{+4} 5 \xrightarrow{+4} 9 \xrightarrow{+4} 13 \xrightarrow{+4} 17 \xrightarrow{+4} 21$$

$$1 \xrightarrow{\cdot 3} 3 \xrightarrow{+4} 7 \xrightarrow{\cdot 3} 21$$

в первом случае получишь десять тумаков, во втором – двенадцать.

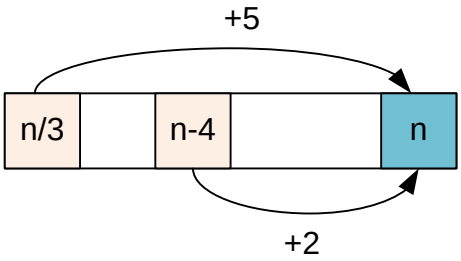
Маугли естественно лучше всех освоил арифметику и быстро придумал, как решить задачу, получив наименьшее количество тумаков. Он также заметил, что не всегда можно выполнить задание хитрого медведя…

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

Выход. Вывести минимальное количество тумаков, которые можно получить за решение задачи. Если решить задачу невозможно, то вывести число 0.

Пример входа	**Пример выхода**
21	10

Обозначим через f(n) минимальное количество тумаков, которые можно получить за решение задачи.



Тогда:

- f(n) = min (f(n / 3) + 5, f(n – 4) + 2), если n делится на 3.
- f(n) = f(n – 4) + 2 , если n не делится на 3.

Например, для $n \le 10^7$ вычислим и запомним значения функции f(n) в линейном массиве m. Для больших значений n запоминания проведем в структуре *map*.

Реализация алгоритма

Объявим структуры для запоминания значений функции f(n).

```
#define MAX 10000000
int m[MAX];
map<int, int> mp;
```

Рекурсивное вычисление с запоминанием функции f(n).

```
int f(int n)
{
  if (n < MAX) return m[n];
  if (mp[n] > 0) return mp[n];
  if (n % 3 == 0)
    mp[n] = f(n/3) + 5;
  else
    mp[n] = f(n-4) + 2;
  return mp[n];
}
```

Основная часть программы. Положим m[i] = - ∞, если значение i Маугли получить не сможет никакими действиями. Вычислим f(i) для i до 10^7 и запомним их в m[i].

```
m[0] = -MAX; m[1] = 0; m[2] = -MAX; m[3] = 5; m[4] = -MAX; m[5] = 2;
for(i = 6; i < MAX; i++)
```

```
  if(i % 3 == 0)
    m[i] = min(m[i / 3] + 5, m[i - 4] + 2);
  else
    m[i] = m[i - 4] + 2;
```

Читаем входное значение *n*. Вычисляем и выводим ответ.

```
scanf("%d",&n);
res = f(n);
```

Если *res* окажется меньше 0, то решить задачу невозможно, выводим число 0.

```
if (res < 0) res = 0;
printf("%d\n",res);
```

10048. Обернуть граф

Задан ориентированный граф с *n* вершинами и *m* ребрами. Вершины графа пронумерованы от 1 до *n*. Найдите наименьшее количество ребер, которое следует обернуть, чтобы существовал хотя бы один путь от вершины 1 до вершины *n*.

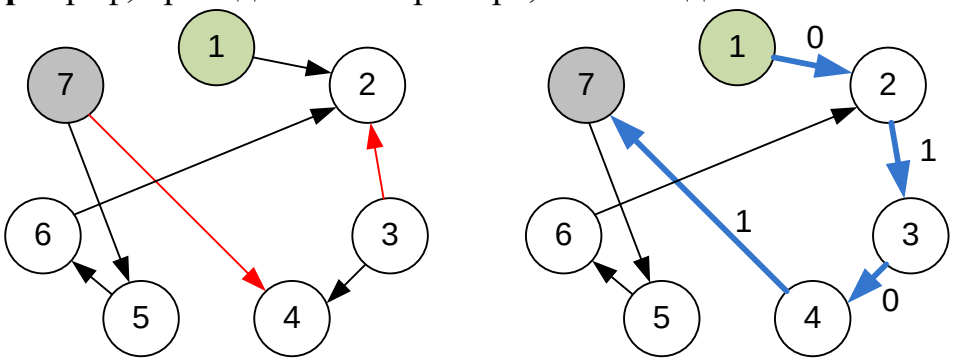
Вход. Первая строка содержит два целых числа *n* и *m* ($1 \le n, m \le 10^5$) – количество вершин и ребер. *i*-ая строка из следующих *m* строк содержит два целых числа x_i и y_i ($1 \le x_i, y_i \le n$), означающих что i-ое ориентированное ребро идет от вершины x_i до вершины y_i.

Выход. Выведите наименьшее количество ребер, которое следует обернуть. Если невозможно получить ни одного пути от 1 до *n*, выведите -1.

Пример входа	**Пример выхода**
7 7 1 2 3 2 3 4 7 4 6 2 5 6 7 5	2

Построим 0-1 граф. Имеющимся ребрам присвоим вес 0, а им обратным 1. Запустим поиск в ширину. Величина кратчайшего пути из вершины 1 в вершину *n* равна наименьшему количеству ребер, которое следует обернуть.

Пример. Граф, приведенный в примере, имеет вид:



Реализация алгоритма

Определим константу бесконечность.

```
#define INF 0x3F3F3F3F
```

Объявим массив кратчайших расстояний *dist* и список смежности графа *g*. Для каждого ребра вместе со смежной вершиной храним вес ребра (0 или 1).

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

Функция ***bfs*** запускает поиск в ширину из вершины *start*.

```
void bfs(int start)
{
  dist = vector<int>(n + 1, INF);
  dist[start] = 0;

  deque<int> q;
  q.push_back(start);

  while (!q.empty())
  {
    int v = q.front(); q.pop_front();
    for (int i = 0; i < g[v].size(); i++)
    {
      int to = g[v][i].first;
      int w = g[v][i].second;

      if (dist[to] > dist[v] + w)
      {
        dist[to] = dist[v] + w;
```

Если вес ребра 1, то новую вершину кладем в конец очереди. Если вес 0, то в начало.

```
        if (w == 1)
          q.push_back(to);
        else
          q.push_front(to);
      }
    }
```

```
  }
}
```

Основная часть программы. Читаем входной граф.

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

Ориентированному ребру $a \rightarrow b$ присваиваем вес 0, обратному ребру присваиваем вес 1.

```
  g[a].push_back(make_pair(b, 0));
  g[b].push_back(make_pair(a, 1));
}
```

Запускаем поиск в ширину из вершины 1.

```
bfs(1);
```

Выводим ответ – кратчайшее расстояние до вершины n.

```
if (dist[n] == INF)
  printf("-1\n");
else
  printf("%d\n", dist[n]);
```

10654. Уникальный цвет

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

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

Найдите все хорошие вершины.

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

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

Пример входа 1

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

Пример выхода 1

```
1
2
3
4
6
```

Пример входа 2

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

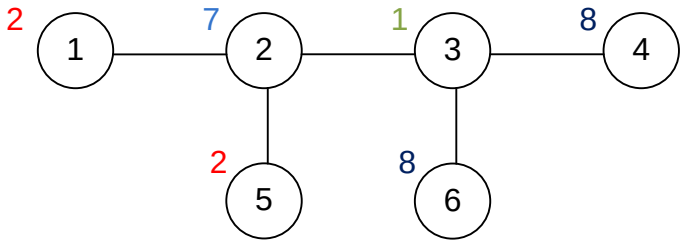
Пример выхода 2

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

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

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

Пример. Граф из первого примера имеет вид:



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

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

Реализация алгоритма

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

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

```
set<int> st;
```

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

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

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

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

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

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

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

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

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

Основная часть программы. Читаем входные данные.

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

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

Запускаем поиск в глубину из вершины 1.

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

Выводим хорошие вершины.

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

10332. Посадка травы

Наступило время для фермера Джона сажать траву на всех своих полях. Вся ферма состоит из n полей, пронумерованных 1 ... n и соединенных n – 1 двунаправленными дорогами таким образом, что с каждого поля можно достичь любое другое поле используя некоторый набор путей.

Фермер Джон может сажать разные типы травы на каждом поле, однако он хочет свести к минимуму количество используемых видов травы в целом, поскольку чем больше видов травы он использует, тем больше затрат он несет.

К сожалению, его коровы стали относиться к выбору травы на ферме довольно снобистски. Если один и тот же тип травы посажен на двух соседних полях (напрямую соединенных дорогой) или даже на двух почти соседних полях (которые напрямую связаны с общим полем дорогами), то коровы будут жаловаться на отсутствие разнообразия в их питательном рационе. Последнее что нужно фермеру Джону, так это жалобы от коров, учитывая, сколько вреда они, как известно, причиняют, будучи недовольны.

Помогите фермеру Джону определить минимальное количество видов травы, которое ему нужно для всей фермы.

Вход. Первая строка содержит число n ($1 \le n \le 10^5$). Каждая из оставшихся n – 1 строк описывает дорогу и содержит два поля, которые она соединяет.

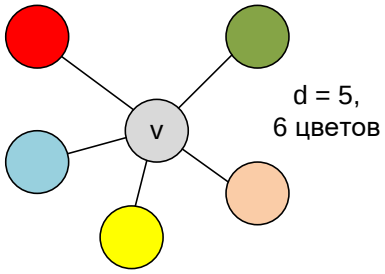
Выход. Выведите минимальное количество видов травы, которое нужно использовать фермеру Джону.

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

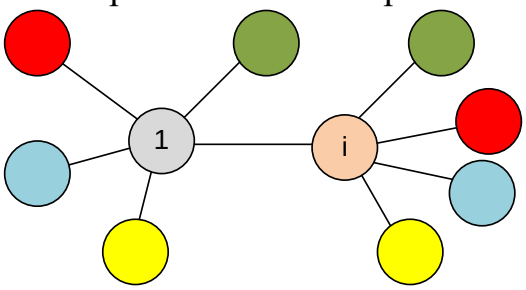
Вершины дерева следует раскрасить наименьшим количеством красок так, чтобы один цвет не имели:

- две вершины, соединенные ребром;
- две вершины на расстоянии двух ребер;

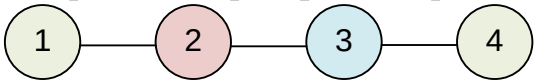
Пусть некоторая вершина v имеет степень d. Это значит, что из вершины v исходит d ребер. Тогда нам необходимо как минимум $d + 1$ цвет для покраски вершины v и всех ее смежных вершин.



Пусть максимальная степень вершины в графе равна *d*. Покажем, что существует требуемая раскраска из *d* + 1 цветов. Возьмем вершину 1, покрасим ее цветом 1. Пусть она имеет *d* соседей, покрасим их в цвета 2, 3, …, *d* + 1. Возьмем вершину *i* – соседа 1, покрашенную в цвет *i*. Пусть она также имеет *d* соседей. Вершина *i* имеет цвет *i*, один из ее соседей покрашен в цвет 1. Остальных соседей вершины *i* можно красить цветами 2, 3, …, *i* – 1, *i* + 1, …, *d* + 1. Поскольку структура – дерево, то таким образом можно покрасить все вершины.



Пример. Граф из примера можно раскрасить тремя цветами:



Реализация алгоритма

Объявим массив для подсчета степеней вершин графа. Степень вершины *i* будем хранить в d[*i*].

```
int d[100000];
```

Читаем входные данные. Подсчитываем степени вершин графа.

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

В переменной *res* находим максимальную степень вершины.

```
res = 0;
for (i = 1; i <= n; i++)
  if (d[i] > res) res = d[i];
```

Выводим ответ.

```
printf("%d\n", res + 1);
```

1599. Динамическая лягушка

В связи с расширением использования пестицидов, местные ручьи и реки оказались настолько загрязненными, что стали почти невозможными для жизни водных животных.

Лягушка Фред находится на левом берегу такой реки. n камней расположено на прямой линии, простирающейся с левого берега на правый. Расстояние между левым и правым берегом d метров. Камни могут быть двух размеров: крупные могут выдержать любой вес, но мелкие начинают тонуть, как только на них окажется тело любой массы. Фреду нужно перейти на правый берег, где он должен собрать подарки и вернуться обратно на левый берег в свой дом.

Фрейд может приземлиться на каждый маленький камень не более чем один раз. Крупные может использовать столько раз, сколько пожелает. В воду он прыгнуть не может, так она чрезвычайно загрязнена.

Можете ли вы спланировать маршрут так, чтобы минимизировать максимальное расстояние одного прыжка?

Вход. Первая строка содержит количество тестов t ($t < 100$). Каждый тест начинается двумя целыми числами n ($0 \le n \le 100$) и d ($1 \le d \le 10^9$). Следующая строка описывает n камней. Каждый камень задается в виде s-m. Симол s укзывает на тип камня – Большой (B) или Маленький (S), а m ($0 < m < d$) определяет расстояние от этого камня до левого берега. Камни задаются в порядке возрастания значений m.

Выход. Для каждого теста вывести его номер и минимально возможное значение наибольшего прыжка.

Пример входа 1

```
3
1 10
B-5
1 10
S-5
2 10
B-3 S-6
```

Пример выхода 1

```
Case 1: 5
Case 2: 10
Case 3: 7
```

Пример входа 2

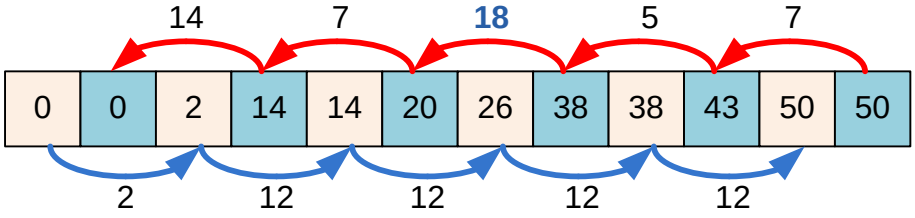
```
1
6 50
S-2 B-14 S-20 S-26 B-38 S-43
```

Пример выхода 2

```
Case 1: 18
```

Очевидно, что на обратном пути лягушка может использовать все имеющиеся у нее на пути камни. Нам необходимо разработать стратегию передвижения лягушки с левого берега на правый. Будем считать, что левый и правый берег являются большими камнями, и изначально лягушка находится на самом левом камне. Теперь каждый большой камень заменим на два маленьких, находящихся в одном и том же месте. Это можно сделать, так как очевидно, что лягушка будет использовать любой большой камень не более двух раз. Поскольку теперь у нас имеется последовательность только маленьких камней, то сформулируем алгоритм движения лягушки: при движении с левого на правый берег она должна каждый раз перепрыгивать через один камень – в этом и состоит принцип жадного подхода.

Пример. Рассмотрим второй тест. Переправа содержит $n = 6$ камней, расстояние между берегами $d = 50$. Левый и правый берег представляем большими камнями. Массив и движение лягушки по нем имеет следующий вид.



Реализация алгоритма

Читаем входные данные.

```
scanf("%d\n",&tests);
for(t = 1; t <= tests; t++)
{
  scanf("%d %d\n",&n,&d);
  memset(m,-1,sizeof(m));
```

Левый берег представляем одним большим камнем, который заменяем на два маленьких.

```
  m[0] = m[1] = 0;
```

Считываем информацию про камни и заполняем массив m. Каждый большой камень заносим в массив дважды, маленький – только один раз.

```
  for(ptr = 2, i = 0; i < n; i++)
  {
    do {scanf("%c",&letter);} while (letter == ' ');
    scanf("-%d",&s);
    if (letter == 'B') {m[ptr] = m[ptr+1] = s; ptr += 2;}
    else {m[ptr] = s; ptr++;}
  }
```

Правый берег представляем одним большим камнем, который заменяем на два маленьких.

```
    m[ptr] = m[ptr+1] = d;
    ptr++;

    scanf("\n");
```

Движемся с самого левого камня до самого правого, перепрыгивая через один. Ищем максимум разностей между i-ым и $(i + 2)$-ым камнями.

```
    for(dist = 0, i = 2; i < ptr; i += 2)
      if (m[i] - m[i-2] > dist) dist = m[i] - m[i-2];
```

Уменьшаем значение i на 1. Теперь движемся справа налево по соседним камням, которые имеют нечетные номера (камни с четными номерами утонули при движении лягушки на правый берег).

```
    for(i--; i >= 2; i -= 2)
      if (m[i] - m[i-2] > dist) dist = m[i] - m[i-2];
```

Выводим ответ.

```
    printf("Case %d: %d\n",t,dist);
}
```

9596. Текстовый процессор

Бесси работает над эссе. Поскольку пишет она некрасиво, она решила набрать эссе в текстовом процессоре.

Эссе содержит n слов, разделённых пробелами. Каждое слово имеет длину от 1 до 15 символов включительно, и состоит только из больших или маленьких латинских букв. В соответствии с правилами, эссе должно быть отформатировано специфическим образом: каждая строк должна содержать не более k символов, не считая пробелы. К счастью, текстовый процессор Бесси может выполнять это требование при использовании следующей стратегии:

- Если Бесси пишет слово которое может поместиться на текущей строке, оно помещается в эту строку.
- Иначе надо переместить слово в следующую строку и продолжить пополнение этой следующей строки.

Конечно, последовательные слова в одной строке должны быть разделены ровно одним пробелом. Не должно быть пробелов в конце любой строки.

К несчастью, текстовый процессор Бесси сломался, помогите ей отформатировать её эссе в соответствии с вышеописанными правилами.

Вход. Первая строка содержит два целых числа n ($1 \le n \le 100$) и k ($1 \le k \le 80$). Следующая строка содержит n слов, разделённых одиночными пробелами. Никакое слово не будет длиннее, чем k символов – максимальное количество символов в одной строке.

Выход. Выведите корректно отформатированное эссе Бесси.

Пример входа	**Пример выхода**
10 7 hello my name is Bessie and this is my essay	hello my name is Bessie and this is my essay

Последовательно читаем входные слова. Текущее слово выводим в текущей строке, если суммарная длина выводимых в строке слов не превышает *k*. Если при добавлении слова к текущей строке длина строки становится больше *k*, то слово выводим в новой строке.

Реализация алгоритма

Входные слова читаем в массив *s*.

```
char s[100];
```

Читаем входные данные.

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

Переменная *ptr* хранит текущую длину выводимой строки.

```
ptr = 0;
```

Последовательно читаем *n* слов.

```
while (n--)
{
  scanf("%s", s);
```

Вычисляем длину слова *s*.

```
  len = strlen(s);
```

Если $ptr + len \leq k$, то слово *s* можно вывести в текущей строке. Длина выводимой строки не превысит *k* символов.

```
  if (ptr + len <= k)
  {
    ptr += len;
    printf("%s ", s);
  }
```

Иначе слово *s* следует вывести в новой строке. Длину *ptr* новой текущей строки положим равной *len*.

```
    else
    {
      ptr = len;
      printf("\n%s ", s);
    }
  }
```

5103. Коза ностра

Пока у школьников идет зачет, преподаватели играют в мафию. В кругу сидит n преподавателей. Ведущий должен раздать кому-то из них карты с тузами (тузов любое количество, возможно 0) – эти преподаватели будут мафией. Однако никакие два мафиози не должны сидеть рядом.

Сколько способов раздать карты есть у ведущего? (Два способа считаются различными, если есть хотя бы один преподаватель, который является мафией в одном случае, но не является в другом).

Вход. Количество преподавателей n ($1 \le n \le 30$), которые сидят в кругу.

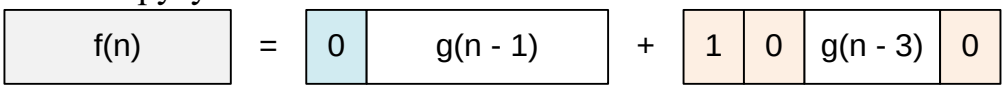
Выход. Выведите одно число – количество способов раздать карты.

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

Пусть g(n) равно количеству способов раздать карты n преподавателям, которые выстроены в ряд (первый не стоит рядом с последним). Тогда задача эквивалентна нахождению количества последовательностей длины n из 0 и 1, где никакие две единицы не стоят рядом. Ее решением будет число Фибоначчи, заданное рекуррентностью:

$$g(n) = \begin{cases} 2, if\ n = 1 \\ 3, if\ n = 2 \\ f(n-1) + f(n-2) \end{cases}$$

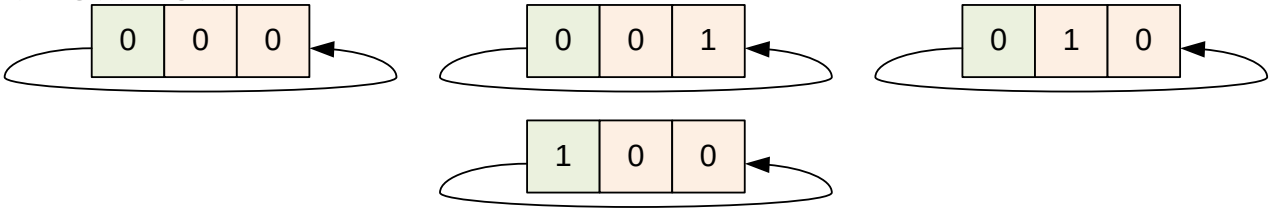
Пусть f(n) равно количеству способов раздать карты n преподавателям, расположенным по кругу.



Если первому преподавателю туз не дали, то следующим n – 1 преподавателям тузы можно раздать g(n – 1) способами. Если первому преподавателю туз дали, то второму и последнему преподавателям тузов выдавать не следует. Оставшимся n – 3 преподавателям тузы можно раздать g(n – 3) способами. Имеем соотношение:

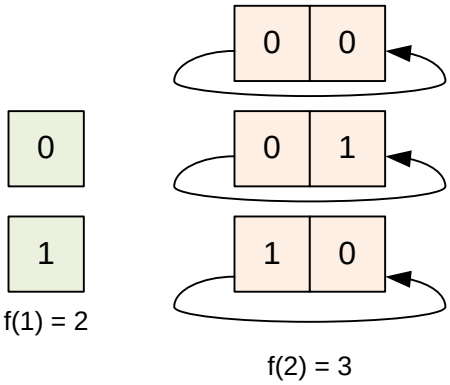
$$f(n) = g(n-1) + g(n-3), \text{ если } n \ge 3$$

При n = 3 нам потребуется значение g(0), которое можно вычислить из равенства g(0) + g(1) = g(2), откуда g(0) = g(2) – g(1) = 3 – 2 = 1. Следовательно f(3) = g(2) + g(0) = 3 + 1 = 4.



Базовые случаи имеют вид:

- f(1) = 2
- f(2) = 3



Реализация алгоритма

Объявим массив для хранения чисел Фибоначчи.

```
#define MAX 46
int fib[MAX];
int n, i, res;
```

Основная часть программы. Вычисляем числа Фибоначчи.

```
fib[0] = 1; fib[1] = 2;
for (int i = 2; i < MAX; i++)
  fib[i] = fib[i - 1] + fib[i - 2];
```

Читаем входное число *n*.

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

Вычисляем ответ *res*.

```
if (n == 1) res = 2; else
if (n == 2) res = 3; else
res = fib[n - 1] + fib[n - 3];
```

Выводим ответ.

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

1618. Наибольшая общая подпоследовательность

Даны две последовательности. Найдите длину их наибольшей общей подпоследовательности. Подпоследовательность – это последовательность, полученная из другой последовательности удалением некоторых элементов без изменения порядка следования оставшихся элементов.

Вход. В первой строке задана длина n первой последовательности ($1 \le n \le 1000$). Во второй строке записаны члены первой последовательности – целые числа, не превосходящие по модулю 10000. В третьей строке задана длина второй последовательности m ($1 \le m \le 1000$). В четвертой строке записаны члены второй последовательности – целые числа, не превосходящие по модулю 10000.

Выход. Вывести длину наибольшей общей подпоследовательности, или 0, если такой не существует.

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

Подпоследовательность заданной последовательности – это набор элементов, которые появляются в порядке слева направо, но не обязательно последовательно. Подпоследовательность может быть получена из последовательности только удалением некоторых ее элементов.

Например, рассмотрим последовательность {2, 1, 3, 5}. Тогда

- {1, 5}, {2}, {2, 3, 5} являются подпоследовательностями;
- {5, 1}, {2, 3, 1} не являются подпоследовательностями;

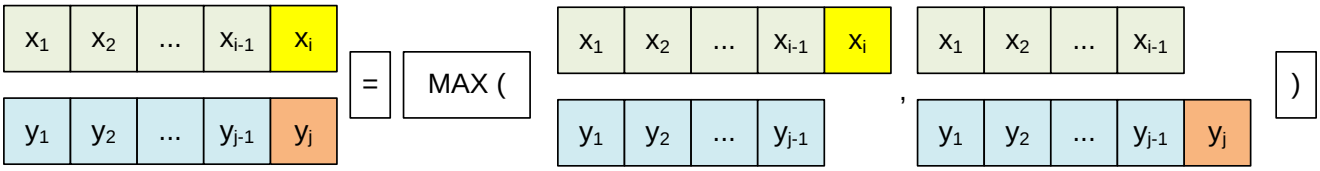
Общая подпоследовательность двух последовательностей – это подпоследовательность, которая встречается в обеих последовательностях. ***Наибольшая общая подпоследовательность*** (НОП) – это общая подпоследовательность наибольшей длины.

Например, наибольшей общей подпоследовательностью {1, 2, 3} и {2, 1, 3, 5} могут быть {1, 3} или {2, 3}. Длина НОП равна 2.

Пусть f(i, j) – длина наибольшей общей подпоследовательности (НОП) последовательностей $x_1x_2\ldots x_i$ и $y_1y_2\ldots y_j$.

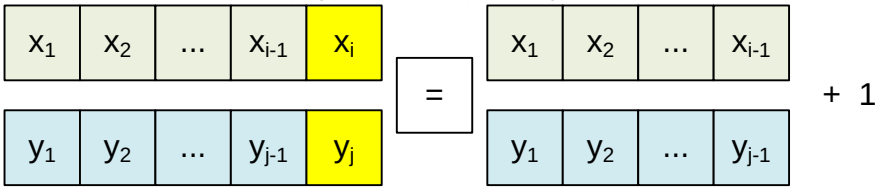
Если $x_i \ne y_j$, то ищем НОП среди $x_1x_2\ldots x_i$ и $y_1y_2\ldots y_{j-1}$, а также среди $x_1x_2\ldots x_{i-1}$ и $y_1y_2\ldots y_j$. Большую из них возвращаем:

$$f(i, j) = \max(f(i, j - 1), f(i - 1, j))$$



Если $x_i = y_j$, то ищем НОП среди $x_1x_2\ldots x_{i-1}$ и $y_1y_2\ldots y_{j-1}$:

$$f(i, j) = 1 + f(i - 1, j - 1)$$

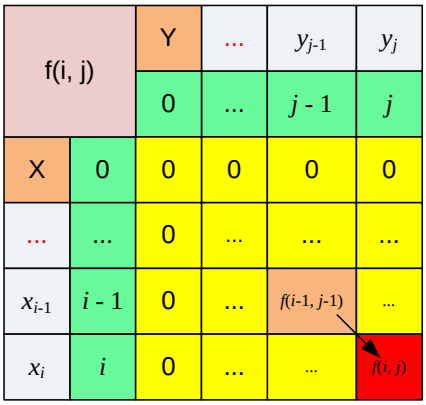


Если одна из последовательностей пустая, то их НОП тоже пустая:

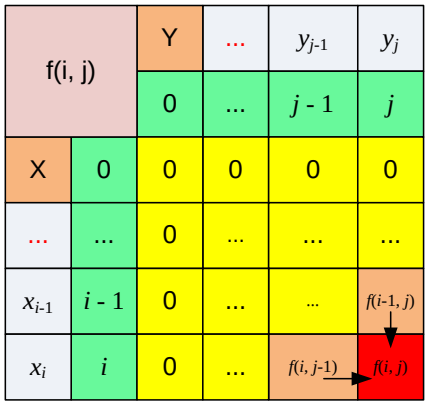
$$f(0, j) = f(i, 0) = 0$$

Подведем итог. Рекуррентное соотношение для нахождения длины НОП имеет вид:

$$f(i, j) = \begin{cases} \max(f(i, j-1), f(i-1, j)), x_i \neq y_j \\ f(i-1, j-1) + 1, x_i = y_j \\ 0, i = 0 \text{ или } j = 0 \end{cases}$$



$x_i = y_j$
$f(i, j) = f(i - 1, j - 1) + 1$



$x_i \neq y_j$
$f(i, j) = \max(f(i, j - 1), f(i - 1, j))$

Значения f(i, j) будем хранить в массиве m[0..1000, 0..1000], где m[0][i] = m[i][0] = 0. Каждая следующая строка массива m[i][j] вычисляется через предыдущую. Поэтому для нахождения ответа достаточно держать в памяти только две строки длины 1000.

Пример. Пусть X = *abcdgh*, Y = *aedfhr*. Наибольшей общей подпоследовательностью будет *adh*, ее длина равна f(6, 6) = 3.

f(i, j)		X	a	b	c	d	g	h
		0	1	2	3	4	5	6
Y	0	0	0	0	0	0	0	0
a	1	0	1(a)	1	1	1	1	1
e	2	0	1	1	1	1	1	1
d	3	0	1	1	1	2(d)	2	2
f	4	0	1	1	1	2	2	2
h	5	0	1	1	1	2	2	3(h)
r	6	0	1	1	1	2	2	3

f(6, 6) = max(f(6, 5), f(5, 6)) = max(2, 3) = 3, так как Y[6] = *r* ≠ *h* = X[6].
f(5, 6) = 1 + f(4, 5) = 1 + 2 = 3, так как Y[5] = *h* = X[6].

Упражнение. Заполните следующую таблицу:

f(i, j)		X	d	f	c	a	b	a
		0	1	2	3	4	5	6
Y	0							
f	1							
d	2							
c	3							
c	4							
a	5							
a	6							

Реализация алгоритма

Массивы x и y содержат входные последовательности, n и m – их длины. Массив m содержит две последние строки динамического преобразования.

```
#define SIZE 1010
int x[SIZE], y[SIZE], mas[2][SIZE];
```

Основная часть программы. Читаем входные последовательности в массивы, начиная с первого индекса. То есть в ячейки x[1..*n*] и y[1..*m*].

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

Обнулим массив mas. Динамически вычисляем значения f(*i*, *j*). Изначально mas[0][*j*] содержит значения f(0, *j*). Далее в mas[1][*j*] заносим значения f(1, *j*). Поскольку для вычисления f(2, *j*) достаточно иметь значения предыдущей строки массива mas, то значения f(2, *j*) можно сохранить в ячейках mas[0][*j*], значения f(3, *j*) – в ячейках mas[1][*j*] и так далее.

```
memset(mas,0,sizeof(mas));
for(i = 1; i <= n; i++)
for(j = 1; j <= m; j++)
  if (x[i] == y[j])
    mas[i%2][j] = 1 + mas[(i+1)%2][j-1];
  else
    mas[i%2][j] = max(mas[(i+1)%2][j],mas[i%2][j-1]);
```

Выводим ответ, который находится в ячейке mas[*n*][*m*]. Первый аргумент берем по модулю 2.

```
printf("%d\n",mas[n%2][m]);
```

Реализация алгоритма – рекурсия

```
#include <stdio.h>
#include <string.h>
#define SIZE 1002

int x[SIZE], y[SIZE], dp[SIZE][SIZE];
int n, m, i, j, res;

int max(int i, int j)
{
  return (i > j) ? i : j;
}

int lcs(int *x, int *y, int m, int n)
{
  if (m == 0 || n == 0)
    return 0;
  if (dp[m][n] != -1) return dp[m][n];

  if (x[m] == y[n])
    return dp[m][n] = 1 + lcs(x, y, m - 1, n - 1);
  else
    return dp[m][n] = max(lcs(x, y, m, n - 1), lcs(x, y, m - 1, n));
}

int main(void)
```

```
{
  scanf("%d", &n);
  for (i = 1; i <= n; i++) scanf("%d", &x[i]);
  scanf("%d", &m);
  for (i = 1; i <= m; i++) scanf("%d", &y[i]);

  memset(dp, -1, sizeof(dp));
  res = lcs(x, y, n, m);
  printf("%d\n", res);
  return 0;
}
```

2585. Прибыль

Коровы открыли новый бизнес, и Фермер Джон хочет видеть, насколько они хорошо его ведут. Бизнес работает n ($1 \le n \le 100000$) дней, и в каждый i-ый день коровы записывают свою чистую прибыль P_i ($-1000 \le P_i \le 1000$).

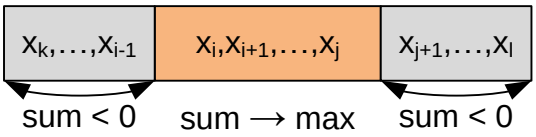
Фермер Джон хочет найти самую большую прибыль, которую получили коровы в течение любого последовательного периода времени (обратите внимание, что последовательный период времени может иметь длину от одного дня до n дней). Помогите ему, написав программу для нахождения величины наибольшей непрерывной прибыли.

Вход. Первая строка содержит целое число n. Каждая из следующих n строк содержит одно целое число P_i.

Выход. Выведите значение максимальной суммы прибыли за любой последовательный период времени.

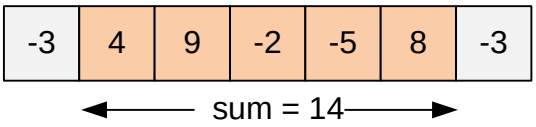
Пример входа	Пример выхода
7 -3 4 9 -2 -5 8 -3	14

В задаче следует найти такую подпоследовательность подряд идущих чисел, которая будет иметь максимально возможную сумму среди всех возможных таких подпоследовательностей. Если на подпоследовательности x_i, x_{i+1}, ..., x_j достигается максимальная сумма, то для любого k, $1 \le k < i$ и l, $j < l \le n$ сумма элементов x_k, ..., x_{i-1} и x_{j+1}, ..., x_l будет отрицательной.



Алгоритм Кадана. Движемся по массиву слева направо и накапливаем в переменной *s* текущую частичную сумму. Если в какой-то момент *s* окажется отрицательной, то присвоим *s* = 0. Максимум из всех значений переменной *s* за время прохода по массиву и будет ответом на задачу.

Пример. Для входной последовательности максимальная прибыль равна 14.



Рассмотрим пример последовательности X. Построим частичные суммы. Текущее значение частичной суммы обнуляем когда сумма становится меньше нуля и начинаем отсчет суммы со следующего числа. Максимальное значение среди всех частичных сумм равно 6, что и является ответом. Искомой подпоследовательностью будет 4, -2, 4.

X	5	-3	1	-7	4	-2	4	-1	-8	2
s	5	2	3	-4 0	4	2	6	5	-3 0	2
				s = 0					s = 0	

Реализация алгоритма

Читаем длину последовательности *n*. Обнуляем значение максимальной прибыли *max* и текущей частичной суммы *s*.

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

Читаем *n* чисел. Каждое прочитанное число *Number* добавляем к текущей сумме *s* и пересчитываем текущее значение прибыли *max*. Если на каком-то шаге значение *s* стало отрицательным, то положим его равным нулю и продолжим процесс.

```
for(i = 0; i < n; i++)
{
  scanf("%d",&Number);
  s += Number;
  if (s > max) max = s;
  if (s < 0) s = 0;
}
```

Выводим результат.

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