

Задача 1. Меценаты

Для решения задачи будет необходимо считать все данные и для каждого дня посчитать сумму, которая будет внесена в фонд. Если подсчитывать сумму с использованием цикла `for`, то будет получен частичный балл.

```
#include <iostream>
#include <vector>

void solve() {
    int n, m;
    unsigned long long ans = 0;
    std::cin >> n >> m;
    for(size_t i=0; i<m; ++i) {
        int a1, a2;
        std::cin >> a1 >> a2;
        int d = a2 - a1;
        for(int i = 0; i < n; ++i) {
            ans += a1;
            a1 += d;
        }
    }
    std::cout << ans;
}

int main() {
    solve();
    return 0;
}
```

Для получения полного балла нужно использовать формулу суммы арифметической прогрессии:

```
from sys import stdin

inp = stdin.readline

def solve():
    n, m = map(int, inp().split())
    s = 0
    for i in range(m):
        a, b = map(int, inp().split())
        d = b - a
        s += (2 * a + d * (n - 1)) * n // 2
    print(s)

solve()
```

При решении на других языках программирования потребуется использовать 64-битный тип данных для хранения ответа.

```
#include <iostream>
#include <vector>

void solve() {
    int n, m;
    unsigned long long ans = 0;
    std::cin >> n >> m;
```

```
for (size_t i=0; i<m; ++i) {
    unsigned long long a1, a2;
    std::cin >> a1 >> a2;
    unsigned long long d = a2 - a1;
    ans += (a1 + a1 + d * (n - 1)) * n / 2;
}
std::cout << ans;
}

int main() {
    solve();
    return 0;
}
```

Задача 2. XXX

Для полного решения задачи необходимо рассмотреть различные случаи, которые могут возникнуть на доске. Для этого имеет смысл написать отдельную функцию проверки того, что игрок выиграл по состоянию игрового поля - в листинге ниже такая функция называется **check** - в ней проверяется, что ход в поле (i, j) возможен и если при этом игрок победит (наберет 3 по одной из вертикалей, горизонталей или диагоналей), то результатом будет **True** иначе **False**.

Также, важно определить кто ходит первым: если ходов сделано нечетное количество, значит первым ходил Ваня, если ходов сделано четное количество - первым ходил Петя.

Алгоритм поиска решения задачи описан ниже в функции **solve**: проверим каждую клетку, в которой еще нет пометки. Если походил в нее Петя выигрывает, то выводим ответ.

Если после проверки всех клеток не нашлось подходящей для победы Пети, то пройдем еще раз по всем клеткам и найдем количество клеток, из которых может победить Ваня. Возможны случаи:

- таких клеток нет, то победитель не определится после ходов Пети и Вани - Петя может походить в любую незанятую клетку
- такая клетка ровно одна - значит Пете обязательно нужно походить в эту клетку и тогда победитель не определится после ходов Пети и Вани
- таких клеток больше 1 - в этом случае как бы не ходил Петя - Ваня выиграет на своем ходе - для Пети необходимо указать любой вариант хода

Код программы на языке Python представлен ниже

```
board = [ [0 for j in range(3)] for i in range(3) ]

n = int(input())
m = 1 + n % 2
for t in range(n):
    i, j = map(int, input().split())
    board[i-1][j-1] = m
    m = 3 - m

def check(i, j, m):
    if board[i][j] != 0:
        return False
    board[i][j] = m
    k1 = k2 = kc = kr = 0
    for t in range(3):
        if board[i][t] == m: kr += 1
        if board[t][j] == m: kc += 1
```

```
        if board[t][t] == m: k1 += 1
        if board[t][2-t] == m: k2 += 1
    board[i][j] = 0
    return max(k1, k2, kc, kr) == 3

def solve():
    a = b = 0
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                a, b = i, j
            if check(i, j, 1):
                print('PETYA')
                print(i+1, j+1)
                return

    k = 0
    for i in range(3):
        for j in range(3):
            if check(i, j, 2):
                k += 1
                a, b = i, j

    if k <= 1:
        print('UNKNOWN')
    else:
        print('VANYA')
    print(a+1, b+1)

solve()
```

Задача 3. Укладка плитки

Для решения задачи необходимо применить известный алгоритм двойного поиска по ответу, прочитать про который можно например здесь: <https://ru.algorithmica.org/cs/interactive/binary-search/>.

Пример решения на языке Python

```
from sys import stdin

inp = stdin.readline

def solve():
    m = int(inp())
    k = int(inp())

    left = 0
    right = 10**18 + 1
    while right - left > 1:
        mid = (left + right) // 2
        if (1 + mid) * mid // 2 * k <= m:
            left = mid
        else:
            right = mid
    print(left)
```

```
solve()
```

Решение на других языках потребует использовать 64-битный целый тип

```
#include <iostream>
#include <vector>

void solve() {
    unsigned long long m;
    int k;
    std::cin >> m >> k;
    auto check = [m, k](int n) { return 1LL * n * (1 + n) / 2 * k <= m; };
    int left = 0, right = (int)2e9;
    while (right - left > 1) {
        int mid = left + (right - left) / 2;
        if (check(mid)) {
            left = mid;
        } else {
            right = mid;
        }
    }
    std::cout << left;
}

int main() {
    solve();
    return 0;
}
```

Задача 4. Приключение на 20 минут

Для решения задачи можно написать в цикле реализовать прямое моделирование движения «хасти». Для удобства во всех примерах ниже ввод данных происходит отдельно от функции, выполняющей поиск ответа. В первом примере кода (который позволяет набрать 60 баллов) моделирование ходов выполняется с помощью метода **solutionLoop**

```
def solutionLoop(board, init, dir, k):
    d = dir # current direction
    i, j = init # current position
    n, m = board # board size
    start = k
    dirs = { (1,1):0, (1,-1):1, (-1,1):2, (-1,-1):3 }

    while k > 0:
        if 0 <= i + d[0] < n:
            i += d[0]
        else:
            d[0] = -d[0]
        if 0 <= j + d[1] < m:
            j += d[1]
        else:
            d[1] = -d[1]
        k -= 1
```

```
    return [i, j]

n, m = map(int, input().split())
y, x = map(int, input().split())
dy, dx = map(int, input().split())
k = int(input())
ay, ax = solutionLoop([n, m], [y-1, x-1], [dy, dx], k)
print(ay+1, ax+1)
```

Данный подход можно ускорить: так как клеток не более 1 млн. и в каждую клетку мы можем попасть с одного из 4 направлений, то всего получается 4 млн. вариантов входа в клетку. Заметим, что если на ходе m мы попали в клетку повторно с той же стороны, с которой уже попадали ранее, то значит имеется цикл. Этот цикл можно отбросить столько раз сколько позволяет величина k , а оставшиеся ходы выполнить. Для того, чтобы запоминать момент времени в который мы попали в ту или иную клетку поля создадим список `time`. Пример функции, реализующий данный подход представлен в листинге ниже. Заметим также, что на языке C++ можно было получить максимальный балл даже с решением, не выполняющим сохранения ходов.

```
def solutionFastLoop(boardSize, initPosition, initDirection, k):
    d = initDirection # current direction
    i, j = initPosition # current position
    n, m = boardSize # board size
    start = k
    dirs = { (1,1):0, (1,-1):1, (-1,1):2, (-1,-1):3 }
    time = [ [ [-1] * m for i in range(n) ] for j in range(4)]
    noLoop = True

    while k > 0:
        if noLoop and time[ dirs[tuple(d)] ][i][j] != -1:
            k = start % (start - k)
            noLoop = False
        else:
            time[ dirs[tuple(d)] ][i][j] = k
            if 0 <= i + d[0] < n:
                i += d[0]
            else:
                d[0] = -d[0]
            if 0 <= j + d[1] < m:
                j += d[1]
            else:
                d[1] = -d[1]
            k -= 1

    return [i, j]
```

У этой задачи есть решение с использованием формул, основанное на целочисленных операциях: каждое отражение от стен игрового поля порождает симметричную позицию для координат до отражения относительно границ клетки, в которой происходит соприкосновение со стеной. Для расчета координат конечной клетки достаточно представить, что движение происходит на поле в двойное большего размера (каждое второе отражение компенсирует первое). Пример решения на языке Python ниже

```
def solution(board, pos, d, k):
    y = (pos[0] + d[0] * k) % (2 * board[0])
    x = (pos[1] + d[1] * k) % (2 * board[1])
```

```
if y >= board[0]:
    y = 2 * board[0] - 1 - y
if x >= board[1]:
    x = 2 * board[1] - 1 - x
return [y, x]
```

Заметим также, что все эти решения работают если от 1 нумерации осуществить переход к 0 нумерации (вычесть 1), а при выводе ответа осуществить обратный переход (добавить 1)

```
ay, ax = solution([n, m], [y-1, x-1], [dy, dx], k)
print(ay+1, ax+1)
```

Задача 5. Раз, два, три... бесконечность

Наивная идея реализации - создадим вектор с отметками какой из лифтов уже уехал. Каждый раз, когда очередной постоялец подходит к лифту будем находить ближайший свободный слева и ближайший свободный справа. Если справа ближе, чем слева, то отправляем на лифте справа и ставим отметку. Если ближе элемент слева, то отправляем на лифте слева и отмечаем, что лифт уже использован. На C++ это решение будет эффективно работать с вектором логических элементов и позволяет набрать 60 баллов.

```
#include <iostream>
#include <vector>

void solve() {
    int n;
    std::cin >> n;
    std::vector<bool> f(n+2, false);

    for(int i=0; i<n; ++i) {
        int j;
        std::cin >> j;
        if (!f[j]) {
            f[j] = true;
            std::cout << j << '\n';
        } else {
            int l = j;
            while (f[l]) --l;
            int r = j;
            while (f[r]) ++r;

            if (r == n + 1 || l > 0 && j - l <= r - j) {
                std::cout << l << '\n';
                f[l] = true;
            } else {
                std::cout << r << '\n';
                f[r] = true;
            }
        }
    }
}

int main() {
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(0);
```

```
std::cout.tie(0);  
solve();  
return 0;  
}
```

Для того, чтобы получить полный балл необходимо научиться быстро находить ближайшие свободные значения слева и справа. Для этого заведем два списка: ближайших слева - **prv** и ближайших справа **nxt**. В начале для каждого элемента $prv[i] = i$ и $nxt[i] = i$. Когда происходит выбор элемента, то нужно соответствующим образом переместить $prv[i]$ и $nxt[i]$ - для этого будем использовать рекурсивную функцию **get**, которая будет переходить на следующий элемент до тех пор пока не найдет свободный.

Для того, чтобы лучше понять этот подход, а также понять, почему это работает достаточно быстро рекомендуется прочитать про «эвристику сжатия путей» при реализации структуры данных «Система непересекающихся множеств»: <https://e-maxx.ru/algo/dsu>

Код реализующий данный подход представлен ниже.

```
#include <iostream>  
#include <vector>  
#include <numeric>  
#include <algorithm>  
  
int get(std::vector<int> & a, int i) {  
    if (a[i] != i) {  
        a[i] = get(a, a[i]);  
    }  
    return a[i];  
}  
  
void solve() {  
    int n;  
    std::cin >> n;  
    std::vector<int> prv(n + 2);  
    std::iota(prv.begin(), prv.end(), 0);  
    std::vector<int> nxt(n + 2);  
    std::iota(nxt.begin(), nxt.end(), 0);  
  
    for(int i=0; i<n; ++i) {  
        int x;  
        std::cin >> x;  
        int left = get(prv, x);  
        int right = get(nxt, x);  
        if ((right == n + 1) || (left != 0 and x - left <= right - x)) {  
            x = left;  
        } else {  
            x = right;  
        }  
        std::cout << x << '\n';  
        prv[x] = get(prv, x - 1);  
        nxt[x] = get(nxt, x + 1);  
    }  
}  
  
int main() {  
    std::ios_base::sync_with_stdio(false);
```

```
std::cin.tie(0);  
std::cout.tie(0);  
solve();  
return 0;  
}
```