Разбор задачи «Юный Сальвадор»

Для начала заметим, что достаточно несложно вычислить высоту и ширину законченного рисунка: это будет квадрат со стороной $a + \left\lceil \frac{k}{2} \right\rceil \cdot 2b + \left\lfloor \frac{k}{2} \right\rfloor \cdot 2c$. Действительно, мы начинаем с квадрата со стороной a, а затем каждый ход увеличиваем сторону на $2b$ или $2c$.

Поэтому для некоторых групп тестов (1, 5 и 6) можно было написать решение вида «создать доску известной высоты и ширины, и закрашивать на ней клетки в соответствии с условием, после чего просто посчитать количество клеток каждого цвета». Разумеется, такое решение требует научиться трансформировать положение клеток относительно центра фигуры в положение относительно начала такой таблицы, но это делается достаточно несложными преобразованиями.

При этом в первых двух группах тестов можно было набрать баллы и более простым решением. Заметим, что если ходов $k \leqslant 2$, то длина стороны фигуры будет равна a, затем $a + 2b$, затем $a+2b+2c$. Можно было в явном виде выписать формулу, задающую количество клеток, добавленных на каждом ходу: a^2, $2b \cdot (2a + 2b)$, $2c \cdot (2a + 4b + 2c)$. После оставалось только сложить количество клеток одного цвета.

В группах 5 и 6 работало описанное выше решение, однако в группе 5 проходило и чуть менее реализационно сложное решение: вместо закрашивания «рамок» вокруг текущей фигуры, достаточно было совершать ходы в конца и закрашивать целиком постепенно уменьшающиеся квадраты. В таком случае итоговый рисунок не поменяется, а для реализации такого решения достаточно просто поддерживать сближающиеся индексы верхней, нижней, левой и правой границ текущего квадрата.

Группы тестов 3 и 4 были рассчитаны на аккуратный вывод формулы: в случае с $a = 0$, фигура будет состоять из рамок ширины b или c чередующихся цветов. Если расписать количество клеток в каждой рамке как разность площадей большого и маленького квадратов, то если сторона очередного квадрата равна d_i, то количество клеток на i-м слое будет равно $d_i^2 - d_{i-1}^2$. При вынесении общих множителей,

- в группе 3 получается формула для количества красных клеток

  $$(2b)^2 + (4b+2c)^2 - (2b+2c)^2 + (6b+4c)^2 - (4b+4c)^2 + \ldots = 2b \cdot (2b + (6b+4c) + (10b+8c) + \ldots),$$

  что затем раскрывается по формуле арифметической прогрессии;

- в группе 4 же все еще проще: черные клетки занимают только исходный квадрат со стороной a, а остальные $(a + 2bk)^2 - a^2$ клеток — красные.

Наконец, полное решение предполагало использование того же наблюдения про количество клеток на каждом слое. Будем просто итерироваться по всем ходам, поддерживая текущий цвет, длину очередной рамки и текущую длину стороны фигуры. На каждом ходу достаточно добавить к счетчику клеток текущего цвета разность квадратов новой длины стороны и старой длины стороны фигуры. Таким образом, решение за время $\mathcal{O}(k)$ набирает полный балл.

Также можно было решить всю задачу формулой за $\mathcal{O}(1)$, однако формула сводится к суммированию нескольких арифметических прогрессий, и выглядит достаточно сложно, поэтому в этой задаче от вас не требовалось выводить эту формулу до конца.

Разбор задачи «Патерны Даттона»

Шаги решения

1. Разбиение строки на группы одинаковых символов:

   - Проходим по строке и разбиваем её на последовательности одинаковых символов, сохраняя символ и его длину.

2. Перебор всех возможных троек групп:

   - Перебираем все возможные тройки групп, чтобы найти те, которые удовлетворяют условию $a_1 \neq a_2 \neq a_3$ и имеют соответствующую длину k, i, j.

3. Проверка соответствия длине:
   - Проверяем, совпадают ли длины групп с заданными k, i, j.
4. Подсчёт количества подстрок:
   - Если все условия выполнены, то данная тройка групп составляет допустимую подстроку.

Пошаговый разбор кода на Python

1. Чтение входных данных:

```
n = int(input())

s = input()

left, middle, right = map(int, input().split())
```

   - n — длина строки.
   - s — сама строка.
   - left, middle, right — количества символов для подстрок.
2. Разбиение строки на группы одинаковых символов:

```
a = []
a.append((s[0], 1))
for i in range(1, n):
    if s[i] == a[-1][0]:
        a[-1] = (a[-1][0], a[-1][1] + 1)
    else:
        a.append((s[i], 1))
```

   - Начинаем с первой группы, состоящей из первого символа строки.
   - Проходим по строке и объединяем подряд идущие одинаковые символы в группы.
3. Перебор всех возможных троек групп и проверка условий:

```
cnt = 0
for i in range(1, len(a)  1):
    if a[i - 1][1] >= left and a[i][1] == middle and a[i + 1][1] >= right:
        cnt += 1
```

   - Перебираем все группы, начиная со второй и заканчивая предпоследней.
   - Проверяем, что количество символов в соседних группах соответствует требованиям left, middle, right.
4. Вывод результата:

```
print(cnt)
```

   - Выводим количество найденных подстрок, соответствующих условиям задачи.

Пример работы кода

Для строки `llvvyywwww` и параметров left = 2, middle = 2, right = 2:

1. Разбиение строки на группы:
   - `ll` → $('l', 2)$
   - `vv` → $('v', 2)$
   - `yy` → $('y', 2)$
   - `wwww` → $('w', 4)$
2. Перебор троек групп и проверка условий:
   - Для $(('l', 2), ('v', 2), ('y', 2))$ условия выполняются.
   - Для $(('v', 2), ('y', 2), ('w', 4))$ условия выполняются.

Результат: 2 допустимые подстроки.

Разбор задачи «Коровы проверяют Даттона»

Решение задачи

1. Чтение входных данных:

```
import sys
from collections import defaultdict

input = sys.stdin.read
data = input().splitlines()

n = int(data[0])
s = data[1]
```

2. Инициализация команд и переменных:

```
comand = {
    'L': (-1, 0),
    'R': (1, 0),
    'U': (0, 1),
    'D': (0, -1),
}

X = defaultdict(int)
Y = defaultdict(int)
x, y = 0, 0
X[0] = 1
Y[0] = 1
cell = {(0, 0): 1}
```

3. Обработка команд и запись значений в клетки:

```
for i in range(n):
    c = s[i]
    x += comand[c][0]
```

```
        y += comand[c][1]
        tmp = cell.get((x, y), 0)
        cell[(x, y)] = i + 2
        X[x] -= tmp
        Y[y] -= tmp

        X[x] += i + 2
        Y[y] += i + 2
```

4. Обработка запросов и вывод результатов:

```
q = int(data[2])
results = []
for it in range(q):
    tmp = list(map(int, data[3 + it].strip().split()))
    tp = tmp[0]
    if tp == 1:
        x = tmp[1]
        results.append(str(X[x]))
    else:
        y = tmp[1]
        results.append(str(Y[y]))

print("\n".join(results))
```

Подробное пояснение к решению

1. Чтение входных данных:
   - Используем модуль *sys* для считывания входных данных, что позволяет быстро обрабатывать большие объемы данных.
   - Считываем количество команд n и строку команд s.
2. Инициализация команд и переменных:
   - Создаем словарь comand, в котором каждой команде (L, R, U, D) сопоставлены изменения координат на поле.
   - Создаем два словаря X и Y для хранения сумм значений в строках и столбцах соответственно. Используем defaultdict для удобства.
   - Инициализируем начальную позицию $(0, 0)$ и записываем в неё значение 1.
3. Обработка команд и запись значений в клетки:
   - Проходим по строке команд и вычисляем новую позицию после каждой команды.
   - Получаем текущее значение в клетке и обновляем его до нового значения.
   - Обновляем сумму значений в текущей строке и столбце, учитывая новое значение и вычитая старое значение.
4. Обработка запросов и вывод результатов:
   - Считываем количество запросов q.
   - Проходим по каждому запросу и в зависимости от типа запроса (1 или 2) выводим сумму значений в указанной строке или столбце.

- Сохраняем результаты и выводим их в конце.

Этот алгоритм эффективно решает задачу, обеспечивая обработку команд и запросов за линейное время, что подходит для ограничений задачи.

Разбор задачи «Забор »

Важный факт заключается в том, что, если пара целых чисел (l, r) удовлетворяет условиям, то также удовлетворяет пара $(l + 1, r)$. Таким образом, существует d_r такая, что:

Для фиксированного r, пара (l, r) удовлетворяет условиям, если и только если l является целым числом меж

Если можно найти эту d_r для каждого r, искомый ответ можно найти как:

$$\sum_{r=1}^{M}(r - d_r + 1).$$

Теперь рассмотрим, как найти эту d_r.

Предположим, что мы уже знаем d_{r-1} и теперь хотим найти d_r. Если не существует i с $R_i = r$, то нет новых ограничений, поэтому $d_r = d_{r-1}$. Если существует такое i, то:

$$d_r = \max(d_{r-1}, L_{\max} + 1),$$

где $L_{\max}$ - это максимальное значение L_i для i, у которых $R_i = r$.

Путем поэтапной оценки этих рекуррентных соотношений, мы можем найти все d_r, начиная с d_1 по порядку.

Задача может быть решена путем корректной реализации этой процедуры. Сложность составляет $O(N + M)$.

Разбор задачи «Подвалы и Математики»

Идея решения

Основная цель — найти максимально возможную площадь правильного многоугольника с k вершинами, который можно разместить на столе размером $n \times m$.

Шаги решения

1. Обработка входных данных

Читаем размеры стола и количество сторон многоугольника:

$$n, m, k$$

где n и m — размеры стола, k — количество сторон многоугольника.

Для простоты пусть $n \leqslant m$.

2. Условия для $k > 4$

Если количество сторон многоугольника $k > 4$, то поле такого многоугольника на столе разместить невозможно, поэтому задача рассматривает только правильные многоугольники с 3 и 4 вершинами:

$$\text{Если } k > 4, \text{ выводим } -1.$$

3. Обработка для правильного треугольника ($k = 3$)

Лемма

В оптимальной посадке мы можем предположить, что правильный треугольник и стол имеют хотя бы одну общую вершину.

Доказательство

Рассмотрим самую верхнюю, самую нижнюю, самую правую и самую левую точки на правильном треугольнике. На самом деле, существует хотя бы одна такая вершина для каждой из этих точек. Также вершина не может одновременно быть самой верхней и самой нижней, или самой правой и самой левой, поэтому треугольник всегда удовлетворяет хотя бы одному из следующих утверждений:

1. Существует вершина, которая является самой верхней и самой правой точкой на треугольнике.
2. Существует вершина, которая является самой верхней и самой левой точкой на треугольнике.
3. Существует вершина, которая является самой нижней и самой правой точкой на треугольнике.
4. Существует вершина, которая является самой нижней и самой левой точкой на треугольнике.

Например, если выполняется первое утверждение, то мы можем сдвинуть треугольник так, чтобы эта вершина совпала хотя бы с одной из вершин прямоугольника, и, следовательно, можно предположить, что оптимальная посадка удовлетворяет этому предположению.

Площадь правильного треугольника зависит от длины стороны, значит надо максимизировать длину стороны.

Для правильного треугольника мы фиксируем первую вершину в точке $(0, 0)$, вторая вершина будет находится в точке (n, mid), где mid будем искать с помощью бинарного поиска.

При увеличении mid длина стороны будет увеличиваться, так как длина стороны $\sqrt{n^2 + \text{mid}^2}$.

Площадь правильного треугольника с данной стороной можно вычислить по формуле:

$$S = \frac{\sqrt{3}}{4} \times a^2$$

где a — длина стороны треугольника.

Бинарный поиск для треугольника

В процессе бинарного поиска мы фиксируем вторую вершину в точке (n, mid), так как мы знаем координаты двух точек треугольника, то можем найти третью.

Вектор $v(n, \text{mid})$ — это координаты второй вершины правильного треугольника. После этого вектор v мы вращаем на угол $\frac{\pi}{3}$ (против часовой стрелки), чтобы получить координаты третьей вершины треугольника. Проверяем, помещается ли эта вершина на столе. Тем самым находим наибольшее значение mid.

После этого вычисляем площадь треугольника с найденной длиной стороны.

4. Обработка для квадрата ($k = 4$)

Если $k = 4$, то можно просто вычислить площадь квадрата как квадрат минимальной из сторон стола. Площадь квадрата будет равна:

$$S = n^2 \quad (\text{где } n \text{ — минимальная сторона стола}).$$

Краткий алгоритм

1. Считываем входные данные n, m, k.
2. Если $k > 4$, выводим -1.
3. Если $k = 3$, выполняем бинарный поиск для нахождения максимально возможной стороны правильного треугольника.
4. Если $k = 4$, вычисляем площадь квадрата, который помещается на столе.

Реализация

```
typedef long double ld;
const ld eps = 0.000'000'001;

ld pow2(ld x){
    return x * x;
}

struct pnt{
    ld x, y;

    pnt(){}

    pnt(ld _x, ld _y){
        this->x = _x;
        this->y = _y;
    }
};

pnt operator-(pnt a, pnt b){
    return pnt(a.x - b.x, a.y - b.y);
}

pnt operator+(pnt a, pnt b){
    return pnt(a.x + b.x, a.y + b.y);
}

pnt rot(pnt v, ld a){
    return pnt(
        cos(a)*v.x - sin(a)*v.y,
        sin(a)*v.x + cos(a)*v.y
    );
}

const ld pi = acos(-1.0);

int main() {
    ld n, m;
    int k;
    cin >> n >> m >> k;
```

```
    if(k > 4){
        cout << -1;
        return;
    }

    if(n > m)
        swap(n, m);

    if(k == 3){
        ld l = 0, r = m;
        for(int i = 0; i < 100; ++i){
            ld mid = (l + r) / 2.0;

            pnt v(n, mid);
            pnt u = rot(v, pi / 3.0);

            if(u.x >= 0 && u.y <= m)
                l = mid;
            else
                r = mid;
        }
        cout << fixed << setprecision(20) << (sqrt(3) * (r*r + n*n)) / 4.0;
    } else {
        cout << fixed << setprecision(20) << n*n;
    }
}
```

Разбор задачи «Ночной жор Даттона»

Алгоритм решения

1. **Прочитать входные данные**:
   - n, k — длина коридора и количество переключаемых лампочек.
   - Массив L — начальные состояния лампочек.
   - A — номер комнаты, где находится кухня.
2. **Подготовка начального состояния**:
   - Если $A > 0$, инвертируем состояния лампочек в комнатах от 1 до A, чтобы они стали противоположными (включены, если были выключены, и наоборот). Это нужно для простоты реализации, так как выключенные лампочки в комнатах от 1 до A надо включать, а после инвертирования надо сделать все элементы равными 0.
3. **Использование подхода "ленивого"пропагирования**:
   - Завести массив *state* для отслеживания необходимости инверсии в текущем и следующих позициях.
   - Проход по массиву лампочек с учетом накопленных инверсий.
   - Если текущая лампочка с индексом i находится в неправильном состоянии, инвертировать её и следующие k лампочек. Для инвертирования в массиве *state* меняем значения в ячейках i и $i + k$ на противоположные.
4. **Проверка на возможность инверсии**:

- Если очередная инверсия выходит за границы массива, то вывести -1.

5. **Вывод результата**:
   - Количество необходимых инверсий.

Реализация на C++

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n, k;
    cin >> n >> k;
    vector<int> L(n);
    for (int i = 0; i < n; ++i) {
        cin >> L[i];
    }
    int A;
    cin >> A;

    // Инвертируем начальное состояние до кухни
    for (int i = 0; i < A; ++i) {
        L[i] = !L[i];
    }

    vector<int> state(n + 1, 0);
    int cnt = 0;
    int cur = 0;

    for (int i = 0; i < n; ++i) {
        cur ^= state[i];
        if ((L[i] ^ cur) == 0) continue;
        cnt++;
        cur ^= 1;
        if (i + k - 1 >= n) {
            cout << "-1\n";
            return 0;
        }
        state[i + k] ^= 1;
    }

    cout << cnt << '\n';
    return 0;
}
```

Пояснение кода

1. **Чтение входных данных**: Считываем длину коридора, количество переключаемых лампочек, массив начальных состояний и номер комнаты, где находится кухня.

2. **Инвертирование состояний до кухни**: Проходим по массиву до комнаты A и инвертируем состояния лампочек.

3. **Алгоритм ленивого пропагирования**: Используем массив *state* для отслеживания необходимости инверсии. Если лампочка в текущей комнате находится в неправильном состоянии, инвертируем её и следующие k лампочек, обновляя *state* и счётчик операций *cnt*.

4. **Проверка возможности инверсии**: Если инверсия выходит за пределы массива, выводим -1.

5. **Вывод результата**: Количество необходимых инверсий.

Разбор задачи «Ильнур и массив»

Алгоритм решения:

1. **Принцип Дирихле**:
   - Если количество пар элементов в массиве $\frac{n(n-1)}{2}$ превышает 1,000,000, то массив точно не может быть хорошим, потому что будет слишком много пар, чтобы все они были уникальны. В этом случае сразу выводим «NO».
   - Поскольку значение $P(i,j) = \lceil \sqrt{a_i \cdot a_j} \rceil$ округляется вверх, различных значений P не может быть больше 1,000,000.

2. **Вычисление величины** $P(i,j)$:
   - Для каждой пары индексов i и j вычислить $P(i,j)$, используя формулу $P(i,j) = \lceil \sqrt{a_i \cdot a_j} \rceil$.
   - Используем округление вверх для получения целого числа.

3. **Проверка уникальности величин**:
   - Храним вычисленные значения $P(i,j)$ в массиве флагов b.
   - Если $P(i,j)$ уже присутствует в массиве флагов, выводим «NO» и завершаем программу.
   - Если все величины уникальны, выводим «YES».

Реализация на Python:

```
import math

N = 1000000
n = int(input())

# Если количество пар больше 1,000,000, то массив не может быть хорошим
if n * (n - 1) // 2 > N:
    print("NO")
    exit(0)

# Чтение массива
s = input().split()
a = [int(s[i]) for i in range(n)]

# Массив для отслеживания уникальности величин P(i, j)
b = [0] * (N + 1)

for i in range(n):
    for j in range(i + 1, n):
```

```
        v = a[i] * a[j]
        u = int(math.sqrt(v))
        # Корректировка u для точного целочисленного значения
        while u * u > v:
            u -= 1
        while u * u < v:
            u += 1

        # Проверка уникальности
        if b[u] == 1:
            print("NO")
            exit(0)
        b[u] = 1

print("YES")
```

Пояснение принципа Дирихле:

Принцип Дирихле утверждает, что если m объектов распределены по n ящикам, и $m > n$, то хотя бы один ящик будет содержать более одного объекта. В контексте данной задачи:

- Значение $P(i, j) = \lceil \sqrt{a_i \cdot a_j} \rceil$ округляется вверх, поэтому различных значений P не может быть больше 1,000,000.
- Если это количество пар превышает 1,000,000, то по принципу Дирихле должно быть хотя бы две пары чисел, для которых значение $P(i, j)$ совпадает. Следовательно, если $\frac{n(n-1)}{2} > 1,000,000$, то массив не может быть хорошим, так как среди этих пар найдутся одинаковые значения P.
- Таким образом, если количество пар превышает 1,000,000, мы сразу выводим «NO».

Разбор задачи «Ночной жор Даттона 7-8»

1. **Перебор позиций**: Для каждой позиции i от 0 до $n - k$:
   - Если лампочка в позиции i выключена (0), переключить k подряд идущих лампочек начиная с позиции i.
   - Увеличить счетчик действий.
2. **Проверка результата**: После обработки всех позиций проверить, все ли лампочки включены. Если да, вывести количество действий. Если нет, вывести -1.

Разбор задачи «Примитивный тетрис»

В первой группе тестов $n = 1$ и $m = 4$. Заметим, что на поле высоты 1 единственная фигура, которая не выйдет за границы поля сразу же — это горизонтальная полоска ширины 4. Заметим также, что такую фигуру очень просто отличить от остальных, даже не смотря на точки и решетки во входных данных — для нее $r_i = 1$ и $c_i = 4$. Так что в данной группе тестов достаточно было выполнить проверку, что первая фигура — полоска 1×4, и в таком случае вывести 2, а иначе — 1.

Вторая группа тестов решалась аналогично, однако требовала заметного перебора случаев. Если у вас не получалось сдать полное решение, можно было попробовать неполноценным решением с какими-то ошибками пройти хотя бы эту группу тестов. Либо же перебор случаев для этой группы можно было сократить, заметим, что мало какие фигуры могут расположиться друг на друге на таком невысоком поле, не выйдя за границу, а также что две не выходящие за границу фигуры

можно расположить рядом по горизонтали только если они имеют высоту $r_i = 2$ или $r_i = 3$, а таких фигур не очень много.

Третья группа гарантировала, что все фигуры — это «столбики» 4×1. Для таких фигур игру достаточно просто обработать: каждый столбец поля будет иметь высоту $4k$, где k — количество упавших в этот столбец фигур. Поэтому достаточно для каждого из m столбцов поля хранить его текущую высоту, и следить, когда высота какого-то из них превысит n.

Четвертая группа абсолютно аналогична третьей с той лишь разницей, что теперь поле заполняется квадратами 2×2. Поскольку все квадраты падают на четные столбцы, заполнение поля будет «плотным», и достаточно так же для каждого столбца поля поддерживать его высоту.

Оставшиеся группы уже требовали реализации более-менее полного решения. Идея остается такой же: для каждого столбца храним его «высоту». Важное отличие в том, что на поле могут быть «пробелы», поэтому будем хранить для каждого столбца h_j — самую верхнюю клетку в нем. Теперь давайте научимся для фигуры понимать, на какой высоте она остановится.

- в пятой группе тестов все фигуры — полоски 1×4, так что они будут падать на высоту $\max(h_j, \ldots, h_{j+3}) + 1$;
- в шестой и седьмой группах достаточно было двигать фигуру в явном виде по одной клетке вниз и проверять на пересечение с текущим замощением поля; в шестой группе маленькие ограничения позволяли выполнять все эти действия произвольным образом с почти любым временем работы;
- для полного решения надо было для каждой фигуры и каждого ее столбца найти mn_j и mx_j — самую нижнюю и самую верхнюю клетку в нем; после чего определить строку, на которой остановится фигура, как $R = \max(h_j - mn_0, \ldots, h_{j+3} - mn_3) + 1$, и пересчитать высоты столбцов как $h_{j+0\ldots3} \leftarrow R + mx_{0\ldots3}$.

Соответственно, полное решение обрабатывает каждую фигуру за $\mathcal{O}(1)$ и работает за $\mathcal{O}(q)$ в сумме.

Разбор задачи «Примитивный тетрис 7-8»

Эта задача на простую симуляцию описанного в условии процесса. Проще было либо нарисовать поле нужного размера на бумаге, либо открыть тест в текстовом редакторе, и затем по очереди размещать на поле заданные фигуры в соответствии с описанным в условии процессом.

Разумеется, при «ручном» решении задачи можно было случайно ошибиться, поэтому альтернативно можно было написать код, который решает эту задачу. Самое простое решение хранит поле в виде двухмерного массива (матрицы) и при обработке каждой фигуры двигает ее вниз по одной строке за раз до тех пор, пока следующее перемещение фигуры вниз приведет к ее пересечению с уже расположенными на поле клетками.