Разбор задач

Задача 1. Лягушка и кузнечик

Расстояние между лягушкой и кузнечиком равно $N-1$ клеток. За одну секунду это расстояние может сократиться на 3, 4 или 5 клеток. Поэтому если $N-1=1$ или $N-1=2$ (то есть при $N<4$) лягушка и кузнечик не смогут встретиться и нужно вывести «-1».

Во всех остальных случаях им понадобится $\lceil \frac{N-1}{5} \rceil$ прыжков (частное от деления $N-1$ на 5, округлённое вверх). Это значение можно вычислить по формуле `(N - 1 + 4) // 5` (где `//` — операция целочисленного деления в языке Python, в языках C++, C#, Java надо использовать просто `/`, в языке Pascal — операцию `div`), или при помощи действительного деления и округления результата вверх при помощи функции `ceil`.

Пример решения на языке Python.

```
n = int(input())
if n < 4:
    print(-1)
else:
    print((n - 1 + 4) // 5)
```

Если решать задачу «моделированием», например, в цикле увеличивать координату лягушки на 3, а координату кузнечика уменьшать на 2, то такое решение наберёт 50 баллов.

Кроме того, частичные решения могут быть основаны на идее перебора. Например, можно перебирать величины f_2, f_3, g_1, g_2 — количество прыжков лягушки на 2 и на 3 клетки и количество прыжков кузнечика на 1 и на 2 клетки соответственно — и в качестве кандидатов на оптимальное решение рассматривать такие варианты, в которых

$$2f_2 + 3f_3 + g_1 + 2g_2 = N - 1$$

и

$$f_2 + f_3 = g_1 + g_2.$$

В этом случае получится очень медленное решение, имеющее вычислительную сложность $O(N^4)$ и набирающее 30 баллов.

Также допустимо перебирать количество секунд s с момента старта и анализировать, могут ли лягушка и кузнечик оказаться в одной клетке по прошествии данного времени. Потребуется выяснить, пересекаются ли множества клеток, в которых они могут оказаться. Нетрудно понять, что для лягушки такое множество — это отрезок полосы с номерами клеток от $1+2s$ до $1+3s$ включительно, а для кузнечика — отрезок с номерами клеток от $N-2s$ до $N-s$ включительно. Таким образом, задача сводится к нахождению такого минимального s, при котором данные отрезки пересекаются. Если этот поиск осуществлять последовательным перебором, решение будет иметь вычислительную сложность $O(N)$ и набирать 50 баллов. Также минимальное подходящее s можно найти методом двоичного поиска, что даёт вычислительную сложность $O(\log N)$ и будет оцениваться полным баллом.

Задача 2. Центральные квадраты

Множество «центральных» квадратов в этой задаче является прямоугольником, расположенным в центре исходного прямоугольника и вытянутым вдоль большей стороны исходного прямоугольника. Будем считать, что n — меньшая сторона исходного прямоугольника, а m — большая, если это не так, то поменяем значения n и m.

Если n нечётное, то множество «центральных» квадратов является прямоугольником шириной 1 клетку и длиной $m-n+1$ клеток. Если же n чётное, то центральный прямоугольник имеет ширину 2 клетки и длину $m-n+2$ клеток. Необходимо разобрать эти варианты и вывести нужное значение.

Пример решения на языке Python.

```
n = int(input())
m = int(input())
```

```
if n > m:
    n, m = m, n

if n % 2 == 1:
    print(m - n + 1)
else:
    print(2 * (m - n + 2))
```

30 баллов можно набрать, если создать двумерный массив размера $n \times m$, записать для каждого элемента массива расстояние от него до края и посчитать количество элементов массива с наибольшим значением расстояния.

60 баллов набирает решение, не содержащее массива размером $n \times m$, но перебирающее все горизонтальные или вертикальные полосы прямоугольника.

Задача 3. Заказ в магазине

Сначала проверим, существует ли решение. Максимальное количество ручек, которое можно заказать, равно $1 + 2 + 3 + ... + N = N(N + 1)/2$ (по формуле суммы арифметической прогрессии), и если $M > N(N + 1)/2$, то решения не существует.

При проверке этого условия можно столкнуться с проблемой переполнения 32-битного целого типа в языках C++, Pascal, Java, C#, поэтому вычисление нужно производить с использованием 64-битных целочисленных переменных. Также если эту сумму вычислять не по формуле, а при помощи цикла, то цикл длины N можеть не уложиться в ограничение по времени. В этом случае стоит прервать цикл, если сумма превысит M, либо заметить, что если $N > \sqrt{2 \cdot 10^9} \approx 44721$, то $1 + 2 + 3 + ... + N > 10^9$ и решение существует.

Если решение существует, то воспользуемся жадным алгоритмом: будем выбирать упаковки максимально возможного размера: N, $N - 1$, $N - 2$ и т.д. Если размер рассматриваемой упаковки s больше или равен M, то выведем значение s и уменьшим M на s.

Если использовать цикл `for` от N до 1, то получится решение сложности $O(N)$, которое наберёт 40 баллов. Надо заметить, что вовсе необязательно перебирать все значения от N до 1, т.к. если в какой-то момент значение M станет меньше рассматриваемого размера упаковки, то достаточно взять одну упаковку размером M, то есть вывести M и завершить работу программы. Такое решение будет иметь сложность $O(\sqrt{M})$.

Пример решения на языке Python.

```
n = int(input())
m = int(input())
if (1 + n) * n // 2 < m :
    print(0)
else:
    while m > n:
        print(n)
        m -= n
        n -= 1
    print(m)
```

Задача 4. Спираль

Решение на 40 баллов создаёт двумерный массив размером $N \times M$ и моделирует движение черепашки, отмечая в массиве закрашенные клетки. Такое решение довольно трудно в реализации, приведем пример реализации на языке C++. Решение имеет сложность $O(NM)$.

```
#include<iostream>
#include<vector>
using namespace std;
```

```
int n, m;
vector <vector<bool>> used;;
unsigned int ans = 0;
int di[4] = {1, 0, -1,  0};
int dj[4] = {0, 1,  0, -1};

bool check_used(int i, int j)
{
    return i >= 0 && i < n && j >= 0 && j < m && used[i][j];
}

bool check_good(int i, int j)
{
    if (i < 0 || i >= n || j < 0 || j >= m)
        return false;
    int count = check_used(i - 1, j) + check_used(i + 1, j)
              + check_used(i, j - 1) + check_used(i, j + 1);
    return count == 1;
}

int main()
{
    cin >> n >> m;
    used.resize(n, vector<bool>(m));
    int i = 0;
    int j = 0;
    int dir = 0;
    while (true)
    {
        ++ans;
        used[i][j] = true;
        int i1 = i + di[dir];
        int j1 = j + dj[dir];
        if (check_good(i1, j1))
        {
            i = i1;
            j = j1;
            continue;
        }
        dir = (dir + 1) % 4;
        i = i + di[dir];
        j = j + dj[dir];
        if (!check_good(i, j))
            break;
    }
    cout << ans << endl;
}
```

Чтобы написать решение на 60 баллов, рассмотрим движение черепашки вдоль левой стороны прямоугольника. Черепашка закрасит столбец из N клеток. Теперь пусть черепашка сдвинется на одну клетку вправо, закрасив суммарно $N + 1$ клетку. Справа от черепашки теперь располагается прямоугольник из $M - 2$ столбцов и N строк, и черепашка начинает обходить его вдоль стороны

длиной $M-2$. То есть мы свели задачу от прямоугольника размером $N \times M$ к прямоугольнику размером $(M-2) \times N$, закрасив $N+1$ клетку. Это можно сделать, если $N \geqslant 2$ и $M \geqslant 2$. Пока соблюдается это условие, будем в цикле прибавлять к ответу $N+1$ и менять числа (N, M) на $(M-2, N)$.

В результате мы получим либо вырожденный пустой прямоугольник (если мы отрежем от него всё), либо прямоугольник, одна из сторон которого равна 1. В этом случае черепашка закрасит весь оставшийся прямоугольник, в котором будет MN клеток.

Такое решение имеет сложность $O(M+N)$ и набирает 60 баллов. Пример решения на языке Python.

```
n = int(input())
m = int(input())
ans = 0
while n >= 2 and m >= 2:
    ans += n + 1
    n, m = m - 2, n
ans += m * n
print(ans)
```

Наконец, для решения на полный балл заметим, что суммируются числа вида $N+1$, $M-2+1$, $N-2+1$, $M-4+1$, $N-4+1$, ... То есть каждое следующее горизонтальное или вертикальное перемещение черепашки окажется на 2 короче предыдущего горизонтального или вертикального перемещения, и мы суммируем арифметические прогрессии, что можно сделать без использования цикла.

Удобней суммировать не две прогрессии, а одну. Рассмотрим передвижения черепашки вдоль сторон прямоугольника: вниз на N клеток, вправо на $M-1$ клеток, затем вверх на 1 клетку. Так мы перейдём к прямоугольнику размером $(N-2) \times (M-2)$, прибавив к ответу $N+M$. На следующем шаге мы поибавим к ответу $(N-2)+(M-2)$, затем опять уменьшим стороны прямоугольника на 2. Мы получим арифметическую прогрессию с начальным членом $N+M$ и разностью -4.

Посчитаем количество членов этой прогрессии — число возможных сокращений сторон прямоугольника на 2, пока не останется прямоугольник, наименьшая сторона которого будет не более 2. Обозначим это число за k. Добавим к ответу сумму арифметической прогрессии из k членов с начальным значением $n+m$ и разностью -4, и разберём случаи разного количества закрашенных клеток в оставшемся прямоугольнике.

Такое решение имеет сложность $O(1)$, пример решения на языке Python.

```
n = int(input())
m = int(input())
k = min(n - 1, m - 1) // 2
ans = ((n + m) + (n + m) - 4 * (k - 1)) * k // 2
n -= 2 * k
m -= 2 * k
if n == 1:
    ans += m
elif m == 1:
    ans += n
elif m == 2:
    ans += n + 1
else: # n == 2, m > 2
    ans += m + 2
print(ans)
```

Задача 5. Надпись на табло

В этой задаче набранные баллы зависят от того, насколько аккуратно были разобраны разные

случаи входных данных. Сама задача требует значительных навыков программирования.

Рассмотрим возможное решение.

```
import sys

def quit(c):
    print(c)
    sys.exit(0)

n = int(input())

a = [[] for i in range(n)]

for i in range(n):
    a[n - 1 - i] = list(input())

x1 = n + 1
x2 = -1
y1 = n + 1
y2 = -1

for y in range(n):
    for x in range(n):
        if a[y][x] == '#':
            x1 = min(x1, x)
            x2 = max(x2, x)
            y1 = min(y1, y)
            y2 = max(y2, y)

if x1 > x2:
    quit("X")

def find_rect():
    for y in range(y1, y2 + 1):
        for x in range(x1, x2 + 1):
            if a[y][x] == '.':
                y3 = y
                x3 = x
                y4 = y
                x4 = x
                while y4 + 1 <= y2 and a[y4 + 1][x] == '.':
                    y4 += 1
                while x4 + 1 <= x2 and a[y][x4 + 1] == '.':
                    x4 += 1
                for y in range(y3, y4 + 1):
                    for x in range(x3, x4 + 1):
                        if a[y][x] == '#':
                            quit("X")
                        else:
                            a[y][x] = '#'
                return (x3, x4, y3, y4)
    return None

res = find_rect()
```

```
if res is None:
    quit("I")
else:
    x3, x4, y3, y4 = res

res = find_rect()
if res is None:
    if  x1 < x3 <= x4 < x2 and y1 < y3 <= y4 < y2:
        quit("O")
    elif x1 < x3 <= x4 == x2 and y1 < y3 <= y4 < y2:
        quit("C")
    elif x1 < x3 <= x4 == x2 and y1 < y3 <= y4 == y2:
        quit("L")
    else:
        quit("X")
else:
    x5, x6, y5, y6 = res

res = find_rect()
if res is None:
    if x1 < x3 == x5 <= x4 == x6 < x2 and y1 == y3 <= y4 < y5 <= y6 == y2:
        quit("H")
    elif x1 < x3 == x5 <= x6 < x4 == x2 and y1 == y3 <= y4 < y5 <= y6 < y2:
        quit("P")
    else:
        quit("X")
else:
    quit("X")
```

В этом решении функция `quit` используется для вывода ответа с последующим завершением работы программы.

После чтения входных данных найдём значения x_1, x_2, y_1, y_2 — минимальное и максимальное значения координаты x и координаты y клетки, в которой стоит символ «#». Это — предполагаемые границы внешнего прямоугольника.

Фукнция `find_rect` используется для выделения внутренних прямоугольников, составленных из символов «.». Она находит самый нижний символ «.», затем определяет высоту и ширину прямоугольника циклами по оси OY и по оси OX, затем во вложенном прямоугольнике заменяет все символы «#» на «.». Если при этом символ «#» будет найден там, где предполагается наличие внутренней части вложенного прямоугольника, сразу же вызываем `quit('X')`. Функция возвращает координаты прямоугольника, а если такого нет, то функция возвращает специальное значение `None`.

Будем вызывать эту функцию для выделения прямоугольников. Если первый же вызов функции вернул `None`, значит, внутри большого прямоугольника нет символов «.», ответом является «I». Если же нам удалось выделить прямоугольник, сохраним его координаты и вызовем функцию `find_rect` повторно. Если второй вызов вернул `None`, значит, внутри есть только один прямоугольник. Проверив условия на его границы, отделим ответы «O», «C», «L» или «X», если прямоугольник один, но условия не выполнены.

Если был найден второй прямоугольник, то убедимся, что больше символов «.» не осталось, вызвав функцию `find_rect` в третий раз. Для двух найденных прямоугольников проверим условия букв «H» и «P», наконец, если ничего не подошло, то выведем «X».

Но у задачи есть и более простое решение. Сначала удалим все нижние и верхние строки, которые не содержат символов «#», а также выключенные столбцы слева и справа.

Теперь «упростим» картинку: будем удалять соседние строки и соседние столбцы, если они совпадают. В результате буква «I» превратится всего лишь в один символ «#», а другие буквы превратятся в небольшие картинки, содержащие не более четырёх строк и трёх столбцов:

```
 O    C    L   H    P
###  ##   #.  #.#  ###
#.#  #.   ##  ###  #.#
###  ##       #.#  ###
                  #..
```

После этого просто проверим совпадение полученного массива с одним из возможных вариантов.

Пример такого решения.

```
n = int(input())
lines = [list(input()) for _ in range(n)]

i = 0
while i + 1 < len(lines):
    if lines[i] == lines[i + 1]:
        lines.pop(i)
    else:
        i += 1

i = 0
while i + 1 < len(lines[0]):
    if [lines[j][i] for j in range(len(lines))] == \
       [lines[j][i + 1] for j in range(len(lines))]:
        for j in lines:
            j.pop(i)
    else:
        i += 1

while lines and set(lines[0]) == {"."}:
    lines.pop(0)
while lines and set(lines[-1]) == {"."}:
    lines.pop()

while lines and lines[0] and \
   set(lines[j][0] for j in range(len(lines))) == {"."}:
    for j in lines:
        j.pop(0)
while lines and lines[-1] and
   set(lines[j][-1] for j in range(len(lines))) == {"."}:
    for j in lines:
        j.pop()

for i in range(len(lines)):
    lines[i] = "".join(lines[i])

if lines == ["#"]:
    print("I")
elif lines == ['###', '#.#', '###']:
    print("O")
elif lines == ["##", "#.", "##"]:
    print("C")
elif lines == ['#.', '##']:
```

```
    print("L")
elif lines == ['#.#', '###', '#.#']:
    print("H")
elif lines == ['###', '#.#', '###', '#..']:
    print("P")
else:
    print("X")
```