ДП

1 A. Платная лестница

Разбор

Несмотря на то, что это простая задача, попробуем сначала порассуждать так, чтобы построить рекуррентную зависимость между состояниями динамики, а затем составим итеративный алгоритм.

Что нам хочется хранить в состоянии динамики? В этой задаче нет никаких значений кроме суммы, которая указана для каждой ступеньки. Значит, мы можем считать её напрямую. Допустим, на последней (n-й) ступеньке мы набрали какую-то сумму x. На n-ю ступеньку мы могли попасть только с $n-1$ либо $n-2$ ступенек. При этом, чтобы сумма x была оптимальной, нам нужно было выбрать минимум из них. Для каждой из этих ступенек такое правило сохраняется. Тогда мы получаем следующее правило для ответа на k-й ступеньке:

$$s_k = \min(s_{k-1}, s_{k-2}) + c_k$$

Понятно, что на нулевой ступеньке ответ - 0, а на первой - стоимость этой ступеньки. Так как для ответа на всех остальных ступеньках выполняется полученное нами правило, то теперь мы можем легко написать итеративный алгоритм.

Асимптотика решения: $O(N)$

Решение

```
n = int(input())
c = list(map(int, input().split()))
dp = [0] * (n + 1)
dp[0] = 0
dp[1] = c[0]
for i in range(2, n + 1):
    dp[i] = min(dp[i - 1], dp[i - 2]) + c[i - 1]
print(dp[-1])
```

2 B. Взрывоопасность-2

Разбор

В этой задаче проведем аналогичные рассуждения. Для начала подумаем, что мы хотим хранить в состоянии динамики и сколько их у нас будет? Понятно, что кроме ответа, кажется, больше нечего хранить. Но помимо этого нам нужно как-то исключать из ответа некорректные последовательности. Однако на самом деле можно сразу считать ответ без них! Давайте смотреть на эту задачу как на цепочку состояний. В этой цепочке на одном шаге есть состояния трех типов (A, B, C). При этом у нас есть ограничение на переходы:

- В состояние типа A мы можем переходить только из состояний B или C
- В состояние типа B мы можем переходить из всех возможных цветов
- В состояние типа C мы аналогично можем переходить из всех возможных цветов

При этом при переходе мы просто складываем ответы из предыдущих состояний в то, куда мы попали. Теперь мы можем легко написать итеративный алгоритм.

Асимптотика решения: $O(N)$

Решение

```
n = int(input())
dp = [[0] * 3 for _ in range(n)]
dp[0][0] = 1
dp[0][1] = 1
dp[0][2] = 1
for i in range(1, n):
    dp[i][0] = dp[i - 1][1] + dp[i - 1][2]
    dp[i][1] = dp[i - 1][0] + dp[i - 1][1] + dp[i - 1][2]
    dp[i][2] = dp[i - 1][0] + dp[i - 1][1] + dp[i - 1][2]
print(sum(dp[-1]))
```

3 C. Ход конем - 2

Разбор

В этой задаче конь **может** ходить выше или левее предыдущей клетки, поэтому нам не подойдёт прямой обход массива состояний.

Изначально мы не были ни в одном состоянии, поэтому инициализируем их каким-нибудь специальным значением (например, -1). Обход начинаем из последней клетки, в которую мы хотим попасть по условию задачи. В неё мы можем попасть из четырех каких-то других состояний, тогда посчитаем ответ для этой клетки *лениво*, перейдя в потомков и посчитав ответ сначала для них. Важно помнить, что конь **не может** совершить циклический обход какого-то набора клеток, поэтому если вдруг в процессе обхода мы попали в уже посещённую точку - просто возьмём значение из неё, считая, что ответ для неё уже посчитан корректно. Определить, была посещена клетка или нет, очень просто – в посещенной клетке значение состояния $dp_{i,j}$ будет отлично от -1.

Асимптотика решения: $O(N \times M)$

Решение

```
def cnt(dp, i, j, h, w):
    if not (0 <= i < h and 0 <= j < w):
        return 0
    if dp[i][j] < 0:
        a = cnt(dp, i - 2, j - 1, h, w)
        b = cnt(dp, i - 2, j + 1, h, w)
        c = cnt(dp, i - 1, j - 2, h, w)
        d = cnt(dp, i + 1, j - 2, h, w)
        dp[i][j] = a + b + c + d
    return dp[i][j]

n, m = map(int, input().split())
dp = [[-1 for _ in range(m)] for _ in range(n)]
dp[0][0] = 1
ans = cnt(dp, n - 1, m - 1, n, m)
print(ans)
```

4 D. Расстояние Дамерау-Левенштейна

Разбор

Будем строить динамику по параметрам i – длина префикса первой строки и j – длина префикса второй строки. Изначально считаем, что строки равны и расстояние между ними равно нулю. Очевидно, что если мы возьмём у одной строки префикс длины 0, то расстояние до второй строки будет строго равно длине префикса второй строки. Что делать в остальных случаях?

- Попробуем продлить префикс второй строки текущим символом первой строки, это даст нам расстояние $dp_{i,j-1} + 1$

- Обратно, попробуем продлить префикс первой строки текущим символом второй строки, это даст нам расстояние $dp_{i-1,j} + 1$
- Попробуем заменить последний рассматриваемый символ, это даст нам расстояние $dp_{i-1,j-1} + 1_{s_i \neq t_j}$
- Наконец, если нам позволяют длины обоих префиксов и $s_{i-1} = t_j$ и $s_i = t_{j-1}$, попробуем проверить перестановку – для этого добавим $dp_{i-2,j-2} + 1$

Наконец, из всех этих вариантов нам надо на каждом шаге выбирать минимальный и при этом подходящий по всем условиям. После подсчета всех состояний динамики ответ будет лежать в нижней правой ячейке таблицы.

Асимптотика решения: $O(N \times M)$

Решение

```
s = input()
t = input()

n, m = len(s), len(t)

dp = [[0] * (m + 1) for _ in range(n + 1)]
for i in range(n + 1):
    for j in range(m + 1):
        if min(i, j) == 0:
            dp[i][j] = max(i, j)
        else:
            dp[i][j] = min(dp[i - 1][j] + 1, dp[i][j - 1] + 1, dp[i - 1][j - 1] + int(s[i - 1]
     != t[j - 1]))
            if i > 1 and j > 1 and s[i - 2] == t[j  - 1] and s[i - 1] == t[j - 2]:
                dp[i][j] = min(dp[i][j], dp[i - 2][j - 2] + 1)
print(dp[n][m])
```

5 E. Наибольший квадрат

Разбор

Прежде чем начать думать о том, как хранить и пересчитывать динамику в этой задаче, давайте попытаемся найти правило, по которому мы можем улучшить ответ. Что нам нужно, чтобы увеличить квадрат до размера k при переходе в ячейку (i, j)? Нам нужно, чтобы выполнялись следующие три условия:

- В ячейке $(i - 1, j - 1)$ был достижим квадрат размером $k - 1$
- В ячейках слева и сверху стояли единицы
- В ячейке (i, j) **также** стояла единица

Допустим, что мы будем хранить в состояниях динамики размер квадрата k, если его можно достичь в этой ячейке. В качестве базы динамики возьмем ячейки по левой и верхней границам нашего массива – в них наибольший достижимый размер квадратов равен значению в ячейке массива.

Как делать переходы? Сразу не очень понятно, как их правильно считать. Поэтому заведем две вспомогательных матрицы u и v, в одной из которых будем хранить число непрерывно идущих единиц выше текущей ячейки, а в другой – число непрерывно идущих единиц левее. В таком случае переход можно описать как $dp_{i,j} = min(dp_{i-1,j-1}, u_{i-1,j}, v_{i,j-1}) + a_{i,j}$. Теперь мы можем упростить это решение – обратим внимание на то, что всё, что нам нужно, уже посчитано в массиве динамики! В самом деле, каждая единица задаст динамике значение 1, а по мере роста квадрата мы будем заполнять его внутренние слои:

1 1 1 1	1 **1 1 1**	1 1 1 1
1 2 2 2	1 **2 2 2**	**1 2 2** 2
1 2 3 3	1 **2 3 3**	**1 2 3** 3
1 2 3 _	1 2 3 _	**1 2 3** _

$$\begin{matrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 \\ 1 & 2 & 3 & 3 \\ 1 & 2 & 3 & \mathbf{4} \end{matrix}$$

Асимптотика решения: $O(N \times M)$

Решение

```
def solve(dp, n, m):
    if n == 1 and m == 1:
        return 1, 1, 1
    max_i, max_j, max_size = 0, 0, 0
    for i in range(1, n):
        for j in range(1, m):
            if dp[i][j] > 0:
                dp[i][j] = min(dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]) + 1
            if dp[i][j] > max_size:
                max_size = dp[i][j]
                max_i, max_j = i - max_size + 1, j - max_size + 1
    return max_size, max_i + 1, max_j + 1

n, m = map(int, input().split())
a = [None] * n
for i in range(n):
    a[i] = list(map(int, input().split()))
x, y, z = solve(a, n, m)
print(x)
print(y, z)
```