Динамическое программирование – 1

1 A. Калькулятор

Разбор

В этой задаче динамическое программирование описывается следующим образом:

- Состояние динамики – минимальное число операций, необходимое для получения числа x. Параметр динамики – число x (результат последовательности операций)
- Переход динамики: инкремент значения из предыдущего состояния на 1
- База динамики: считаем, что для получения единицы нам необходимо совершить 0 операций
- Порядок обхода: слева направо в порядке возрастания числа x

Для восстановления последовательности операций будем записывать в массив предков число, из которого мы попали в текущее. Когда мы посчитаем динамику, просто пройдёмся по этому массиву в обратном порядке.

Сложность решения: $O(N)$

Решение

```
n = int(input())
dp, path = [0] * (n + 1), [0] * (n + 1)
for x in range(2, n + 1):
    dp[x], path[x] = dp[x - 1] + 1, x - 1
    if x % 2 == 0 and dp[x // 2] + 1 < dp[x]:
        dp[x], path[x] = dp[x // 2] + 1, x // 2
    if x % 3 == 0 and dp[x // 3] + 1 < dp[x]:
        dp[x], path[x] = dp[x // 3] + 1, x // 3

ans, x = [], n
op_to_diff = {
    '1': lambda x, y: y == x - 1,
    '2': lambda x, y: x % 2 == 0 and y == x // 2,
    '3': lambda x, y: x % 3 == 0 and y == x // 3,
}
while dp[x] != 0:
    new_x = path[x]
    for op, diff in op_to_diff.items():
        if diff(x, new_x):
            ans.append(op)
            break
    x = new_x
print(''.join(ans[::-1]))
```

2 B. Конём ходи!

Разбор

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

Тогда давайте начнём обход с последней клетки. Из неё мы можем попасть в 4 каких-то других состояния и *лениво* посчитать ответ для них, перейдя в потомков. 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.

Сложность 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 = [[0 for _ in range(m)] for _ in range(n)]
dp[0][0] = 1
ans = cnt(dp, n - 1, m - 1, n, m)
print(ans)
```

3 C. Наибольший квадрат

Разбор

В этой задаче опишем динамическое программирование следующим образом:

- Состояние динамики: наибольший размер квадрата, достижимый в подпрямоугольнике размера $i \times j$. Параметрами динамики в таком случае будут являться ширина подпрямоугольника i и высота подпрямоугольника j
- База динамики: считаем, что максимально достижимый размер квадрата равен нулю
- Порядок обхода: последовательно вдоль всех строк и столбцов

Сразу не очень понятно, как считать переход между состояниями динамики. Введём два вспомогательных массива:

- В u будем поддерживать максимальное непрерывное число единиц по горизонтали в строке i
- В v будем поддерживать максимальное непрерывное число единиц по вертикали в столбце j

Тогда становится понятно, что переход можно описать как $dp_{i,j} = min(dp_{i-1,j-1}, u_{i-1}, v_{j-1}) + a_{i,j}$. 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 2 2 2
1 2 3 3
1 2 3 4
```

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(n):
        for j in range(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)
```

4 D. Гвоздики

Разбор

Опишем состояния динамического программирования в этой задаче следующим образом:

- Состояние динамики: суммарная минимальная длина ниточек, которыми мы можем связать гвоздики вплоть до i-го. Введём параметр динамики j, обозначающий следующее:
  - $dp_{i,0}$ - ответ для i-го гвоздика, если мы его **не связали** с предыдущим
  - $dp_{i,1}$ - ответ для i-го гвоздика, если мы его связали с предыдущим
- Переход динамики:
  - $dp_{i,0} = dp_{i-1,1}$ - если мы не связали этот гвоздик с предыдущим, то предыдущий точно должен быть связан с предшествующим ему гвоздиком
  - $dp_{i,1} = \min(dp_{i-1,0}, dp_{i-1,1}) + (a_i - a_{i-1})$ - если мы связываем этот гвоздик с предыдущим, мы можем выбрать лучший ответ на предыдущем шаге и прибавить к нему расстояние для текущего шага
- База динамики: Поймём, что мы обязательно должны связать ниточкой первые два гвоздика. Тогда $dp_{1,j}$ для любого значения параметра j будет равен расстоянию от 2-го гвоздика до 1-го.
- Обход состояний выполняется слева направо, ответ лежит в $dp_{n-1,1}$, так как мы также обязательно должны связать последние два гвоздика.

Особое коварство этой задачи заключается в том, что в открытом тестовом примере координаты гвоздиков идут упорядоченно, хотя на самом деле порядок координат не гарантируется. Отсюда следует, что перед решением задачи мы также должны отсортировать входной массив.

Асимптотика решения: $O(N \times \log N)$

Решение

```
n = int(input())
a = list(map(int, input().split()))
a.sort()
dp = [[0 for _ in range(2)] for _ in range(n)]
dp[1][0] = a[1] - a[0]
dp[1][1] = a[1] - a[0]
```

```
for i in range(2, n):
    dp[i][1] = min(dp[i - 1][0], dp[i - 1][1]) + (a[i] - a[i - 1])
    dp[i][0] = dp[i - 1][1]
print(dp[n - 1][1])
```

5 E. Горные виды

Разбор

Разбор

Для решения задачи воспользуемся идеей о том, что нам проще сначала посчитать количество всех возможных пейзажей, а потом вычесть плохие.

Для начала поймем сколько у нас вообще существует плохих пейзажей? Для этого воспользуемся определением: пусть высота равнины x, тогда нам нужна полоска длины $W \cdot x$. То есть нам нужно найти количество неотрицательных целых решений неравенства $W \cdot xL \wedge xHx\frac{L}{W} \wedge xH$.

Теперь остается понять как посчитать все пейзажи. Для этого заметим, что наша задача похожа на задачу о рюкзаке, только вместо очередного предмета вы выбираем вес от 0 до H. Если более формально: $\texttt{dp[i][L]} = \sum_{\texttt{j}=0}^{H} \texttt{dp[i-1][L-j]}$. Тогда ответ $\sum_{\texttt{j}=0}^{L} \texttt{dp[n][j]} - \min(H, \lfloor \frac{L}{W} \rfloor)$. Получили решение за $\mathcal{O}(WLH)$

Но если попробовать сдать такое решение на Python'е, то скорее всего вас ждет `Time Limit exceeded`. Дабы его избежать можно а) переписать решение на быстрый язык программирования б) ускорить переход. Для этого заметим, что если закрепить `i`, то нас всегда будет интересовать отрезок длины `H`, при этом, при сдвиге `L` на один, отрезок тоже сдвинется на один, следовательно, можно поддерживать сумму на таком отрезке и сдвигать его при увеличении `L`. Тогда решение будет работать за $\mathcal{O}(WL)$, что уже укладывается в ограничения по времени.

Решение

```
MOD = 10 ** 9 + 7
l, w, h = map(int, input().split())

dp = [[0 for _ in range(l + 1)] for _ in range(w + 1)]
for i in range(l + 1):
    dp[1][i] = 1 if i <= h else 0
for i in range(2, w + 1):
    for j in range(l + 1):
        diff = 0 if j <= h else dp[i - 1][j - h - 1]
        dp[i][j] = (dp[i - 1][j] + dp[i][j - 1] - diff + MOD) % MOD
s = MOD - min(h, l // w)
for i in range(1, l + 1):
    s = (s + dp[-1][i]) % MOD
print(s % MOD)
```

6 F. Лесенки

Разбор

При решении этой задачи нужно было заметить, что когда мы уменьшаем число блоков в самой нижней строке, мы получаем лесенки, ответы для которых мы уже посчитали ранее. Используем эту идею для построения динамики!

- Параметрами динамики будут являться общее число блоков i и число j блоков, лежащих в основании лесенки. В качестве состояния динамики возьмём максимальное число лесенок, которое можно построить, используя i блоков и храня j блоков в основании

- Переходом динамики будет являться переиспользование ранее посчитанных ответов для лесенок – возьмём лесенку из $i - j$ блоков и, перебрав дополнительный параметр k, прибавим её ответы к ответам для лесенки из i блоков
- В качестве ответа возьмём сумму всех значений в n-й строке

Асимптотика решения: $O(N^3)$

Решение

```
n = int(input())

dp = [[0 for _ in range(n + 1)] for _ in range(n + 1)]
dp[0][0] = 1

for num in range(1, n + 1):
    for line1_num in range(1, num + 1):
        for row in range(line1_num):
            dp[num][line1_num] += dp[num - line1_num][row]
print(sum(dp[n]))
```

7 G. Упаковка сокровищ

Разбор

Эта задача похожа на обычный рюкзак за исключением того, что мы имеем как делимые, так и неделимые предметы. Какая может прийти первая мысль? Можно посчитать динамикой вообще все предметы и в оставшийся вес запихнуть только делимые предметы.

Какой недостаток у этого подхода? Некоторые делимые предметы имеют бОльшую удельную ценность, чем неделимые, однако мы этого не узнаем на этапе набора рюкзака. Что можно сделать в этом случае?

1. Посчитаем динамику только для неделимых предметов
2. Для каждого изменения стоимости найдём наименьший вес, при котором можно набрать эту стоимость
3. Для каждой из гипотез будем жадно набирать делимые сокровища в порядке убывания их удельной ценности и обновлять максимальную возможную стоимость

Асимптотика решения: $O(W \times N)$

Решение

```
w, n = map(int, input().split())
items = [None] * n
for i in range(n):
    w_i, c_i, s_i = input().split()
    items[i] = ((int(w_i), int(c_i), s_i == 'Y'))
items.sort(key=lambda i: i[1] / i[0], reverse=True)

dp = [[0 for _ in range(w + 1)] for _ in range(n + 1)]
for i in range(1, n + 1):
    w_i, c_i, s_i = items[i - 1]
    for j in range(w + 1):
        dp[i][j] = dp[i - 1][j]
        if s_i:
            continue
        if j >= w_i:
            dp[i][j] = max(dp[i][j], dp[i - 1][j - w_i] + c_i)

possible_ans = []
best_cost = -1
```

```
for i in range(w + 1):
    if dp[-1][i] > best_cost:
        best_cost = dp[-1][i]
        possible_ans.append((i, best_cost))

ans = 0
for w_p, c_p in possible_ans:
    remain_w, i = w - w_p, 0
    while i < n and remain_w > 0:
        w_i, c_i, s_i = items[i]
        i += 1
        if not s_i:
            continue
        use_w = min(remain_w, w_i)
        remain_w -= use_w
        c_p += use_w * (c_i / w_i)
    ans = max(ans, c_p)
print(ans)
```