Динамическое программирование – 1

1 A. Числа Фибоначчи

Разбор

В этой задаче требовалось найти n-е по счёту число Фибоначчи. Вспомним, что последовательность Фибоначчи определяется как

$$f(n) = f(n-1) + f(n-2), f(0) = 0, f(1) = 1$$

Тогда

- Состоянием динамики в этой задаче является число Фибоначчи на i-м шаге
- Переход динамики: формула для вычисления n-го числа Фибоначчи
- База динамики: $f(0) = 0, f(1) = 1$
- Обход выполняется слева направо по массиву, ответ лежит в n-м элементе

Асимптотика решения: $O(N)$ на запрос
Бонус: сущестует решение за $O(\log N)$

Решение

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

2 B. Без трёх единиц

Разбор

Начать решать эту задачу можно было со следующих рассуждений: введём параметр динамики j, показывающий число единиц в суффиксе. Тогда мы имеем следующие состояния:

- $dp_{i,0}$ - число последовательностей, оканчивающихся на 0
- $dp_{i,1}$ - число последовательностей, оканчивающихся на одну единицу
- $dp_{i,1}$ - число последовательностей, оканчивающихся на две единицы

Заполнив первые несколько строк такой таблицы, можно было заметить, что эта динамика имеет следующие переходы:

- $dp_{i,0} = dp_{i-1,0} + dp_{i-1,0} + dp_{i-1,2}$

- $dp_{i,1} = dp_{i-1,0}$
- $dp_{i,2} = dp_{i-1,1}$

Эти переходы на самом деле эквивалентны одномерной динамике с переходом $dp_i = dp_{i-1} + dp_{i-2} + dp_{i-3}$. База динамики: $dp_0 = 1, dp_1 = 2, dp_2 = 4$

Асимптотика решения: $O(N)$

Решение

```
n = int(input())
dp = [0] * (n + 1)
dp[0] = 1
if n > 0:
    dp[1] = 2
if n > 1:
    dp[2] = 4
for i in range(3, n + 1):
    dp[i] = dp[i - 1] + dp[i - 2] + dp[i - 3]
print(dp[n])
```

3 C. Маршрут максимальной стоимости

Разбор

В этой задаче используется двумерная динамика, описывающаяся следующим образом:

- База динамики: начальное состояние черепашки, то есть, число монет, которое она соберёт, не двигаясь с места - $dp_{0,0} = f_{0,0}$
- Состояние динамики: число монет, которое черепашка соберёт на текущем ходу $dp_{i,j}$
- Переход динамики: черепашка может ходить только вниз и влево, поэтому мы хотим брать максимум из двух предыдущих состояний - $dp_{i,j} = \max(dp_{i-1,j}, dp_{i,j-1}) + f_{i,j}$
- Обход выполняется последовательно по всем клеткам массива, результат находится в правой нижней клетке массива динамики

При решении следовало обратить внимание на крайние случаи при заполнении динамики (например, в крайнем левом столбце и краней верхней строке). Эту проблему можно решать двумя способами - добавлением фиктивных столбца и строки либо предподсчётом значений до запуска основного цикла обхода.

Асимптотика решения: $O(N \times M)$

Решение

```
n, m = map(int, input().split())
f = [None] * n
dp = [[0 for _ in range(m)] for _ in range(n)]
for i in range(n):
    f[i] = list(map(int, input().split()))
dp[0][0] = f[0][0]
for i in range(1, n):
    dp[i][0] = dp[i - 1][0] + f[i][0]
for j in range(1, m):
    dp[0][j] = dp[0][j - 1] + f[0][j]
for i in range(1, n):
```

```
    for j in range(1, m):
        dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]) + f[i][j]
path = []
r, c = n - 1, m - 1
while r > 0 or c > 0:
    if r > 0 and c > 0:
        if dp[r - 1][c] > dp[r][c - 1]:
            r -= 1
            path.append('D')
        else:
            c -= 1
            path.append('R')
    elif r > 0:
        r -= 1
        path.append('D')
    elif c > 0:
        c -= 1
        path.append('R')
path.reverse()
print(dp[n - 1][m - 1])
print(' '.join(path))
```

4 D. Красивый массив

Разбор

При решении этой задачи возникает следующая идея: давайте брать максимальный по модулю подотрезок и умножать его на x, а затем ещё раз искать подотрезок с максимальной суммой. К сожалению, этот подход работает только для положительных x.

Контрпример для неположительного x:

5 0

10 -1 10 10 -2

Тогда давайте решать эту задачу следующим образом: поймём, что какой-то непрерывный отрезок $[i, l]$ принадлежит исходному (неизменённому) массиву, какой-то непрерывный отрезок $[l+1, r]$ принадлежит массиву, домноженному на x, и, наконец, какой-то отрезок $[r+1, j]$ принадлежит снова исходному массиву. Представим состояния как "бутерброд"из трёх массивов: $A|A \times x|A$

Тогда динамика строится очень просто:

- $dp_{i,0} = dp_{i-1,0} + a_i$
- $dp_{i,1} = \max(dp_{i-1,0}, dp_{i-1,1}) + a_i \times x$
- $dp_{i,2} = \max(dp_{i-1,0}, dp_{i-1,1}, dp_{i-1,2}) + a_i$

Однако, если мы реализуем задачу таким образом, нам придётся дополнительно думать, как найти границы i и j подмассива с наибольшей суммой на отрезке. Введём ещё две идеи:

- Нам невыгодно брать какой-то префикс (первые k элементов) с отрицательной суммой. Действительно, если мы отбросим какой-то префикс с отрицательной суммой, мы всегда улучшим ответ. Это избавляет нас от необходимости искать левую границу.
- Заметим, что если в массиве есть суффикс с отрицательной суммой, то мы также можем его отбросить и улучшить ответ. Тогда просто считаем ответом максимум между уже найденным лучшим ответом и состояниями динамики на этом шаге. Это избавляет нас от необходимости искать правую границу.

Асимптотика решения: $O(N)$

Решение

```
n, x = map(int, input().split())
a = list(map(int, input().split()))
dp = [[0 for _ in range(n + 1)] for _ in range(3)]
ans = 0
for i in range(n):
    dp[0][i + 1] = max(0, dp[0][i] + a[i])
    dp[1][i + 1] = max(dp[0][i], dp[1][i]) + a[i] * x
    dp[2][i + 1] = max(dp[0][i], dp[1][i], dp[2][i]) + a[i]
    ans = max(ans, dp[0][i + 1], dp[1][i + 1], dp[2][i + 1])
print(ans)
```

5 E. Покупка билетов

Разбор

В этой задаче динамика описывается следующим образом:

- Состояние динамики: минимальное время, требуемое для обслуживания первых n посетителей
- Переход динамики: минимум среди следующих значений
  - Ответ для шага $n - 1$ и время на обслуживание только n-го посетителей
  - Ответ для шага $n - 2$ и время на обслуживание n-го и $n - 1$-го посетителей
  - Ответ для шага $n - 3$ и время на обслуживание n-го, $n - 1$-го и $n - 2$-го посетителей
- База динамики: посчитанные с помощью перехода ответы для первых трёх посетителей
- Обход выполняется слева направо, ответ лежит в последнем элементе массива динамики

Асимптотика решения: $O(N)$

Решение

```
n = int(input())
a, b, c, dp = [0] * n, [0] * n, [0] * n, [0] * n
for i in range(n):
    a[i], b[i], c[i] = map(int, input().split())
dp[0] = a[0]
if n > 1:
    dp[1] = min(dp[0] + a[1], b[0])
if n > 2:
    dp[2] = min(dp[1] + a[2], dp[0] + b[1], c[0])
for i in range(3, n):
    dp[i] = min(dp[i - 1] + a[i], dp[i - 2] + b[i - 1], dp[i - 3] +
     c[i - 2])
print(dp[n - 1])
```

6 F. Ход конём

Разбор

В этой задаче динамика описывается следующим образом:

- Состояние динамики: число ходов конём, ведущих в клетку (i, j)
- Переход динамики: ответ, полученный для клеток, из которых конь мог прийти - $dp_{i,j} = dp_{i-1,j-2} + dp_{i-2,j-1}$
- База динамики: Конь мог прийти единственным образом в начальную точку (т.к. стоит в ней), а также единственным образом в точки $(1, 2)$ и $(2, 1)$
- Порядок обхода: слева направо и сверху вниз, т.к. конь не может пойти в клетку левее или выше текущей. Ответ находится в правой нижней ячейке.

Сложность решения: $O(N \times M)$

Решение

```
n, m = map(int, input().split())
dp = [[0 for _ in range(m)] for _ in range(n)]
dp[0][0] = 1
if n > 1 and m > 2:
    dp[1][2] = 1
if n > 2 and m > 1:
    dp[2][1] = 1
for i in range(2, n):
    for j in range(2, m):
        dp[i][j] = dp[i - 1][j - 2] + dp[i - 2][j - 1]
print(dp[n - 1][m - 1])
```

7 G. Ход конём - 2

Разбор

В этой задаче динамика описывается почти так же, как и в предыдущей, за исключением одного момента: конь **может** ходить выше или левее предыдущей клетки, поэтому порядок обхода в этой задаче меняется.

Давайте начнём обход состояний динамики 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)
```

8 H. Аппарат Констанции

Разбор

Обратим внимание, что если входная строка содержит символы 'm' или 'w', то ответ равен нулю.

Иначе давайте определим, сколькими способами можно интерпретировать последовательность символов 'n' или 'u' заданной длины:

- 'n': 1
- 'nn': 2 ('nn' или 'm')
- 'nnn': 3 ('nnn', 'mn', 'nm')
- 'nnnn': 5 ('nnnn', 'mnn', 'nmn', 'nnm', 'mm')

Нетрудно убедиться, что если мы продолжим увеличивать число символов, то количество строк, которые можно получить из них на каждом шаге, образует последовательность Фибоначчи.

Тогда мы можем

1. Заранее посчитать все числа Фибоначчи до заданного значения n по модулю m
2. Пройтись по строке и подсчитать длины всех отрезков, содержащих 'n' или 'u' более 1 раза
3. Изначально ответ равен 1
4. Каждой найденной длине отрезка мы можем сопоставить число Фибоначчи, тогда умножим посчитанный на предыдущем шаге ответ на значение числа Фибоначчи и найдём остаток по модулю m

Асимптотика решения: $O(N)$

Решение

```
def fibonacci(n, m):
    f = [0] * max(n, 2)
    f[0] = 2
    f[1] = 3
    for i in range(2, n):
        f[i] = (f[i - 1] + f[i - 2]) % m
    return f


mod = 10 ** 9 + 7
s = input()
if 'm' in s or 'w' in s:
    print(0)
else:
    fib = fibonacci(len(s), mod)
    cnt, n = [], 0
    for i in range(1, len(s)):
        if s[i] == s[i - 1]:
            if s[i] == 'n' or s[i] == 'u':
                n += 1
        else:
            if n > 0:
                cnt.append(n)
                n = 0
    if n > 0:
        cnt.append(n)
    res = 1
    for c in cnt:
        res = (res * fib[c - 1]) % mod
    print(res)
```

9 I. Гвоздики

Разбор

Опишем состояния динамики 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])
```