Динамическое программирование – 2

1 A. НОПроблемо

Разбор

В этой задаче нужно было найти длину наибольшей общей подпоследовательности двух последовательностей a и b. Опишем динамику следующим образом:

- Состояние динамики: длина НОП для префикса последовательности a длиной i и префикса последовательности длины b длиной j
- База динамики: пустые префиксы обеих последовательностей имеют общую длину 0
- Переход динамики: мы можем выбрать максимум из двух ранее рассмотренных состояний: когда префикс последовательности a был короче на единицу и когда префикс последовательности b был короче на единицу, а затем прибавить 1 в том случае, если элементы i и j соответствующих префиксов совпадают
- Обход выполняется сверху вниз слева направо, ответ лежит в состоянии $dp_{n,m}$

Асимптотика решения: $O(N \times M)$

Решение

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

2 B. Расстояние Дамерау-Левенштейна

Разбор

Воспользуемся идеей задачи НОП и 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 $s_i = tj - 1$, 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])
```

3 C. Почти палиндромы

Разбор

Начнём рассуждение с того, что построим динамику для поиска всех подстрок, являющихся подпалиндромами при $K = 0$. В качестве параметров динамики возьмём индекс начала подпалиндрома (его крайнего левого символа) и конца подпалиндрома (его крайнего правого символа). Переход будем делать следующим образом:

$$dp_{i-1,j+1} = 1_{dp_{i,j} \wedge s_{i-1}=s_{j+1}}$$

Тогда ответ будет являться просто суммой по всем $dp_{i,j}$.

Для данного решения нам потребуется хранить квадрат памяти. Прежде чем рассматривать решение для любых K, давайте научимся решать эту задачу за константу памяти. Как это сделать? Заметим, что нам не обязательно поддерживать все состояния – так как мы всё равно тратим квадрат времени на обход, мы можем считать всё онлайн! Будем рассматривать последовательно все подстроки $s_{l..r}$, начиная с некоторого центра s_i для всех i и увеличивать общий счётчик при выполнении ровно того же условия: если $s_{l-1} = s_{r+1}$, то $cnt := cnt + 1$.

Теперь решим исходную задачу. Зададим некоторое $k := K$ и будем действовать по описанному выше принципу. Только в случае, если $s_{l-1} \neq s_{r+1}$, будем всё равно совершать переход и отнимать от k единицу. Если символы не равны и $k = 0$, то считаем, что для данных l, r достигнут максимальный подпалиндром.

Асимптотика решения: $O(N^2)$

Решение

```
n, k = map(int, input().split())
s = input()

cnt = 0
for i in range(n):
    c, l, r = 0, i, i
    while l >= 0 and r < n:
        if s[l] != s[r]:
            c += 1
        if c > k:
            break
        l -= 1
        r += 1
```

```
        cnt += 1
for i in range(1, n):
    c, l, r = 0, i - 1, i
    while l >= 0 and r < n:
        if s[l] != s[r]:
            c += 1
        if c > k:
            break
        l -= 1
        r += 1
        cnt += 1
print(cnt)
```

4 D. Число возрастающих подпоследовательностей

Разбор

В этой задаче нужно было просто посчитать ровно то, что требуется по условию. Будем поддерживать динамику dp_i – в состоянии динамики будем хранить пару: длину максимальной НВП l_i, достижимой в этом элементе, и количество таких НВП c_i. Для того, чтобы совершить переход, посчитаем максимальную длину НВП L, которую мы могли набрать ранее. А затем для всех элементов $l_j : j < i$ и $l_j = L$ выполним два присвоения:

- $c_i := c_i + c_j \pmod M$
- $l_i := L + 1$

Чтобы найти ответ, просто посчитаем сумму по всем c_i для наибольшей возможной длины, набранной за время обхода.

Асимптотика решения: $O(N^2)$

Решение

```
MOD = 10 ** 9 + 7
n = int(input())
a = list(map(int, input().split()))

max_len = 0
dp = [[1] * 2 for _ in range(n)]
for i in range(n):
    has_less, max_len_prev = False, 0
    for j in range(i):
        if a[i] > a[j]:
            dp[i] = [0, 0]
            max_len_prev = max(max_len_prev, dp[j][1])
    for j in range(i):
        if a[i] > a[j] and dp[j][1] == max_len_prev:
            dp[i][0] = (dp[i][0] + dp[j][0]) % MOD
            dp[i][1] = dp[j][1] + 1
    max_len = max(max_len, dp[i][1])
ans = 0
for i in range(n):
    if dp[i][1] == max_len:
        ans = (ans + dp[i][0]) % MOD
print(ans)
```

5 E. Опять сжимаешь, шакал...

Разбор

В этой задаче рассуждения нужно было начинать со следующего: нам дана строка s и нужно посчитать в ней какие-то циклические подстроки, причём эти подстроки могут пересекаться. Что мы можем сделать

со строкой, чтобы попробовать её сжать? Мы можем попытаться разрезать её на две части и проверить, подходит ли какой-то из кусочков для сжатия.

Тогда будем рассматривать подстроки, начинающиеся в l и заканчивающиеся в r. Для данной подстроки $s_{l..r}$ проверим, можем ли мы её сжать – для этого переберём все возможные длины цикла и проверим, является ли строка цикличной. Если да, то возьмём наиболее короткую из двух строк – исходной и после сжатия.

Далее, независимо от того, сократили мы строку сжатием или нет, попробуем найти ответ для других возможных разрезов строки – переберём все возможные точки разреза от l до r и посчитаем рекурсивно ответ для них.

Наконец, чтобы наша рекурсия не работала слишком долго, добавим отсечения. Если мы уже ранее посчитали ответ, то просто вернём $dp_{l,r}$. Если $r - l < 5$, то мы точно не можем улучшить ответ, поэтому вернём $s_{l..r}$

Асимптотика решения: $O(N^3)$

Решение

```
s = input()
n = len(s)

dp = [[None] * (n + 1) for _ in range(n + 1)]

def ok(s, l, r, p):
    t = (r - l) // p
    for i in range(p):
        for j in range(1, t):
            if s[l + p * j + i] != s[l + p * (j - 1) + i]:
                return False
    return True

def solve(s, l, r):
    if dp[l][r]:
        return dp[l][r]
    res, m = s[l:r], r - l
    if m <= 4:
        return res
    res = compress(s, l, r, 1, res)
    for i in range(2, m):
        if m % i != 0:
            continue
        res = compress(s, l, r, i, res)
        res = compress(s, l, r, m // i, res)
    for i in range(l + 1, r):
        sub = solve(s, l, i) + solve(s, i, r)
        if len(sub) < len(res):
            res = sub
    dp[l][r] = res
    return dp[l][r]

def compress(s, l, r, mul, prev):
    m = r - l
    cur = prev
    if ok(s, l, r, mul):
        sub = solve(s, l, l + mul)
        cur = f'{m // mul}({sub})'
    return min(cur, prev, key=lambda x: len(x))

res = solve(s, 0, n)
print(res)
```