

САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №7 по
курсу «Алгоритмы и структуры
данных»

Вариант 7

Выполнила:
Пожидаева Е.Р.

Санкт-Петербург
2024 г.

Оглавление

Задача №3. Редакционное расстояние.....	3
Задача №4. Наибольшая общая подпоследовательность двух последовательностей	7
Задача №5. Наибольшая общая подпоследовательность трех последовательностей	10
Задача №6. Наибольшая возрастающая подпоследовательность	13

Задача №3 . Редакционное расстояние

Текст задачи.

Редакционное расстояние между двумя строками – это минимальное количество операций (вставки, удаления и замены символов) для преобразования одной строки в другую. Это мера сходства двух строк. У редакционного расстояния есть применения, например, в вычислительной биологии, обработке текстов на естественном языке и проверке орфографии. Ваша цель в этой задаче – вычислить расстояние редактирования между двумя строками.

Листинг кода:

```
with open("input.txt", "r") as file:
    s1 = file.readline().strip()
    s2 = file.readline().strip()

def edit_distance(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1)
           for _ in range(m + 1)]
    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i - 1] == s2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = 1 + min(dp[i - 1][j],
                                   dp[i][j - 1], dp[i - 1][j - 1])

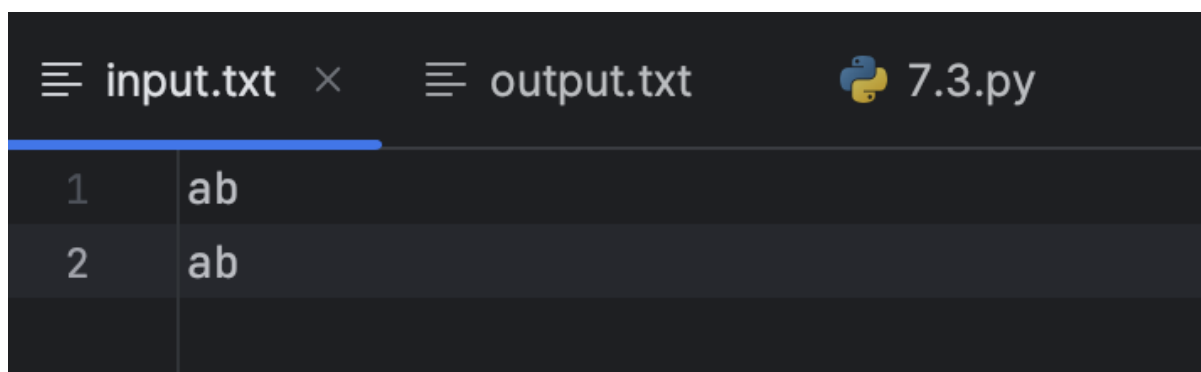
    return dp[m][n]

result = edit_distance(s1, s2)
with open("output.txt", "w") as file:
    file.write(str(result))
```

Текстовое объяснение:

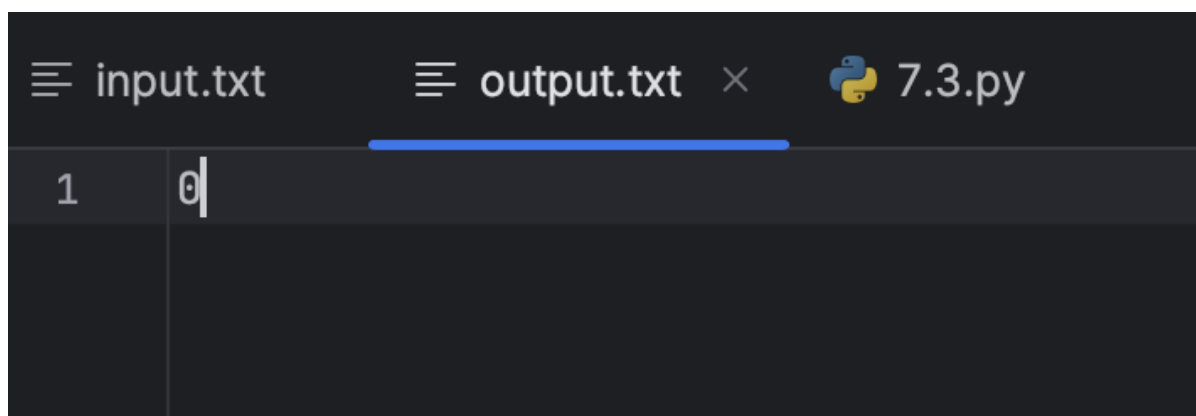
Этот код открывает файл "input.txt" для чтения и считывает из него две строки. Затем он определяет функцию `edit_distance()`, которая вычисляет расстояние между двумя строками `s1` и `s2`. Это расстояние - это минимальное количество операций (вставки символов, удаление символов, замена символов), необходимых для превращения одной строки в другую. Функция `edit_distance()` использует динамическое программирование для заполнения матрицы `dp`, где `dp[i][j]` представляет минимальное расстояние между подстроками `s1[:i]` и `s2[:j]`. Затем она возвращает `dp[m][n]`, где `m` и `n` - длины строк `s1` и `s2` соответственно. Входные данные считываются из файла `input.txt`. Результат записывается в `output.txt`.

Пример работы кода:



The screenshot shows a code editor with two tabs: `input.txt` and `output.txt`. The `input.txt` tab is active and contains two lines of text, both reading "ab".

1	ab
2	ab



The screenshot shows the same code editor with the `output.txt` tab now active. It contains a single line with the character "0".

1	0
---	---

```
input.txt × output.txt 7.3.py
1 short
2 ports|
```

```
input.txt output.txt × 7.3.py
1 3|
```

```
input.txt × output.txt 7.3.py
1 editing
2 distance
```

```
input.txt output.txt × 7.3.py
1 5|
```

	Время выполнения (сек)	Затраты памяти (Мб)
Пример из задачи	0.00052976608276367 19	6021120

Пример из задачи	0.00107693672180175 78	6217728
Пример из задачи	0.00116705894470214 84	6066176

Вывод по задаче: поработала с динамическим
программированием

Задача №4. Наибольшая общая подпоследовательность двух последовательностей

Текст задачи.

4 задача. Наибольшая общая подпоследовательность двух последовательностей

Вычислить длину самой длинной общей подпоследовательности из двух последовательностей.

Даны две последовательности $A = (a_1, a_2, \dots, a_n)$ и $B = (b_1, b_2, \dots, b_m)$, найти длину их самой длинной общей подпоследовательности, т.е. наибольшее неотрицательное целое число p такое, что существуют индексы $1 \leq i_1 < i_2 < \dots < i_p \leq n$ и $1 \leq j_1 < j_2 < \dots < j_p \leq m$ такие, что $a_{i_1} = b_{j_1}, \dots, a_{i_p} = b_{j_p}$.

Листинг кода

```
def longest_common_subsequence(A, B):
    n = len(A)
    m = len(B)
    dp = [[0] * (m+1) for _ in range(n+1)]

    for i in range(1, n+1):
        for j in range(1, m+1):
            if A[i-1] == B[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j],
                                dp[i][j-1])
    return dp[n][m]

with open('input.txt') as f:
    n = int(f.readline())
    A = [int(i) for i in f.readline().split()]
    m = int(f.readline())
    B = [int(i) for i in f.readline().split()]
```

```
p = longest_common_subsequence(A,B)

with open('output.txt', 'w') as f:
    f.write(str(int(p)))
```

Текстовое объяснение:

Функция `longest_common_subsequence(A, B)` принимает два списка `A` и `B` как аргументы и возвращает длину наибольшей общей подпоследовательности. Первым делом, в функции определяются длины списков `A` и `B` - `n` и `m` соответственно. Затем создается двумерный массив `dp` размером $(n+1) \times (m+1)$, заполненный нулями. Этот массив будет использоваться для сохранения результатов подзадач. Затем используются два вложенных цикла для заполнения массива `dp`. На каждой итерации проверяется, равны ли элементы `A[i-1]` и `B[j-1]`. Если да, то `dp[i][j]` равно `dp[i-1][j-1] + 1`, что означает, что длина наибольшей общей подпоследовательности увеличивается на 1. Если элементы не равны, то `dp[i][j]` равно максимуму между `dp[i-1][j]` и `dp[i][j-1]`, что означает, что длина наибольшей общей подпоследовательности не изменяется. Функция возвращает значение `dp[n][m]`, которое содержит длину наибольшей общей подпоследовательности. Входные данные считываются из файла `input.txt`. Результат записывается в `output.txt`.

Пример работы кода:


```
≡ input.txt × ≡ output.txt 7.4.py
1      3
2      2 7 5
3      2
4      2 5
```

```
≡ input.txt ≡ output.txt × 7.4.py
1      2
```

	Время выполнения (сек)	Затраты памяти (Мб)
Пример из задачи	0.0005459785461425	6053888

Задача №5. Наибольшая общая подпоследовательность трех последовательностей

Текст задачи.

Вычислить длину самой длинной общей подпоследовательности из трех последовательностей.

Листинг кода.

```
def longest_common_subsequence_length(A, B, C):
    m = len(A)
    n = len(B)
    l = len(C)

    dp = [[[0 for _ in range(l+1)] for _ in
range(n+1)] for _ in range(m+1)]
    for i in range(1, m+1):
        for j in range(1, n+1):
            for k in range(1, l+1):
                if A[i-1] == B[j-1] and A[i-1] ==
C[k-1]:
                    dp[i][j][k] = dp[i-1][j-1][k-1] +
1
                else:
                    dp[i][j][k] = max(dp[i-1][j][k],
dp[i][j-1][k], dp[i][j][k-1])
    return dp[m][n][l]

with open("input.txt", "r") as file:
    n = int(file.readline())
    A = list(map(int, file.readline().split()))
    m = int(file.readline())
    B = list(map(int, file.readline().split()))
    l = int(file.readline())
    C = list(map(int, file.readline().split()))

result = longest_common_subsequence_length(A, B, C)

with open("output.txt", "w") as file:
    file.write(str(result))
```

Текстовое объяснение решения.

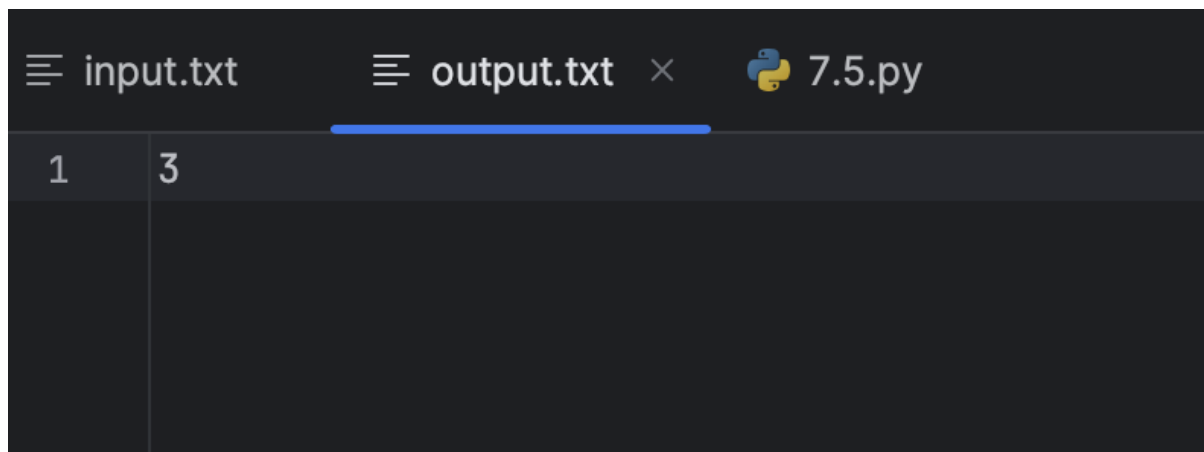
Данный код реализует функцию `longest_common_subsequence_length`, которая вычисляет длину наибольшей общей подпоследовательности трех массивов A, B и C.

Результат работы кода на примерах из текста задачи

input.txt ×		output.txt		7.5.py
1	3			
2	1 2 3			
3	3			
4	2 1 3			
5	3			
6	1 3 5			

input.txt		output.txt ×		7.5.py
1	2			

input.txt ×		output.txt		7.5.py
1	5			
2	8 3 2 1 7			
3	7			
4	8 2 1 3 8 10 7			
5	6			
6	6 8 3 1 4 7			



	Время выполнения (сек)	Затраты памяти (Мб)
Пример из задачи	0.00182914733886718 7	6332416
Пример из задачи	0.00852069139480590 8	6144000

Вывод по задаче: удалось вычислить длину наибольшей общей подпоследовательности трех массивов А, В и С.

Задача №6. Наибольшая возрастающая подпоследовательность

Текст задачи.

Дана последовательность, требуется найти ее наибольшую возрастающую подпоследовательность.

Листинг кода

```
def biggest_subs(a):
    subs = [[] for i in range (len(a))]
    biggest_subs = [a[0]]
    for i in range(len(a)):
        for j in range(i):
            if a[i] > a[j] and len(subs[j]) >
len(subs[i]):
                subs[i] = subs[j].copy()
            subs[i].append(a[i])
            if len(subs[i]) > len(biggest_subs):
                biggest_subs = subs[i]
    return biggest_subs

with open('input.txt', 'r') as f:
    _ = int(f.readline())
    seq = list(map(int, f.readline().split()))
    bis = biggest_subs(seq)
with open('output.txt', 'w') as f:
    f.write(str(len(bis)) + '\n' + ' '.join(map(str,
bis)))
```

Текстовое объяснение решения.

Первая строка во входных данных содержит размер последовательности, который не используется в коде. Вторая строка содержит саму последовательность, которая считывается и преобразуется в список seq с помощью функции map(). Функция biggest_subs(a) создает список subs,

который содержит все возможные подпоследовательности элементов списка `a`. Затем создается список `biggest_subs`, который изначально содержит первый элемент списка `a`. Далее прогоняются все элементы списка `a`. Для каждого элемента `i` проверяется каждый элемент `j` с индексом меньшим `i`. Если элемент `j` меньше элемента `i` и длина подпоследовательности `subs[j]` больше длины подпоследовательности `subs[i]`, то подпоследовательность `subs[j]` копируется в `subs[i]`. Затем копия `subs[i]` расширяется элементом `i`. Если длина полученной подпоследовательности `subs[i]` больше длины текущей наибольшей подпоследовательности `biggest_subs`, то `biggest_subs` обновляется. Затем вызывается функция `biggest_subs(seq)`, которая находит наибольшую возрастающую подпоследовательность в списке `seq` и возвращает ее.

Результат работы кода на примерах из текста задачи

The image shows two screenshots of a code editor interface. The top screenshot shows the 'input.txt' file with the following content:

```
1 6
2 3 29 5 5 28 6
```

The bottom screenshot shows the 'output.txt' file with the following content:

```
1 6
2 3 5 28
```

	Время выполнения (сек)	Затраты памяти (Мб)
Пример из задачи	0.00178933143615722	6053888

Вывод по задаче: Я решила задачу поиска наибольшей возрастающей подпоследовательности в заданной последовательности чисел.

