

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ**

**ОТЧЕТ
по лабораторной работе №3
по дисциплине «Построение и анализ алгоритмов»
Тема: Расстояние Левенштейна. Вариант 10.**

Студент гр. 3343

Иванов П. Д.

Преподаватель

Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы

Нахождения редакционного предписания алгоритмом Вагнера-Фишера.

Задание

№1 - Над строкой ε (будем считать строкой непрерывную последовательность из латинских букв) заданы следующие операции:

$\text{replace}(\varepsilon, a, b)$ – заменить символ a на символ b .

$\text{insert}(\varepsilon, a)$ – вставить в строку символ a (на любую позицию).

$\text{delete}(\varepsilon, b)$ – удалить из строки символ b .

Каждая операция может иметь некоторую цену выполнения (положительное число).

Даны две строки A и B , а также три числа, отвечающие за цену каждой операции. Определите минимальную стоимость операций, которые необходимы для превращения строки A в строку B .

Входные данные: первая строка – три числа: цена операции replace , цена операции insert , цена операции delete ; вторая строка – A ; третья строка – B .

Выходные данные: одно число – минимальная стоимость операций.

Sample Input:

1 1 1

entrance

reenterable

Sample Output:

5

№2 - Над строкой ε (будем считать строкой непрерывную последовательность из латинских букв) заданы следующие операции:

$\text{replace}(\varepsilon, a, b)$ – заменить символ a на символ b .

$\text{insert}(\varepsilon, a)$ – вставить в строку символ a (на любую позицию).

$\text{delete}(\varepsilon, b)$ – удалить из строки символ b .

Каждая операция может иметь некоторую цену выполнения (положительное число).

Даны две строки A и B , а также три числа, отвечающие за цену каждой операции. Определите последовательность операций (редакционное

предписание) с минимальной стоимостью, которые необходимы для превращения строки A в строку B.

Входные данные: первая строка – три числа: цена операции replace, цена операции insert, цена операции delete; вторая строка – A; третья строка – B.

Выходные данные: первая строка – последовательность операций (M – совпадение, ничего делать не надо; R – заменить символ на другой; I – вставить символ на текущую позицию; D – удалить символ из строки); вторая строка – исходная строка A; третья строка – исходная строка B.

Sample Input:

1 1 1

entrance

reenterable

Sample Output:

IMIMMIMMRRM

entrance

reenterable

Модификация: Вар. 10 После выполнения вычислений базовой части пользователь выбирает строку для расширения (первую или вторую) и вводит продолжение этой строки. Требуется вычислить редакционное расстояние с учётом изменившейся строки, используя уже вычисленные ранее таблицы (временная сложность должна зависеть от длины добавления и длины другой строки и не должна зависеть от изначальной длины расширенной строки).

Выполнение работы

Код для решения задачи №1 реализует расстояние Левенштейна с произвольными весами операций: замены (replace), вставки (insert), удаления (delete).

Алгоритм строит матрицу dp размером $(\text{len}(s1) + 1) \times (\text{len}(s2) + 1)$, где $dp[i][j]$ — минимальная стоимость преобразования первых i символов строки $s1$ в первые j символов строки $s2$.

Шаги алгоритма:

Инициализация:

- Первая строка матрицы заполняется как стоимость вставок (от пустой строки к $s2$).
- Первый столбец — стоимость удалений (от $s1$ к пустой строке).

Заполнение матрицы:

- Для каждой ячейки $dp[i][j]$ выбирается минимум из:
- $dp[i-1][j-1] + \text{cost_replace}$ — если заменять $s1[i-1]$ на $s2[j-1]$ (или оставить, если совпадают),
- $dp[i][j-1] + \text{cost_insert}$ — если вставить символ $s2[j-1]$,
- $dp[i-1][j] + \text{cost_delete}$ — если удалить $s1[i-1]$.

Расширение строк (индивидуальное задание):

- Функции `extend_levenshtein_first` и `extend_levenshtein_second` дополняют уже вычисленную матрицу без пересчета всей матрицы:
- При добавлении символов в $s1$ — добавляются новые строки.
- При добавлении символов в $s2$ — добавляются новые столбцы.

Оценка сложности алгоритма

Базовая функция `levenshtein()`:

Время: $O(m \cdot n)$, где $m = \text{len}(s1)$, $n = \text{len}(s2)$ — так как каждая из $m \times n$ ячеек матрицы заполняется за $O(1)$.

Память: $O(m \cdot n)$ требуется полная матрица $m+1 \times n+1$ для хранения промежуточных значений.

Функции расширения (`extend_levenshtein_first`, `extend_levenshtein_second`):

Пусть k — длина добавочной строки (`extension`).

`extend_levenshtein_first` (расширение $s1$ на k символов):

Время: $O(k \cdot n)$

Память: $O(k \cdot n)$

`extend_levenshtein_second` (расширение $s2$ на k символов):

Время: $O(k \cdot m)$

Память: $O(k \cdot m)$

Для выполнения задания №2, был написан код, который решает задачу нахождения редакционного расстояния, а также восстанавливает последовательность операций для преобразования исходной строки в целевую.

Основные этапы:

Инициализация:

- Матрица `dp`: Создается матрица размера $(m+1) \times (n+1)$, где $m = \text{len}(A)$ и $n = \text{len}(B)$.
- Первая строка (индекс 0) заполняется стоимостью вставок (превращение пустой строки в первые j символов строки B).
- Первый столбец (индекс 0) заполняется стоимостью удалений (превращение первых i символов строки A в пустую строку).

Заполнение матрицы:

- Для каждого индекса i (от 1 до m) и j (от 1 до n):
 - **Замена или совпадение:** Если символы $A[i-1]$ и $B[j-1]$ совпадают, то операция замены не требует затрат и используется значение $dp[i-1][j-1]$. Если символы различны, стоимость определяется как $dp[i-1][j-1] + \text{price}[0]$.
 - **Вставка:** Стоимость вычисляется как $dp[i][j-1] + \text{price}[1]$.
 - **Удаление:** Стоимость вычисляется как $dp[i-1][j] + \text{price}[2]$.

- Выбирается минимальное значение из указанных вариантов, и оно записывается в $dp[i][j]$.

Восстановление последовательности операций:

- Функция $backtrace(dp, price, A, B)$:
 - Начинается с нижнего правого угла матрицы ($dp[m][n]$) и осуществляется обратный ход, чтобы восстановить последовательность операций преобразования:
 - Если достигнут первый ряд ($i = 0$), остается лишь вставка оставшихся символов строки B .
 - Если достигнут первый столбец ($j = 0$), остаётся выполнить операции удаления для оставшихся символов строки A .
 - Если символы $A[i-1]$ и $B[j-1]$ равны и стоимость текущей позиции совпадает с $dp[i-1][j-1]$, операция считается совпадением (обозначается символом 'M').
 - Если значение в $dp[i][j]$ получено за счет замены, вставки или удаления, выбирается соответствующая операция (обозначается соответственно 'R', 'I' и 'D').

В процессе обратного прохода формируется последовательность операций, которая затем переворачивается для получения правильного порядка преобразований.

Оценка сложности алгоритма

Время: Полное время работы алгоритма определяется доминирующим этапом заполнения матрицы: $O(m \cdot n)$

Память: Пространственная сложность алгоритма составляет: $O(m \cdot n)$ для хранения матрицы.

Тестирование

Результаты тестирования представлены в таблице 1.

№ п/п	Входные данные	Выходные данные	Комментарии
1.	1 1 1 qwerty qwsdeg	Редакционное расстояние: 4	Алгоритм Вагнера-Фишера. Результат вычислен верно.
2.	5 2 3 abc cbaa	Редакционное расстояние: 12	Алгоритм Вагнера-Фишера. Результат вычислен верно.
3.	100 100 100 asdf asdf	Редакционное расстояние: 0	Алгоритм Вагнера-Фишера. Результат вычислен верно.
4.	1 1 1 wierghwij sooidfhgi	IRRRRRMRMD wierghwij sooidfhgi	Алгоритм Вагнера-Фишера с восстановлением действий. Результат вычислен верно.

Табл. 1. – Результаты тестирования

Выводы

Был реализован алгоритм Вагнера-Фишера для вычисления редакционного предписания, определяя минимальное количество операций (вставки, удаления, замены) для преобразования одной строки в другую.

ПРИЛОЖЕНИЕ А

```
1 - Файл main.py
DEBUG = False # Общий режим отладки
DEBUG_VERBOSE = True # Выводить матрицу после каждой итерации

def print_matrix(matrix, s1, s2):
    print("      ", end="")
    for ch in " " + s2:
        print(f"{ch:>5}", end="")
    print()
    for i in range(len(matrix)):
        ch = " " if i == 0 else s1[i - 1] if i - 1 < len(s1) else " "
        print(f"{ch:>3} |", end="")
        for j in range(len(matrix[0])):
            print(f"{matrix[i][j]:5}", end="")
        print()
    print()

def levenshtein(price, s1, s2):
    # price[0] - replace, price[1] - insert, price[2] - delete
    m, n = len(s1), len(s2)
    matrix = [[0] * (n + 1) for _ in range(m + 1)]

    if DEBUG:
        print("==> Инициализация первой строки (вставки):")
    for j in range(1, n + 1):
        matrix[0][j] = j * price[1]
        if DEBUG:
            print(f" Вставить '{s2[j - 1]}' в позицию 0: {j} * {price[1]}
= {matrix[0][j]}")

    if DEBUG:
        print("\n==> Инициализация первого столбца (удаления):")
    for i in range(1, m + 1):
        matrix[i][0] = i * price[2]
        if DEBUG:
            print(f" Удалить '{s1[i - 1]}' из позиции {i - 1}: {i} *
{price[2]} = {matrix[i][0]}")

    if DEBUG_VERBOSE:
```

```

print("\nНачальная матрица:")
print_matrix(matrix, s1, s2)

for i in range(1, m + 1):
    for j in range(1, n + 1):
        a, b = s1[i - 1], s2[j - 1]
        match_or_replace = 0 if a == b else price[0]
        cost_replace = matrix[i - 1][j - 1] + match_or_replace
        cost_insert = matrix[i][j - 1] + price[1]
        cost_delete = matrix[i - 1][j] + price[2]
        matrix[i][j] = min(cost_replace, cost_insert, cost_delete)

        if DEBUG:
            print(f"\n==> Позиция s1[{i - 1}]='{a}' и s2[{j - 1}]='{b}' (i={i}, j={j}):")
            if a == b:
                print("Символы совпадают")
                print(f"Стоимость: dp[{i - 1}][{j - 1}] = {matrix[i - 1][j - 1]}")
            else:
                print("Символы разные")
                print(f"Стоимость замены: dp[{i - 1}][{j - 1}] + {price[0]} = {cost_replace}")
                print(f"Стоимость вставки : dp[{i}][{j - 1}] + {price[1]} = {cost_insert}")
                print(f"Стоимость удаления : dp[{i - 1}][{j}] + {price[2]} = {cost_delete}")
                print(f"--> Выбрано минимальное значение: {matrix[i][j]}")

        if DEBUG_VERBOSE:
            print("\nТекущая матрица:")
            print_matrix(matrix, s1, s2)

    if DEBUG:
        print("\n==> Финальная матрица:")
        print_matrix(matrix, s1, s2)

return matrix[m][n], matrix

def extend_levenshtein_first(matrix, price, s1, s2, extension):
    """

```

Расширение первой строки. Предполагается, что matrix уже содержит матрицу расстояний для старой s1.

После расширения s1 = s1 + extension, вычисляем только новые строки.

Временная сложность: $O(|\text{extension}| * \text{len}(s2))$

```
"""
old_m = len(s1)
s1_extended = s1 + extension
new_m = len(s1_extended)
n = len(s2)

if DEBUG:
    print(f"\n==> Расширяем первую строку '{s1}' на: '{extension}'")

# Для новых строк создаём строки в матрицы
for i in range(old_m + 1, new_m + 1):
    # Вычисляем dp[i][0]: удаление всех символов s1_extended[0:i]
    new_row = [0] * (n + 1)
    new_row[0] = i * price[2]
    for j in range(1, n + 1):
        a = s1_extended[i - 1]
        b = s2[j - 1]
        cost_replace = matrix[i - 1][j - 1] + (0 if a == b else
price[0])

        cost_insert = new_row[j - 1] + price[1]
        cost_delete = matrix[i - 1][j] + price[2]
        new_row[j] = min(cost_replace, cost_insert, cost_delete)

    if DEBUG:
        print(f"\n==> Расширение: новая позиция s1[{i - 1}]='{a}'
и s2[{j - 1}]='{b}' (i={i}, j={j}):")
        if a == b:
            print(" Символы совпадают")
            print(f" Стоимость: dp[{i - 1}][{j - 1}] = {matrix[i
- 1][j - 1]}")
        else:
            print(f" Стоимость замены: dp[{i - 1}][{j - 1}] +
{price[0]} = {cost_replace}")
            print(f" Стоимость вставки: dp[{i}][{j - 1}] + {price[1]}
= {cost_insert}")
            print(f" Стоимость удаления: dp[{i - 1}][{j}] + {price[2]}
= {cost_delete}")
            print(f" --> Выбрано: {new_row[j]}")
    matrix.append(new_row)
```

```

if DEBUG:
    print("\n==> Финальная матрица после расширения первой строки:")
    print_matrix(matrix, s1_extended, s2)
    return matrix[new_m][n], s1_extended

def extend_levenshtein_second(matrix, price, s1, s2, extension):
    """
    Расширение второй строки. Предполагается, что matrix уже содержит
    матрицу расстояний для старой s2.
    После расширения s2 = s2 + extension, вычисляем только новые столбцы.
    Временная сложность: O( |extension| * len(s1) )
    """
    old_n = len(s2)
    s2_extended = s2 + extension
    new_n = len(s2_extended)
    m = len(s1)

    if DEBUG:
        print(f"\n==> Расширяем вторую '{s2}' строку на: '{extension}'")

    # Добавляем новые столбцы ко всем строкам матрицы
    for i in range(m + 1):
        # Для строки 0, базовая инициализация, если i == 0: dp[0][j] = j *
price[1]
        for j in range(old_n + 1, new_n + 1):
            if i == 0:
                val = j * price[1]
                matrix[0].append(val)
            else:
                a = s1[i - 1]
                b = s2_extended[j - 1]
                cost_replace = matrix[i - 1][j - 1] + (0 if a == b else
price[0])

                cost_insert = matrix[i][j - 1] + price[1]
                cost_delete = matrix[i - 1][j] + price[2]
                matrix[i].append(min(cost_replace, cost_insert,
cost_delete))

    if DEBUG:
        print(f"\n==> Расширение: новая позиция s1[{i -
1}]='{a}' и s2[{j - 1}]='{b}' (i={i}, j={j}):")

```

```

        if a == b:
            print(" Символы совпадают")
            print(f"    Стоимость: dp[{i - 1}][{j - 1}] =
{matrix[i - 1][j - 1]}")
        else:
            print(f"    Стоимость замены: dp[{i - 1}][{j - 1}] +
{price[0]} = {cost_replace}")
            print(f"    Стоимость вставки: dp[{i}][{j - 1}] +
{price[1]} = {cost_insert}")
            print(f"    Стоимость удаления: dp[{i - 1}][{j}] +
{price[2]} = {cost_delete}")
            print(f" --> Выбрано: {matrix[i][j]}")

    if DEBUG:
        print("\n==> Финальная матрица после расширения второй строки:")
        print_matrix(matrix, s1, s2_extended)
    return matrix[m][new_n], s2_extended

if __name__ == '__main__':
    if not DEBUG:
        DEBUG_VERBOSE = False

    price = list(map(int, input().split()))
    s1 = input().strip()
    s2 = input().strip()

    dist, matrix = levenshtein(price, s1, s2)
    print(f"\nРедакционное расстояние: {dist}")

    print("\nВыберите строку для расширения:")
    print("1 - расширить первую строку")
    print("2 - расширить вторую строку")
    print("0 - не расширять")
    choice = input("Ваш выбор: ").strip()

    if choice == "1":
        extension = input("Введите продолжение первой строки: ")
        new_dist, s1_extended = extend_levenshtein_first(matrix, price, s1,
s2, extension)
        print(f"\nНовое редакционное расстояние для '{s1_extended}' и
'{s2}': {new_dist}")
    elif choice == "2":

```

```

        extension = input("Введите продолжение второй строки: ")
        new_dist, s2_extended = extend_levenshtein_second(matrix, price,
s1, s2, extension)
        print(f"\nНовое редакционное расстояние для '{s1}' и
'{s2_extended}': {new_dist}")
    else:
        print("\nРасширение не выполнено.")

```

2 - Файл back.py

DEBUG = False

```

def print_matrix(matrix, A, B):
    m, n = len(matrix), len(matrix[0])
    header = " " + " ".join([" "] + list(B))
    print(header)
    for i in range(m):
        label = " " if i == 0 else A[i - 1]
        row = " ".join(f"{matrix[i][j]:2}" for j in range(n))
        print(f"{label:>3} | {row}")
    print()

```

```

def compute_dp(price, A, B):
    m, n = len(A), len(B)
    matrix = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        matrix[i][0] = i * price[2]
        if DEBUG:
            print(f"Инициализация dp[{i}][0] = {matrix[i][0]} (удаление
символа '{A[i - 1]}')")

    for j in range(1, n + 1):
        matrix[0][j] = j * price[1]
        if DEBUG:
            print(f"Инициализация dp[0][{j}] = {matrix[0][j]} (вставка
символа '{B[j - 1]}')")

    if DEBUG:
        print("\nНачальная матрица DP:")
        print_matrix(matrix, A, B)

```

```

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            cost_replace = matrix[i - 1][j - 1] + (0 if A[i - 1] == B[j - 1] else price[0])

            cost_insert = matrix[i][j - 1] + price[1]
            cost_delete = matrix[i - 1][j] + price[2]

            matrix[i][j] = min(cost_replace, cost_insert, cost_delete)

            if DEBUG:
                print(f"\nВычисляем dp[{i}][{j}] для A[{i - 1}]='{A[i - 1]}' и B[{j - 1}]='{B[j - 1]}' :",
                      f"      Заменить      = {cost_replace},      Вставить      = {cost_insert},
Удалить = {cost_delete}    => dp[{i}][{j}] = {matrix[i][j]}")

        if DEBUG:
            print("\nИтоговая матрица DP:")
            print_matrix(matrix, A, B)

    return matrix

def backtrace(dp, price, A, B):
    i, j = len(A), len(B)
    ops = []
    if DEBUG:
        print("\nОбратный ход по матрице для восстановления операций:")
    while i > 0 or j > 0:
        if DEBUG:
            print(f"\nНа позиции dp[{i}][{j}], текущая стоимость: {dp[i][j]}")

        if i == 0:
            ops.append('I')
            if DEBUG:
                print(f"(I) Вставляем символ '{B[j - 1]}' (i={i}, j={j})")
            j -= 1
        elif j == 0:
            ops.append('D')
            if DEBUG:
                print(f"(D) Удаляем символ '{A[i - 1]}' (i={i}, j={j})")
            i -= 1
        elif A[i - 1] == B[j - 1] and dp[i][j] == dp[i - 1][j - 1]:

```



```

ops.append('M')
if DEBUG:
    print(f"(M) Символы равны: A[{i - 1}]='{A[i - 1]}' и B[{j - 1}]='{B[j - 1]}'")
    i -= 1
    j -= 1
elif dp[i][j] == dp[i - 1][j - 1] + price[0]:
    ops.append('R')
    if DEBUG:
        print(f"(R) Заменяем A[{i - 1}]='{A[i - 1]}' на B[{j - 1}]='{B[j - 1]}'")
        i -= 1
        j -= 1
elif dp[i][j] == dp[i][j - 1] + price[1]:
    ops.append('I')
    if DEBUG:
        print(f"(I) Вставляем символ '{B[j - 1]}'")
        j -= 1
elif dp[i][j] == dp[i - 1][j] + price[2]:
    ops.append('D')
    if DEBUG:
        print(f"(D) Удаляем символ '{A[i - 1]}'")
        i -= 1

ops.reverse()
if DEBUG:
    print("\nПоследовательность операций:", "".join(ops))
return "".join(ops)

if __name__ == '__main__':
    price = list(map(int, input().split()))
    A = input().strip()
    B = input().strip()

    dp = compute_dp(price, A, B)
    ops = backtrace(dp, price, A, B)

    print(ops)
    print(A)
    print(B)

```