МИНОБРНАУКИ РОССИИ
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: Редакционное расстояние
Вариант 14а.

Студент гр. 3388 _____ Лексин М.В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

Цель работы.

Разработка и реализация эффективного алгоритма нахождения редакционного расстояния и предписания для решения задач нахождения редакционного расстояния, редакционного предписания и расстояния Левенштейна.

Задание.

Над строкой ε (будем считать строкой непрерывную последовательность из латинских букв) заданы следующие операции:

1. *replace(ε, a, b)* – заменить символ *a* на символ *b*.
2. *insert(ε, a)* – вставить в строку символ *a* (на любую позицию).
3. *delete(ε, b)* – удалить из строки символ *b*.

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

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

Задание 1:

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*.

Выходные данные: одно число – минимальная стоимость операций.

Sample Input:

1 1 1
entrance
reenterable

Sample Output:

5

Задание 2:

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*.

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

Задание 3:

Расстоянием Левенштейна назовём минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.
Разработайте программу, осуществляющую поиск расстояния Левенштейна между двумя строками.

Пример:

Для строк pedestal и stien расстояние Левенштейна равно 7:

- Сначала нужно совершить четыре операции удаления символа: pedestal -> stal.
- Затем необходимо заменить два последних символа: stal -> stie.
- Потом нужно добавить символ в конец строки: stie -> stien.

Параметры входных данных:

Первая строка входных данных содержит строку из строчных латинских букв. (S, $1 \leq |S| \leq 2550$).

Вторая строка входных данных содержит строку из строчных латинских букв. (T, $1 \leq |T| \leq 2550$).

Параметры выходных данных:

Одно число L, равное расстоянию Левенштейна между строками S и T.

Sample Input:

pedestal

stien

Sample Output:

7

Вариант 14а:

Методом динамического программирования вычислить длину наибольшей общей подпоследовательности двух строк.

Описание алгоритма.

Алгоритм должен определить минимальную стоимость и оптимальную последовательность операций редактирования, необходимых для

преобразования одной строки в другую. Операции включают замену символа, вставку символа и удаление символа, причём каждая может иметь свою стоимость.

Для решения этой задачи используется метод динамического программирования, который обеспечивает оптимальное решение.

Основные этапы алгоритма:

1. Инициализация матрицы DP (динамического программирования). Создаётся двумерный массив dp размером (m+1)*(n+1), где m и n – длины исходной и целевой строк соответственно, dp[i][j] представляет минимальную стоимость преобразования первых i символов начальной строки в первые j символов конечной строки.
    - Базовый случай: dp[0][0] = 0 (пустая строка в пустую)
    - Для i от 1 до m: dp[i][0] = dp[i-1][0] + стоимость удаления (стоимость удаления i символов)
    - Для j от 1 до n: dp[0][j] = dp[0][j-1] + стоимость вставки (стоимость вставки j символов)
2. Заполнение матрицы DP. Для каждой ячейки dp[i][j] рассматриваются три возможные операции:
    - Если A[i-1] == B[j-1] (символы совпадают): dp[i][j] = dp[i-1][j-1] (стоимость не увеличивается)
    - Иначе вычисляем стоимость для трёх вариантов:
    - Замена: dp[i-1][j-1] + стоимость замены
    - Удаление: dp[i-1][j] + стоимость удаления
    - Вставка: dp[i][j-1] + стоимость вставки
    - Выбираем минимальную из этих трёх стоимостей
3. Получение результата. Значение dp[m][n] содержит минимальную стоимость преобразования всей исходной строки A в целевую строку B.
4. Восстановление последовательности операций (для второго задания). Используется обратный проход от позиции dp[m][n] к dp[0][0]:

- Если A[i-1] == B[j-1]: добавляем операцию ‘M’ (совпадение) и идём к dp[i-1][j-1]
- Иначе выбираем операцию с минимальной стоимостью
- Если выбрана замена: добавляем ‘R’ и идём к dp[i-1][j-1]
- Если выбрана вставка: добавляем ‘I’ и идём к dp[i][j-1]
- Если выбрано удаление: добавляем ‘D’ и идём к dp[i-1][j]
- Поскольку проход осуществляется с конца, последовательность операций переворачивается.

5. Расстояние Левенштейна (третье задание). Является частным случаем редакционного расстояния, где все операции имеют одинаковую стоимость, равную 1.

Помимо основного алгоритма, для каждого из трёх заданий реализована дополнительная функциональность – нахождение наибольшей общей подпоследовательности (НОП) двух строк. Эта задача также решается методом динамического программирования.

Основные этапы алгоритма нахождения НОП:

1. Инициализация матрицы LCS (НОП). Создаётся двумерный массив lcs размером (m+1)*(n+1), где lcs[i][j] представляет длину наибольшей общей подпоследовательности для первых i символов начальной строки A и первых j символов целевой строки B. Начальные значения – 0.
2. Заполнение матрицы LCS. Для каждой ячейки lcs[i][j]:
    - Если A[i-1] == B[j-1] (символы совпадают): lcs[i][j] = lcs[i-1][j-1] + 1 (добавляем 1 к длине НОП)
    - Иначе: lcs[i][j] = max(lcs[i-1][j], lcs[i][j-1]) (берём максимум из двух вариантов – пропуск символа в первой или второй строке)
3. Получение результата. Значение lcs[m][n] содержит длину наибольшей общей подпоследовательности для строк A и B.
4. Восстановление НОП (опционально). Используется обратный проход от позиции lcs[m][n] к lcs[0][0]:

- Если A[i-1] == B[j-1]: добавляем символ в НОП и идём в lcs[i-1][j-1]
- Иначе идём в направлении максимального значения: либо к lcs[i-1][j], либо к lcs[i][j-1]
- Поскольку проход осуществляется с конца, последовательность символов переворачивается

Оценка сложности.

Временная сложность алгоритмов определяется процессом заполнения динамических матриц и восстановления решений:

Алгоритм нахождения редакционного расстояния.

- Основная операция – заполнение матрицы dp размером (m+1)*(n+1), где m и n – длины исходной и целевой строк:
- Для каждой ячейки dp[i][j] требуется константное время на сравнение символов и выбор минимума из трёх вариантов
- Общее количество ячеек: (m+1)*(n+1). Соответственно, временная сложность составляет O(m*n)

Алгоритм построения редакционного предписания. Включает два этапа:

- Заполнение матрицы dp: O(m*n)
- Восстановление последовательности операций через обратный проход: в худшем случае требуется пройти от позиции (m, n) до (0, 0), что занимает O(m+n) операций
- Итоговая временная сложность определяется первым этапом и составляет O(m*n)

Алгоритм нахождения расстояния Левенштейна. Является частным случаем алгоритма редакционного расстояния:

- Временная сложность идентична: O(m*n)

Алгоритм нахождения наибольшей общей подпоследовательности.

- Аналогично заполняется матрица lcs размером (m+1)*(n+1)
- Для каждой ячейки требуется константное время на сравнение и выбор максимума
- Временная сложность O(m*n)

Пространственная сложность определяется размерами используемых матриц и дополнительных структур данных:

- Матрица dp для редакционного расстояния и редакционного предписания:
- Размер матрицы: (m+1)*(n+1) ячеек. Пространственная сложность O(m*n)
- Матрица lcs для наибольшей общей подпоследовательности:
- Размер матрицы: (m+1)*(n+1) ячеек. Пространственная сложность O(m*n)
- Хранение редакционного предписания: в худшем случае длина предписания составляет O(max(m, n)), если каждый символ требует отдельной операции. Дополнительная пространственная сложность: O(max(m, n))
- Таким образом, общая пространственная сложность алгоритмов составляет O(m*n).

Тестирование.

Алгоритм был протестирован на различных наборах входных данных. Тестирование первой задачи:

Табл.1

Входные данные	Выходные данные
1 1 1	=== ИТОГОВЫЙ РЕЗУЛЬТАТ === 0 Длина наибольшей общей подпоследовательности: 0

1 1 1 abc	=== ИТОГОВЫЙ РЕЗУЛЬТАТ === 3 Длина наибольшей общей подпоследовательности: 0
1 1 1 abcde abcde	=== ИТОГОВЫЙ РЕЗУЛЬТАТ === 0 Длина наибольшей общей подпоследовательности: 5
3 1 4 a abcde	=== ИТОГОВЫЙ РЕЗУЛЬТАТ === 4 Длина наибольшей общей подпоследовательности: 1
3 1 2 algorithm logarithm	=== ИТОГОВЫЙ РЕЗУЛЬТАТ === 6 Длина наибольшей общей подпоследовательности: 7

Тестирование второй задачи:

Табл.2

Входные данные	Выходные данные
1 1 1	=== ИТОГОВЫЙ РЕЗУЛЬТАТ === Длина наибольшей общей подпоследовательности: 0
1 1 1 abc	=== ИТОГОВЫЙ РЕЗУЛЬТАТ === III abc Длина наибольшей общей подпоследовательности: 0
1 1 1 abcde abcde	=== ИТОГОВЫЙ РЕЗУЛЬТАТ === MMMMM abcde abcde Длина наибольшей общей подпоследовательности: 5
1 1 1 cat bat	=== ИТОГОВЫЙ РЕЗУЛЬТАТ === RMM cat bat Длина наибольшей общей подпоследовательности: 2
2 1 1 algorithm logarithm	=== ИТОГОВЫЙ РЕЗУЛЬТАТ === DMIMRMMMMM algorithm logarithm Длина наибольшей общей подпоследовательности: 7

Тестирование третьей задачи:

Табл.3

Входные данные	Выходные данные
	=== ИТОГОВЫЙ РЕЗУЛЬТАТ === 0 Длина наибольшей общей подпоследовательности: 0
abcde	=== ИТОГОВЫЙ РЕЗУЛЬТАТ === 5 Длина наибольшей общей подпоследовательности: 0
hello hello	=== ИТОГОВЫЙ РЕЗУЛЬТАТ === 0 Длина наибольшей общей подпоследовательности: 5
qwerty asdfgh	=== ИТОГОВЫЙ РЕЗУЛЬТАТ === 6 Длина наибольшей общей подпоследовательности: 0
apple app	=== ИТОГОВЫЙ РЕЗУЛЬТАТ === 2 Длина наибольшей общей подпоследовательности: 3

Выводы.

В ходе выполнения лабораторной работы реализован алгоритм нахождения редакционного расстояния и предписания для решения задач нахождения редакционного расстояния, редакционного предписания и расстояния Левенштейна.

ПРИЛОЖЕНИЕ

Код.

Файл: main1.py

```
def minCostTransform(stringA, stringB, replaceCost, insertCost, deleteCost,
debug=False):
    m, n = len(stringA), len(stringB)
    dpTable = [[0] * (n + 1) for _ in range(m + 1)]

    if debug:
        print("\n=== ИНИЦИАЛИЗАЦИЯ ТАБЛИЦЫ DP ===")
        print("Строка A:", stringA)
        print("Строка B:", stringB)
        print("Стоимость операций: Замена={}, Вставка={}, Удаление={}".format(
            replaceCost, insertCost, deleteCost))

    for i in range(1, m + 1):
        dpTable[i][0] = dpTable[i-1][0] + deleteCost
        if debug:
            print(f"Инициализация: dpTable[{i}][0] = {dpTable[i][0]} (удаление
символа '{stringA[i-1]}')")

    for j in range(1, n + 1):
        dpTable[0][j] = dpTable[0][j-1] + insertCost
        if debug:
            print(f"Инициализация: dpTable[0][{j}] = {dpTable[0][j]} (вставка
символа '{stringB[j-1]}')")

    if debug:
        print("\n=== ЗАПОЛНЕНИЕ ТАБЛИЦЫ DP ===")

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if stringA[i-1] == stringB[j-1]:
                dpTable[i][j] = dpTable[i-1][j-1]
                if debug:
                    print(f"Символы '{stringA[i-1]}' и '{stringB[j-1]}'
совпадают: dpTable[{i}][{j}] = {dpTable[i][j]}")
            else:
                replaceOption = dpTable[i-1][j-1] + replaceCost
                deleteOption = dpTable[i-1][j] + deleteCost
                insertOption = dpTable[i][j-1] + insertCost

                dpTable[i][j] = min(replaceOption, deleteOption, insertOption)

                if debug:
                    print(f"Символы '{stringA[i-1]}' и '{stringB[j-1]}' не
совпадают:")
                    print(f"  Вариант замены: {dpTable[i-1][j-1]} +
{replaceCost} = {replaceOption}")
                    print(f"  Вариант удаления: {dpTable[i-1][j]} + {deleteCost}
= {deleteOption}")
                    print(f"  Вариант вставки: {dpTable[i][j-1]} + {insertCost}
= {insertOption}")
                    print(f"  Выбран минимальный вариант: dpTable[{i}][{j}] =
{dpTable[i][j]}")

    if debug:
        print(f"\nМинимальная стоимость преобразования: {dpTable[m][n]}")
```

```
    return dpTable[m][n]


def findLcsLength(stringA, stringB, debug=False):
    """Функция для нахождения длины наибольшей общей подпоследовательности"""
    m, n = len(stringA), len(stringB)
    lcsTable = [[0] * (n + 1) for _ in range(m + 1)]

    if debug:
        print("\n=== АЛГОРИТМ НАХОЖДЕНИЯ НАИБОЛЬШЕЙ ОБЩЕЙ ПОДПОСЛЕДОВАТЕЛЬНОСТИ
===")
        print("Строка A:", stringA)
        print("Строка B:", stringB)

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if stringA[i-1] == stringB[j-1]:
                lcsTable[i][j] = lcsTable[i-1][j-1] + 1
                if debug:
                    print(f"Символы '{stringA[i-1]}' и '{stringB[j-1]}'
совпадают: lcsTable[{i}][{j}] = {lcsTable[i][j]}")
            else:
                lcsTable[i][j] = max(lcsTable[i-1][j], lcsTable[i][j-1])
                if debug:
                    print(f"Символы '{stringA[i-1]}' и '{stringB[j-1]}' не
совпадают:")
                    print(f"  Макс. из: lcsTable[{i-1}][{j}]={lcsTable[i-1][j]}
и lcsTable[{i}][{j-1}]={lcsTable[i][j-1]}")
                    print(f"  Результат: lcsTable[{i}][{j}] = {lcsTable[i][j]}")

    if debug:

        i, j = m, n
        lcs = []

        print("\n=== ВОССТАНОВЛЕНИЕ НОП ===")
        while i > 0 and j > 0:
            if stringA[i-1] == stringB[j-1]:
                lcs.append(stringA[i-1])
                print(f"Символы совпадают: '{stringA[i-1]}' добавляем в НОП")
                i -= 1
                j -= 1
            elif lcsTable[i-1][j] >= lcsTable[i][j-1]:
                print(f"Пропускаем символ '{stringA[i-1]}' в строке A")
                i -= 1
            else:
                print(f"Пропускаем символ '{stringB[j-1]}' в строке B")
                j -= 1

        lcs.reverse()
        print(f"\nНайденная НОП: {''.join(lcs)}")
        print(f"Длина НОП: {lcsTable[m][n]}")

    return lcsTable[m][n]


def main():
    costs = list(map(int, input().split()))
    replaceCost, insertCost, deleteCost = costs

    StringA = input().strip()
    StringB = input().strip()

    debug = True
```

```
    result = minCostTransform(StringA, StringB, replaceCost, insertCost,
deleteCost, debug)

    lcsLength = findLcsLength(StringA, StringB, debug)

    print("\n=== ИТОГОВЫЙ РЕЗУЛЬТАТ ===")
    print(result)
    print(f"Длина наибольшей общей подпоследовательности: {lcsLength}")


if __name__ == "__main__":
    main()
```

Файл: main2.py

```
def minCostTransform(StringA, StringB, replaceCost, insertCost, deleteCost,
debug=False):
    m, n = len(StringA), len(StringB)

    dpTable = [[0] * (n + 1) for _ in range(m + 1)]

    if debug:
        print("\n=== ИНИЦИАЛИЗАЦИЯ ТАБЛИЦЫ DP ===")
        print("Исходная строка:", StringA)
        print("Целевая строка:", StringB)
        print("Стоимость операций: Замена={}, Вставка={}, Удаление={}".format(
            replaceCost, insertCost, deleteCost))

    for i in range(1, m + 1):
        dpTable[i][0] = dpTable[i-1][0] + deleteCost
        if debug:
            print(f"Инициализация: dpTable[{i}][0] = {dpTable[i][0]} (удаление
символа '{StringA[i-1]}')")

    for j in range(1, n + 1):
        dpTable[0][j] = dpTable[0][j-1] + insertCost
        if debug:
            print(f"Инициализация: dpTable[0][{j}] = {dpTable[0][j]} (вставка
символа '{StringB[j-1]}')")

    if debug:
        print("\n=== ЗАПОЛНЕНИЕ ТАБЛИЦЫ DP ===")

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if StringA[i-1] == StringB[j-1]:
                dpTable[i][j] = dpTable[i-1][j-1]
                if debug:
                    print(f"Символы '{StringA[i-1]}' и '{StringB[j-1]}'
совпадают: dpTable[{i}][{j}] = {dpTable[i][j]}")
            else:
                replaceOption = dpTable[i-1][j-1] + replaceCost
                deleteOption = dpTable[i-1][j] + deleteCost
                insertOption = dpTable[i][j-1] + insertCost

                dpTable[i][j] = min(replaceOption, deleteOption, insertOption)

                if debug:
                    print(f"Символы '{StringA[i-1]}' и '{StringB[j-1]}' не
совпадают:")
                    print(f"  Вариант замены: {dpTable[i-1][j-1]} +
{replaceCost} = {replaceOption}")
```

```
                    print(f"  Вариант удаления: {dpTable[i-1][j]} + {deleteCost}
= {deleteOption}")
                    print(f"  Вариант вставки: {dpTable[i][j-1]} + {insertCost}
= {insertOption}")
                    print(f"  Выбран минимальный вариант: dpTable[{i}][{j}] =
{dpTable[i][j]}")

    operations = []
    i, j = m, n

    if debug:
        print("\n=== ОБРАТНЫЙ ПРОХОД ДЛЯ НАХОЖДЕНИЯ ПОСЛЕДОВАТЕЛЬНОСТИ ОПЕРАЦИЙ
===")
        print(f"Начинаем с позиции ({i}, {j})")

    while i > 0 or j > 0:
        if i > 0 and j > 0 and StringA[i-1] == StringB[j-1]:
            operations.append('M')
            if debug:
                print(f"Символы '{StringA[i-1]}' и '{StringB[j-1]}' совпадают:
операция 'M' (совпадение)")
                print(f"Переходим к позиции ({i-1}, {j-1})")
            i -= 1
            j -= 1
        elif i > 0 and j > 0 and dpTable[i][j] == dpTable[i-1][j-1] +
replaceCost:
            operations.append('R')
            if debug:
                print(f"Заменяем символ '{StringA[i-1]}' на '{StringB[j-1]}':
операция 'R' (замена)")
                print(f"Переходим к позиции ({i-1}, {j-1})")
            i -= 1
            j -= 1
        elif j > 0 and dpTable[i][j] == dpTable[i][j-1] + insertCost:
            operations.append('I')
            if debug:
                print(f"Вставляем символ '{StringB[j-1]}': операция 'I'
(вставка)")
                print(f"Переходим к позиции ({i}, {j-1})")
            j -= 1
        elif i > 0 and dpTable[i][j] == dpTable[i-1][j] + deleteCost:
            operations.append('D')
            if debug:
                print(f"Удаляем символ '{StringA[i-1]}': операция 'D'
(удаление)")
                print(f"Переходим к позиции ({i-1}, {j})")
            i -= 1
        else:
            if debug:
                print("Что-то пошло не так, выходим из цикла")
            break

    operations.reverse()
    operationsString = ''.join(operations)

    if debug:
        print("\n=== РЕЗУЛЬТАТ ===")
        print(f"Последовательность операций: {operationsString}")
        print(f"Исходная строка: {StringA}")
        print(f"Целевая строка: {StringB}")

        print("\n=== ВИЗУАЛИЗАЦИЯ ПРОЦЕССА ПРЕОБРАЗОВАНИЯ ===")
        currentString = ""
        posA = 0
        posB = 0
```

```
        for op in operationsString:
            if op == 'M':
                currentString += StringA[posA]
                print(f"Совпадение: '{StringA[posA]}' остаётся без изменений")
                posA += 1
                posB += 1
            elif op == 'R':
                currentString += StringB[posB]
                print(f"Замена: '{StringA[posA]}' -> '{StringB[posB]}'")
                posA += 1
                posB += 1
            elif op == 'I':
                currentString += StringB[posB]
                print(f"Вставка: добавляем '{StringB[posB]}'")
                posB += 1
            elif op == 'D':
                print(f"Удаление: удаляем '{StringA[posA]}'")
                posA += 1

            print(f"Текущая строка: {currentString}")

        print(f"\nИтоговая строка: {currentString}")
        print(f"Целевая строка:  {StringB}")
        assert currentString == StringB, "Ошибка в преобразовании!"

    return operationsString


def findLcsLength(StringA, StringB, debug=False):
    """Функция для нахождения длины наибольшей общей подпоследовательности"""
    m, n = len(StringA), len(StringB)
    lcsTable = [[0] * (n + 1) for _ in range(m + 1)]

    if debug:
        print("\n=== АЛГОРИТМ НАХОЖДЕНИЯ НАИБОЛЬШЕЙ ОБЩЕЙ ПОДПОСЛЕДОВАТЕЛЬНОСТИ
===")
        print("Строка A:", StringA)
        print("Строка B:", StringB)

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if StringA[i-1] == StringB[j-1]:
                lcsTable[i][j] = lcsTable[i-1][j-1] + 1
                if debug:
                    print(f"Символы '{StringA[i-1]}' и '{StringB[j-1]}'
совпадают: lcsTable[{i}][{j}] = {lcsTable[i][j]}")
            else:
                lcsTable[i][j] = max(lcsTable[i-1][j], lcsTable[i][j-1])
                if debug:
                    print(f"Символы '{StringA[i-1]}' и '{StringB[j-1]}' не
совпадают:")
                    print(f"  Макс. из: lcsTable[{i-1}][{j}]={lcsTable[i-1][j]}
и lcsTable[{i}][{j-1}]={lcsTable[i][j-1]}")
                    print(f"  Результат: lcsTable[{i}][{j}] = {lcsTable[i][j]}")

    if debug:

        i, j = m, n
        lcs = []

        print("\n=== ВОССТАНОВЛЕНИЕ НОП ===")
        while i > 0 and j > 0:
            if StringA[i-1] == StringB[j-1]:
                lcs.append(StringA[i-1])
```

```
                print(f"Символы совпадают: '{StringA[i-1]}' добавляем в НОП")
                i -= 1
                j -= 1
            elif lcsTable[i-1][j] >= lcsTable[i][j-1]:
                print(f"Пропускаем символ '{StringA[i-1]}' в строке A")
                i -= 1
            else:
                print(f"Пропускаем символ '{StringB[j-1]}' в строке B")
                j -= 1

        lcs.reverse()
        print(f"\nНайденная НОП: {''.join(lcs)}")
        print(f"Длина НОП: {lcsTable[m][n]}")

    return lcsTable[m][n]


def main():
    costs = list(map(int, input().split()))
    replaceCost, insertCost, deleteCost = costs

    StringA = input().strip()
    StringB = input().strip()

    debug = True

    operations = minCostTransform(StringA, StringB, replaceCost, insertCost,
deleteCost, debug)

    lcsLength = findLcsLength(StringA, StringB, debug)

    print("\n=== ИТОГОВЫЙ РЕЗУЛЬТАТ ===")
    print(operations)
    print(StringA)
    print(StringB)
    print(f"Длина наибольшей общей подпоследовательности: {lcsLength}")


if __name__ == "__main__":
    main()
```

Файл: main3.py

```
def levenshteinDistance(sourceString, targetString, debug=False):
    m, n = len(sourceString), len(targetString)
    dpTable = [[0] * (n + 1) for _ in range(m + 1)]

    if debug:
        print("\n=== РАССТОЯНИЕ ЛЕВЕНШТЕЙНА ===")
        print("Исходная строка:", sourceString)
        print("Целевая строка:", targetString)
        print("Все операции имеют стоимость 1")

    for i in range(1, m + 1):
        dpTable[i][0] = i
        if debug:
            print(f"Инициализация: dpTable[{i}][0] = {i} (удаление {i}
символов)")

    for j in range(1, n + 1):
        dpTable[0][j] = j
        if debug:
```

```
            print(f"Инициализация: dpTable[0][{j}] = {j} (вставка {j}
символов)")

    if debug:
        print("\n=== ЗАПОЛНЕНИЕ ТАБЛИЦЫ DP ===")

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if sourceString[i-1] == targetString[j-1]:
                dpTable[i][j] = dpTable[i-1][j-1]
                if debug:
                    print(f"Символы '{sourceString[i-1]}' и
'{targetString[j-1]}' совпадают: dpTable[{i}][{j}] = {dpTable[i][j]}")
            else:
                replaceOption = dpTable[i-1][j-1] + 1
                deleteOption = dpTable[i-1][j] + 1
                insertOption = dpTable[i][j-1] + 1

                dpTable[i][j] = min(replaceOption, deleteOption, insertOption)

                if debug:
                    print(f"Символы '{sourceString[i-1]}' и
'{targetString[j-1]}' не совпадают:")
                    print(f"  Вариант замены: {dpTable[i-1][j-1]} + 1 =
{replaceOption}")
                    print(f"  Вариант удаления: {dpTable[i-1][j]} + 1 =
{deleteOption}")
                    print(f"  Вариант вставки: {dpTable[i][j-1]} + 1 =
{insertOption}")
                    operation = ""
                    if dpTable[i][j] == replaceOption:
                        operation = "замена"
                    elif dpTable[i][j] == deleteOption:
                        operation = "удаление"
                    else:
                        operation = "вставка"
                    print(f"  Выбран минимальный вариант ({operation}):
dpTable[{i}][{j}] = {dpTable[i][j]}")

    if debug:
        print("\n=== ВОССТАНОВЛЕНИЕ ПОСЛЕДОВАТЕЛЬНОСТИ ОПЕРАЦИЙ ===")
        operations = []
        i, j = m, n

        while i > 0 or j > 0:
            if i > 0 and j > 0 and sourceString[i-1] == targetString[j-1]:
                operations.append(('M', i-1, j-1))
                print(f"Символы '{sourceString[i-1]}' и '{targetString[j-1]}'
совпадают - нет операции")
                i -= 1
                j -= 1
            elif i > 0 and j > 0 and dpTable[i][j] == dpTable[i-1][j-1] + 1:
                operations.append(('R', i-1, j-1))
                print(f"Заменяем символ '{sourceString[i-1]}' на
'{targetString[j-1]}'")
                i -= 1
                j -= 1
            elif j > 0 and dpTable[i][j] == dpTable[i][j-1] + 1:
                operations.append(('I', i, j-1))
                print(f"Вставляем символ '{targetString[j-1]}'")
                j -= 1
            elif i > 0 and dpTable[i][j] == dpTable[i-1][j] + 1:
                operations.append(('D', i-1, j))
                print(f"Удаляем символ '{sourceString[i-1]}'")
                i -= 1
```

```
            else:
                print("Что-то пошло не так, выходим из цикла")
                break

        operations.reverse()

        print("\n=== ВИЗУАЛИЗАЦИЯ ПРОЦЕССА ТРАНСФОРМАЦИИ ===")
        currentString = ""

        for op, i, j in operations:
            if op == 'M':
                currentString += sourceString[i]
                print(f"Сохраняем символ '{sourceString[i]}': {currentString}")
            elif op == 'R':
                currentString += targetString[j]
                print(f"Заменяем '{sourceString[i]}' на '{targetString[j]}':
{currentString}")
            elif op == 'I':
                currentString += targetString[j]
                print(f"Вставляем символ '{targetString[j]}': {currentString}")
            elif op == 'D':
                print(f"Удаляем символ '{sourceString[i]}': {currentString}")

        print(f"\nИтоговая строка: {currentString}")
        print(f"Целевая строка:  {targetString}")
        print(f"\nРасстояние Левенштейна: {dpTable[m][n]}")

    return dpTable[m][n]


def findLcsLength(sourceString, targetString, debug=False):
    """Функция для нахождения длины наибольшей общей подпоследовательности"""
    m, n = len(sourceString), len(targetString)
    lcsTable = [[0] * (n + 1) for _ in range(m + 1)]

    if debug:
        print("\n=== АЛГОРИТМ НАХОЖДЕНИЯ НАИБОЛЬШЕЙ ОБЩЕЙ ПОДПОСЛЕДОВАТЕЛЬНОСТИ
===")
        print("Строка A:", sourceString)
        print("Строка B:", targetString)

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if sourceString[i-1] == targetString[j-1]:
                lcsTable[i][j] = lcsTable[i-1][j-1] + 1
                if debug:
                    print(f"Символы '{sourceString[i-1]}' и
'{targetString[j-1]}' совпадают: lcsTable[{i}][{j}] = {lcsTable[i][j]}")
            else:
                lcsTable[i][j] = max(lcsTable[i-1][j], lcsTable[i][j-1])
                if debug:
                    print(f"Символы '{sourceString[i-1]}' и
'{targetString[j-1]}' не совпадают:")
                    print(f"  Макс. из: lcsTable[{i-1}][{j}]={lcsTable[i-1][j]}
и lcsTable[{i}][{j-1}]={lcsTable[i][j-1]}")
                    print(f"  Результат: lcsTable[{i}][{j}] = {lcsTable[i][j]}")

    if debug:

        i, j = m, n
        lcs = []

        print("\n=== ВОССТАНОВЛЕНИЕ НОП ===")
        while i > 0 and j > 0:
            if sourceString[i-1] == targetString[j-1]:
```

```
                lcs.append(sourceString[i-1])
                print(f"Символы совпадают: '{sourceString[i-1]}' добавляем в
НОП")
                i -= 1
                j -= 1
            elif lcsTable[i-1][j] >= lcsTable[i][j-1]:
                print(f"Пропускаем символ '{sourceString[i-1]}' в строке A")
                i -= 1
            else:
                print(f"Пропускаем символ '{targetString[j-1]}' в строке B")
                j -= 1

        lcs.reverse()
        print(f"\nНайденная НОП: {''.join(lcs)}")
        print(f"Длина НОП: {lcsTable[m][n]}")

    return lcsTable[m][n]


def main():
    sourceString = input().strip()
    targetString = input().strip()

    debug = True

    result = levenshteinDistance(sourceString, targetString, debug)

    print("\n=== ИТОГОВЫЙ РЕЗУЛЬТАТ ===")
    print(result)

    lcsLength = findLcsLength(sourceString, targetString, debug)
    print(f"Длина наибольшей общей подпоследовательности: {lcsLength}")


if __name__ == "__main__":
    main()
```