+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: Расстояние Левенштейна. Алгоритм Вагнера-Фишера

Студентка гр. 3343 _____ Гельман П.Е.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы.

Цель данной лабораторной работы состоит в изучении нахождения расстояния Левенштейна (редакционного расстояния) и алгоритма Вагнера-Фишера, его реализации.

Задание.

№1

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).

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:

111

entrance

reenterable

Sample Output:

5

№2

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).

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.

Пример (все операции стоят одинаково)

M	M	M	R	I	M	R	R
C	**O**	**N**	**N**		**E**	**C**	**T**
C	O	N	**E**	**H**	E	A	D

Пример (цена замены 3, остальные операции по 1)

M	M	M	D	M	I	I	I	I	D	D
C	**O**	**N**	**N**	**E**					**C**	**T**
C	O	N		E	**H**	**E**	**A**	**D**		

Рисунок 1 - Пример

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:

111

entrance

reenterable

Sample Output:

IMIMMIMMRRM

entrance

reenterable

№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

Задание варианта:

5а. Добавляется 4-я операция со своей стоимостью: удаление двух последовательных разных символов.

Примечания для варианта:

1) Предполагается, что для весов операций действует правило треугольника: если две последовательные операции можно заменить одной, то это не ухудшает общую цену.

2) При выполнении операций запрещается применять операции к символам, которые уже были получены в результате выполнения операций (т.е. строка преобразуется всегда только слева направо).

Выполнение работы.

Расстояние Левенштейна (также редакционное расстояние или дистанция редактирования) между двумя строками — это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Для расстояния Левенштейна справедливы следующие утверждения:

- $d(S1, S2) \geqslant ||S1| - |S2||$
- $d(S1, S2) \leqslant \max(|S1|, |S2|)$
- $d(S1, S2) = 0 \Leftrightarrow S1 = S2$

где $d(S1, S2)$ — расстояние Левенштейна между строками S1 и S2, а $|S|$ — длина строки S.

В ходе выполнения работы были реализованы следующие функции:

1. wagner_fischer(s1, s2, replace_cost, insert_cost, delete_cost, double_delete_cost) -> tuple – функция, которая вычисляет редакционное расстояние, основываясь на алгоритме Вагнера-Фишера. Она принимает в качестве аргументов две строки, а также цену каждой имеющейся операции (вставка, удаление, замена, удаление двух разных символов подряд). В теле функции создается матрица размером len(s1)*len(s2). Далее заполняется верхний правый элемент нулем, а также первая строка и первый столбец значениями, соответствующими операциям вставки и удаления. Затем в цикле происходит посимвольное сравнение строк, сравниваются уже известные результаты в клетках матрицы, и выбирается наименьшее число среди: клетки слева, через одну сверху, сверху и диагонально от текущей + стоимость выбранного действия. Удаление двух символов, идущих подряд и различных, происходит только в

случае, если мы не выходим за границы матрицы (т.е. не выходим за пределы индексов строки). Таким образом и заполняется вся таблица. Последняя ячейка слева снизу – редакционное расстояние между строками s1 и s2 с учетом всех стоимостей операций.

2. print_table(dp, s1, s2) -> None – функция для печати матрицы.
3. reconstruct_operations(op, s1, s2) -> str – функция, которая восстанавливает последовательность операций, примененных к строке s1 для получения s2. В качестве аргументов передается матрица, содержащая символы операций, которая заполняется в ходе поиска расстояния Левенштейна, и две строки s1 и s2. Функция «обратно» проходит по самому короткому пути, который был найден и записан в последнюю клетку матрицы, и восстанавливает примененные действия и их последовательность.
4. levenshtein_distance(s1, s2) -> int – поиск расстояния Левенштейна (частный случай алгоритма Вагнера-Фишера, так как все стоимости операций равны 1). Данная функция работает аналогично wagner_fisher() за исключением того, что здесь нет действия двойного удаления.
5. levenshtein_double(s1, s2) -> int – поиск редакционного расстояния Левенштейна, включающий 4-ую операцию. Работает так же, как и wagner_fisher(), но все стоимости также равны 1.

Оценка сложности.

Сложность по времени O((N+1)(M+1)), так как необходимо пройти по всей матрице, размеры которой – произведение длин строк + нулевой элемент пустой и там, и там.

Сложность по памяти в обычном виде O((N+1)(M+1)), но так как мы храним еще матрицу совершенных операций, то имеем: O(2(N+1)(M+1))

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	1 1 1 1 cat dots	Алгоритм Вагнера-Фишера: 3 Последовательность операций: RRMI Расстояние Левенштейна (без двойного удаления): 3 Расстояние Левенштейна (с двойным удалением): 3	Верно
2.	1 2 3 4 polynomial exponential	Алгоритм Вагнера-Фишера: 8 Последовательность операций: RRRRMIRRMMM Расстояние Левенштейна (без двойного удаления): 6 Расстояние Левенштейна (с двойным удалением): 6	Верно
3.	4 2 8 2 kged mn	Алгоритм Вагнера-Фишера: 8 Последовательность операций: OOII Расстояние Левенштейна (без двойного удаления): 4 Расстояние Левенштейна (с двойным удалением): 3	Верно
4.	3 4 67 8 ggggggg ccccccc	Алгоритм Вагнера-Фишера: 21 Последовательность операций: RRRRRRR Расстояние Левенштейна (без двойного удаления): 7 Расстояние Левенштейна (с двойным удалением): 7	Верно
5.	1 1 1 1 pedeestal staill	Алгоритм Вагнера-Фишера: 5 Последовательность операций: OODMMMIIM Расстояние Левенштейна (без двойного удаления): 7 Расстояние Левенштейна (с двойным удалением): 5	Верно
6.	1 1 1 1 pedeeeestal clear	Алгоритм Вагнера-Фишера: 6 Последовательность операций: OORRMOMR Расстояние Левенштейна (без двойного удаления): 9 Расстояние Левенштейна (с двойным удалением): 6	Верно

Выводы.

В ходе лабораторной работы был реализован алгоритм Вагнера-Фишера для поиска редакционного расстояния Левенштейна, проанализирована его временная сложность и сложность по памяти.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
def wagner_fischer(s1, s2, replace_cost, insert_cost, delete_cost,
double_delete_cost) -> tuple:
    print("==== Алгоритм Вагнера-Фишера ====")
    m, n = len(s1), len(s2)
    print("Длина первой строки -", m, "\nДлина второй строки -", n)
    d = [[0] * (n + 1) for _ in range(m + 1)]
    op = [[None] * (n + 1) for _ in range(m + 1)]
    print('=' * 100)
    print(f"Стоимость вставки = {insert_cost}, стоимость удаления =
{delete_cost},"
          f" стоимость замены = {replace_cost}, стомость двойного удаления
= {double_delete_cost}")
    print('=' * 100)
    print("Заполняем ячейку [0][0] нулем")
    d[0][0] = 0

    print("Заполняем первый столбец и первую строку")
    for j in range(1, n + 1):
        d[0][j] = d[0][j - 1] + insert_cost
        print(f"d[0][{j}] = {d[0][j]} (вставка {insert_cost})")
        op[0][j] = 'I'

    print('=' * 100)

    for i in range(1, m + 1):
        d[i][0] = d[i - 1][0] + delete_cost
        op[i][0] = 'D'
        if i >= 2 and s1[i - 2] != s1[i - 1]:
            if d[i][0] > d[i - 2][0] + double_delete_cost:
                d[i][0] = d[i - 2][0] + double_delete_cost
                op[i][0] = 'O'
        print(f"d[{i}][0] = {d[i][0]} (удаление {delete_cost})")
    print('=' * 100)

    print("Заполняем остальные ячейки матрицы")
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i - 1] == s2[j - 1]:
                d[i][j] = d[i - 1][j - 1]
                print(f"Значения совпали (s1[{i - 1}] = {s1[i - 1]}),
(s2[{j - 1}] = {s2[j - 1]}),"
                      f" d[{i}][{j}] = {d[i][j]} ")
                op[i][j] = 'M'
            else:
                replace = d[i - 1][j - 1] + replace_cost
                insert = d[i][j - 1] + insert_cost
                delete = d[i - 1][j] + delete_cost
                double_delete = float('inf')
                if i >= 2 and s1[i - 2] != s1[i - 1]:
                    double_delete = d[i - 2][j] + double_delete_cost
                print("Выбираем менее дорогостоящую операцию")
                print(f"replace - {replace}, insert - {insert}, delete -
{delete}, ddelete - {double_delete}")
                d[i][j] = min(replace, insert, delete, double_delete)

                if d[i][j] == replace:
```

```
                    print(f"Была выбрана замена (s1[{i - 1}] = {s1[i -
1]}), (s2[{j - 1}] = {s2[j - 1]}),"
                          f" d[{i}][{j}] = {d[i][j]}")
                    op[i][j] = 'R'
                elif d[i][j] == insert:
                    print(f"Была выбрана вставка (s1[{i - 1}] = {s1[i -
1]}), (s2[{j - 1}] = {s2[j - 1]}),"
                          f" d[{i}][{j}] = {d[i][j]}")
                    op[i][j] = 'I'
                elif d[i][j] == delete:
                    print(
                        f"Было выбрано удаление 1 символа (s1[{i - 1}] =
{s1[i - 1]}), (s2[{j - 1}] = {s2[j - 1]}),"
                        f" d[{i}][{j}] = {d[i][j]}")
                    op[i][j] = 'D'

                else:
                    print(
                        f"Было выбрано удаление 2-х отличающихся символов
(s1[{i - 1}] = {s1[i - 1]}),"
                        f" (s1[{i - 2}] = {s1[i - 2]}), (s2[{j - 1}] =
{s2[j - 1]}), d[{i}][{j}] = {d[i][j]}")
                    op[i][j] = 'O'
    print('=' * 100)

    print_table(d, s1, s2)
    return d, op


def print_table(dp, s1, s2) -> None:
    n = len(s1)
    m = len(s2)
    max_val = max(max(row) for row in dp)
    cell_width = max(3, len(str(max_val)) + 2)

    header = [" "] + [" "] + list(s2)
    print("-" + ("-" * (cell_width + 2) + "-") * (m + 2))

    header_row = "|"
    for h in header:
        header_row += f" {h:^{cell_width}} |"
    print(header_row)

    print("-" + ("-" * (cell_width + 2) + "-") * (m + 2))

    for i in range(n + 1):
        row_header = " " if i == 0 else s1[i - 1]
        row = [f" {row_header:^{cell_width}} |"]
        for j in range(m + 1):
            cell = dp[i][j]
            cell_str = f"{cell:^{cell_width}}"
            row.append(f" {cell_str} |")
        print("|" + "".join(row))
        print("-" + ("-" * (cell_width + 2) + "-") * (m + 2))


def reconstruct_operations(op, s1, s2) -> str:
    operations = []
    i, j = len(s1), len(s2)
    print('=' * 100)
    print("\nВосстановление последовательности операций:")
    while i > 0 or j > 0:
        current_op = op[i][j]
        operations.append(current_op)
```

```
        print(f"Операция {current_op} на позиции ({i}, {j})")
        if current_op == 'M' or current_op == 'R':
            i -= 1
            j -= 1
        elif current_op == 'I':
            j -= 1
        elif current_op == 'D':
            i -= 1
        elif current_op == 'O':
            i -= 2

    operations.reverse()
    return ''.join(operations)


def levenshtein_distance(s1, s2) -> int:
    m, n = len(s1), len(s2)
    d = [[0] * (n + 1) for _ in range(m + 1)]

    print("\nВычисление расстояния Левенштейна без двойного удаления:")
    d[0][0] = 0
    print('=' * 100)
    print("Заполнение первого столбца и первой строки")
    for j in range(1, n + 1):
        d[0][j] = d[0][j - 1] + 1
        print(f"d[0][{j}] = {d[0][j]}")

    for i in range(1, m + 1):
        d[i][0] = d[i - 1][0] + 1
        print(f"d[{i}][0] = {d[i][0]}")
    print('=' * 100)
    print("Заполнение остальных ячеек")
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i - 1] == s2[j - 1]:
                print(f"Значения совпали (s1[{i - 1}] = {s1[i - 1]}),
(s2[{j - 1}] = {s2[j - 1]}), "
                      f"d[{i}][{j}] = {d[i][j]} ")
                d[i][j] = d[i - 1][j - 1]
            else:
                replace = d[i - 1][j - 1] + 1
                insert = d[i][j - 1] + 1
                delete = d[i - 1][j] + 1
                print("Выбираем менее дорогостоящую операцию")
                print(f"replace - {replace}, insert - {insert}, delete -
{delete}")
                d[i][j] = min(replace, insert, delete)
                if d[i][j] == replace:
                    print(f"Была выбрана замена (s1[{i - 1}] = {s1[i -
1]}), (s2[{j - 1}] = "
                          f"{s2[j - 1]}), d[{i}][{j}] = {d[i][j]}")
                elif d[i][j] == insert:
                    print(f"Была выбрана вставка (s1[{i - 1}] = {s1[i -
1]}), (s2[{j - 1}] = {s2[j - 1]}),"
                          f" d[{i}][{j}] = {d[i][j]}")
                elif d[i][j] == delete:
                    print(f"Было выбрано удаление 1 символа (s1[{i - 1}] =
{s1[i - 1]}), (s2[{j - 1}] = {s2[j - 1]}), "
                          f"d[{i}][{j}] = {d[i][j]}")
    print('=' * 100)
    print_table(d, s1, s2)

    return d[m][n]
```

```
def levenshtein_double(s1, s2) -> int:
    m, n = len(s1), len(s2)
    d = [[0] * (n + 1) for _ in range(m + 1)]

    print("\nВычисление расстояния Левенштейна с двойным удалением:")
    print('=' * 100)
    print("Заполнение первого столбца и первой строки")
    for i in range(1, m + 1):
        d[i][0] = d[i - 1][0] + 1
        if i >= 2 and s1[i - 2] != s1[i - 1]:
            if d[i][0] > d[i - 2][0] + 1:
                d[i][0] = d[i - 2][0] + 1
        print(f"d[{i}][0] = {d[i][0]}")

    for j in range(1, n + 1):
        d[0][j] = d[0][j - 1] + 1
        print(f"d[0][{j}] = {d[0][j]}")
    print('=' * 100)
    print("Заполнение остальных ячеек")
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i - 1] == s2[j - 1]:
                d[i][j] = d[i - 1][j - 1]
                print(f"Значения совпали (s1[{i - 1}] = {s1[i - 1]}),
(s2[{j - 1}] = {s2[j - 1]}),"
                      f" d[{i}][{j}] = {d[i][j]} ")
            else:
                replace = d[i - 1][j - 1] + 1
                insert = d[i][j - 1] + 1
                delete = d[i - 1][j] + 1
                double_delete = float('inf')
                if i >= 2 and s1[i - 2] != s1[i - 1]:
                    double_delete = d[i - 2][j] + 1
                print("Выбираем менее дорогостоящую операцию")
                print(f"replace - {replace}, insert - {insert}, delete -
{delete}, ddelete - {double_delete}")
                d[i][j] = min(replace, insert, delete, double_delete)
                if d[i][j] == replace:
                    print(f"Была выбрана замена (s1[{i - 1}] = {s1[i -
1]}), (s2[{j - 1}] = {s2[j - 1]}),"
                          f" d[{i}][{j}] = {d[i][j]}")
                elif d[i][j] == insert:
                    print(f"Была выбрана вставка (s1[{i - 1}] = {s1[i -
1]}), (s2[{j - 1}] = {s2[j - 1]}), "
                          f"d[{i}][{j}] = {d[i][j]}")
                elif d[i][j] == delete:
                    print(f"Было выбрано удаление 1 символа (s1[{i - 1}] =
{s1[i - 1]}), (s2[{j - 1}] = {s2[j - 1]}), "
                          f"d[{i}][{j}] = {d[i][j]}")
                else:
                    print(f"Было выбрано удаление 2-х отличающихся символов
(s1[{i - 1}] = {s1[i - 1]}),"
                          f" (s1[{i - 2}] = {s1[i - 2]}), (s2[{j - 1}] =
{s2[j - 1]}), d[{i}][{j}] = {d[i][j]}")
    print('=' * 100)
    print_table(d, s1, s2)
    return d[m][n]


def main() -> None:
    replace_cost, insert_cost, delete_cost, double_delete_cost = map(int,
input(
```

```
        "Введите стоимость Replace, Insert, Delete, DoubleDelete через
пробел: ").split())
    s1 = input("Введите первую строку: ").strip()
    s2 = input("Введите вторую строку: ").strip()

    d, op = wagner_fischer(s1, s2, replace_cost, insert_cost, delete_cost,
double_delete_cost)

    operations_sequence = reconstruct_operations(op, s1, s2)

    lev_distance = levenshtein_distance(s1, s2)
    lev_distance2 = levenshtein_double(s1, s2)
    print('=' * 100)
    print("\nРезультаты:")
    print("Алгоритм Вагнера-Фишера:", d[-1][-1])
    print("Последовательность операций:", operations_sequence)
    print(s1)
    print(s2)
    print("Расстояние Левенштейна (без двойного удаления):", lev_distance)
    print("Расстояние Левенштейна (с двойным удалением):", lev_distance2)


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