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

Студент гр. 3343 _____ Малиновский А.А,

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы.

Нахождения редакционного предписания алгоритмом Вагнера-Фишера.

Задание.

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*S*, 1≤|S|≤25501≤|*S*|≤2550).

Вторая строка входных данных содержит строку из строчных латинских букв. (T*T*, 1≤|T|≤25501≤|*T*|≤2550).

Параметры выходных данных:

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

Sample Input:

pedestal

stien

Sample Output:

7

Индивидуализация

Вариант 2

"Особый заменитель и особо удаляемый символ": цена замены на определённый символ отличается от обычной цены замены; цена удаления другого (или того же) определённого символа отличается от обычной цены удаления. Особый заменитель и цена замены на него, особо удаляемый символ и цена его удаления — дополнительные входные данные.

Описание алгоритма.

Алгоритм Вагнера-Фишера — это метод динамического программирования для вычисления расстояния Левенштейна между двумя строками, то есть минимального числа операций вставки, удаления или замены символов, нужных для превращения одной строки в другую. Сначала создаётся матрица размером (n+1)×(m+1), где n и m — длины строк. Первая строка заполняется числами от 0 до n (стоимость удаления символов первой строки), а первый столбец — от 0 до m (стоимость вставки символов второй строки). Затем для каждой ячейки матрицы вычисляется минимальная стоимость операций: удаление (берётся значение сверху и прибавляется 1), вставка (значение слева плюс 1) или замена (значение по диагонали плюс 1, если символы разные, или без изменений, если они совпадают). Результат — число в правом нижнем углу матрицы, которое и есть расстояние Левенштейна.

Чтобы восстановить последовательность операций, нужно пройти от конца матрицы к началу, выбирая путь с наименьшей стоимостью. Движение вверх означает удаление символа первой строки, влево — вставку символа второй строки, а по диагонали — либо совпадение символов (если они равны), либо замену (если разные).

Сложность по времени:

Требуется заполнить матрицу размером n*m, где n-длина первой, m – длина второй строки. Итого O(n*m).

Сложность по памяти:

Если полностью хранить матрицу, то требуется O(n*m) памяти. Можно улучшить храня только одну строки матрицы, так как нам чтобы заполнить ячейку матрицы требуется смотреть на 3 значения: слева, сверху и по диагонали. Таким образом получаем O(m).

Описание функций.

1. **print_dp_matrix(s1, s2, dp, current_i, current_j, cells)**
   Выводит матрицу динамического программирования (DP) с подсветкой текущей ячейки или указанных ячеек. Поддерживает цветное выделение для наглядности.
2. **visualize_levenshtein(s1, s2, costs)**
   Визуализирует алгоритм Левенштейна, заполняя DP-матрицу с учетом стоимости операций. Пошагово выводит изменения матрицы и логику выбора минимальной стоимости.
3. **get_edit_sequence(s1, s2, costs, dp)**
   Восстанавливает последовательность операций редактирования (вставка, удаление, замена, совпадение) из заполненной DP-матрицы. Возвращает список операций и ячеек пути.
4. **print_operations(operations)**
   Выводит последовательность операций редактирования с цветовой подсветкой для разных типов операций (совпадение, замена, вставка, удаление).
5. **display_operation_steps(s1, s2, operations)**
   Отображает последовательность операций в виде таблицы с цветовой разметкой, показывая соответствие символов исходной и целевой строк.

Тестирование.

№	Входные данные	Выходные данные	Комментарий
1	ab abfagfab	6	Верно
2	hello world	4	Верно
4	pedestal stien	7	Верно
5	connect conehead	4	Верно

Результат работы программы с отладочным выводом (см. рис 1, 2, 3).

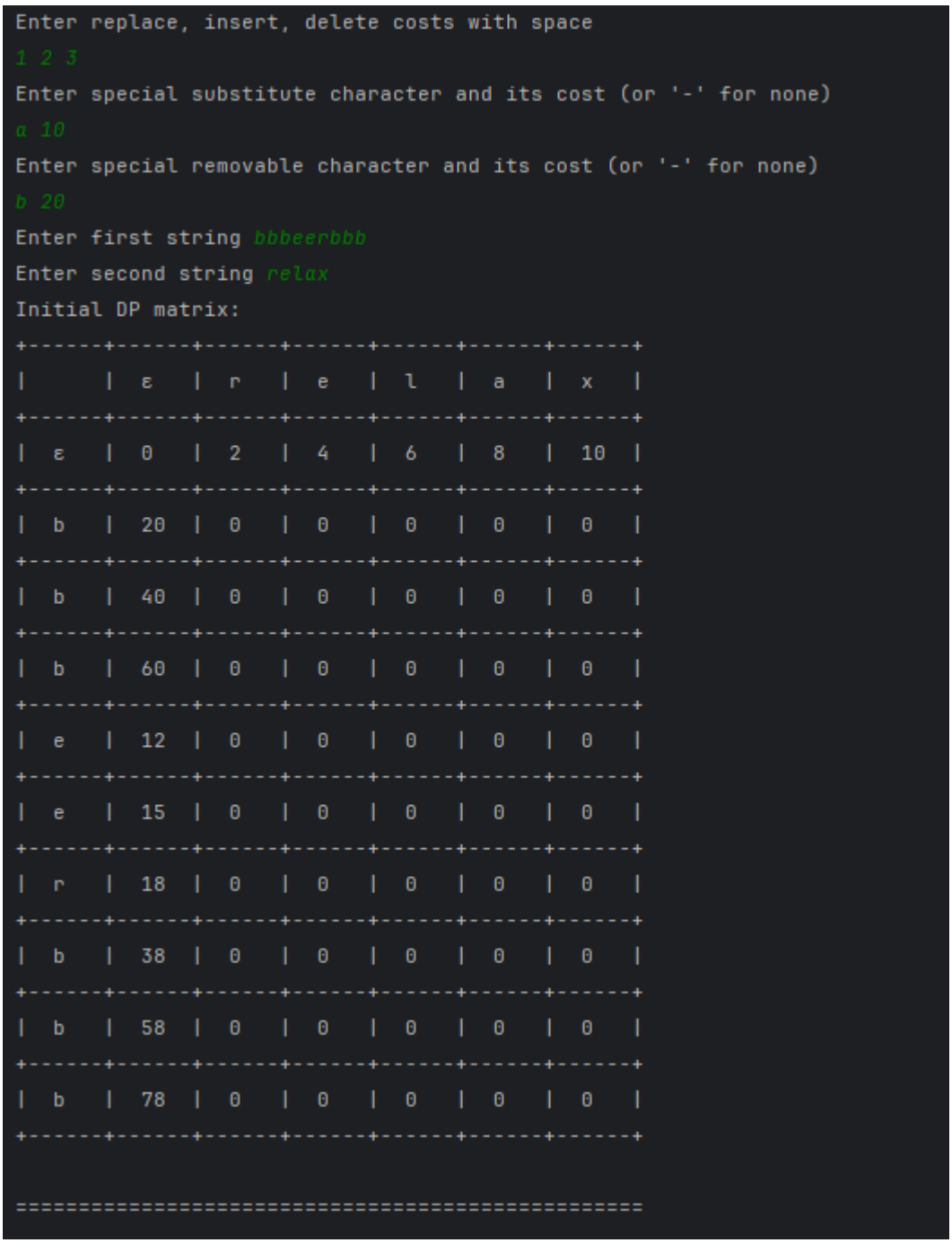

Рисунок 1 – Начало вывода, далее каждое изменение ячейки выводится на экран

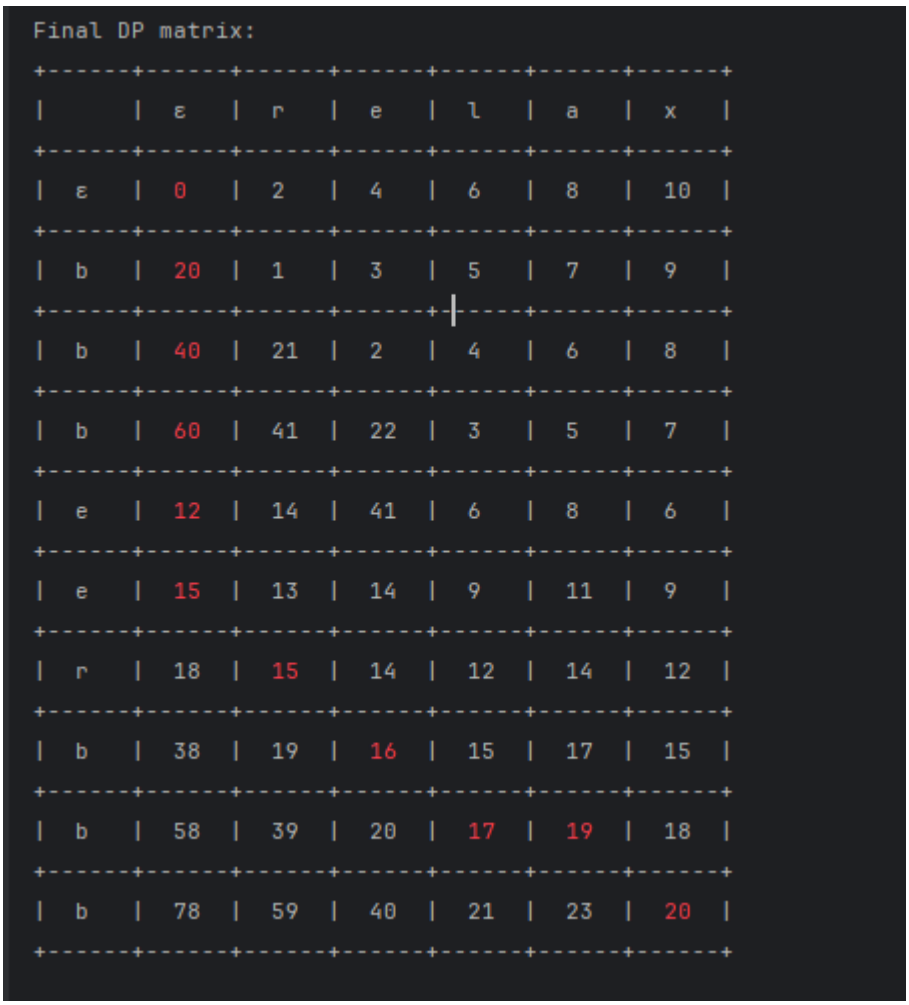

Рисунок 3 – Шаги по которым составляется РП

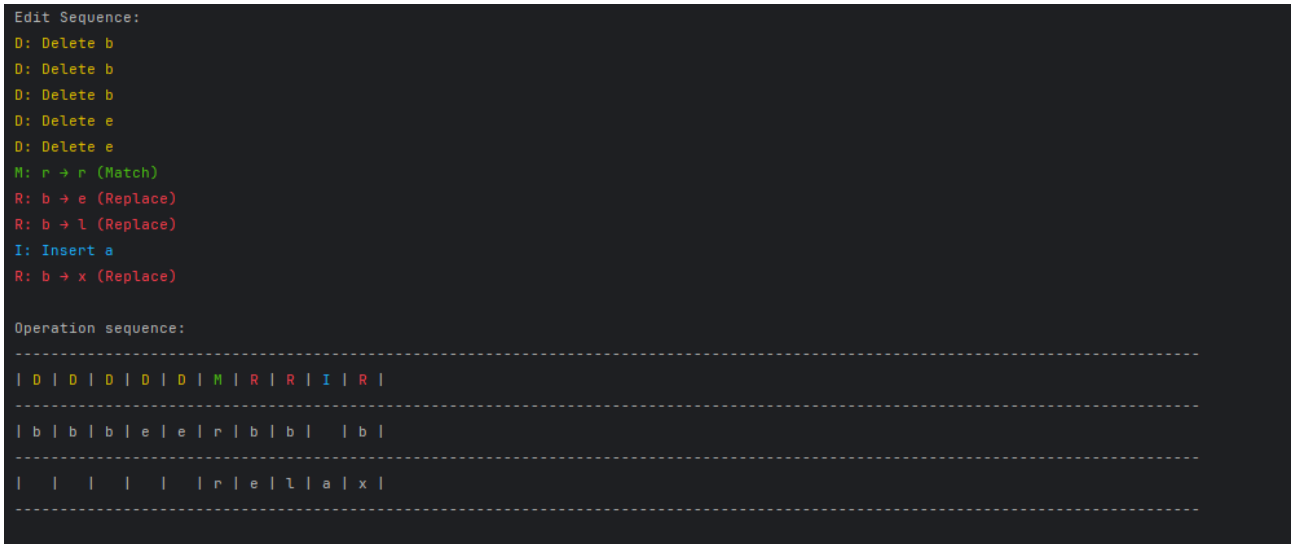

Рисунок 4 – Итоговое редакционное предписание

Исследование.

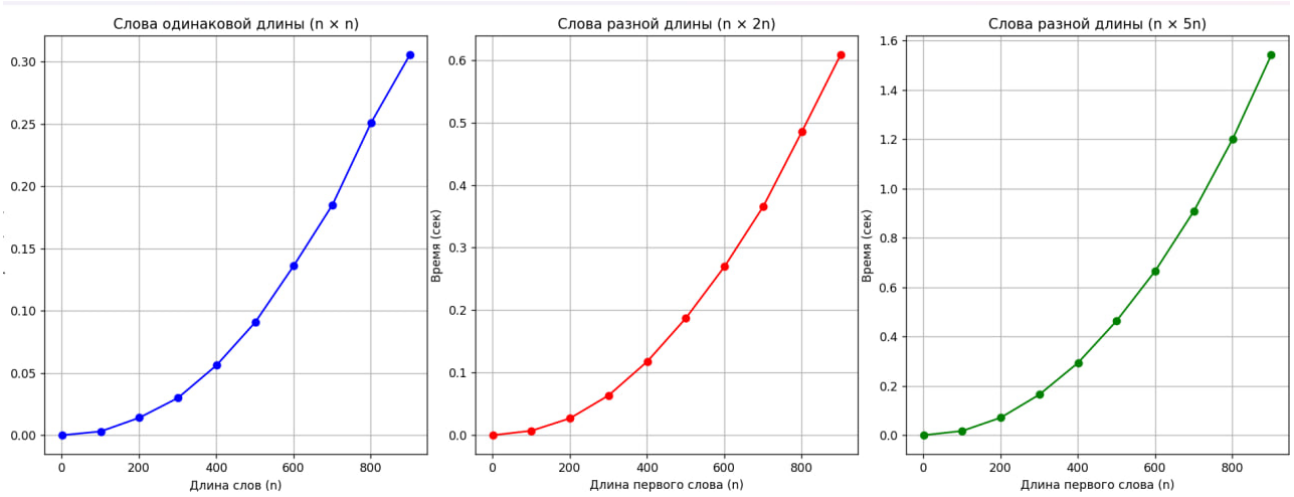

Рисунок 8 – Тестирование алгоритма на разных данных

Как видно практическое время выполнения совпадает с теоретическим.

Выводы.

Был реализован алгоритм Вагнера-Фишера для вычисления редакционного предписания, определяя минимальное количество операций (вставки, удаления, замены) для преобразования одной строки в другую. Алгоритм эффективно решает задачи сравнения строк, исправления опечаток и других приложений, связанных с обработкой текста.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла:visualization.py

```
    def print_dp_matrix(s1, s2, dp, current_i=None, current_j=None,
cells=None):
        n = len(s1)
        m = len(s2)

        # Determine the maximum width needed for any cell
        max_val = max(max(row) for row in dp)
        cell_width = max(3, len(str(max_val)) + 2)  # At least 3 for
single digits + padding

        # Column headers (s2 characters)
        header = [" "] + ["ε"] + list(s2)

        # Print top border
        print("+" + ("-" * (cell_width + 2) + "+") * (m + 2))

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

        # Print separator after header
        print("+" + ("-" * (cell_width + 2) + "+") * (m + 2))

        for i in range(n + 1):
            # Row header (s1 characters)
            row_header = "ε" if i == 0 else s1[i - 1]
            row = [f" {row_header:^{cell_width}} |"]

            for j in range(m + 1):
                cell = dp[i][j]
                # Highlight current cell if specified
                if cells and (i, j) in cells:
```

```
                    cell_str = f"\033[91m{cell:^{cell_width}}\033[0m"  #
Red highlight
                elif current_i == i and current_j == j:
                    cell_str = f"\033[91m{cell:^{cell_width}}\033[0m"  #
Red highlight
                else:
                    cell_str = f"{cell:^{cell_width}}"
                row.append(f" {cell_str} |")

            # Print row with borders
            print("|" + "".join(row))
            # Print separator after each row
            print("+" + ("-" * (cell_width + 2) + "+") * (m + 2))


    def visualize_levenshtein(s1, s2, costs):
        n = len(s1)
        m = len(s2)
        dp = [[0 for _ in range(m + 1)] for _ in range(n + 1)]

        # Initialize first row and column
        for i in range(n + 1):
            if (costs["special_delete"]["char"] is not None and
                    i    >    0    and    s1[i    -    1]    ==
costs["special_delete"]["char"]):
                dp[i][0] = dp[i - 1][0] + costs["special_delete"]["cost"]
            else:
                dp[i][0] = i * costs["delete"]

        for j in range(m + 1):
            dp[0][j] = j * costs["insert"]

        print("Initial DP matrix:")
        print_dp_matrix(s1, s2, dp)
        print("\n" + "=" * 50 + "\n")

        for i in range(1, n + 1):
            for j in range(1, m + 1):
```

```
                print(f"Processing cell ({i}, {j}):")
                if s1[i - 1] == s2[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1]
                    print(f"  Characters match: '{s1[i - 1]}' == '{s2[j
- 1]}'")
                    print(f"   dp[{i}][{j}] = dp[{i - 1}][{j - 1}] =
{dp[i][j]}")
                else:
                    # Check for special substitution
                    if (costs["special_replace"]["char"] is not None and
                            s2[j - 1] ==
costs["special_replace"]["char"]):
                        replace_cost = dp[i - 1][j - 1] +
costs["special_replace"]["cost"]
                        print(f"          Special replace with
'{costs['special_replace']['char']}' cost: {replace_cost}")
                    else:
                        replace_cost = dp[i - 1][j - 1] +
costs["replace"]
                        print(f"  Normal replace cost: {replace_cost}")

                    insert_cost = dp[i][j - 1] + costs["insert"]
                    print(f"  Insert cost: {insert_cost}")

                    # Check for special deletion
                    if (costs["special_delete"]["char"] is not None and
                            s1[i - 1] ==
costs["special_delete"]["char"]):
                        delete_cost = dp[i - 1][j] +
costs["special_delete"]["cost"]
                        print(f"           Special delete of
'{costs['special_delete']['char']}' cost: {delete_cost}")
                    else:
                        delete_cost = dp[i - 1][j] + costs["delete"]
                        print(f"  Normal delete cost: {delete_cost}")

                    dp[i][j] = min(replace_cost, insert_cost,
delete_cost)
```

```
                    print(f"  Selected min cost: {dp[i][j]}")

                print("\nCurrent DP matrix:")
                print_dp_matrix(s1, s2, dp, i, j)
                print("\n" + "-" * 50 + "\n")

        return dp


    def get_edit_sequence(s1, s2, costs, dp):
        i = len(s1)
        j = len(s2)
        operations = []
        path_cells = [(i, j)]
        while i > 0 or j > 0:
            if i > 0 and j > 0 and s1[i - 1] == s2[j - 1]:
                operations.append(('M', s1[i - 1], s2[j - 1]))
                i -= 1
                j -= 1
            else:
                if j > 0 and (i == 0 or dp[i][j] == dp[i][j - 1] +
costs["insert"]):
                    operations.append(('I', '', s2[j - 1]))
                    j -= 1
                elif i > 0 and (j == 0 or dp[i][j] == dp[i - 1][j] + (
                        costs["special_delete"]["cost"] if (
                                costs["special_delete"]["char"]  is  not
None and
                                s1[i          -          1]          ==
costs["special_delete"]["char"]
                        ) else costs["delete"]
                )):
                    operations.append(('D', s1[i - 1], ''))
                    i -= 1
                elif i > 0 and j > 0 and dp[i][j] == dp[i - 1][j - 1] +
(
                        costs["special_replace"]["cost"] if (
```

```
                                costs["special_replace"]["char"] is not
None and
                                s2[j - 1] ==
costs["special_replace"]["char"]
                            ) else costs["replace"]
                ):
                    operations.append(('R', s1[i - 1], s2[j - 1]))
                    i -= 1
                    j -= 1
            path_cells.append((i, j))

        operations.reverse()
        return operations, path_cells


    def print_operations(operations):
        print("\nEdit Sequence:")
        color_codes = {
            'M': '\033[92m',  # Green
            'R': '\033[91m',  # Red
            'I': '\033[94m',  # Blue
            'D': '\033[93m',  # Yellow
        }
        reset_color = '\033[0m'

        for op in operations:
            color = color_codes[op[0]]
            if op[0] == 'M':
                print(f"{color}{op[0]}: {op[1]} → {op[2]}
(Match){reset_color}")
            elif op[0] == 'R':
                print(f"{color}{op[0]}: {op[1]} → {op[2]}
(Replace){reset_color}")
            elif op[0] == 'I':
                print(f"{color}{op[0]}: Insert {op[2]}{reset_color}")
            elif op[0] == 'D':
                print(f"{color}{op[0]}: Delete {op[1]}{reset_color}")
```

```
    def display_operation_steps(s1, s2, operations):
        # Color codes
        colors = {
            'M': '\033[92m',  # Green
            'R': '\033[91m',  # Red
            'I': '\033[94m',  # Blue
            'D': '\033[93m',  # Yellow
        }
        reset = '\033[0m'

        # Prepare rows with colors
        op_row = "|" + "|".join(f" {colors[op[0]]}{op[0]}{reset} " for
op in operations) + "|"
        s1_row = "|" + "|".join(f" {op[1] if op[1] else ' '} " for op in
operations) + "|"
        s2_row = "|" + "|".join(f" {op[2] if op[2] else ' '} " for op in
operations) + "|"

        # Print the table
        print("\nOperation sequence:")
        separator = "-" * len(op_row)
        print(separator)
        print(op_row)
        print(separator)
        print(s1_row)
        print(separator)
        print(s2_row)
        print(separator)
```

Название файла: levenstein.py

```
    def lev_distance(i, j, s1, s2, matrix):
        if i == 0 and j == 0:
            return 0
        elif j == 0 and i > 0:
            return i
        elif i == 0 and j > 0:
            return j
```

```
            else:
                m = 0 if s1[i - 1] == s2[j - 1] else 1
                return min(matrix[i][j - 1] + 1, matrix[i - 1][j] + 1,
matrix[i - 1][j - 1] + m)


        def calculate_levenshtein_distance(s1, s2):
            n = len(s1)
            m = len(s2)
            matrix = [[0 for i in range(m + 1)] for j in range(n + 1)]
            for i in range(n + 1):
                for j in range(m + 1):
                    matrix[i][j] = lev_distance(i, j, s1, s2, matrix)
            return matrix[n][m]


        if __name__=="__main__":
            s1 = input()
            s2 = input()
    print(calculate_levenshtein_distance(s1, s2))
```

Название файла: optimized_levenstein.py

```
def calculate_levenshtein_distance_optimized(s1, s2):
    n = len(s1)
    m = len(s2)

    if n == 0:
        return m
    if m == 0:
        return n

    prev_row = [0] * (m + 1)
    curr_row = [0] * (m + 1)

    for j in range(m + 1):
        prev_row[j] = j

    for i in range(1, n + 1):
        curr_row[0] = i
```

```
        for j in range(1, m + 1):
            cost = 0 if s1[i - 1] == s2[j - 1] else 1
            curr_row[j] = min(
                curr_row[j - 1] + 1,  # Вставка
                prev_row[j] + 1,  # Удаление
                prev_row[j - 1] + cost  # Замена
            )

        # Обмениваем строки для следующей итерации
        prev_row, curr_row = curr_row, prev_row
    return prev_row[m]

if __name__=="__main__":
    s1 = input()
    s2 = input()
    print(calculate_levenshtein_distance_optimized(s1, s2))
```