**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).

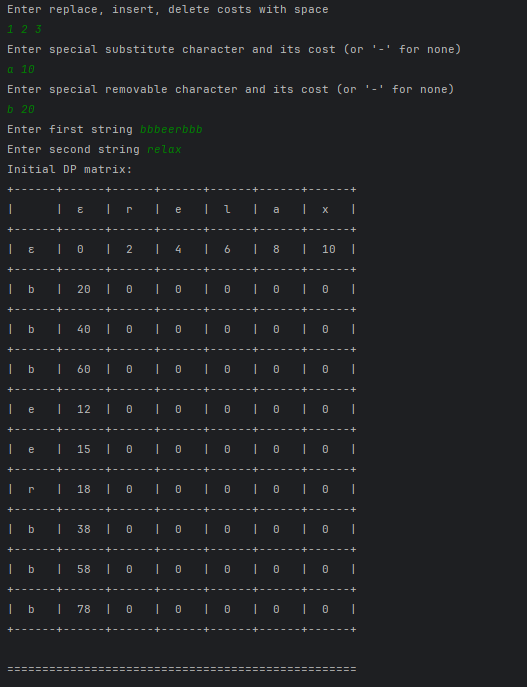


Рисунок 1 – Начало вывода, далее каждое изменение ячейки выводится на экран

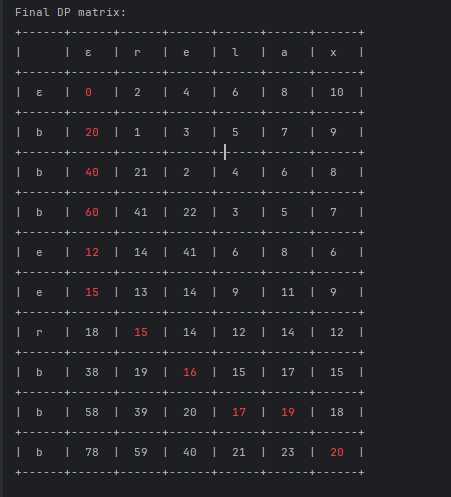


Рисунок 3 – Шаги по которым составляется РП

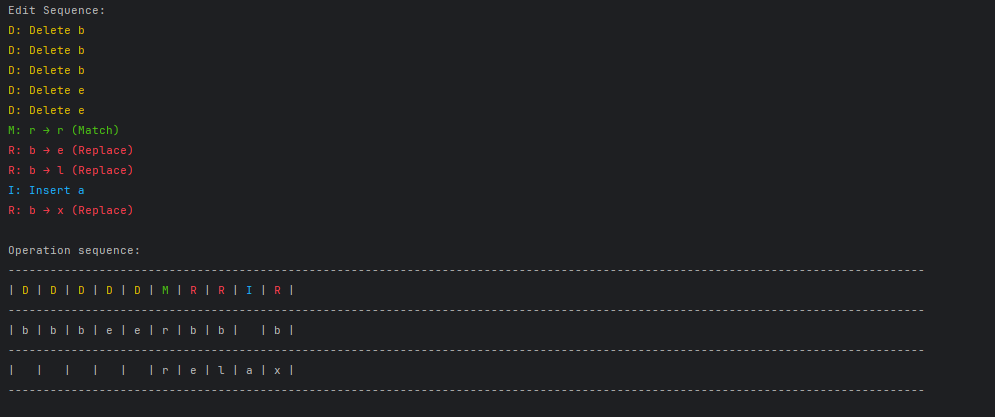


Рисунок 4 – Итоговое редакционное предписание

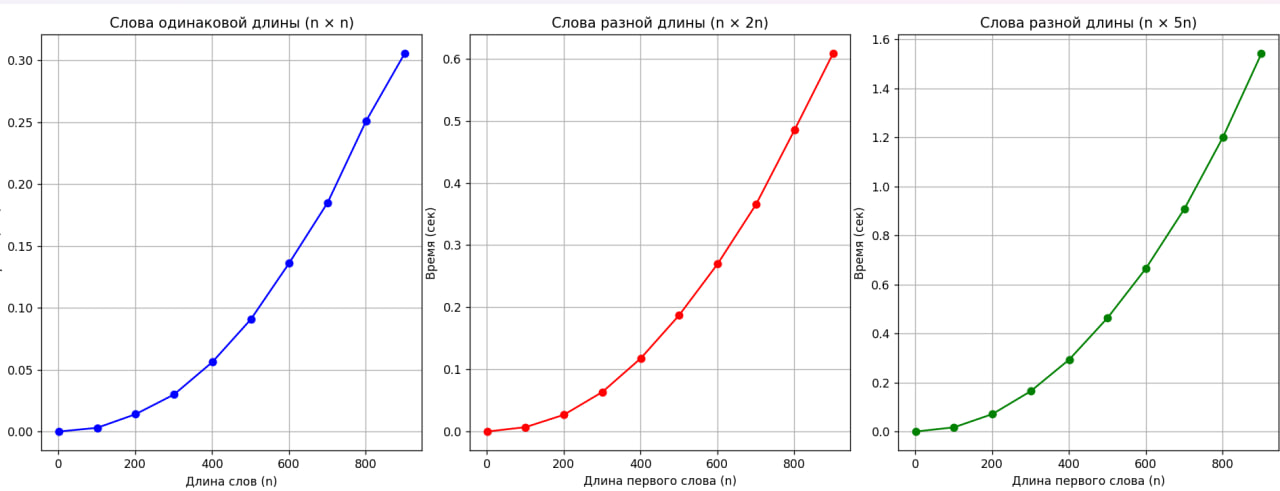
**Исследование.**

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))