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 4а.

Студент гр. 3343 _____ Какира Умар

Преподаватель _____ Жангиров Т.

Санкт-Петербург
2025

Цель работы

Нахождения редакционного предписания алгоритмом Вагнера-Фишера.

Выполнение работы.

Описание реализованного алгоритма

В данной работе реализован алгоритм вычисления редакционного расстояния Левенштейна с произвольными весами операций:

- замены одного символа на другой (replace)
- вставки символа (insert)
- удаления символа (delete)
- замены одного символа на два (replace-to-two)

В коде используются две основные функции:

- getLevenshteinDistance — вычисляет стоимость преобразования строк с учетом заданных операций.
- getPrescription — восстанавливает последовательность операций, необходимых для получения строки B из строки A с минимальной стоимостью.

Алгоритм расчета минимальной стоимости (getLevenshteinDistance)

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

Инициализация:

- Первая строка (dp[0][j]) заполняется стоимостью вставок.
- Первый столбец (dp[i][0]) — стоимостью удалений.

Заполнение матрицы:

Для каждой позиции dp[i][j] вычисляется минимум из:

- Удаление: dp[i-1][j] + deleteCost
- Вставка: dp[i][j-1] + insertCost
- Замена: 0, если символы равны, иначе replaceCost

- Замена одного символа на два (если j ≥ 2): dp[i-1][j-2] + replaceToTwoCost

Алгоритм восстановления последовательности операций (getPrescription)

После построения матрицы dp, функция getPrescription по обратному пути из dp[s1.length][s2.length] к dp[0][0] восстанавливает редакционное предписание.

Для каждой позиции:

- M — совпадение символов
- R — замена
- D — удаление
- I — вставка
- T — замена одного символа на два

Формируется строка операций в обратном порядке, затем переворачивается.

Оценка сложности алгоритма

По времени:

Основной этап — заполнение матрицы размером (m+1) × (n+1), где m = len(s1), n = len(s2)

Время: O(m·n)

По памяти:

Память: O(m·n) — требуется полная матрица для хранения промежуточных значений.

Тестирование

Результаты тестирования представлены в таблице 1.

№ п/п	Входные данные	Выходные данные	Комментарии
1.	1 1 1 1 qwerty ytrewq	Редакционное расстояние: 5	Алгоритм Вагнера-Фишера. Результат вычислен верно.
2.	5 2 3 1 aabc lksa	Редакционное расстояние: 8	Алгоритм Вагнера-Фишера. Результат вычислен верно.
3.	100 100 100 100 asdf asdf	Редакционное расстояние: 0	Алгоритм Вагнера-Фишера. Результат вычислен верно.
4.	1 1 1 1 wierghwij sooidfhgi	7 TRRRRMRMD 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

Название файла: main.py

```
def levenshtein_distance_with_steps(s,
t):
    m = len(s)
    n = len(t)

    if m < n:
        return
levenshtein_distance_with_steps(t, s)

    # DP table to store distances
    dp = [[0] * (n + 1) for _ in
range(m + 1)]
    # Operation table to store
operations
    ops = [[''] * (n + 1) for _ in
range(m + 1)]

    # Initialize the first row and
column
    for j in range(n + 1):
        dp[0][j] = j
        ops[0][j] = 'I' * j  #
Insertions to build t[:j]
    for i in range(m + 1):
        dp[i][0] = i
        ops[i][0] = 'D' * i  #
Deletions to reduce s[:i] to empty

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s[i - 1] == t[j - 1]:
                dp[i][j] = dp[i - 1][j
- 1]
                ops[i][j] = ops[i -
1][j - 1] + 'M'  # Match
            else:
                # Possible operations
and their costs
                delete = dp[i - 1][j]
```

```
+ 1
                insert = dp[i][j - 1]
+ 1
                substitute = dp[i -
1][j - 1] + 1

                # Find the minimum
cost operation
                if delete <= insert
and delete <= substitute:
                    dp[i][j] = delete
                    ops[i][j] = ops[i
- 1][j] + 'D'  # Deletion
                elif insert <= delete
and insert <= substitute:
                    dp[i][j] = insert
                    ops[i][j] =
ops[i][j - 1] + 'I'  # Insertion
                else:
                    dp[i][j] =
substitute
                    ops[i][j] = ops[i
- 1][j - 1] + 'S'  # Substitution

    # Backtrack to get the sequence of
operations
    operations = []
    i, j = m, n
    while i > 0 or j > 0:
        if i > 0 and j > 0 and s[i -
1] == t[j - 1]:
            operations.append('M')  #
Match
            i -= 1
            j -= 1
        else:
            if i > 0 and dp[i][j] ==
dp[i - 1][j] + 1:
                operations.append('D')
# Deletion
                i -= 1
            elif j > 0 and dp[i][j] ==
dp[i][j - 1] + 1:
```

```
                operations.append('I')
# Insertion
                j -= 1
            elif i > 0 and j > 0 and
dp[i][j] == dp[i - 1][j - 1] + 1:
                operations.append('S')
# Substitution
                i -= 1
                j -= 1

    operations.reverse()  # Reverse to
get the correct order

    # Print the operations and steps
    print("Operations:",
''.join(operations))
    current = list(s)
    ptr_s = 0
    ptr_t = 0
    steps = []
    for op in operations:
        if op == 'M':
            steps.append(f"Match
'{s[ptr_s]}'")
            ptr_s += 1
            ptr_t += 1
        elif op == 'D':
            steps.append(f"Delete
'{s[ptr_s]}'")
            del current[ptr_s]
        elif op == 'I':
            steps.append(f"Insert
'{t[ptr_t]}'")
            current.insert(ptr_s,
t[ptr_t])
            ptr_s += 1
            ptr_t += 1
        elif op == 'S':
            steps.append(f"Substitute
'{s[ptr_s]}' with '{t[ptr_t]}'")
            current[ptr_s] = t[ptr_t]
            ptr_s += 1
            ptr_t += 1
```

```
        steps.append("Current string:
" + ''.join(current))

    print("\nSteps:")
    for step in steps:
        print(step)

    return dp[m][n]

s = input().strip()
t = input().strip()
distance =
levenshtein_distance_with_steps(s, t)
print("\nLevenshtein distance:",
distance)
```