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 _____ Гребнев Е.Д,

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, $1 \leq |S| \leq 25501 \leq |S| \leq 2550$).

Вторая строка входных данных содержит строку из строчных латинских букв. (T, $1 \leq |T| \leq 25501 \leq |T| \leq 2550$).

Параметры выходных данных:

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

Sample Input:

pedestal

stien

Sample Output:

7

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. *__init__(self, special_replacer: str = '*', special_replace_cost: float = 0.5, special_deletion_symbol: str = '#', special_deletion_cost: float = 0.5)*

   Конструктор класса. Инициализирует специальные символы и их стоимости для операций редактирования.

2. *_round_cost(self, cost: float) -> float*

   Округляет значение стоимости до 2 знаков после запятой.

3. *_deletion_cost(self, ch: str) -> float*

   Возвращает стоимость удаления символа: специальную стоимость для special_deletion_symbol, иначе 1.0.

4. *_substitution_cost(self, a: str, b: str) -> float*

   Возвращает стоимость замены символа a на b: 0.0 если символы одинаковые, специальную стоимость если b - special_replacer, иначе 1.0.

5. *_initialize_matrices(self, n: int, m: int) -> Tuple[List[List[float]], List[List[str]]]*

   Создает и возвращает две матрицы (n+1)x(m+1): для стоимостей (float) и для операций (str).

6. *_print_matrix(self, matrix: List[List[float]], title: str) -> None*

   Выводит матрицу в консоль с цветным форматированием и указанным заголовком.

7. *_fill_base_cases(self, dp: List[List[float]], ops: List[List[str]], s: str, t: str, verbose: bool) -> None*

   Заполняет базовые случаи в матрицах (первую строку и первый столбец) согласно алгоритму Левенштейна.

8. *_fill_dp_matrix(self, dp: List[List[float]], ops: List[List[str]], s: str, t: str, verbose: bool) -> None*

   Заполняет основную часть матрицы динамического программирования, вычисляя минимальные стоимости операций.

9. *_trace_operations(self, ops: List[List[str]], s: str, t: str, verbose: bool) -> None*

   Восстанавливает и выводит последовательность операций преобразования из матрицы операций.

10. *calculate(self, s: str, t: str, verbose: bool = False) -> float*
    Основная функция для расчета расстояния Левенштейна между строками s и t. Возвращает окончательное расстояние.

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, 4).

```
Computing distance between: '??h#l0 W?ld!' and 'Hello, World!'

=== INITIAL MATRIX ===
       0     1     2     3     4     5     6     7     8     9    10    11    12    13
 0:  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
 1:  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
 2:  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
 3:  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
 4:  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
 5:  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
 6:  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
 7:  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
 8:  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
 9:  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
10:  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
11:  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
12:  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0

=== INITIALIZING BASE CASES ===
Filling first column (deleting all from source):
  dp[1][0] = 0.0 + 0.1 (special, cost=0.1) = 0.1
  dp[2][0] = 0.1 + 0.1 (special, cost=0.1) = 0.2
  dp[3][0] = 0.2 + 1.0 = 1.2
  dp[4][0] = 1.2 + 1.0 = 2.2
  dp[5][0] = 2.2 + 1.0 = 3.2
  dp[6][0] = 3.2 + 1.0 = 4.2
  dp[7][0] = 4.2 + 1.0 = 5.2
  dp[8][0] = 5.2 + 1.0 = 6.2
  dp[9][0] = 6.2 + 0.1 (special, cost=0.1) = 6.3
  dp[10][0] = 6.3 + 1.0 = 7.3
  dp[11][0] = 7.3 + 1.0 = 8.3
  dp[12][0] = 8.3 + 1.0 = 9.3

Filling first row (inserting all to empty string):
  dp[0][1] = 0.0 + 1 = 1.0
  dp[0][2] = 1.0 + 1 = 2.0
  dp[0][3] = 2.0 + 1 = 3.0
  dp[0][4] = 3.0 + 1 = 4.0
  dp[0][5] = 4.0 + 1 = 5.0
  dp[0][6] = 5.0 + 1 = 6.0
  dp[0][7] = 6.0 + 1 = 7.0
  dp[0][8] = 7.0 + 1 = 8.0
  dp[0][9] = 8.0 + 1 = 9.0
  dp[0][10] = 9.0 + 1 = 10.0
  dp[0][11] = 10.0 + 1 = 11.0
  dp[0][12] = 11.0 + 1 = 12.0
  dp[0][13] = 12.0 + 1 = 13.0
```

Рисунок 1. Начало работы программы

```
=== BASE MATRIX ===
       0     1     2     3     4     5     6     7     8     9    10    11    12    13
  0:  0.0   1.0   2.0   3.0   4.0   5.0   6.0   7.0   8.0   9.0  10.0  11.0  12.0  13.0
  1:  0.1   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
  2:  0.2   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
  3:  1.2   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
  4:  2.2   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
  5:  3.2   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
  6:  4.2   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
  7:  5.2   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
  8:  6.2   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
  9:  6.3   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
 10:  7.3   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
 11:  8.3   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
 12:  9.3   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
```

Рисунок 2. Базовая матрица

```
=== FILLING DP MATRIX ===

Cell [1][1] ('?' → 'H'):
  Del: 1.0 + 0.1 = 1.1
  Ins: 0.1 + 1 = 1.1
  Sub: 0.0 + 1.0 = 1.0
  RESULT: 1.0 - Sub '?'→'H'(1.0)

Cell [1][2] ('?' → 'e'):
  Del: 2.0 + 0.1 = 2.1
  Ins: 1.0 + 1 = 2.0
  Sub: 1.0 + 0.5 = 1.5
  RESULT: 1.5 - Sub '?'→'e'(0.5)

Cell [1][3] ('?' → 'l'):
  Del: 3.0 + 0.1 = 3.1
  Ins: 1.5 + 1 = 2.5
  Sub: 2.0 + 1.0 = 3.0
  RESULT: 2.5 - Ins 'l'(1)

Cell [1][4] ('?' → 'l'):
  Del: 4.0 + 0.1 = 4.1
  Ins: 2.5 + 1 = 3.5
  Sub: 3.0 + 1.0 = 4.0
  RESULT: 3.5 - Ins 'l'(1)
```

Рисунок 3. Ход алгоритма

```
=== FINAL MATRIX ===
       0     1     2     3     4     5     6     7     8     9    10    11    12    13
  0:  0.0   1.0   2.0   3.0   4.0   5.0   6.0   7.0   8.0   9.0  10.0  11.0  12.0  13.0
  1:  0.1   1.0   1.5   2.5   3.5   4.5   5.5   6.5   7.5   8.5   9.5  10.5  11.5  12.5
  2:  0.2   1.1   1.5   2.5   3.5   4.5   5.5   6.5   7.5   8.5   9.5  10.5  11.5  12.5
  3:  1.2   1.2   1.6   2.5   3.5   4.5   5.5   6.5   7.5   8.5   9.5  10.5  11.5  12.5
  4:  2.2   2.2   1.7   2.6   3.5   4.5   5.5   6.5   7.5   8.5   9.5  10.5  11.5  12.5
  5:  3.2   3.2   2.7   1.7   2.6   3.6   4.6   5.6   6.6   7.6   8.6   9.5  10.5  11.5
  6:  4.2   4.2   3.7   2.7   2.7   3.6   4.6   5.6   6.6   7.6   8.6   9.6  10.5  11.5
  7:  5.2   5.2   4.7   3.7   3.7   3.7   4.6   4.6   5.6   6.6   7.6   8.6   9.6  10.6
  8:  6.2   6.2   5.7   4.7   4.7   4.7   4.7   5.6   4.6   5.6   6.6   7.6   8.6   9.6
  9:  6.3   6.3   5.8   4.8   4.8   4.8   4.8   5.7   4.7   5.6   6.6   7.6   8.6   9.6
 10:  7.3   7.3   6.8   5.8   4.8   5.8   5.8   5.8   5.7   5.7   6.6   6.6   7.6   8.6
 11:  8.3   8.3   7.8   6.8   5.8   5.8   6.8   6.8   6.7   6.7   6.7   7.6   6.6   7.6
 12:  9.3   9.3   8.8   7.8   6.8   6.8   6.8   7.8   7.7   7.7   7.7   7.7   7.6   6.6

=== OPERATION SEQUENCE ===
1. Del '?'(0.1)
2. Sub '?'→'H'(1.0)
3. Sub 'h'→'e'(0.5)
4. Sub '#'→'l'(1.0)
5. Keep 'l'
6. Ins 'o'(1)
7. Sub '0'→','(1.0)
8. Keep ' '
9. Keep 'W'
10. Ins 'o'(1)
11. Sub '?'→'r'(1.0)
12. Keep 'l'
13. Keep 'd'
14. Keep '!'

РЕЗУЛЬТАТ: Расстояние Левенштейна = 6.6
```

Рисунок 3. Итоговый вывод

Исследование.

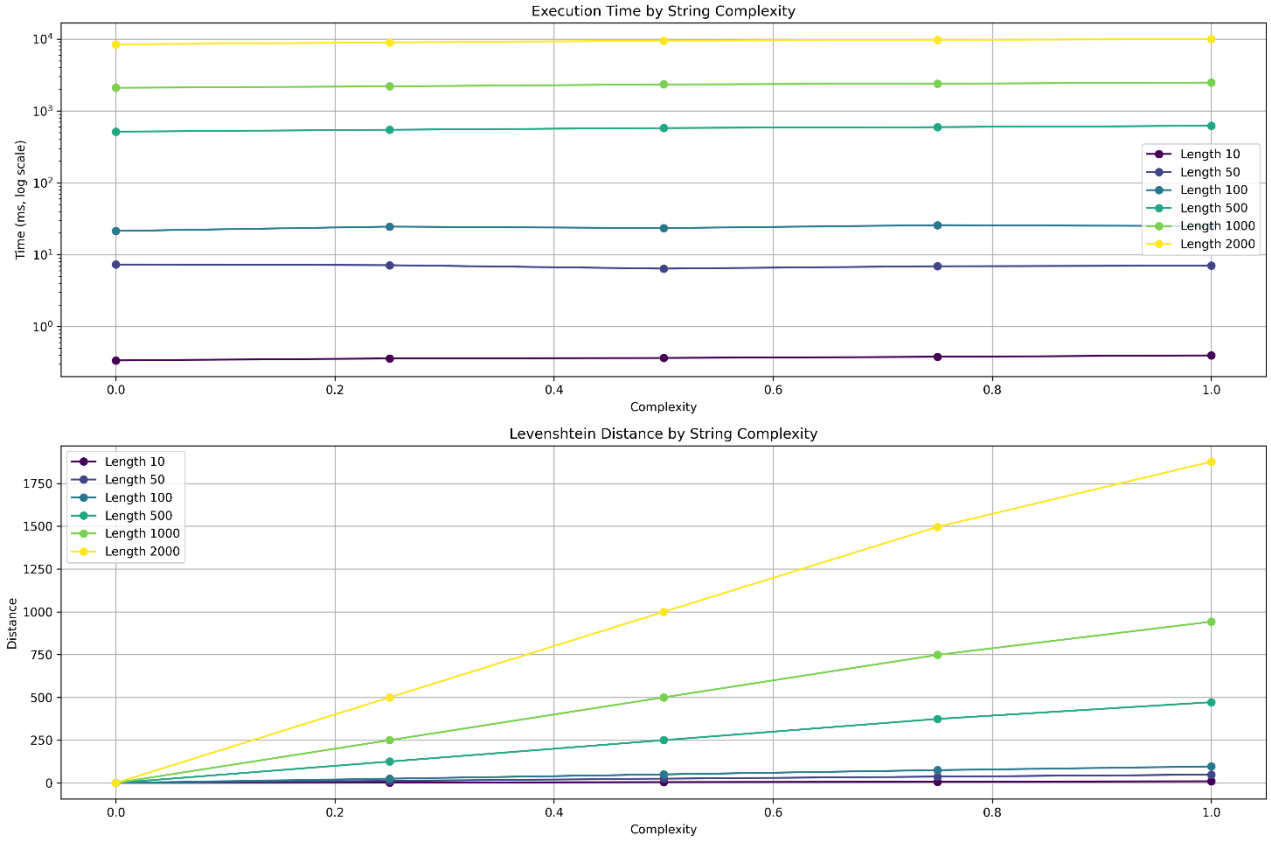

Рисунок 8 – Тестирование алгоритма на разных данных

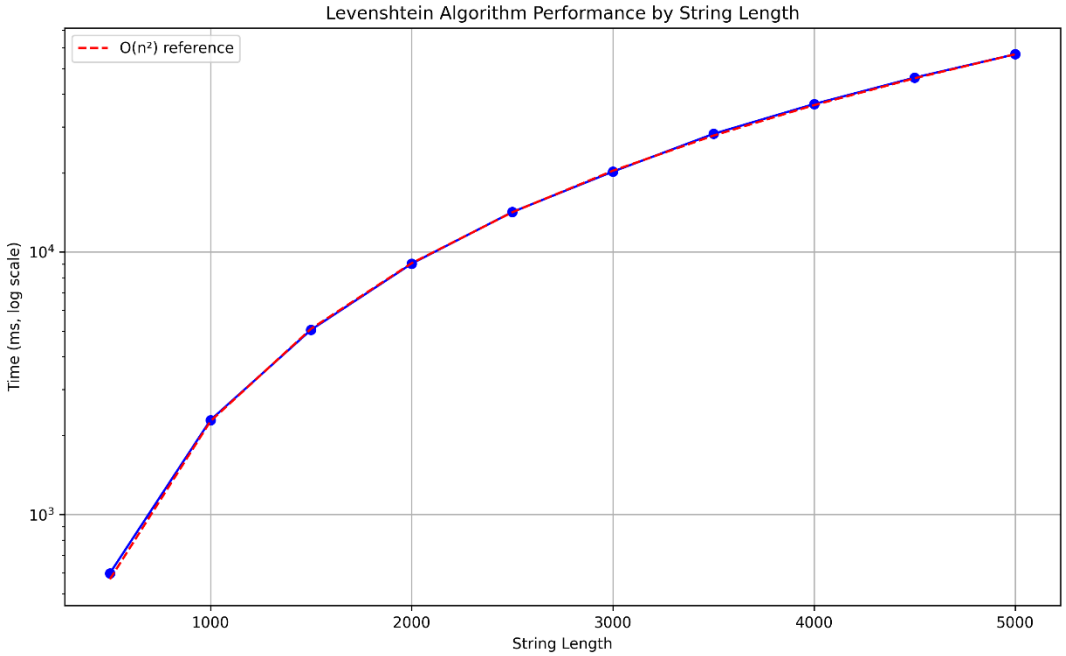

Рисунок 9 – Оценка сложности алгоритма

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:levenshtein_calculator.py

```
from typing import List, Tuple, Optional
from colorama import init, Fore, Back, Style

init(autoreset=True)

class LevenshteinCalculator:
    def __init__(self,
                 special_replacer: str = '*',
                 special_replace_cost: float = 0.5,
                 special_deletion_symbol: str = '#',
                 special_deletion_cost: float = 0.5):
        self.special_replacer = special_replacer
```

```
            self.special_replace_cost = special_replace_cost
            self.special_deletion_symbol = special_deletion_symbol
            self.special_deletion_cost = special_deletion_cost

        def _round_cost(self, cost: float) -> float:
            """Округляет стоимость до 2 знаков после запятой."""
            return round(cost, 2)


        def _deletion_cost(self, ch: str) -> float:
            """Функция стоимости удаления."""
            return self.special_deletion_cost if ch ==
self.special_deletion_symbol else 1.0

        def _substitution_cost(self, a: str, b: str) -> float:
            """Функция стоимости замены."""
            if a == b:
                return 0.0
            return self.special_replace_cost if b ==
self.special_replacer else 1.0

        def _initialize_matrices(self, n: int, m: int) ->
Tuple[List[List[float]], List[List[str]]]:
            """
            Инициализация матриц с предварительным выделением памяти.
            Используется float для поддержки дробных стоимостей.
            """
            dp = [[0.0] * (m + 1) for _ in range(n + 1)]
            ops = [[""] * (m + 1) for _ in range(n + 1)]
            return dp, ops

        def _print_matrix(self, matrix: List[List[float]], title: str) -
> None:
            """Вывод матрицы с форматированием."""
            print(Fore.YELLOW + f"\n=== {title} ===")
            if not matrix:
                print("Empty matrix")
                return
```

```
            header = "       " + "  ".join(f"{j:>5}" for j in
range(len(matrix[0])))
            print(Fore.CYAN + header)

            for i, row in enumerate(matrix):
                prefix = f"{i:>2}:" if i > 0 else "  0:"
                print(Fore.CYAN + prefix + " ".join(f"{cell:>5.1f}" if
isinstance(cell, float) else f"{cell:>5}" for cell in row))

        def _fill_base_cases(self,
                             dp: List[List[float]],
                             ops: List[List[str]],
                             s: str,
                             t: str,
                             verbose: bool) -> None:
            """Заполнение базовых случаев."""
            if verbose:
                print(Fore.YELLOW + "\n=== INITIALIZING BASE CASES ===")
                print(Fore.GREEN + "Filling first column (deleting all
from source):")

            # Fill first column (deletions)
            for i in range(1, len(s) + 1):
                cost = self._deletion_cost(s[i-1])
                dp[i][0] = self._round_cost(dp[i-1][0] + cost)
                ops[i][0] = f"Del '{s[i-1]}'({cost})"
                if verbose:
                    note = f" (special,
cost={self.special_deletion_cost})" if s[i-1] ==
self.special_deletion_symbol else ""
                    print(f"  dp[{i}][0] = {dp[i-1][0]} + {cost}{note} =
{dp[i][0]}")

            if verbose:
                print(Fore.GREEN + "\nFilling first row (inserting all
to empty string):")
```

```
            # Fill first row (insertions)
            for j in range(1, len(t) + 1):
                dp[0][j] = self._round_cost(dp[0][j-1] + 1.0)
                ops[0][j] = f"Ins '{t[j-1]}'(1)"
                if verbose:
                    print(f"     dp[0][{j}]   =   {dp[0][j-1]}   +   1   =
{dp[0][j]}")
            if verbose:
                self._print_matrix(dp, "BASE MATRIX")

        def _fill_dp_matrix(self,
                            dp: List[List[float]],
                            ops: List[List[str]],
                            s: str,
                            t: str,
                            verbose: bool) -> None:
            """Заполнение DP матрицы с минимизацией вычислений."""
            if verbose:
                print(Fore.YELLOW + "\n=== FILLING DP MATRIX ===")

            for i in range(1, len(s) + 1):
                for j in range(1, len(t) + 1):
                    # Вычисляем все возможные стоимости
                    del_cost     =     self._round_cost(dp[i-1][j]     +
self._deletion_cost(s[i-1]))
                    ins_cost = self._round_cost(dp[i][j-1] + 1.0)
                    sub_cost    =    self._round_cost(dp[i-1][j-1]    +
self._substitution_cost(s[i-1], t[j-1]))

                    if verbose:
                        print(Fore.MAGENTA + f"\nCell [{i}][{j}] ('{s[i-
1]}' → '{t[j-1]}'):")
                        print(f"           Del:      {dp[i-1][j]}      +
{self._deletion_cost(s[i-1])} = {del_cost}")
                        print(f"  Ins: {dp[i][j-1]} + 1 = {ins_cost}")
                        print(f"          Sub:     {dp[i-1][j-1]}     +
{self._substitution_cost(s[i-1], t[j-1])} = {sub_cost}")
```

```
                    # Находим минимальную стоимость
                    if sub_cost <= ins_cost and sub_cost <= del_cost:
                        dp[i][j] = sub_cost
                        if s[i-1] == t[j-1]:
                            ops[i][j] = f"Keep '{s[i-1]}'"
                        else:
                            cost = self._substitution_cost(s[i-1], t[j-
1])
                            ops[i][j]    =    f"Sub    '{s[i-1]}'→'{t[j-
1]}'({cost})"
                    elif ins_cost <= del_cost:
                        dp[i][j] = ins_cost
                        ops[i][j] = f"Ins '{t[j-1]}'(1)"
                    else:
                        dp[i][j] = del_cost
                        ops[i][j]         =         f"Del         '{s[i-
1]}'({self._deletion_cost(s[i-1])})"

                    if verbose:
                        print(Fore.BLUE  +  f"    RESULT:  {dp[i][j]}  -
{ops[i][j]}")

        def _trace_operations(self, ops: List[List[str]], s: str, t: str,
verbose: bool) -> None:
            """Восстановление последовательности операций."""
            if not verbose:
                return

            print(Fore.YELLOW + "\n=== OPERATION SEQUENCE ===")
            i, j = len(s), len(t)
            path = []

            while i > 0 or j > 0:
                op = ops[i][j]
                path.append(op)
                if "Sub" in op or "Keep" in op:
                    i -= 1
                    j -= 1
```

```
                elif "Ins" in op:
                    j -= 1
                else:
                    i -= 1

            for step, op in enumerate(reversed(path), 1):
                print(Fore.CYAN + f"{step}. {op}")

        def calculate(self, s: str, t: str, verbose: bool = False) ->
float:
            """
            Расчет расстояния Левенштейна.
            Возвращает float для поддержки дробных стоимостей.
            """
            if not s and not t:
                return 0.0

            n, m = len(s), len(t)
            dp, ops = self._initialize_matrices(n, m)

            if verbose:
                print(Fore.YELLOW + f"\nComputing distance between:
'{s}' and '{t}'")
                self._print_matrix(dp, "INITIAL MATRIX")

            self._fill_base_cases(dp, ops, s, t, verbose)
            self._fill_dp_matrix(dp, ops, s, t, verbose)

            if verbose:
                self._print_matrix(dp, "FINAL MATRIX")
                self._trace_operations(ops, s, t, verbose)

            return self._round_cost(dp[n][m])
```