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

1. *\_round\_cost(self, cost: float) -> float*

Округляет значение стоимости до 2 знаков после запятой.

1. *\_deletion\_cost(self, ch: str) -> float*

Возвращает стоимость удаления символа: специальную стоимость для special\_deletion\_symbol, иначе 1.0.

1. *\_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.

1. \_*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).

1. *\_print\_matrix(self, matrix: List[List[float]], title: str) -> None*

Выводит матрицу в консоль с цветным форматированием и указанным заголовком.

1. *\_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.

1. *\_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.

1. *\_trace\_operations(self, ops: List[List[str]], s: str, t: str, verbose: bool) -> None*

Восстанавливает и выводит последовательность операций преобразования из матрицы операций.

1. *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).

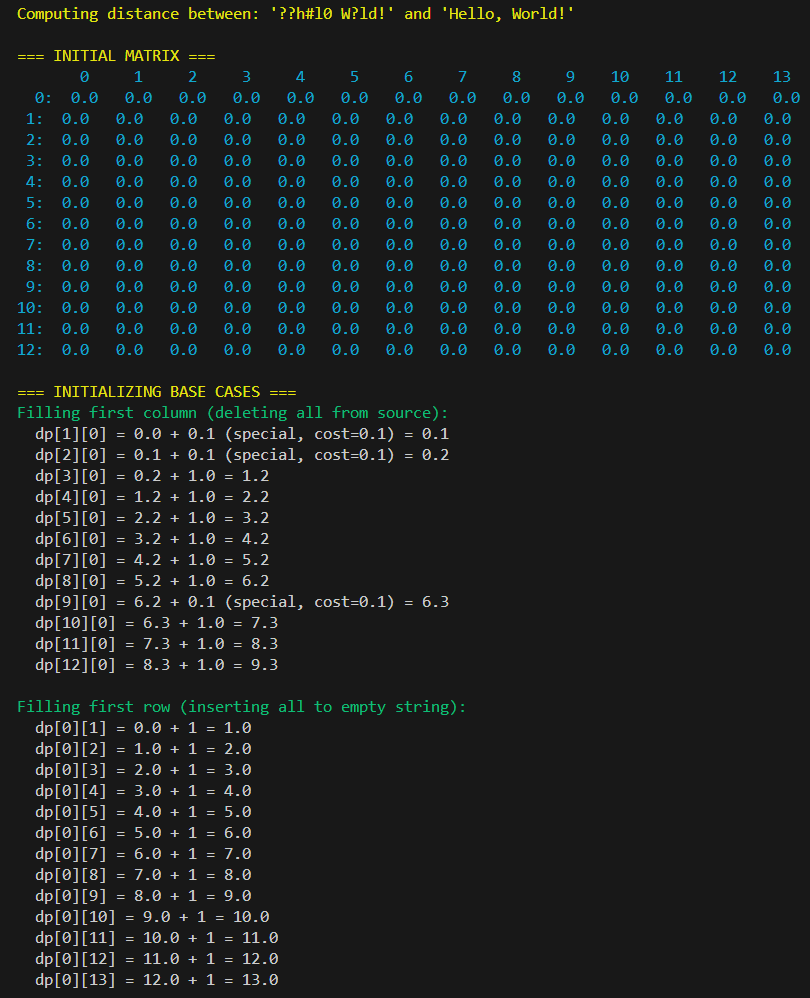


Рисунок 1. Начало работы программы

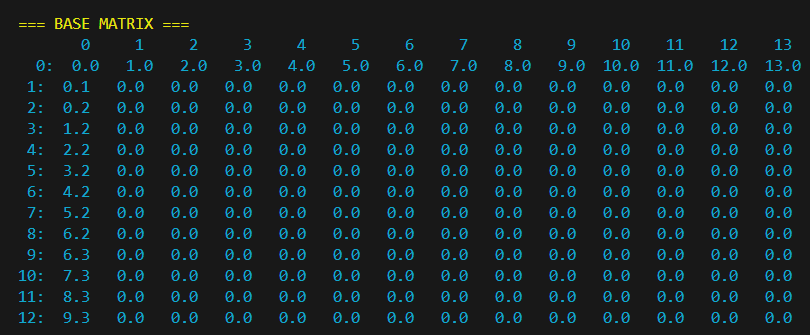


Рисунок 2. Базовая матрица

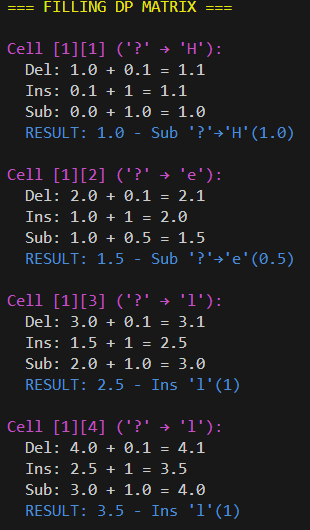


Рисунок 3. Ход алгоритма

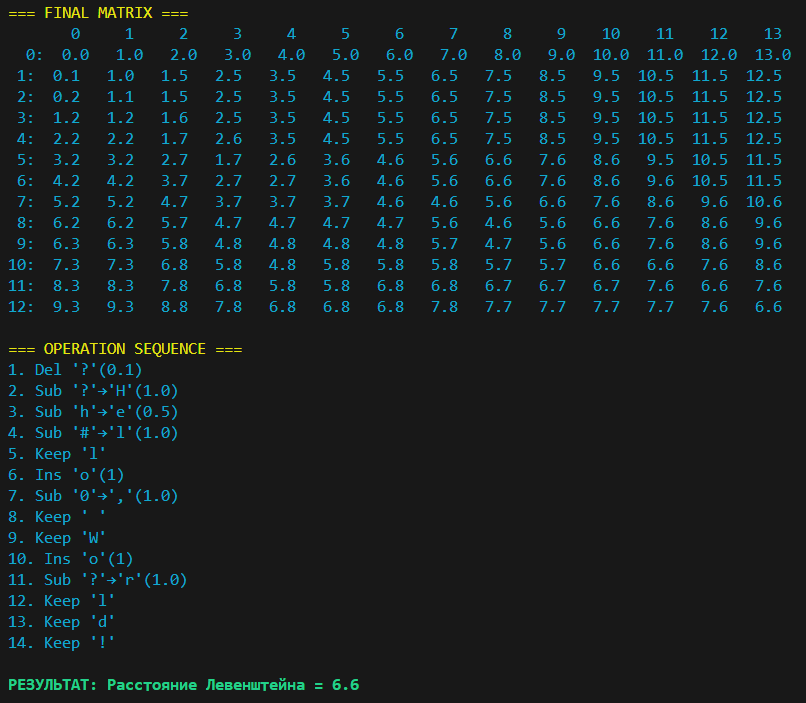


Рисунок 3. Итоговый вывод

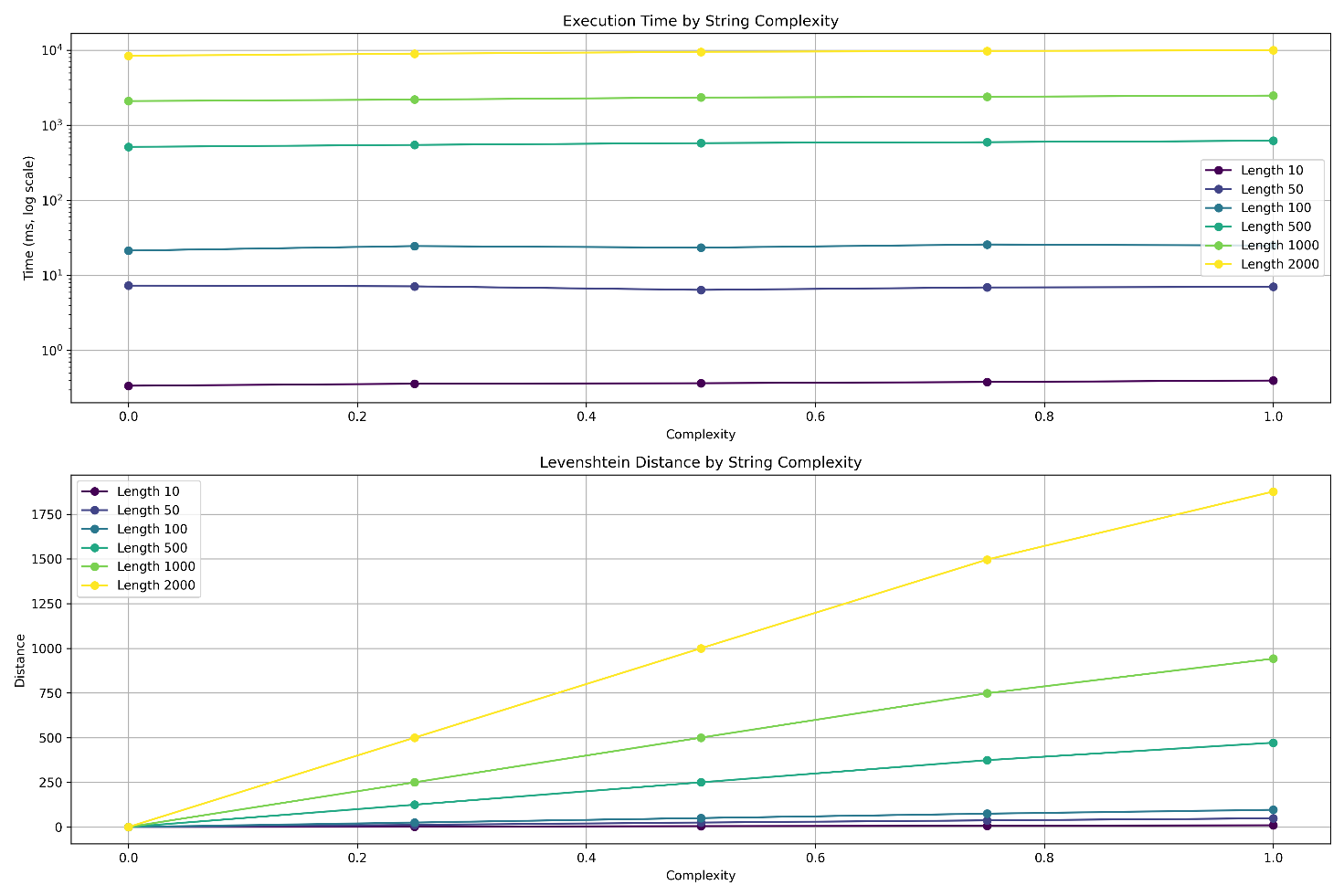
**Исследование.**

Рисунок 8 – Тестирование алгоритма на разных данных

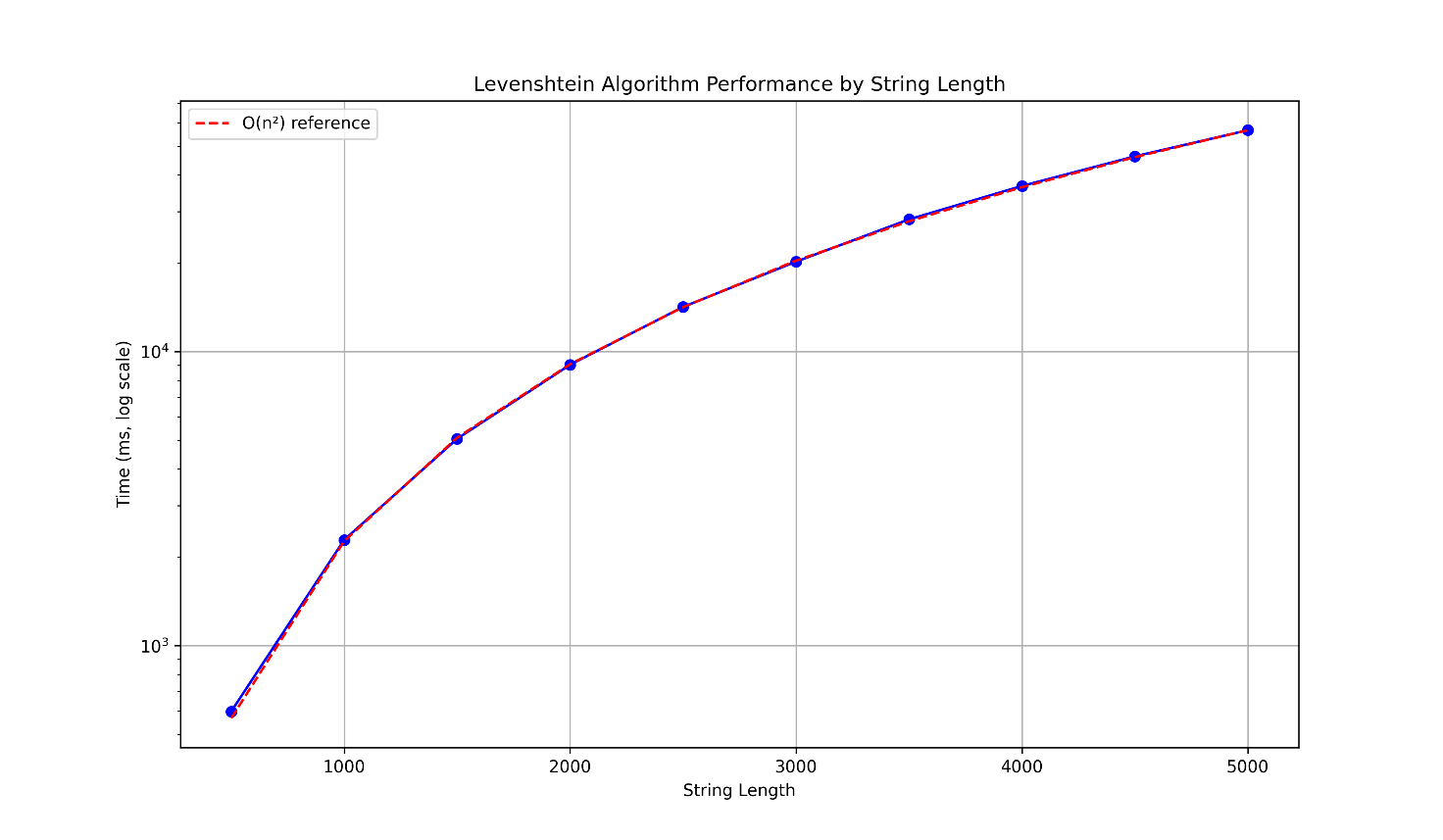


Рисунок 9 – Оценка сложности алгоритма

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])