Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ « Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №1

Группа ИУ7-51Б

Тема работы **Расстояние Левенштейна**

Студент Баранов Николай Алексеевич

Преподаватель Волкова Лилия Леонидовна

2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ **6**

1 Аналитическая часть **7**

1.1 Расстояние Левенштейна 7

1.1.1 Рекурсивный алгоритм 7

1.1.2 Матричный алгоритм 7

1.1.3 Рекурсивный алгоритм с мемоизацией 8

1.2 Расстояние Дамерау-Левенштейна 8

1.3 Вывод 8

2 Конструкторская часть **9**

2.1 Разработка алгоритмов 9

2.2 Используемые типы и структуры данных 13

2.3 Выводы 13

3 Технологическая часть **14**

3.1 Средства реализации 14

3.2 Реализация алгоритмов 14

3.3 Функциональные тесты 15

3.4 Вывод 16

4 Исследовательская часть **17**

4.1 Замеры времени работы 17

4.2 Вывод 18

ЗАКЛЮЧЕНИЕ **19**

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ **20**

ВВЕДЕНИЕ

В данной лабораторной работе рассматривается расстояние Левенштейна между двумя строками – это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения строки в другую [1].

Расстояние Левенштейна применяется для решения следующих задач:

— для исправления ошибок в слове поискового запроса;

— в формах заполнения информации на сайтах;

— для распознавания рукописных символов;

— в базах данных.

Расстояние Дамерау-Левенштейна отличается наличием ещё одной операции транспозиции (перестановки двух соседних символов).

Цель: исследование алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

Задачи:

1) описать различные алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна;
2) написать программу, реализующую несколько версий алгоритма поиска расстояния Левенштейна и одну версию алгоритма поиска расстояния Дамерау-Левенштейна;
3) выбрать инструменты для замера процессорного времени выполнения реализаций алгоритмов;
4) провести анализ затрат реализаций алгоритмов по времени с использованием микропроцессора $STM32$.

1 Аналитическая часть

1.1 Расстояние Левенштейна

1.1.1 Рекурсивный алгоритм

Пусть S_1 и S_2 – 2 строки (длиной n и m соответственно) над некоторым алфавитом [2]. Тогда расстояние Левенштейна $d(S_1, S_2)$ можно подсчитать по формуле (1.1).

$$D[i,j] = \begin{cases} 0 & ; \text{i = 0, j = 0} \\ i & ; \text{j = 0, i > 0} \\ j & ; \text{i = 0, j > 0} \\ \min\{ & \\ \quad D[i,j-1]+1 & ; \text{i > 0, j > 0} \\ \quad D[i-1,j]+1 & \\ \quad D[i-1,j-1]+m(S_1[i],S_2[j]) & \\ \} & \end{cases}, \tag{1.1}$$

Функция (1.2) показывает, была ли осуществлена замена символа.

$$m(a,b) = \begin{cases} 0 & \text{если a = b,} \\ 1 & \text{иначе} \end{cases}. \tag{1.2}$$

На основе формулы (1.1) получается рекурсивный алгоритм.

1.1.2 Матричный алгоритм

При реализации функции с использованием формулы (1.1) может возникнуть ситуация, при которой будет возникать повторное вычисление значения

для одних и тех же входных параметров. Для решения этой проблемы можно создать матрицу для сохранения промежуточных значений, после чего построчно заполнить её.

1.1.3 Рекурсивный алгоритм с мемоизацией

Данный алгоритм получается путём объединении двух предыдущих алгоритмов. Он также использует рекурсию, но вычисленные значения сохраняются в таблице, что позволяет избежать повторного вычисления одних и тех же значений.

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна отличается от расстояния Левенштейна тем, что для него также определена операция перестановки двух соседних символов. Для его вычисления к каждому результату, полученному с помощью формулы (1.1) необходимо применить формулу (1.3) [2]:

$$D[i,j] = \begin{cases} \min\{ & \\ \quad D[i,j] & \text{i > 1, j > 1,} \\ \quad D[i-2,j-2] + m(S_1[i], S_2[j]) & S_1[i] = S_2[j-1], \\ \} & S_1[i-1] = S_2[j] \\ D[i,j] & \text{иначе} \end{cases} \tag{1.3}$$

1.3 Вывод

В данном разделе были рассмотрены 3 метода для вычисления расстояний Левенштейна и Дамерау-Левенштейна.

2 Конструкторская часть

2.1 Разработка алгоритмов

На рисунках 2.1, 2.2, 2.3 и 2.4 представлены схемы алгоритмов. На вход каждому из алгоритмов подаются строки first и second длиной n и m соответственно. Алгоритм с мемоизацией также получает матрицу cache размером n+1 на m+1.

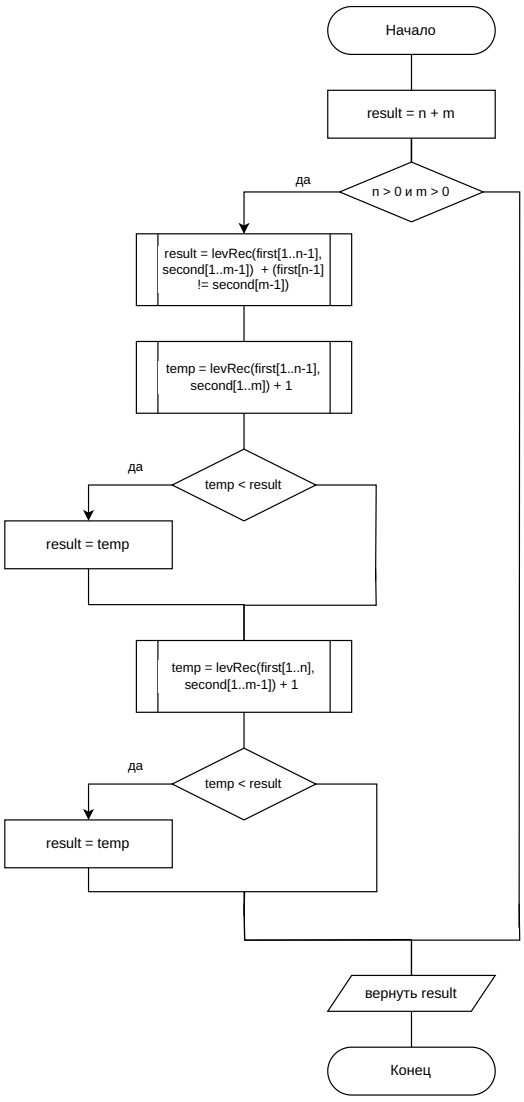

Рисунок 2.1 — Схема рекурсивного алгоритма нахождения расстояния Левенштейна

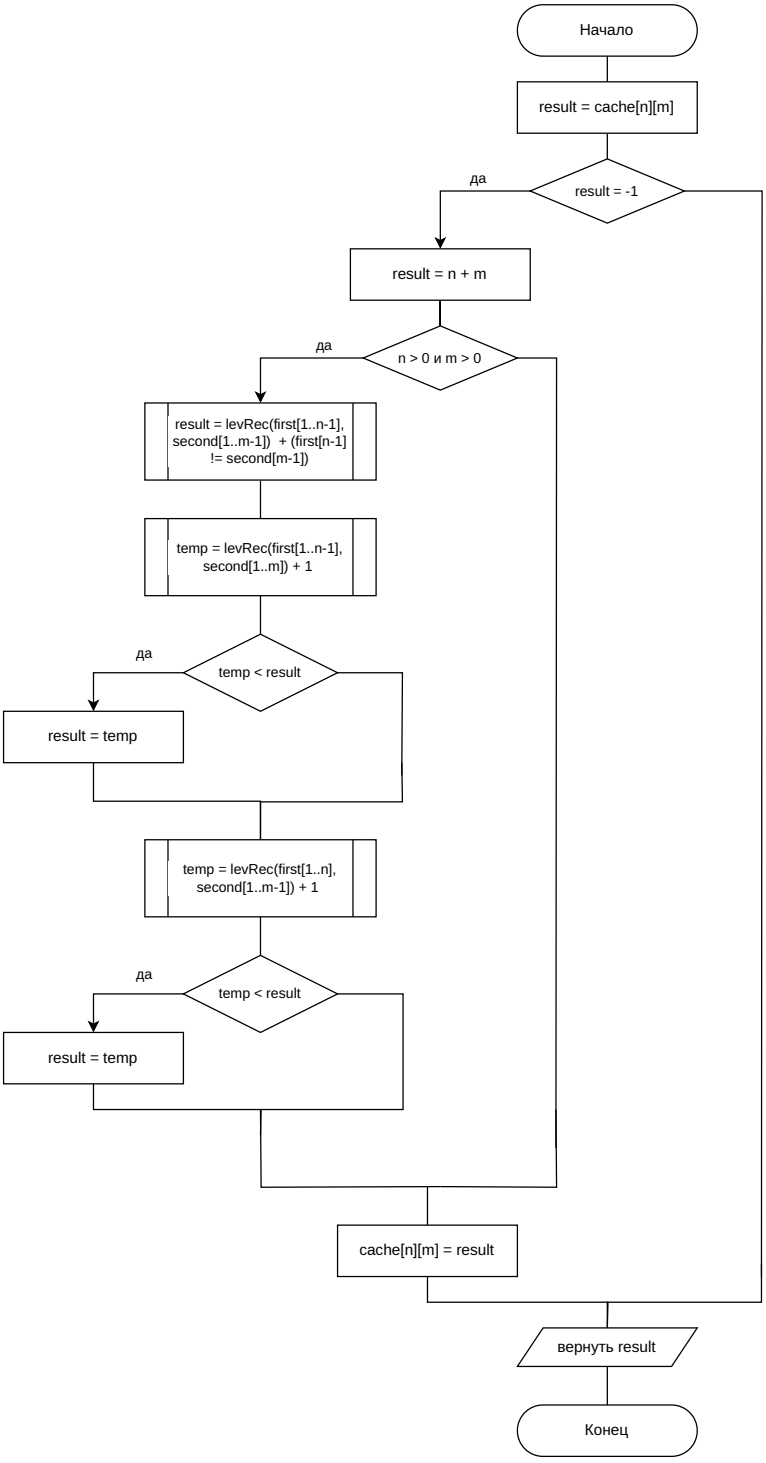

Рисунок 2.2 — Схема рекурсивного алгоритма нахождения расстояния Левенштейна с мемоизацией

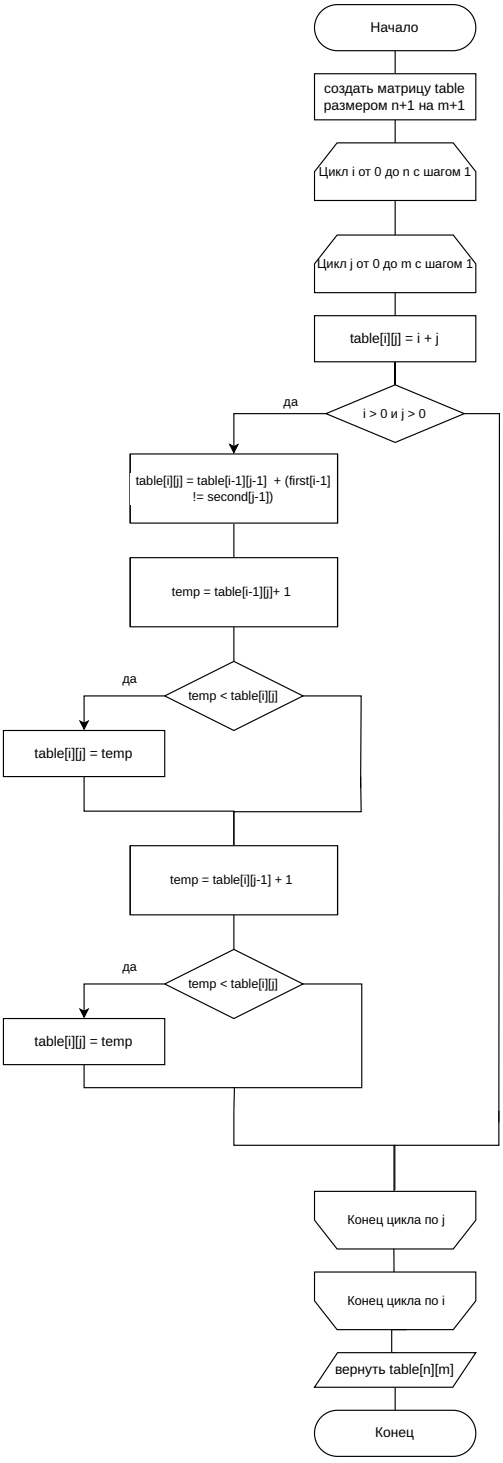

Рисунок 2.3 — Схема матричного алгоритма нахождения расстояния Левенштейна

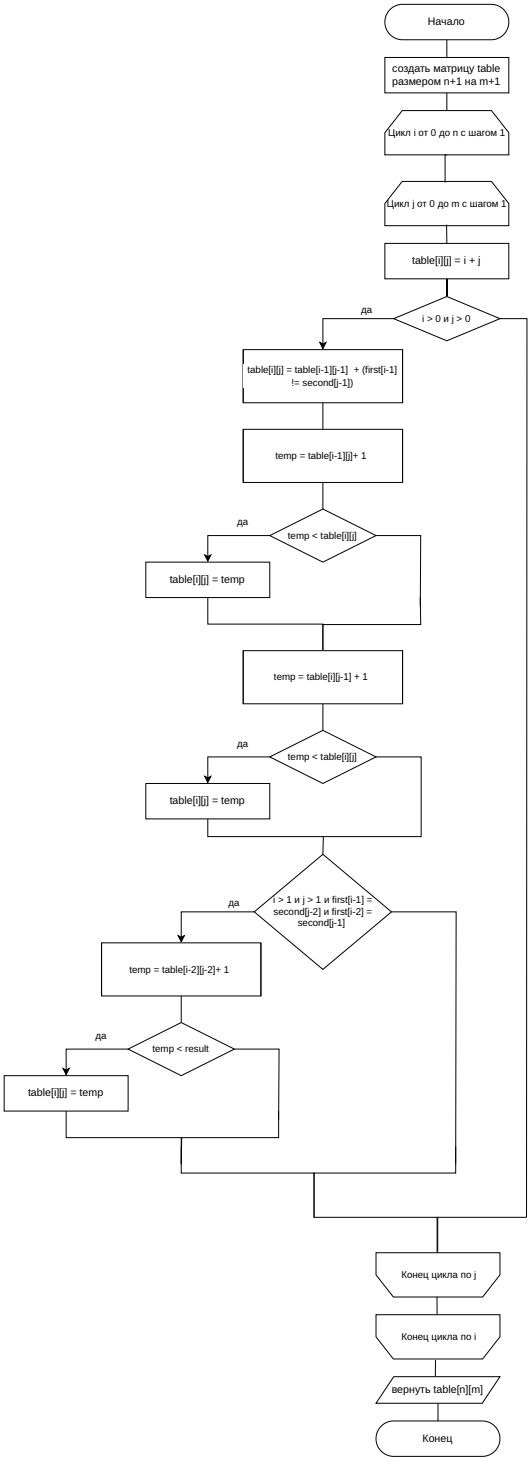

Рисунок 2.4 — Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

2.2 Используемые типы и структуры данных

При реализации алгоритмов использованы следующие структуры данных:

— строка – массив символов;

— длина строки – целое беззнаковое число;

— матрица – двумерный массив целых чисел.

2.3 Выводы

В данном разделе были построены схемы алгоритмов и выбраны структуры данных.

3 Технологическая часть

3.1 Средства реализации

Для реализации данной лабораторной работы был выбран язык Python, так как он содержит все необходимые средства для реализации алгоритмов.

3.2 Реализация алгоритмов

В листингах 3.1, 3.3, 3.2, 3.4 представлены реализации алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 — Метод рекурсивного нахождения расстояния Левенштейна

```
def levenshteinRecursive(a, b):
    result = len(a) + len(b)
    if len(a) > 0 and len(b) > 0:
        result = min(levenshteinRecursive(a[:-1], b) + 1,
           levenshteinRecursive(a, b[:-1]) + 1,
           levenshteinRecursive(a[:-1], b[:-1]) + (a[-1] != b
           [-1]))
    return result
```

Листинг 3.2 — Метод рекурсивного нахождения расстояния Левенштейна с мемоизацией

```
def levenshteinMemoised(a, b, cache = None):
    if cache is None:
        cache = [[-1 for j in range(len(b) + 1)] for i in range(
           len(a) + 1)]
    result = len(a) + len(b)
    if cache[len(a)][len(b)] != -1:
        result = cache[len(a)][len(b)]
    elif len(a) > 0 and len(b) > 0:
        result = min(levenshteinMemoised(a[:-1], b, cache) + 1,
           levenshteinMemoised(a, b[:-1], cache) + 1,
           levenshteinMemoised(a[:-1], b[:-1], cache) + (a[-1] !=
            b[-1]))
    cache[len(a)][len(b)] = result
```

```
    return result
```

Листинг 3.3 — Метод матричного нахождения расстояния Левенштейна

```
def levenshteinTable(a, b):
    table = [[i + j for j in range(len(b) + 1)] for i in range(
       len(a) + 1)]
    for i in range(1, len(a) + 1):
        for j in range(1, len(b) + 1):
            table[i][j] = min(table[i - 1][j] + 1, table[i][j -
               1] + 1, table[i - 1][j - 1] + (a[i - 1] != b[j -
               1]))
    return table[-1][-1]
```

Листинг 3.4 — Метод матричного нахождения расстояния Дамерау-Левенштейна

```
def damerauLevenshtein(a, b):
    table = [[i + j for j in range(len(b) + 1)] for i in range(
       len(a) + 1)]
    for i in range(1, len(a) + 1):
        for j in range(1, len(b) + 1):
            table[i][j] = min(table[i - 1][j] + 1, table[i][j -
               1] + 1,table[i - 1][j - 1] + (a[i - 1] != b[j -
               1]))
            if i > 1 and j > 1 and a[i - 1] == b[j - 2] and a[i -
                2] == b[j - 1]:
                table[i][j] = min(table[i][j], table[i - 2][j -
                   2] + 1)
    return table[-1][-1]
```

3.3 Функциональные тесты

В таблице 3.1 представлены функциональные тесты и результаты их выполнения. 4 числа в результатах означают возвращаемые значения всех 4 реализаций алгоритмов в следующем порядке: результат рекурсивной реализации алгоритма поиска расстояния Левенштейна, результат рекурсивной реализации с мемоизацией, результат матричной реализации, результат матричной реализации алгоритма поиска расстояния Дамерау-Левенштейна.

Таблица 3.1 — Результаты выполнения функциональных тестов

№	first	second	Ожидаемый результат	Полученный результат	Описание теста
1			0 0 0 0	0 0 0 0	Пустой ввод
2	Слово	Слово	0 0 0 0	0 0 0 0	Одинаковые слова
3	один	семь	4 4 4 4	4 4 4 4	Слова, не имеющие ни одной одинаковой буквы
4	бобер	ребро	4 4 4 4	4 4 4 4	Слова, имеющие одинаковые расстояния Левенштейна и Дамерау-Левенштейна
5	лоск	локс	2 2 2 1	2 2 2 1	Слова, имеющие различные расстояния Левенштейна и Дамерау-Левенштейна
6	белка	лак	4 4 4 3	4 4 4 3	Первое слово длиннее второго
7	лак	белка	4 4 4 3	4 4 4 3	Второе слово длиннее первого

3.4 Вывод

В данном разделе был выбран язык программирования для написания программы, были реализованы все ранее описанные алгоритмы, а также были описаны тесты с ожидаемым и полученным результатом.

4 Исследовательская часть

4.1 Замеры времени работы

Для замеров времени работы функции запускались 100 раз, на каждой итерации генерировались 2 строки одинаковой длины, состоящие из случайных цифр. Результаты измерений суммировались, после чего выводилось среднее значение. Время работы было замерено с помощью функции $ticks\_us()$ из модуля $utime$ [3]. Все замеры проводились на микропроцессоре $STM32F767ZIT6$.

При всех используемых вариантах замеры проводились на строках длины 1, 2, 3, 4, 5. Результаты представлены на рисунке 4.1.

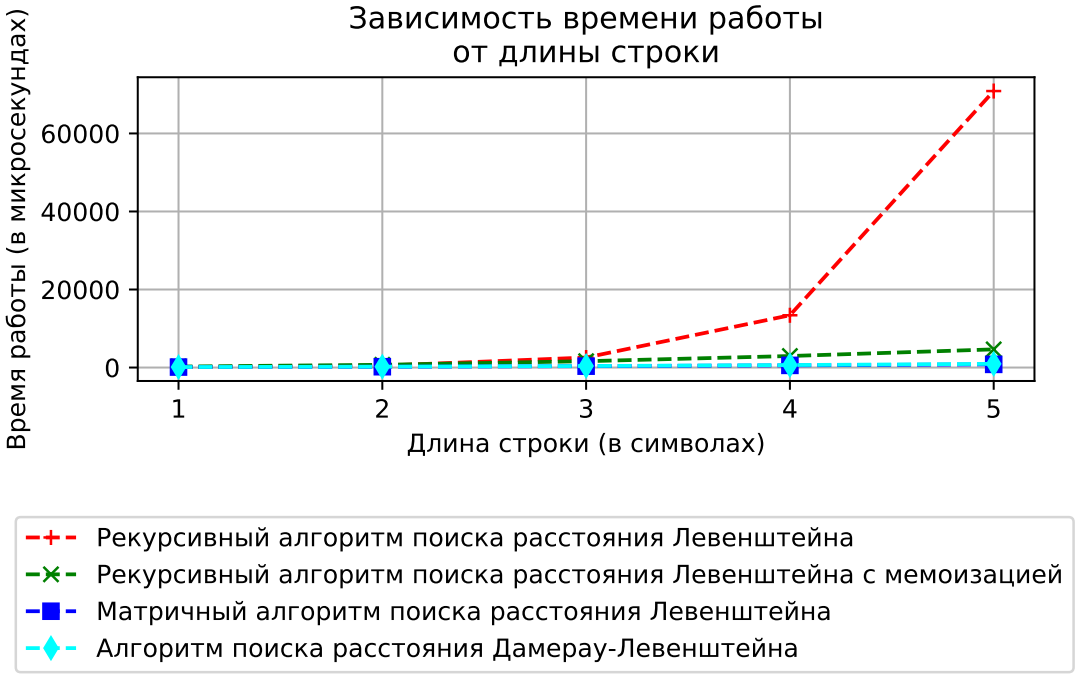

Рисунок 4.1 — Результаты замера времени работы разных алгоритмов на строках длины 1, 2, 3, 4, 5.

Рекурсивная реализация без мемоизации работает дольше остальных. Однако есть проблемы со сравнением других реализаций. На рисунке 4.2 представлены результаты для всех реализаций, кроме рекурсивной.

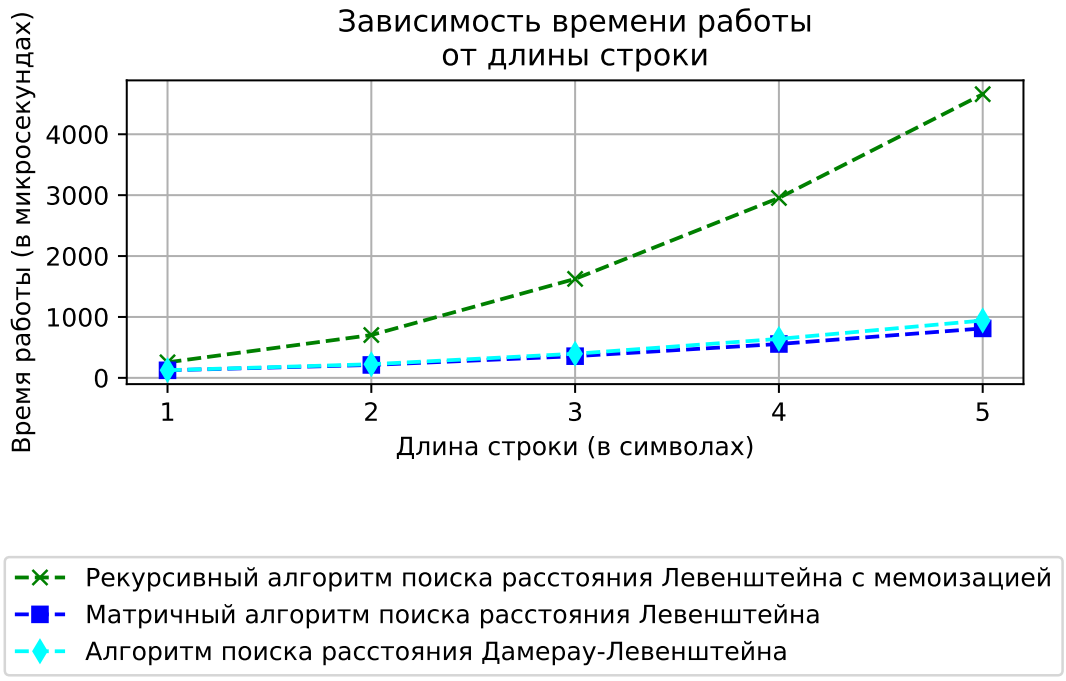

Рисунок 4.2 — Результаты замера времени работы разных алгоритмов на строках длины 1, 2, 3, 4, 5 (кроме рекурсивной реализации).

Наиболее эффективной по времени оказалась матричная реализация алгоритма поиска расстояния Левенштейна, наименее эффективной из оставшихся – рекурсивный алгоритм с мемоизацией.

4.2 Вывод

В данном разделе были проведены замеры времени. Самыми эффективными оказались матричные реализации алгоритмов, причём расстояние Левенштейна считается быстрее, чем расстояние Дамерау-Левенштейна. Далее идёт рекурсивный алгоритм с мемоизацией, и самым медленной оказалась обычная рекурсивная реализация.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы было выявлено, что наиболее эффективными оказались матричные реализации алгоритмов, при этом функции поиска расстояния Левенштейна работали быстрее, чем функции поиска расстояния Дамерау-Левенштейна. Цель работы была достигнута, для чего были описаны различные алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна, написана программа, реализующая некоторые из этих алгоритмов, выбраны инструменты для замера процессорного времени, а также проведён анализ затрат реализаций алгоритмов по времени.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Погорелов Д. А. Тарзанов А. М. Сравнительный анализ алгоритмов редакционного расстояния Левенштейна и Дамерау-Левенштейна. Международный научный журнал «Синергия наук», 2019. — С. 803–811.

2. Черненький В. Е. Гапанюк Ю. Е. Методика идентификации пассажира по установочным данным. Вестник МГТУ им. Н. Э. Баумана, 2012. — С. 31–34.

3. Документация языка $Micropython$, модуль $utime$, функция $ticks\_us$ [Электронный ресурс]. Режим доступа: `https://docs.micropython.org/en/v1.15/library/utime.html#utime.ticks_us` (Дата обращения: 01.11.2024).