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

По дисциплине: Анализ Алгоритмов

Тема: Расстояние Левенштейна и Дамерау-Левенштейна

Студент Казакова Э.М.

Группа ИУ7-56Б

Оценка (баллы) _____

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2021 г.

Содержание

Введение

Редакционное расстояние, или расстояние Левенштейна — метрика, позволяющая определить «схожесть» двух строк — минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для исправления ошибок в слове; сравнения текстовых файлов утилитой diff; в биоинформатике для сравнения генов, хромосом и белков;

Целью данной лабораторной работы является изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна.

Задачи данной лабораторной работы:

1) описать алгоритмы поиска Левенштейна и алгоритм поиска Дамерау-Левенштейна для нахождения расстояния между строками;

2) получение практических навыков реализации указанных алгоритмов;

3) сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояний между строками по затрачиваемым ресурсам(времени);

4) привести примеры работы всех указанных алгоритмов.

1 Аналитическая часть

В данной части будут рассмотрены основные теоретические аспекты, связанные с алгоритмами нахождения минимальногоо редакционного расстояния.

1.1 Описание алгоритма Левенштейна

Для поиска расстояния Левенштейна, чаще всего используют алгоритм в котором необходимо заполнить матрицу D, размером n + 1 на m + 1, где n и m – длины сравниваемых строк A и B, по следующим правилам:

- $D_{0,0} = 0$;
- $D_{i,j}$ = минимальное из:
  - $D_{i-1,j-1+0}$ (символы одинаковые), либо $D_{i-1,j-1}$ + Creplace(замена символа);
  - $D_{i,j-1}$ + Cdelete (удаление символа);
  - $D_{i-1,j}$ + Cinsert (вставка символа);

Cdelete, Cinsert, Creplace – цена или вес удаления, вставки и замены символа. При этом $D_{n-1,m-1}$ – содержит значение расстояния Левенштейна.

Пусть S1 и S2 - две строки (длиной n и m соответственно) над некоторым алфавитом, тогда редакционное расстояние D(S1,S2) можно подсчитать по следующей рекуррентной формуле (1) :

$$D(S_1[i]S_2[j]) = min \begin{cases} D(S_1[1..i], S_2[1..j-1]) + 1 \\ D(S_1[1..i-1], S_2[1..j]) + 1 \\ D(S_1[1..i-1], S_2[1..j-1]) + \begin{cases} 0, & \text{если } S_{1[i]} = S_2[j]. \\ 1, & \text{иначе.} \end{cases} \end{cases} \quad (1)$$

Рассмотрим формулу более подробно. Здесь шаг по i символизирует удаление (D) из первой строки, по j - вставку (I) в первую строку, а шаг по обоим индексам символизирует замену символа (R) или отсутствие изменений (M). Очевидно, что редакционное расстояние между двумя пустыми строками равно нулю. Так же очевидно то, что чтобы получить пустую строку из строки длиной i, следует совершить i операций удаления, а чтобы получить строку длиной j из пустой, требуется произвести j операций вставки. В нетривиальном случае необходимо выбрать минимальную «стоимость» из трёх вариантов. Вставка/удаление будет в любом случае стоить одну операцию, а вот замена может не понадобиться, если символы равны - тогда шаг по обоим индексам бесплатный. Формализация этих рассуждений приводит к формуле, указанной выше.

1.2 Описание алгоритма Дамерау-Левенштейна

В автоматической обработке естественного языка (например, при автоматической проверке орфографии) часто бывает нужно определить, насколько различны два написанных слова. Одна из количественных мер, используемых для этого, называется расстоянием Дамерау–Левенштейна - в честь Владимира Левенштейна и Фредерика Дамерау. Левенштейн придумал способ измерения «расстояний» между словами, а Дамерау независимо от него выделил несколько классов, в которые попадает большинство опечаток.

Эта вариация алгоритма вносит в определение расстояния Левенштейна еще одно правило — транспозиция (перестановка) двух соседних букв также учитывается как одна операция, наряду со вставками, удалениями и заменами.

Чтобы вычислять такое расстояние, достаточно немного модифицировать алгоритм нахождения обычного расстояния Левенштейна следующим образом: хранить не две, а три последних строки матрицы, а также добавить соответствующее дополнительное условие — в случае обнаружения транспозиции при расчете расстояния также учитывать и её стоимость.

$$D(S_1[i]S_2[j]) = min \begin{cases} D(S_1[1..i], S_2[1..j-1]) + 1 \\ D(S_1[1..i-1], S_2[1..j]) + 1 \\ D(S_1[1..i-1], S_2[1..j-1]) + \begin{cases} 0, & \text{если } S_{1[i]} = S_2[j]. \\ 1, & \text{иначе.} \end{cases} \\ D(S_1[1..i-1], S_2[1..j-1]) + \begin{cases} 1, & \text{если } S_{1[i]} = S_2[j-1] \text{ и } S_{1[i]} = S_2[j-1]. \\ \infty, & \text{иначе.} \end{cases} \end{cases}. \tag{2}$$

2 Конструкторская часть

В данном разделе будут рассмотрены схемы алгоритмов: табличный алгоритм Левенштейна, рекурсивный алгоритм Левенштейна, табличный алгоритм Дамерау-Левенштейна, рекурсивный алгоритм Дамерау-Левенштейна.

2.1 Схемы алгоритмов

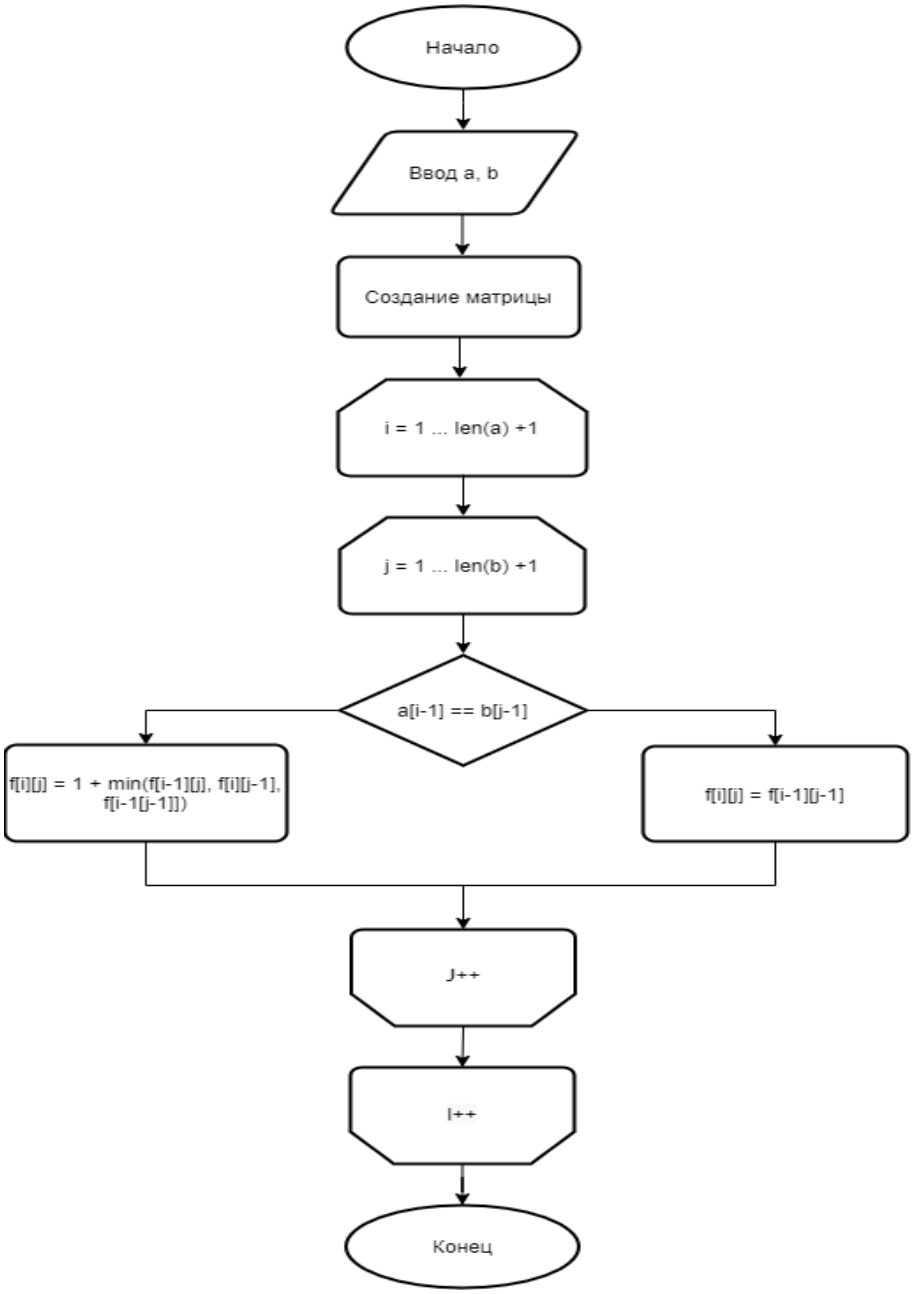

Рис. 1: Схема матричного алгоритма нахождения расстояния Левенштейна.

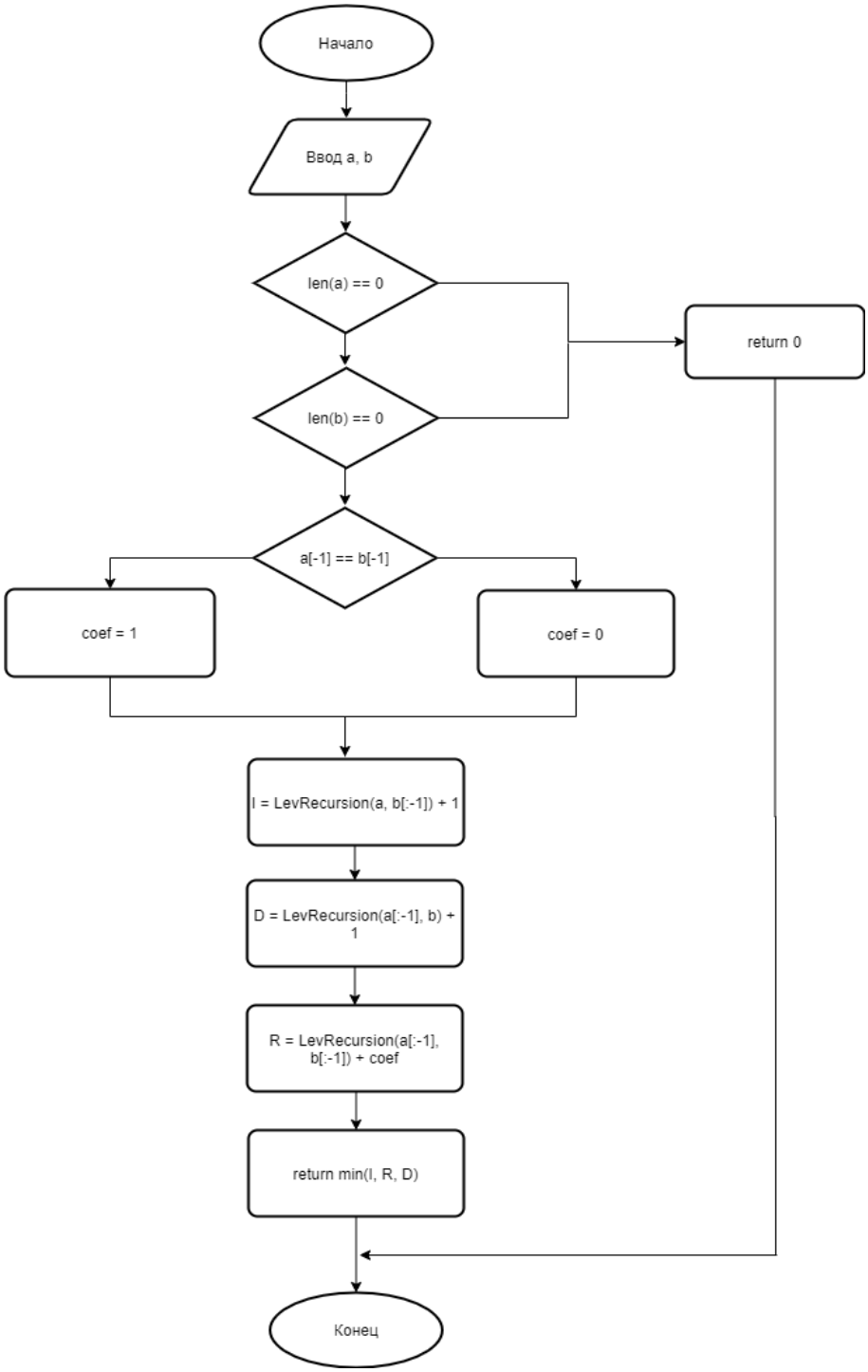

Рис. 2: Схема рекурсивного алгоритма нахождения расстояния Левенштейна.

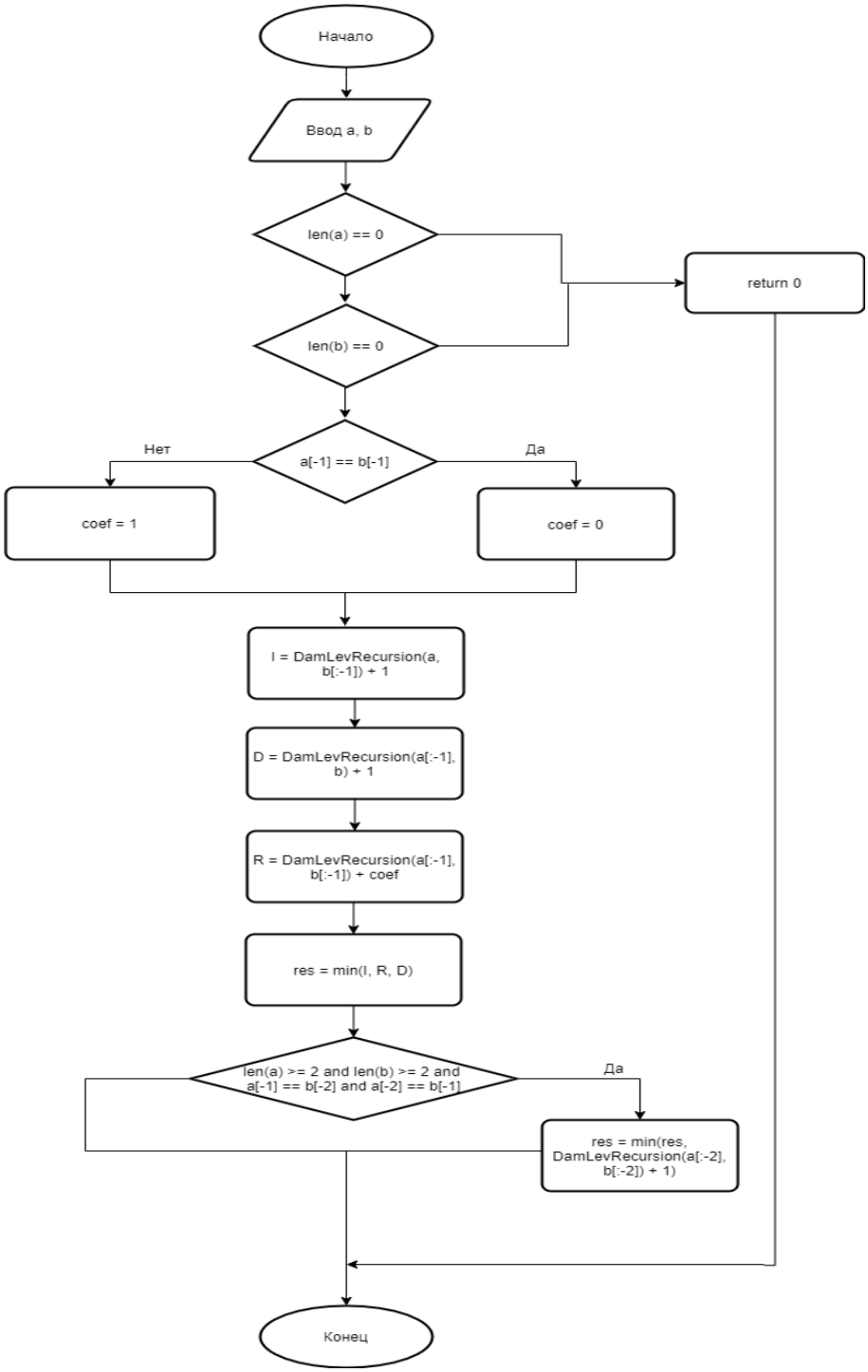

Рис. 3: Схема рекурсивного алгоритма нахождения расстояния ДамерауЛевенштейна.

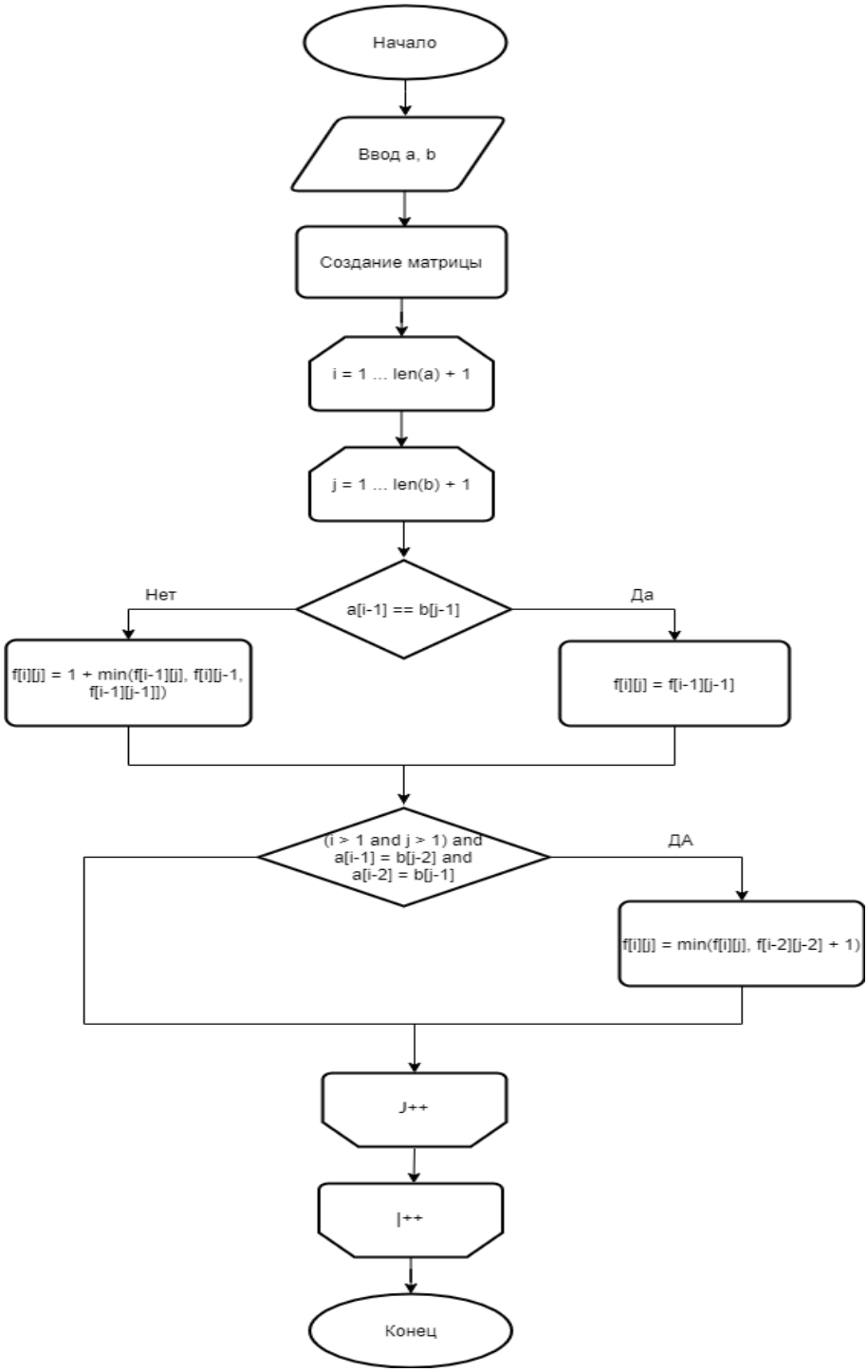

Рис. 4: Схема матричного алгоритма нахождения расстояния ДамерауЛевенштейна

3 Технологическая часть

В данном разделе будут рассмотрены требования к программному обеспечению, средства реализации и представлен листинг кода.

3.1 Требования к программному обеспечению

Программное обеспечение должно предоставлять возможность ввода двух строк (пустая строка считается корректным вводом), на выходе пользователь должен получить кратчайшее расстояние, вывод матрицы для всех алгоритмов, кроме Левенштейна с рекурсией. Также программное обеспечение должно обеспечить вывод замеров времени работы каждого из алгоритма.

3.2 Средства реализации

В данной работе используется язык программирования Python, так как ЯП позволяет написать программу за кратчайшее время.В качестве среды разработки выбрана IDLE. Для замеров времени была выбран метод process_time() модуля time, он возвращает значение (в долях секунды) системного процессорного времени текущего процесса.

3.3 Листинг кода

В листингах 3.1 - 3.4 представлена реализация алгоритмов Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Расстояние Левинштейна

```
def levinstein(a, b):
        m = [[i+j if i*j == 0 else 0 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):
                        if a[i-1] == b[j-1]:
                                m[i][j] = m[i-1][j-1]
                        else:
                                m[i][j] = 1 + min(m[i-1][j], m[i][j-1]
                                                        , m[i-1][j-1])
        return m[len(a)][len(b)]
```

Листинг 3.2 – Рекурсивное расстояние Левенштейна

```
def levinstein_recursive(a, b):
        if a ==  "" or b == "":
                return abs(len(a) - len(b))
        coef = 0 if (a[-1] == b[-1]) else 1
        return min(levinstein_recursive(a, b[:-1]) + 1,
                                levinstein_recursive(a[:-1], b) + 1,
                        levinstein_recursive(a[:-1], b[:-1]) + coef)
```

Листинг 3.3 – Расстояние Дамерау-Левенштейна

```
def damerau_levinstein(a, b):
        m = [[i+j if i*j == 0 else 0 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):
                if a[i-1] == b[j-1]:
                        m[i][j] = m[i-1][j-1]
                else:
                        m[i][j] = 1 + min(m[i-1][j], m[i][j-1], m[i-1][j-1])
                if (i > 1 and j > 1) and a[i-1] == b[j-2]
                                        and a[i-2] == b[j-1]:
                        m[i][j] = min(m[i][j], m[i-2][j-2] + 1)
        return m[len(a)][len(b)]
```

Листинг 3.4 – Рекурсивное расстояние Дамерау-Левенштейна

```
def damerau_lev_recursion(a, b):
        if a == "" or b == "":
                return abs(len(a) - len(b))
        coef = 0 if (a[-1] == b[-1]) else 1
        res = min(damerau_lev_recursion(a, b[:-1]) + 1,
                        damerau_lev_recursion(a[:-1], b) + 1,
                        damerau_lev_recursion(a[:-1], b[:-1]) + coef)
        if (len(a) >= 2 and len(b) >= 2 and a[-1] == b[-2] and a[-2] == b[-1
                res = min(res, damerau_lev_recursion(a[:-2], b[:-2]) + 1)
        return res
```

3.4 Тестирование

На рисунке 5 показаны результаты тестирования. Колонки A и B - заранее приготовленные данные.

A	B	Лев	Дамерау-Лев	Лев(рек)	Дамерау-Лев(рек)
кухня	снег	5	5	5	5
		0	0	0	0
бабушка	бабушка	0	0	0	0
пила	ипла	2	1	2	1
кинотеатр	киноо	5	5	5	5

Рис. 5: Результаты работы программы на тестовых данных

3.5 Выводы

В данном разделе была представлена реализация алгоритмов нахождения расстояния Левенштейна, Дамерау-Левенштейна, а также рекурсивные алгоритмы Левенштейна и Дамерау-Левенштейна. Программа корректно сработала на ввод пустых строк и строк, содержащих одинаковые символы, а так же при выполнении операций удаления, добавления, замены и транспозиции.

4 Экспериментальная часть

В данной части производится экспериментальное сравнение работы четырех реализованных алгоритмов (зависимость времени выполнения от длины входных слов).

4.1 Постановка эксперимента

В рамках данной лабораторной работы были проведены следующие эксперименты:

1. Сравнение алгоритмов Левенштейна и Дамерау-Левенштейна. Количество символов в слове от 0 до 800 с шагом 50;
2. Сравнение рекурсивного Левенштейна, матричного Левенштейна и рекурсивного Дамерау-Левенштейна. Количество символов в слове от 1 до 10 с шагом 2;

4.2 Сравнительный анализ на материале экспериментальных данных

На рисунке 6 можно увидеть сравнение времени работы алгоритмов Левенштейна и Дамерау-Левенштейна.

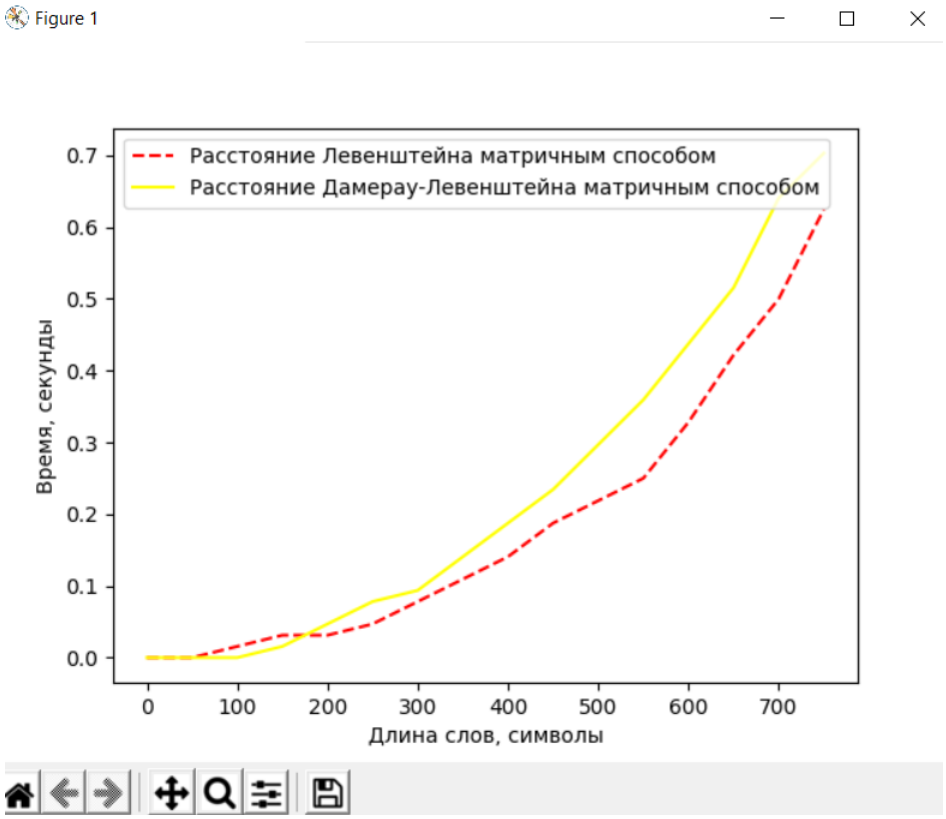

Рис. 6: График работы алгоритмов Левенштейна и Дамерау-Левенштейна

На рисунке 7 сравнение времени работы рекурсивных алгоритмов Левенштейна и Дамерау-Левенштейна.

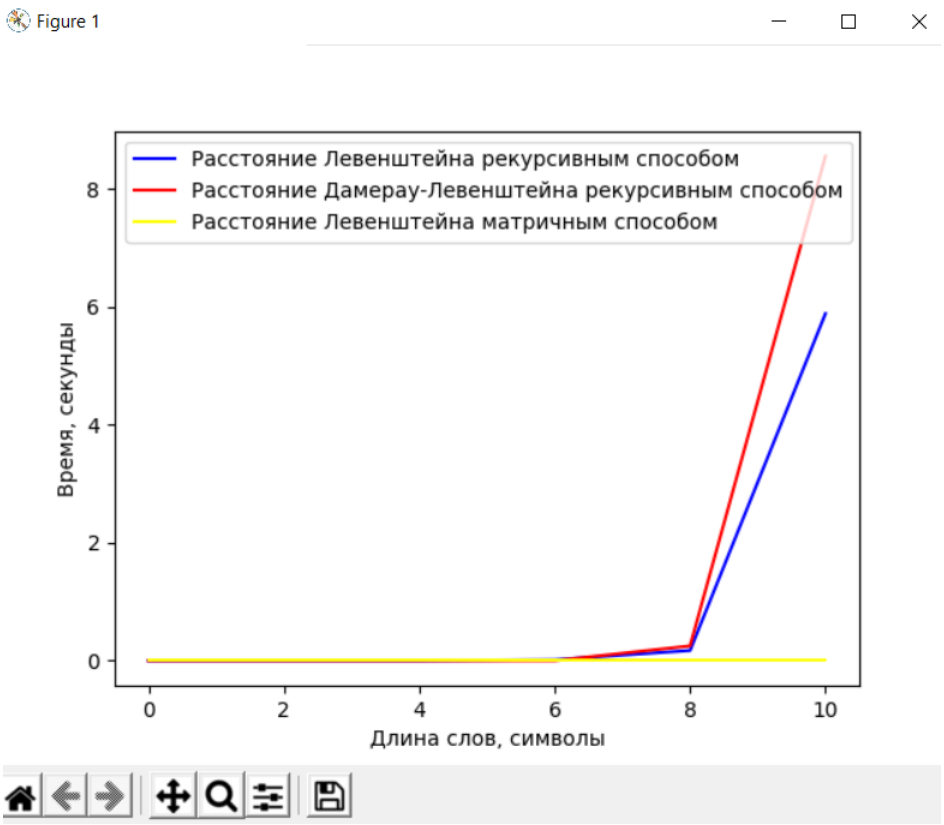

Рис. 7: График работы рекурсивных алгоритмов Левенштейна и Дамерау-Левенштейна, и матричного алгоритма Левенштейна

4.3 Выводы

В результате проведенного эксперимента был получен следующий вывод: рекурсивный алгоритм Левенштейна работает намного медленнее матричной реализации. Алгоритм Дамерау-Левенштейна незначительно проигрывает во времени алгоритму Левенштейна, это связано с дополнительными операциями, выполняющимися в ходе работы алгоритма. При сравнении графиков работы рекурсивных алгоритмой Левенштейна и Дамерау-Левенштейна можно заметить, что оба алгоритмы имеют одинаковый характер, но по скорости рекурсивный алгоритм Левенштейна работает быстрее, чем рекурсивный алгоритм Дамерау-Левенштейна.

Заключение

В ходе работы были изучены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна (рекурсивный и матричный). Выполнено сравнение рекурсивного и матричного алгоритмов ДамерауЛевенштейна. При сравнении данных алгоритмов пришли к выводу, что рекурсивный алгоритм является самым медленным, поэтому по времени намного эффективнее использовать матричные алгоритмы.