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 Криков А. В.

Группа ИУ7-53Б

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

Москва — 2021 г.

Содержание

Введение

В данной лабораторной работе будет рассмотрен алгоритм под названием "расстояние Левенштейна".

Данное расстояние показывает минимальное количество редакторских операций (вставки, замены и удаления), которые необходимы для перевода одной строки в другую. Это расстояние помогает определить схожесть двух строк.

Расстояние Левенштейна используется в компьютерной лингвистике для:

- исправления ошибок в слове
- сравнения текстовых файлов утилитой diff
- в биоинформатике для сравнения генов и белков

Однако кроме упомянутых трех ошибок (вставка лишнего символа, пропуск символа, замена одного символа другим), пользователь может нажимать на нужные клавиши не в том порядке. С этой проблемой поможет справится расстояние Дамерау-Левенштейна. Данное расстояние задействует еще одну редакторскую операцию - транспозицию.

Целью данной работы является анализ и реализация алгоритма Дамерау-Левенштейна и Левенштейна.

Для достижения поставленной цели необходимо выполнить следующие задачи:

- исследовать и сравнить алгоритмы нахождения редакционного расстояния (алгоритмы Левенштейна и Дамерау–Левенштейна);
- привести схемы рассматриваемых алгоритмов;
- описать используемые структуры данных;
- оценить объем памяти для хранения данных;
- описать структуру разрабатываемого ПО;
- определить средства прогарммной реализации;

- протестировать разработанное ПО;
- провести сравнительный анализ алгоритмов на основе экспериментальных;
- подготовить отчет по лабораторной работе.

1 Аналитическая часть

В данном разделе будут рассмотрены алгоритмы нахождения [1] редакционного расстояния, используемые в данной лабораторной работе.

Для преобразования одного слова в другое используются следующие операции:

- D - удаление
- I - вставка
- R - замена

Будем считать стоимость каждой вышеизложенной операции - 1. Введем понятие совпадения - M. Его стоимость будет равна - 0.

1.1 Расстояние Левенштейна

Расстояние Левенштейна между двумя строками a и b может быть вычислено по формуле:

$$D(i,j) = \begin{cases} 0 & \text{i = 0, j = 0} \\ i & \text{j = 0, i > 0} \\ j & \text{i = 0, j > 0} \\ \min\{ & \\ \qquad D(i, j-1) + 1 & \\ \qquad D(i-1, j) + 1 & \text{i > 0, j > 0} \\ \qquad D(i-1, j-1) + m(a[i], b[j]) & (1.2) \\ \} & \end{cases}, \tag{1.1}$$

где функция 1.2 определена как:

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

1.2 Расстояние Дамерау — Левенштейна

В расстоянии Дамерау - Левенштейна задействуют еще одну операцию - транспозицию T. Расстояние Дамерау — Левенштейна может быть найдено по формуле 1.3.

$$D_{a,b}(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j)=0, \\ \min\{ & \\ \quad D_{a,b}(i,j-1)+1, & \\ \quad D_{a,b}(i-1,j)+1, & \\ \quad D_{a,b}(i-1,j-1)+m(a[i],b[j]), & \text{иначе} \\ \quad \left[\begin{array}{ll} D_{a,b}(i-2,j-2)+1, & \text{если } i,j>1; \\ & a[i]=b[j-1]; \\ & b[j]=a[i-1] \\ \infty, & \text{иначе} \end{array} \right. & \\ \} & \end{cases}, \tag{1.3}$$

Вывод

Формулы Левенштейна и Дамерау — Левенштейна для рассчета расстояния между строками задаются рекурсивно, а следовательно, алгоритмы могут быть реализованы рекурсивно или итерационно. Входными данными являются две строки на русском или английском языке в любом регистре. Выходными данными является целое число — искомое расстояние для всех четырех методов и матрицы расстояний для всех методов, за исключением рекурсивного. В связи с ограничениями на ПО, входные данные должны быть корректными.

2 Конструкторская часть

В данном разделе будут рассмотрены схемы алгоритмов нахождения редакционного расстояния. Также будут описаны структуры данных и будет оценена используемая алгоритмами память.

2.1 Разработка алгоритмов

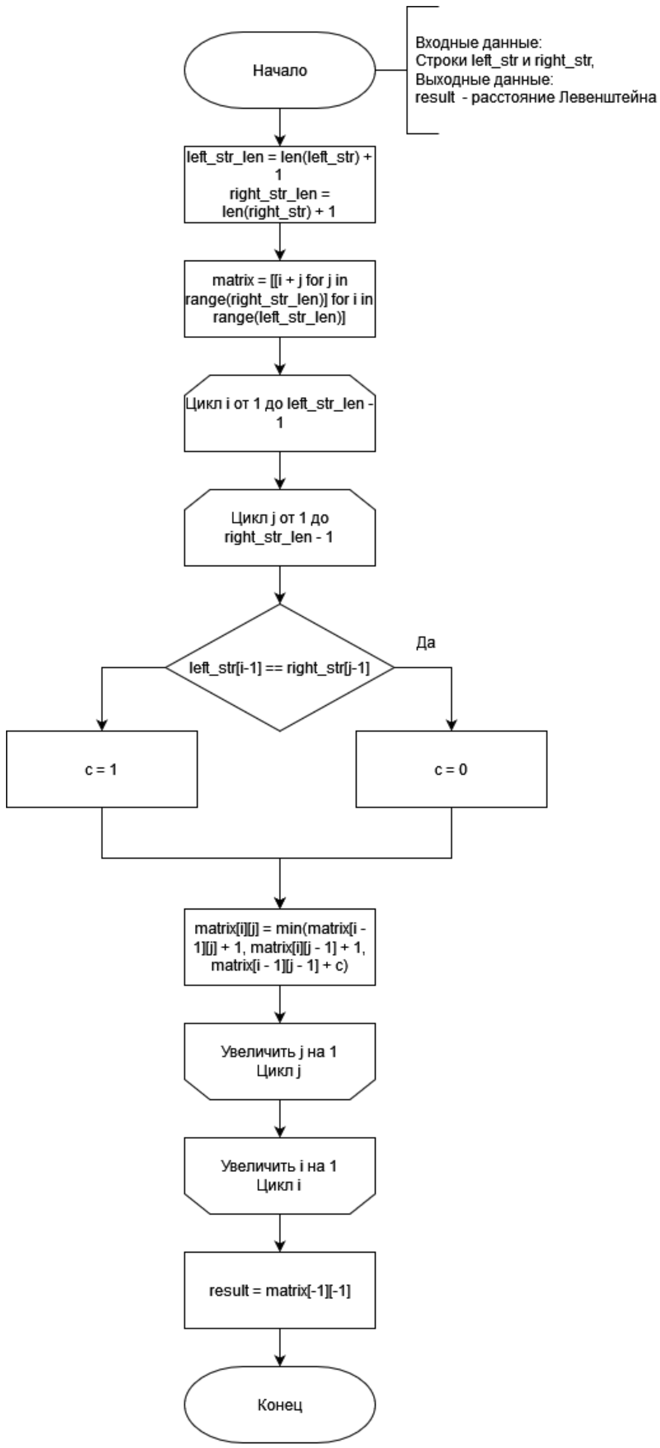

Рисунок 2.1 – Схема алгоритма поиска расстояния Левенштейна

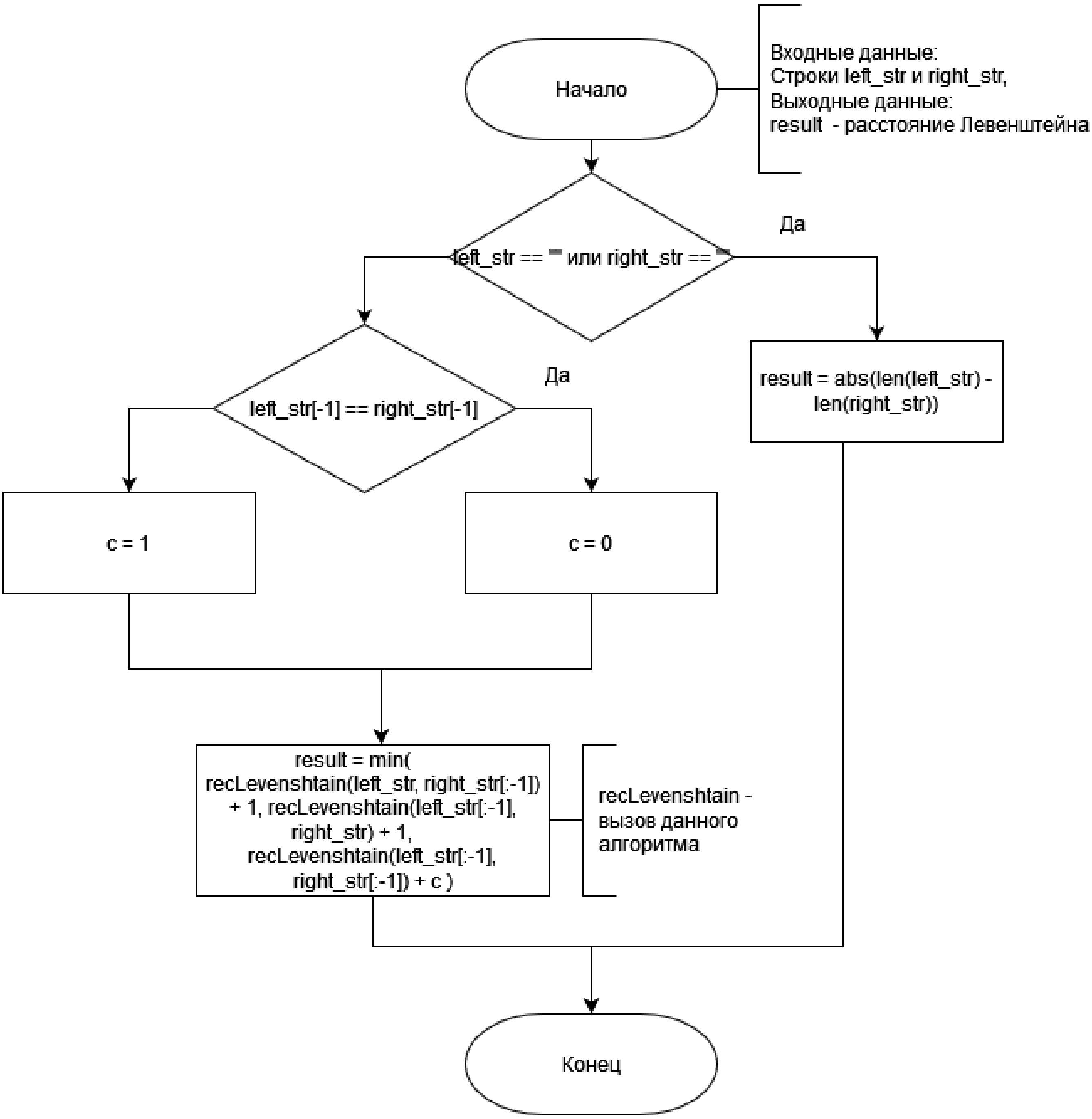

Рисунок 2.2 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна

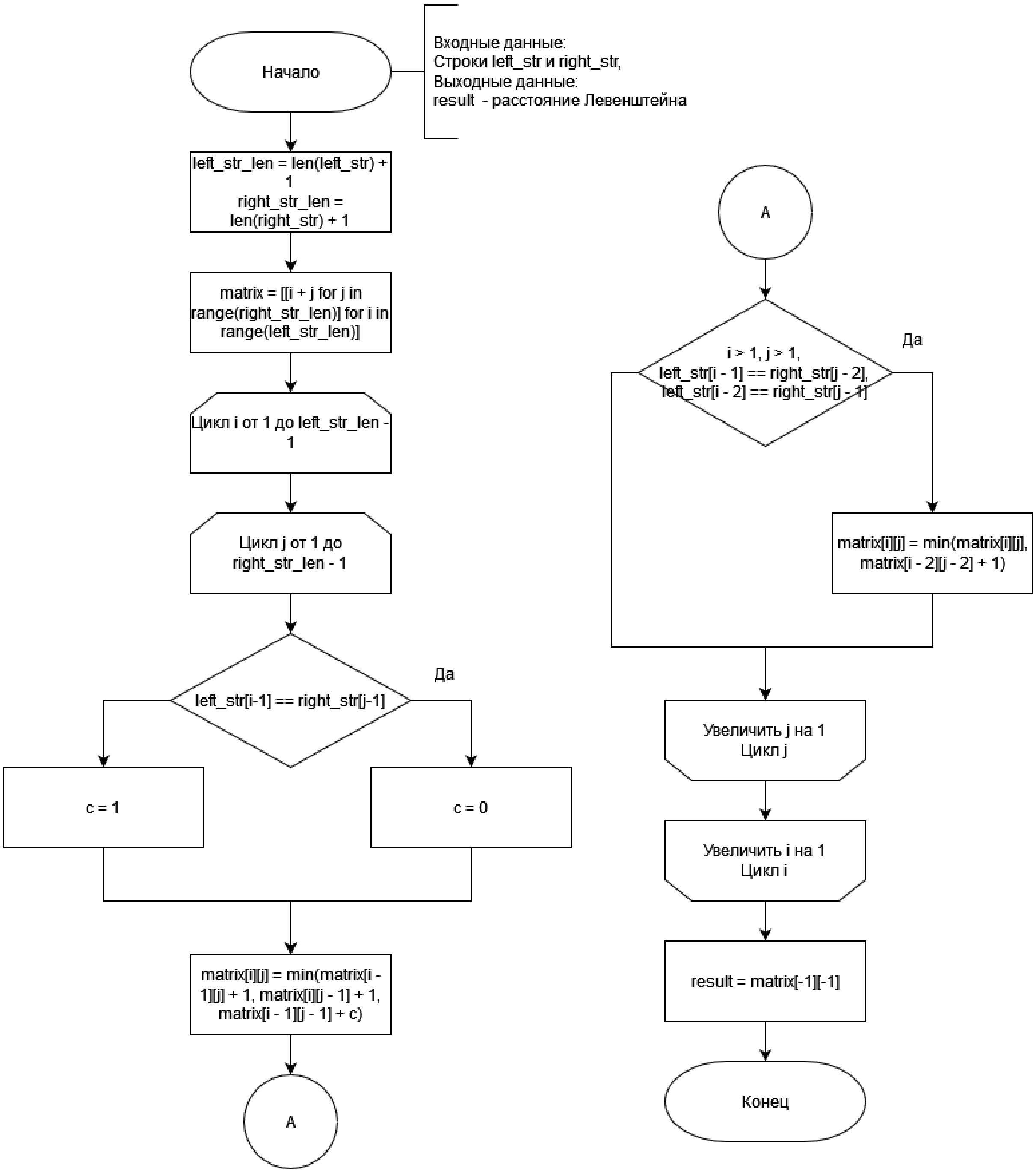

Рисунок 2.3 – Схема алгоритма нахождения расстояния Дамерау-Левенштейна

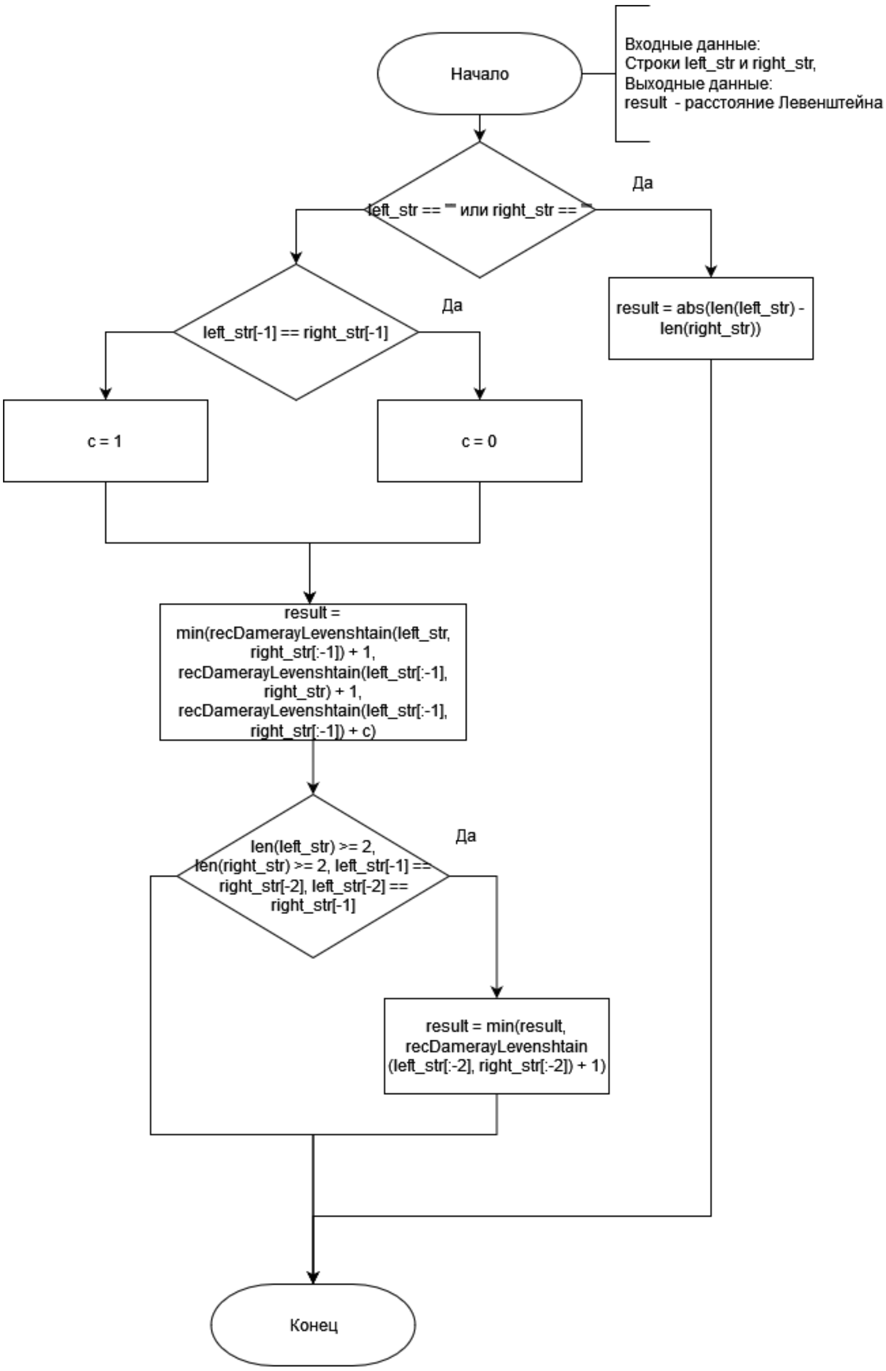

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

2.2 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

- строка типа String заданного размера;
- длина строки - целое число типа int;

2.3 Описание способов тестирования

Данные алгоритмы нахождения редакционного расстояния можно протестировать функционально. При этом можно выделить следующие классы эквивалентности:

- Две пустые строки
- Две одинаковые строки
- Нахождение расстояния при помощи операции вставки
- Нахождение расстояния при помощи операции замены
- Нахождение расстояния при помощи операции удаления
- Нахождения расстояния при помощи всех трех операций

2.4 Характеристики по памяти

Алгоритмы нахождения расстояний Левенштейна и Дамерау — Левенштейна не отличаются друг от друга с точки зрения использования памяти, следовательно, достаточно рассмотреть лишь разницу рекурсивной и матричной реализаций этих алгоритмов.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, при этом для каждого вызова рекурсии в моей реализации требуется:

- переменная типа int, в моем случае: 4 байта;
- 2 аргумента типа строка: $2 \cdot 24 = 48$ байт;
- адрес возврата: 8 байт;
- место для записи возвращаемого функцией значения: 8 байт.

Таким образом получается, что при обычной рекурсии на один вызов требуется (2.1):

$$M_{percall} = 4 + 48 + 8 + 8 = 68 \tag{2.1}$$

Следовательно память, расходуемая в момент, когда стек вызовов максимален, равна (2.2):

$$M_{recursive} = 80 \cdot depth \tag{2.2}$$

где $depth$ - максимальная глубина стека вызовов, которая равна (2.3):

$$depth = |S_1| + |S_2| \tag{2.3}$$

где S_1, S_2 - строки.

Память, требуемая для при итеративной реализации, состоит из следующего:

- 2 локальные переменные типа int, в моем случае: $2 \cdot 4 = 8$ байт;
- 2 аргумента типа строка: $2 \cdot 24 = 48$ байт;
- адрес возврата: 8 байт;
- место для записи возвращаемого функцией значения: 8 байт;
- матрица: M_{Matrix} размером $4 \cdot (n + 1) \cdot (m + 1)$.

Таким образом общая расходуемая память итеративных алгоритмов (2.4):

$$M_{iter} = M_{Matrix} + 72 \tag{2.4}$$

Вывод

На основе теоретических данных, полученных из аналитического раздела, были построены схемы обоих алгоритмов (Левенштейна и Дамерау-Левенштейна). Оценены их трудоемкости в лучшем и худшем случаях. Также были описаны используемые структуры данных.

3 Технологическая часть

В данном разделе приведены требования к программному обеспечению, выбор ЯП и листинги кода.

3.1 Средства реализации

Для разработки ПО был выбран язык Java [2], поскольку он предоставляет разработчику широкий спектр возможностей и позволяет разрабатывать кроссплатформенные приложения. В качестве среды разработки была выбрана IntelliJ IDEA. IntelliJ IDEA [3] подходит не только для Windows, но и для Linux.

3.2 Листинг кода

В листинге 3.1 приведена реализация алгоритмов нахождения расстояния Левенштейна и Дамерау — Левенштейна, а также вспомогательные функции.

Листинг 3.1 – Листинг с алгоритмами

```java
import java.util.ArrayList;

public class Levenshtein {
    public static int getIntBoolean(char first, char second) {
        if (first == second) {
            return 0;
        }
        return 1;
    }

    public static int Levenshtein(String first, String second) {
        int n = first.length();
        int m = second.length();

        int[][] matrix = new int[n + 1][m + 1];
        matrix[0][0] = 0;
        for (int i = 1; i < n + 1; i++) {
            matrix[i][0] = i;
        }
```

```java
        for (int i = 1; i < m + 1; i++) {
            matrix[0][i] = i;
        }

        for (int i = 1; i < n + 1; i++) {
            for (int j = 1; j < m + 1; j++) {
                matrix[i][j] = Math.min(
                    Math.min(
                            matrix[i - 1][j - 1] + getIntBoolean(first.charAt(i - 1),
                            second.charAt(j -1)), matrix[i - 1][j] + 1),
                    matrix[i][j - 1] + 1
                );
            }
        }

        return matrix[n][m];
    }

    public static int LevenshteinRecursion(String first, String second) {
        if (first == "" || second == "") {
            return Math.abs(first.length() - second.length());
        }

        int temp = (first.charAt(first.length() - 1) == second.charAt(second.length() -
            1)) ? 0 : 1;
        return Math.min(
                Math.min(
                        LevenshteinRecursion(first, second.substring(0, second.length() -
                            1)) + 1,
                        LevenshteinRecursion(first.substring(0, first.length() - 1),
                            second) + 1),
                LevenshteinRecursion(first.substring(0, first.length() - 1),
                    second.substring(0, second.length() - 1)) + temp
        );
    }


    public static int DamerauLevenshteinRecursion(String first, String second) {
        if (first == "" || second == "") {
            return Math.abs(first.length() - second.length());
        }
        int temp = (first.charAt(first.length() - 1) == second.charAt(second.length() -
            1)) ? 0 : 1;

        int result = Math.min(
                DamerauLevenshteinRecursion(first, second.substring(0, second.length() -
                    1)) + 1,
                Math.min(
```

```
                        DamerauLevenshteinRecursion(first.substring(0, first.length() - 1), second) + 1,
                        DamerauLevenshteinRecursion(first.substring(0, first.length() - 1), second.substring(0, second.length() - 1)) + temp
                )
        );

        if (first.length() > 1 && second.length() > 1 &&
            first.charAt(first.length() - 1) == second.charAt(second.length() - 2) &&
            first.charAt(first.length() - 2) == second.charAt(second.length() - 1) ) {
                result = Math.min(
                        result,
                        DamerauLevenshteinRecursion(first.substring(0, first.length() - 2),
                                second.substring(0, second.length() - 2)) + 1);
            }

        return result;
    }


    public static int DamerauLevenshtein(String first, String second) {
        int n = first.length();
        int m = second.length();

        int[][] matrix = new int[n + 1][m + 1];
        matrix[0][0] = 0;
        for (int i = 1; i < n + 1; i++) {
            matrix[i][0] = i;
        }
        for (int i = 1; i < m + 1; i++) {
            matrix[0][i] = i;
        }

        for (int i = 1; i < n + 1; i++) {
            for (int j = 1; j < m + 1; j++) {
                matrix[i][j] = Math.min(
                        Math.min(
                                matrix[i - 1][j - 1] + getIntBoolean(first.charAt(i - 1), second.charAt(j -1)),
                                matrix[i - 1][j] + 1),
                        matrix[i][j - 1] + 1
                );
                if (i > 1 && j > 1 && first.charAt(i - 1) == second.charAt(j - 2) &&
                    first.charAt(i - 2) == second.charAt(j - 1)) {
                    matrix[i][j] = Math.min(matrix[i][j], matrix[i - 2][j - 2] + 1);
                }
            }
        }
```

```
        return matrix[n][m];
    }
}
```

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна и Дамерау — Левенштейна. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Строка 1	Строка 2	Ожидаемый результат	
		Левенштейн	**Дамерау — Левенштейн**
cvd	dvc	2	2
mother	money	3	3
just	jest	1	1
turnover	turnovre	2	1
member	morning	6	6
death	health	2	2

Вывод

В данном разделе были рассмотрены листинги кода, был выбран язык программирования и среда разработки, а также было произведено функциональное тестирование. Сравнивая листинги программ видно, что написание рекуррентных подпрограмм значительно проще, чем матричных.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Windows 10. [4]
- Память: 16 GiB.
- Процессор: Intel(R) Core(TM) i7-4700HQ CPU @ 2.40GHz. [5]

4.2 Временные характеристики

Для сравнения времени выполнения программ брались строки длиной [10, 20, 30, 50, 100, 200] символов. Для получения точных временных характеристик замеры времени прогонялись 500 раз.

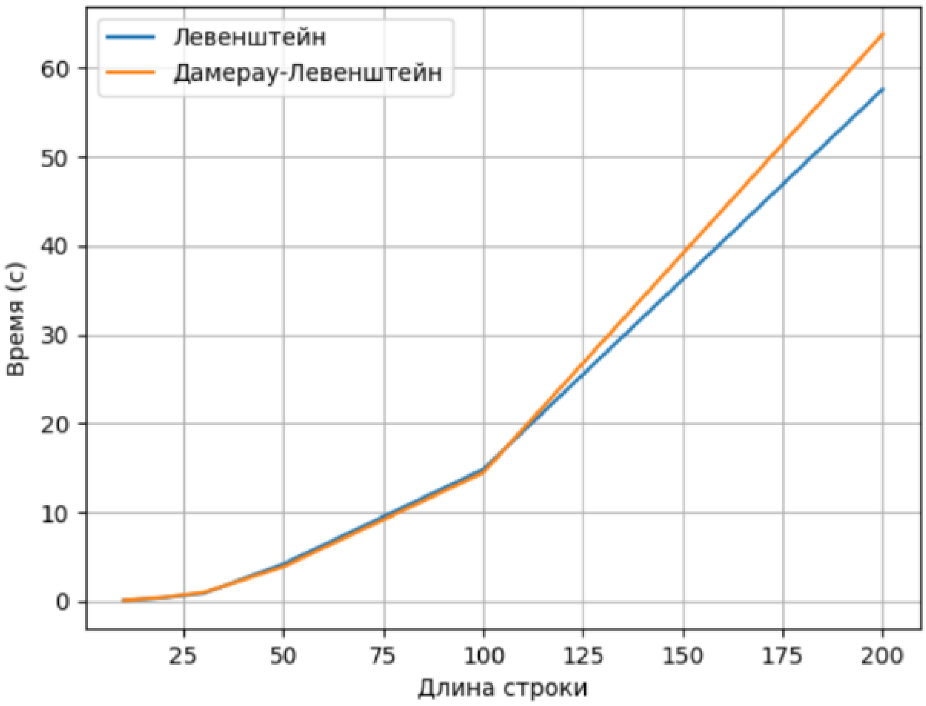

Рисунок 4.1 – Сравнение времени работы алгоритма поиска расстояния Левенштейна и Дамерау-Левенштейна

Вывод

В результате эксперимента было получено, что на случайных данных, алгоритм Дамерау-Левенштейна работает быстрее алгоритма Левенштейна. Например при длине строки в 200 символов алгоритм Дамерау-Левенштейна работает на 10% быстрее алгоритма Левенштейна. Также в результате эксперимента было получено, что при длине строки меньше 100 символов алгоритмы работают за одинаковое время. Можно сделать вывод, что при размерности строк > 100 символов лучше использовать алгоритм Дамерау-Левенштейна.

Заключение

Алгоритмы Левенштейна и Дамерау-Левенштейна являются самыми популярными алгоритмами, которые помогают найти редакторское расстояние.

В ходе выполнения лабораторной работы была проделана следующая работа:

- были теоретически изучены алгоритмы нахождения расстояний Левенштейна и Дамерау–Левенштейна;
- для некоторых реализаций были применены методы динамического программирования, что позволило сделать алгоритмы быстрее;
- были практически реализованы алгоритмы в 2 вариантах: рекурсивном и итеративном;
- на основе полученных в ходе экспериментов данных были сделаны выводы по поводу эффективности всех реализованных алгоритмов;
- был подготовлен отчет по ЛР.

Анализируя результат проведенных экспериментов, приходим к выводу, что наиболее эффективным алгоритмом для нахождения редакционного расстояния наиболее эффективным является алгоритм Дамерау-Левенштейна. Также важно отметить что при длине строки < 100 символов алгоритмы Левенштейна и Дамерау-Левенштейна работают за одинаковое время.

Список литературы

[1] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. – М.: Доклады АН СССР, 1965. Т. 163. С. 845–848.

[2] Документация по Java [Электронный ресурс]. Режим доступа: `https://docs.oracle.com/en/java/`.

[3] Visual Studio Code. 2005. URL: https://code.visualstudio.com/.

[4] Windows. 1985. URL: https://www.microsoft.com/ru-ru/windows.

[5] Процессор Intel® Core™ i7-8550U [Электронный ресурс]. Режим доступа: `https://ark.intel.com/content/www/ru/ru/ark/products/122589/intel-core-i7-8550u-processor-8m-cache-up-to-4-00-ghz.html`.