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

Студент Шацкий Р.Е.

Группа ИУ7-55Б

Оценка (баллы) _________________

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

Москва — 2021 г.

Оглавление

Введение

Расстояние Левенштейна — минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для преобразования одной строки в другую.

Ниже перечислены области применения расстояния Левенштейна в теории информации и компьютерной лингвистике.

- исправления ошибок в слове;
- сравнения текстовых файлов утилитой diff;
- сравнения генов, хромосом и белков в биоинформатике.

Ниже перечислены цели лабораторной работы.

1. Изучить методы динамического программирования на основе алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.
2. Оценить реализации алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Ниже перечислены задачи лабораторной работы.

1. Изучить принципы работы алгоритмов Левенштейна и Дамерау-Левенштейна.
2. Применить методы динамического программирования для матричной реализации указанных алгоритмов.
3. Получить практические навыки реализации данных алгоритмов: итерационные и рекурсивные версии.
4. Сравнить итерационные и рекурсивные реализации алгоритмов по затрачиваемым ресурсам (время и память).
5. Получить экспериментальным методом различия во временной эффективности алгоритмов.
6. Описать и обосновать полученные результаты в отчете о выполненной лабораторной работе в формате расчетно-пояснительной записки.

1 Аналитическая часть

Расстояние Левенштейна [?] между двумя строками — минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для преобразования одной строки в другую.

Цена каждой операции может зависеть от вида операции или от участвующих в ней символов, отражающих вероятность разных ошибок при вводе текста.
Далее рассмотрены виды операций и их цены.

1. Вставка (insert), $w(a, \lambda)$ — цена удаления символа a.
2. Удаление (delete), $w(\lambda, b)$ — цена вставки символа b.
3. Замена (replace), $w(a, b)$ — цена замены символа a на символ b.

Для решения задачи о нахождении редакционного расстояния необходимо найти последовательность операций, при которых суммарная цена операций будет минимальной. Расстояние Левенштейна - частный случай решения этой задачи при перечисленных ниже условиях.

- $w(a, a) = 0$;
- $w(a, b) = 1, a \neq b$;
- $w(a, \lambda) = 1$;
- $w(\lambda, b) = 1$.

1.1 Рекурсивный алгоритм нахождения расстояния Левенштейна

В основе вычисления расстояние Левенштейна между двумя строками a и b лежит формула 1.1.

- $|a|$ — длина строки a;
- $a[i]$ — i-ый символ строки a.

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

Функция 1.2 позволяет сравнить два символа:

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

Рекурсивный алгоритм реализует формулу 1.1. Логика функции D описана ниже.

1. Для получения из пустой строки пустой строки, требуется 0 операций.
2. Для получения из пустой строки строки b требуется $|b|$ операций (все - insert).
3. Для получения из строки a пустой строки требуется $|a|$ операций (все - delete).
4. Для получения из строки a строки b требуется выполнить несколько операций. Обозначая a' и b' за строки a и b без последнего символа соответсвенно, цену преобразования из строки a в b можно выразить следующим образом:
   (a) Сумма цены преобразования строки a' в b и цены операции удаления (для преобразования a' в a)
   (b) Сумма цены преобразования строки a в b' и цены операции вставки (для преобразования b' в b)
   (c) Сумма цены преобразования строки a' в b' и операции замены (если a и b оканчиваются на разные символы)
   (d) Цена преобразования строки a' в b' (если a и b оканчиваются на одинаковый символ)

   Минимальная цена преобразования — минимальное значение из приведенных выше вариантов.

1.2 Итерационный алгоритм нахождения расстояния Левенштейна с использованием матрицы

Прямая реализация формулы 1.1 может быть неэффективна при больших значениях длин строк, поскольку промежуточные значения функции $D(i,j)$ вычисляются несколько раз. Для оптимизации алгоритма можно использовать матрицу для хранения промежуточных значений функции $D(i,j)$. В таком случае алгоритм выполняет построчное заполнение матрицы, пока не дойдет до крайнего правого элемента, в котором будет записано итоговое расстояние Левенштейна.

1.3 Рекурсивный алгоритм нахождения расстояния Левенштейна с использованием матрицы

Основной недостаток обычного рекурсивного алгоритма — многократное вычисления промежуточных значений функции D(i, j). Этот недостаток можно устранить, и сделать таким образом алгоритм более эффективным по времени выполнения, если добавить матрицу, которая будет заполняться промежуточными значениями D(i, j). Если рекурсивный алгоритм обрабатывает данные, которые еще ни разу не были поданы, результат записывается в матрицу. Если рекурсивный алгоритм обрабатывает данные, которые уже были обработаны, берется старый результат из матрицы.

1.4 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна очень похоже на расстояние Левенштейна, но в его поиске используется еще одна операция - **транспозиция** (перестановка двух соседних символов).

Расстояние Дамерау-Левенштейна между двумя строками a и b определяется функцией 1.3:

$$d_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{если } \min(i,j) = 0 \\ \min \begin{cases} d_{a,b}(i-1,j)+1 \\ d_{a,b}(i,j-1)+1 \\ d_{a,b}(i-1,j-1)+m(a[i],b[j]) \\ d_{a,b}(i-2,j-2)+1 \end{cases} & \begin{array}{c} \text{если } i,j > 1 \\ \text{и } a[i] = b[j-1] \\ \text{и } a[i-1] = b[j] \end{array} \\ \min \begin{cases} d_{a,b}(i-1,j)+1 \\ d_{a,b}(i,j-1)+1 \\ d_{a,b}(i-1,j-1)+m(a[i],b[j]) \end{cases} & \text{иначе} \end{cases} \tag{1.3}$$

Формула выводится по тем же соображениям, что и 1.1. Прямое применение рекурсивной функции тоже неэффективно по времени исполнения, так что аналогично методу, описанному в 1.1 добавляется матрица для хранения промежуточных значений.

1.5 Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна. Формулы Левенштейна и Дамерау-Левенштейна для нахождения расстояния между строками задаются рекурсивно и программно могут быть реализованы как рекурсивно, так и итерационно.

2 Конструкторская часть

2.1 Блок-схемы алгоритмов

В данной части будут рассмотрены схемы алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна. На рисунках 2.1 - 2.4 представлены рассматриваемые алгоритмы

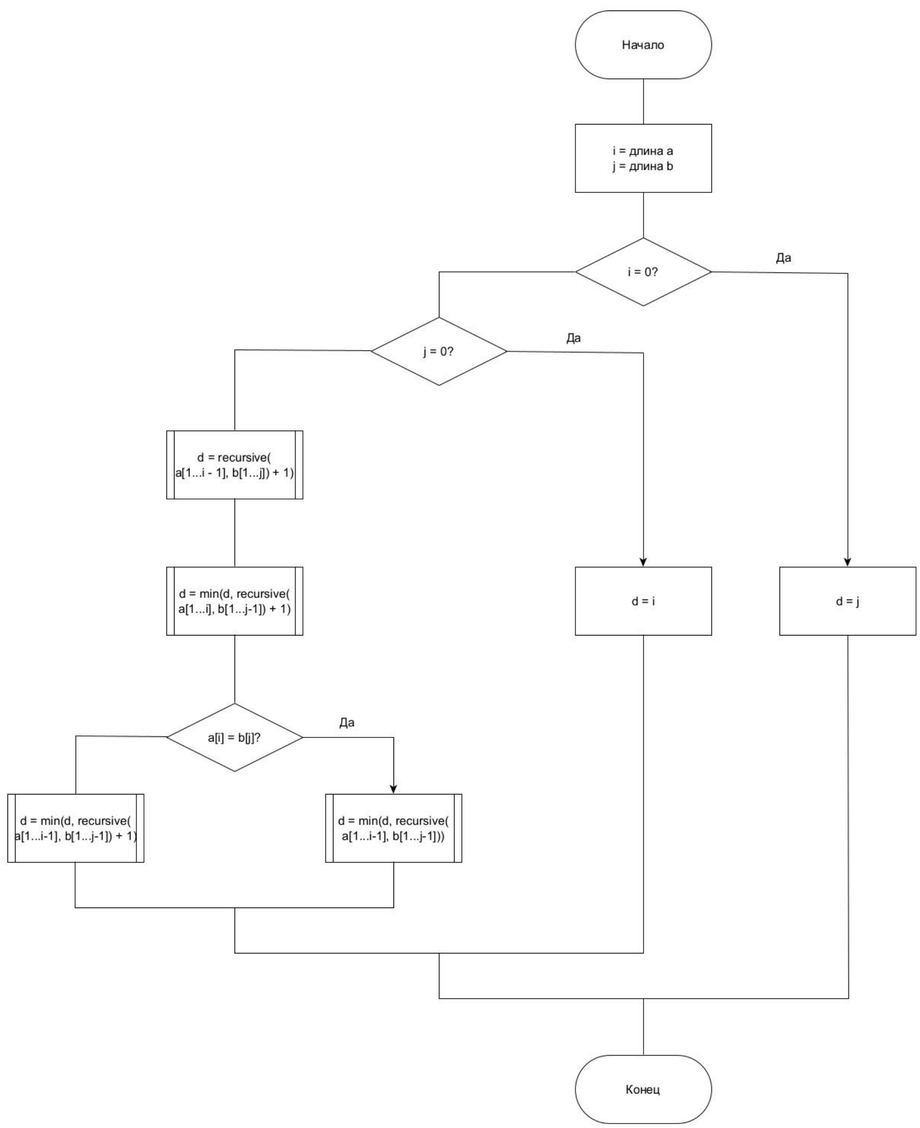

Рис. 2.1: Блок-схема рекурсивного агоритма нахождения расстояния Левенштейна

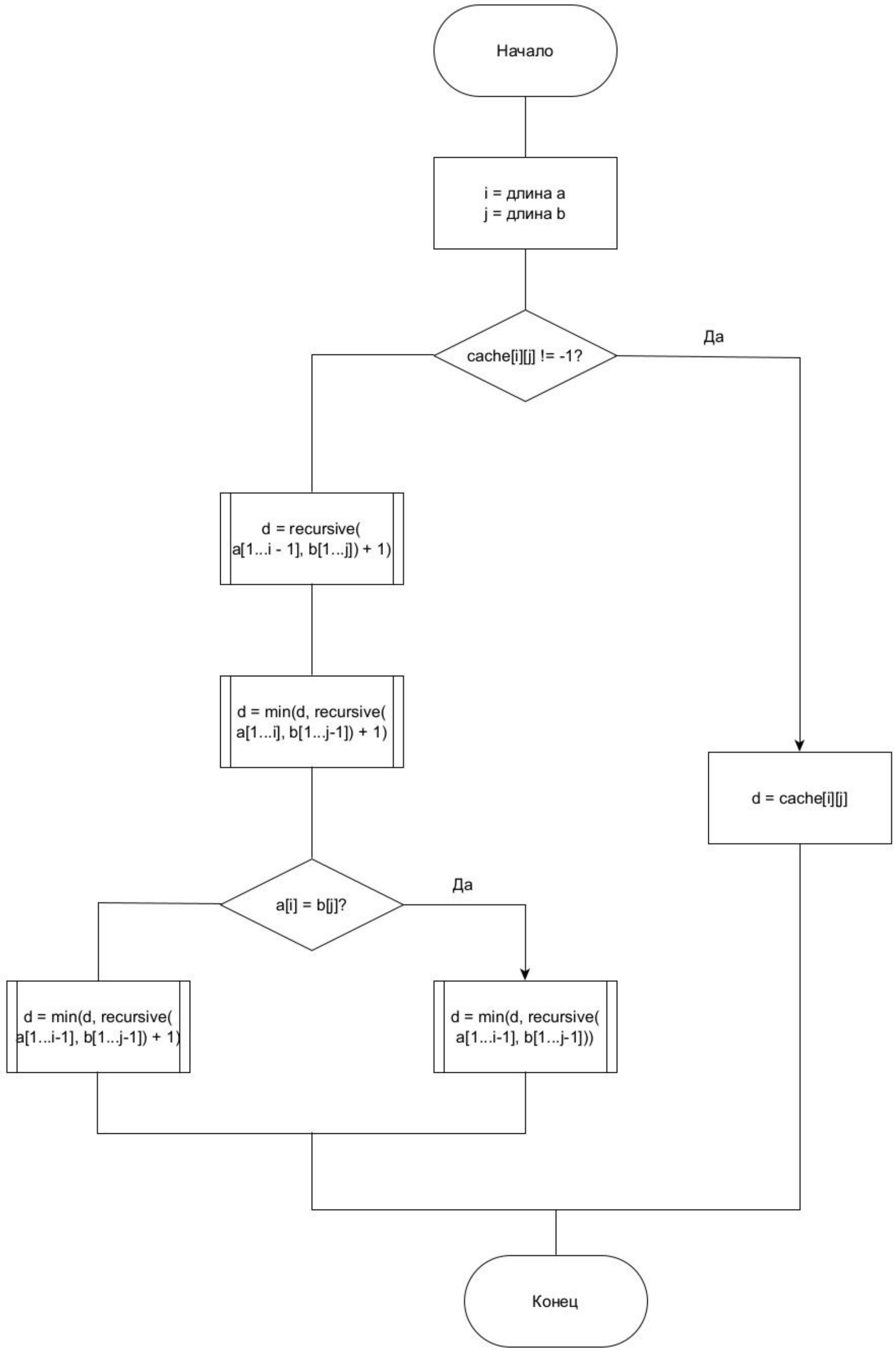

Рис. 2.2: Блок-схема рекурсивного агоритма нахождения расстояния Левенштейна с использованием матрицы

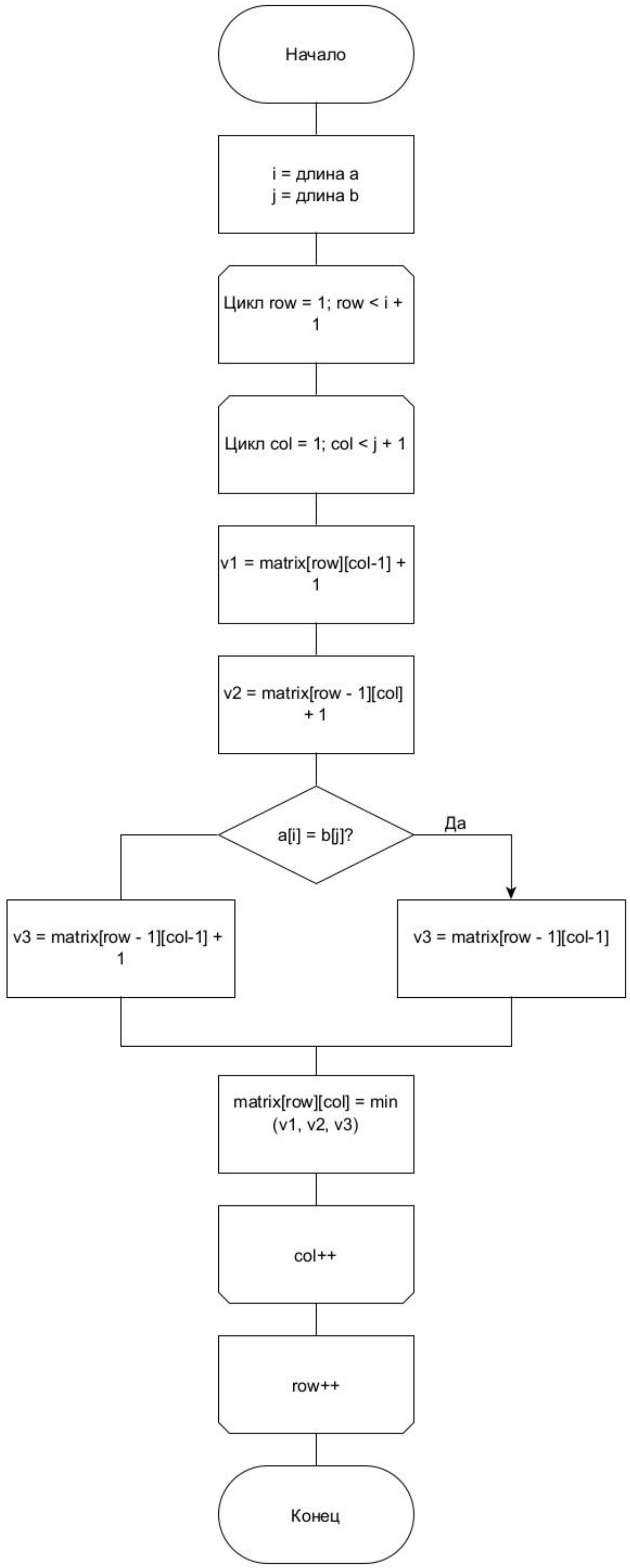

Рис. 2.3: Блок-схема итерационного агоритма нахождения расстояния Левенштейна

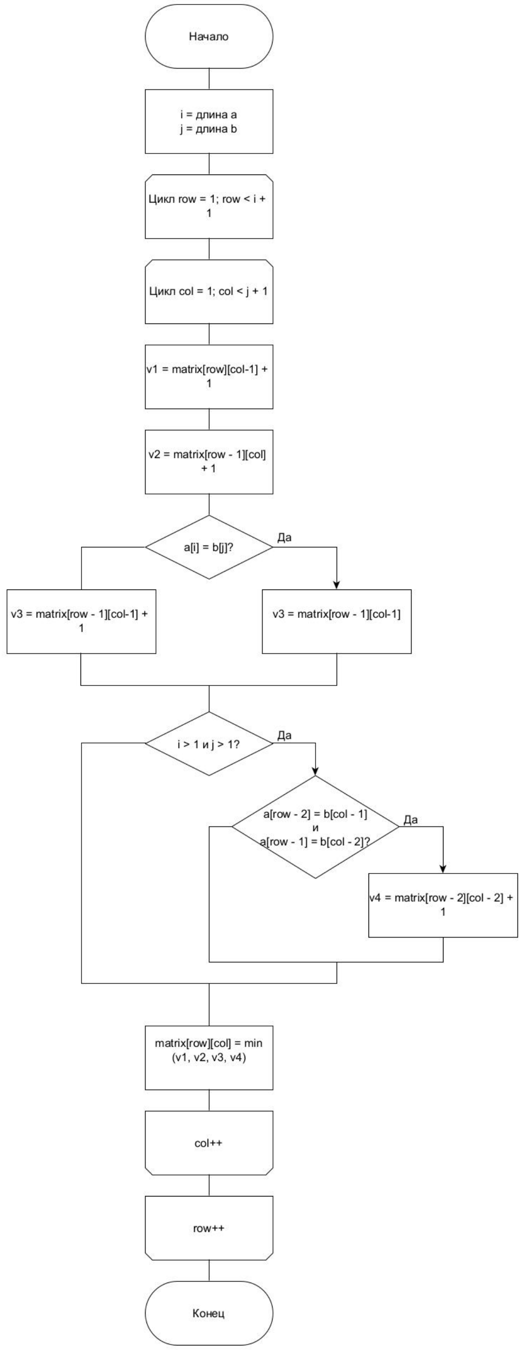

Рис. 2.4: Блок-схема итерационного агоритма нахождения расстояния Дамерау-Левенштейна

2.2 Вывод

На основе теоретических данных, полученных в аналитическом разделе, были построены блок-схемы исследуемых алгоритмов

3 Технологическая часть

3.1 Требование к ПО

1. Подготовить тесты поиска расстояния между строками;
2. ПО должно выводить количество использованного процессорного времени;
3. ПО должно работать корректно;

3.2 Средства реализации

Для реализации программы нахождения расстояния Левенштейна и Дамерау-Левенштейна я выбрал язык программирования Java [?]. Такой выбор обусловлен моим желанием расширить свои знания в области применения данного языка программирования.

3.3 Реализация алгоритмов

В листингах приведена реализация алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Листинг 3.1: Метод для нахождения расстояния Левенштейна рекурсивно

```java
public class LevRecursion extends StringDistanceAlgorithm {
    public LevRecursion(String a, String b) {
        super(a, b);
    }

    public int findDifference() {
        return findDifference(a, a.length(), b, b.length());
    }

    private int findDifference(String a, int aLen, String b, int bLen) {
        if (aLen == 0) {
            return bLen;
        }
        if (bLen == 0) {
            return aLen;
        }

```

```java
        int deletion = findDifference(a, aLen - 1, b, bLen) + 1;
        int insertion = findDifference(a, aLen, b, bLen - 1) + 1;

        int match = Objects.equals(a.charAt(aLen - 1), b.charAt(bLen - 1)) ?
            0 : 1;
        int substitution = findDifference(a, aLen - 1, b, bLen - 1) + match;

        return Math.min(substitution, Math.min(deletion, insertion));
    }
}
```

Листинг 3.2: Метод для нахождения расстояния Левенштейна рекурсивно с матрицей

```java
public class LevRecursionCache extends StringDistanceAlgorithm {
    int[][] cache;

    public LevRecursionCache(String a, String b) {
        super(a, b);
    }

    public int findDifference() {
        cache = new int[a.length() + 1][b.length() + 1];
        for (int[] a : cache) {
            Arrays.fill(a, -1);
        }

        return findDifference(a, a.length(), b, b.length());
    }

    private int findDifference(String a, int aLen, String b, int bLen) {
        if (cache[aLen][bLen] != - 1) {
            return cache[aLen][bLen];
        }
        if (aLen == 0) {
            return bLen;
        }
        if (bLen == 0) {
            return aLen;
        }

        int deletion = findDifference(a, aLen - 1, b, bLen) + 1;
        int insertion = findDifference(a, aLen, b, bLen - 1) + 1;

        int match = Objects.equals(a.charAt(aLen - 1), b.charAt(bLen - 1)) ?
            0 : 1;
        int substitution = findDifference(a, aLen - 1, b, bLen - 1) + match;

        int min = Math.min(substitution, Math.min(deletion, insertion));
        cache[aLen][bLen] = min;
        return min;
```

```
    }
}
```

Листинг 3.3: Метод для нахождения расстояния Левенштейна итерационно

```
public class LevCasualMatrix extends StringDistanceAlgorithm {
    private final int[][] matrix;

    public LevCasualMatrix(String a, String b) {
        super(a, b);
        matrix = new int[a.length() + 1][b.length() + 1];
    }

    public int findDifference() {
        for (int col = 0; col < matrix[0].length; col++) {
            matrix[0][col] = col;
        }

        for (int row = 1; row < matrix.length; row++) {
            matrix[row][0] = row;
        }

        for (int row = 1; row < matrix.length; row++) {
            for (int col = 1; col < matrix[0].length; col++) {
                int valueFromTheRight = matrix[row][col - 1] + 1;
                int valueFromTheUp = matrix[row - 1][col] + 1;

                int match = Objects.equals(a.charAt(row - 1), b.charAt(col -
                    1)) ? 0 : 1;
                int valueFromDiagonal = matrix[row - 1][col - 1] + match;

                matrix[row][col] = Math.min(valueFromTheRight, Math.min(
                   valueFromTheUp, valueFromDiagonal));
            }
        }

        return matrix[a.length()][b.length()];
    }
}
```

Листинг 3.4: Метод для нахождения расстояния Дамерау-Левенштейна рекурсивно

```
public class LevDamRecursion extends StringDistanceAlgorithm {
    public LevDamRecursion(String a, String b) {
        super(a, b);
    }

    public int findDifference() {
        return findDifference(a, a.length(), b, b.length());
    }

```

```
    private int findDifference(String a, int aLen, String b, int bLen) {
        if (aLen == 0) {
            return bLen;
        }
        if (bLen == 0) {
            return aLen;
        }

        int deletion = findDifference(a, aLen - 1, b, bLen) + 1;
        int insertion = findDifference(a, aLen, b, bLen - 1) + 1;

        int match = Objects.equals(a.charAt(aLen - 1), b.charAt(bLen - 1)) ?
            0 : 1;
        int substitution = findDifference(a, aLen - 1, b, bLen - 1) + match;

        int swap = Integer.MAX_VALUE;
        if (aLen > 1 && bLen > 1) {
            if (Objects.equals(a.charAt(aLen - 1), b.charAt(bLen - 2)) &&
                    Objects.equals(a.charAt(aLen - 2), b.charAt(bLen - 1)))
                        {
                swap = findDifference(a, aLen - 2, b, bLen - 2) + 1;
            }
        }

        return Math.min(Math.min(swap, substitution), Math.min(deletion,
           insertion));
    }
}
```

Листинг 3.5: Метод для нахождения расстояния Дамерау-Левенштейна рекурсивно с матрицей

```
public class LevDamRecursionCache extends StringDistanceAlgorithm {
    private int[][] cache;

    public LevDamRecursionCache(String a, String b) {
        super(a, b);
    }

    public int findDifference() {
        cache = new int[a.length() + 1][b.length() + 1];
        for (int[] a : cache) {
            Arrays.fill(a, -1);
        }

        return findDifference(a, a.length(), b, b.length());
    }

    private int findDifference(String a, int aLen, String b, int bLen) {
        if (cache[aLen][bLen] != -1) {
```

```
            return cache[aLen][bLen];
        }
        if (aLen == 0) {
            return bLen;
        }
        if (bLen == 0) {
            return aLen;
        }

        int deletion = findDifference(a, aLen - 1, b, bLen) + 1;
        int insertion = findDifference(a, aLen, b, bLen - 1) + 1;

        int match = Objects.equals(a.charAt(aLen - 1), b.charAt(bLen - 1)) ? 0 : 1;
        int substitution = findDifference(a, aLen - 1, b, bLen - 1) + match;

        int swap = Integer.MAX_VALUE;
        if (aLen > 1 && bLen > 1) {
            if (Objects.equals(a.charAt(aLen - 1), b.charAt(bLen - 2)) &&
                    Objects.equals(a.charAt(aLen - 2), b.charAt(bLen - 1))) {
                swap = findDifference(a, aLen - 2, b, bLen - 2) + 1;
            }
        }

        int min = Math.min(Math.min(swap, substitution), Math.min(deletion, insertion));
        cache[aLen][bLen] = min;
        return min;
    }
}
```

Листинг 3.6: Метод для нахождения расстояния Дамерау-Левенштейна итерационно

```
public class LevDamCasualMatrix extends StringDistanceAlgorithm {
    private int[][] matrix;

    public LevDamCasualMatrix(String a, String b) {
        super(a, b);
        matrix = new int[a.length() + 1][b.length() + 1];
    }

    public int findDifference() {

        for (int col = 0; col < matrix[0].length; col++) {
            matrix[0][col] = col;
        }

        for (int row = 1; row < matrix.length; row++) {
            matrix[row][0] = row;
```

```
        }

        for (int row = 1; row < matrix.length; row++) {
            for (int col = 1; col < matrix[0].length; col++) {
                int valueFromTheRight = matrix[row][col - 1] + 1;
                int valueFromTheUp = matrix[row - 1][col] + 1;

                int match = Objects.equals(a.charAt(row - 1), b.charAt(col - 1)) ? 0 : 1;
                int valueFromDiagonal = matrix[row - 1][col - 1] + match;

                matrix[row][col] = Math.min(valueFromTheRight, Math.min(valueFromTheUp, valueFromDiagonal));

                int swap = Integer.MAX_VALUE;
                if (row > 1 && col > 1) {
                    if (Objects.equals(a.charAt(row - 1), b.charAt(col - 2)) &&
                            Objects.equals(a.charAt(row - 2), b.charAt(col - 1))) {
                        swap = matrix[row - 2][col - 2] + 1;
                    }
                }

                matrix[row][col] = Math.min(Math.min(valueFromTheRight, valueFromTheUp),
                        Math.min(swap, valueFromDiagonal));
            }
        }

        return matrix[a.length()][b.length()];
    }
}
```

3.4 Тестовые данные

В таблице 3.1 приведены тестовые данные, на которых было протестировано разработанное ПО.

Таблица 3.1: Тестовые данные

№	Первое слово	Второе слово	Ожидаемый результат	Полученный результат
1	kot	skat	2, 2	2, 2
2	solution	slouiton	4, 2	4, 2
3	fumo	mofu	4, 4	4, 4
4			0, 0	0,0
5		aoa	3, 3	3, 3
6	aoa		3, 3	3, 3
7	текст	ткскт	2, 2	2, 2

3.5 Вывод

В данном разделе были разработаны исходные коды шести алгоритмов: вычисления расстояния Левенштейна и Дамерау-Левенштейна рекурсивно, рекурсивно с матрицей и итерационно

4 Исследовательская часть

4.1 Пример работы программы

Демонстрация работы программы приведена на рисунке 4.1

```
-----------------
Слово 1 - текст, слово 2 - тксткт
Количество повторов - 1000000, для рекурсивных - 1000
Алгоритм Левенштейна (Итерационный)
    CPU time: 9359 ns
    Ответ: 2
Алгоритм Левенштейна (Рекурсивный)
    CPU time: 9390625 ns
    Ответ: 2
Алгоритм Левенштейна (Рекурсивный с кэшем)
    CPU time: 10015 ns
    Ответ: 2
Алгоритм Дамерау-Левенштейна (Итерационный)
    CPU time: 10484 ns
    Ответ: 2
Алгоритм Дамерау-Левенштейна (Рекурсивный)
    CPU time: 10531250 ns
    Ответ: 2
Алгоритм Дамерау-Левенштейна (Рекурсивный с кэшем)
    CPU time: 11343 ns
    Ответ: 2
```

Рис. 4.1: Демонстрация работы программы

4.2 Технические характеристики

Ниже приведены технические характеристики устройства, на котором было проведено тестирование ПО:

- Операционная система — Windows [?] 10 64-bit
- Оперативная память — 4 GB (для Java)

- Процессор — Intel(R) Core(TM) i5-7600 CPU @ 3.50GHz [?]

4.3 Время выполнения алгоритма

Время выполнения алгоритмов замерялось с помощью Java-интерфейса для управления потоками виртуальной машины Java - ThreadMXBean [?], позволяющего вычислить процессорное время, затраченное на определенный процесс.

В таблице 4.1 представлены замеры времени работы для каждого из алгоритмов.

Таблица 4.1: Таблица времени выполнения алгоритмов (в наносекундах)

Длина строк	LevIter	LevRec	LevRecCache	LevDamIter	LevDamRec	LevDamRecCache
10	750	41718750	2500	3570	112187500	6125
20	3156	—	8406	14906	—	22734
50	17500	—	45264	89218	—	139843
100	67656	—	193593	356562	—	546875

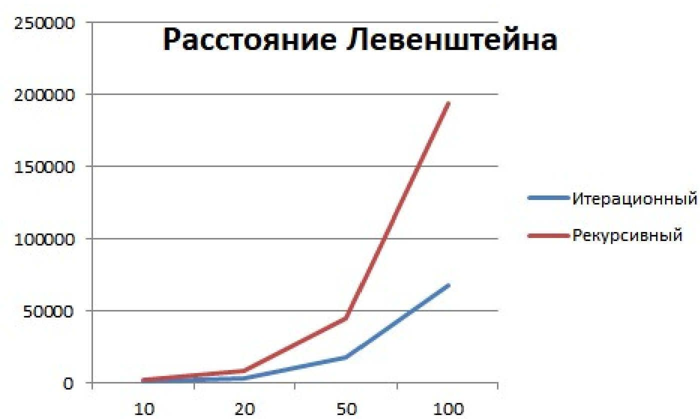

Рис. 4.2: График времени выполнения алгоритмов поиска расстояния Левенштейна

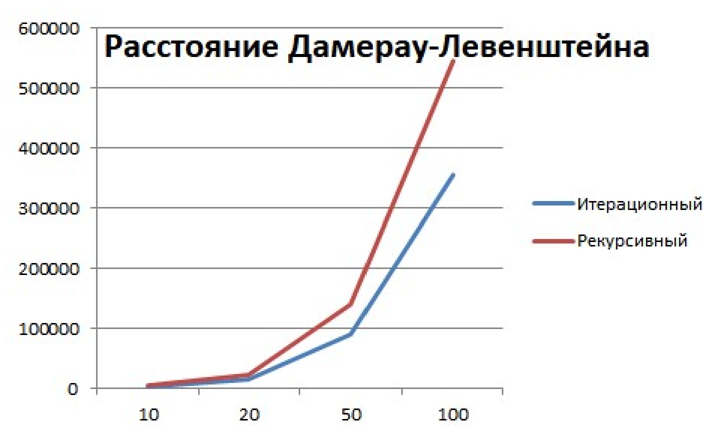

Рис. 4.3: График времени выполнения алгоритмов поиска расстояния Дамерау-Левенштейна

4.4 Использование памяти

Алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна практически не отличаются друг от друга с точки зрения используемой памяти.

Максимальная глубина стека вызовов при рекурсивной реализации — сумма длин входных строк. Тогда максимальный объем требуемой памяти равен:

$$(len(Str_1) + len(Str_2)) \cdot (2 \cdot K(String) + 7 \cdot K(int)), \tag{4.1}$$

len — оператор вычисления длины строки, Str_1 и Str_2 — входные строки, где K — оператор вычисления размера, $String$ — строковый тип, int — целочисленный тип.

Объем используемой памяти при итерационной реализации равен (при оптимизации матрицы до 2 строк):

$$2 \cdot (len(Str_2) + 1) \cdot K(int) + 4 \cdot K(int) \tag{4.2}$$

4.5 Вывод

Рекурсивные реализации алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна работают дольше итерационных реализаций. Время работы реализаций увеличивается в геометрической прогрессии. Рекурсивный алгоритм с матрицей работает немногим медленнее итерационного.

Заключение

В ходе проделанной работы был изучен метод динамического программирования на материале реализации алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна. Были изучены алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна и получены навыки реализации указанных алгоритмов в матричной и рекурсивных версиях, в том числе и с кэшированием результатов.

Экспериментально подтверждено различие во временной эффективности рекурсивной и итерационной реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна при помощи разработанного программного обеспечения с замерами процессорного времени. Рекурсивные реализации проигрывают по времени в несколько раз.

Теоретически было рассчитано использование памяти в каждой из реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна. Рекурсивный алгоритм с матрицей (кэшированием) использует больше всего памяти.