



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

<b>Введение</b>	<b>2</b>
<b>1 Аналитическая часть</b>	<b>3</b>
1.1 Рекурсивный алгоритм нахождения расстояния Левенштейна . . . . .	3
1.2 Итерационный алгоритм нахождения расстояния Левенштейна с использова- нием матрицы . . . . .	4
1.3 Рекурсивный алгоритм нахождения расстояния Левенштейна с использованием матрицы . . . . .	5
1.4 Расстояние Дамерау-Левенштейна . . . . .	5
1.5 Вывод . . . . .	5
<b>2 Конструкторская часть</b>	<b>6</b>
2.1 Разработка алгоритмов . . . . .	6
2.2 Вывод . . . . .	12
<b>3 Технологическая часть</b>	<b>13</b>
3.1 Требование к ПО . . . . .	13
3.2 Средства реализации . . . . .	13
3.3 Реализация алгоритмов . . . . .	13
3.4 Тестовые данные . . . . .	18
3.5 Вывод . . . . .	19
<b>4 Исследовательская часть</b>	<b>20</b>
4.1 Пример работы программы . . . . .	20
4.2 Технические характеристики . . . . .	20
4.3 Время выполнения алгоритма . . . . .	21
4.4 Использование памяти . . . . .	22
4.5 Вывод . . . . .	22
<b>Заключение</b>	<b>23</b>
<b>Литература</b>	<b>24</b>

# 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 Дamerau-Левенштейна.
2. Оценить реализации алгоритмов нахождения расстояния Левенштейна и Дamerau-Левенштейна.

Ниже перечислены задачи лабораторной работы.

1. Изучить принципы работы алгоритмов Левенштейна и Дamerau-Левенштейна.
2. Подготовить тесты поиска расстояния между строками.
3. Применить методы динамического программирования для матричной реализации указанных алгоритмов.
4. Получить практические навыки реализации данных алгоритмов: итерационные и рекурсивные версии.
5. Сравнить итерационные и рекурсивные реализации алгоритмов по затрачиваемым ресурсам (время и память).
6. Получить экспериментальным методом различия во временной эффективности алгоритмов.
7. 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 расстояния Левенштейна между двумя строками  $a$  и  $b$  лежит формула 1.1. При описании алгоритмов и формул будут использоваться указанные ниже обозначения.

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

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

Функция 1.2 позволяет сравнить два символа:

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

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 соответственно, 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:

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

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 Разработка алгоритмов

В данной части будут рассмотрены схемы алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна. На рисунках 2.1 - 2.6 представлены рассматриваемые алгоритмы

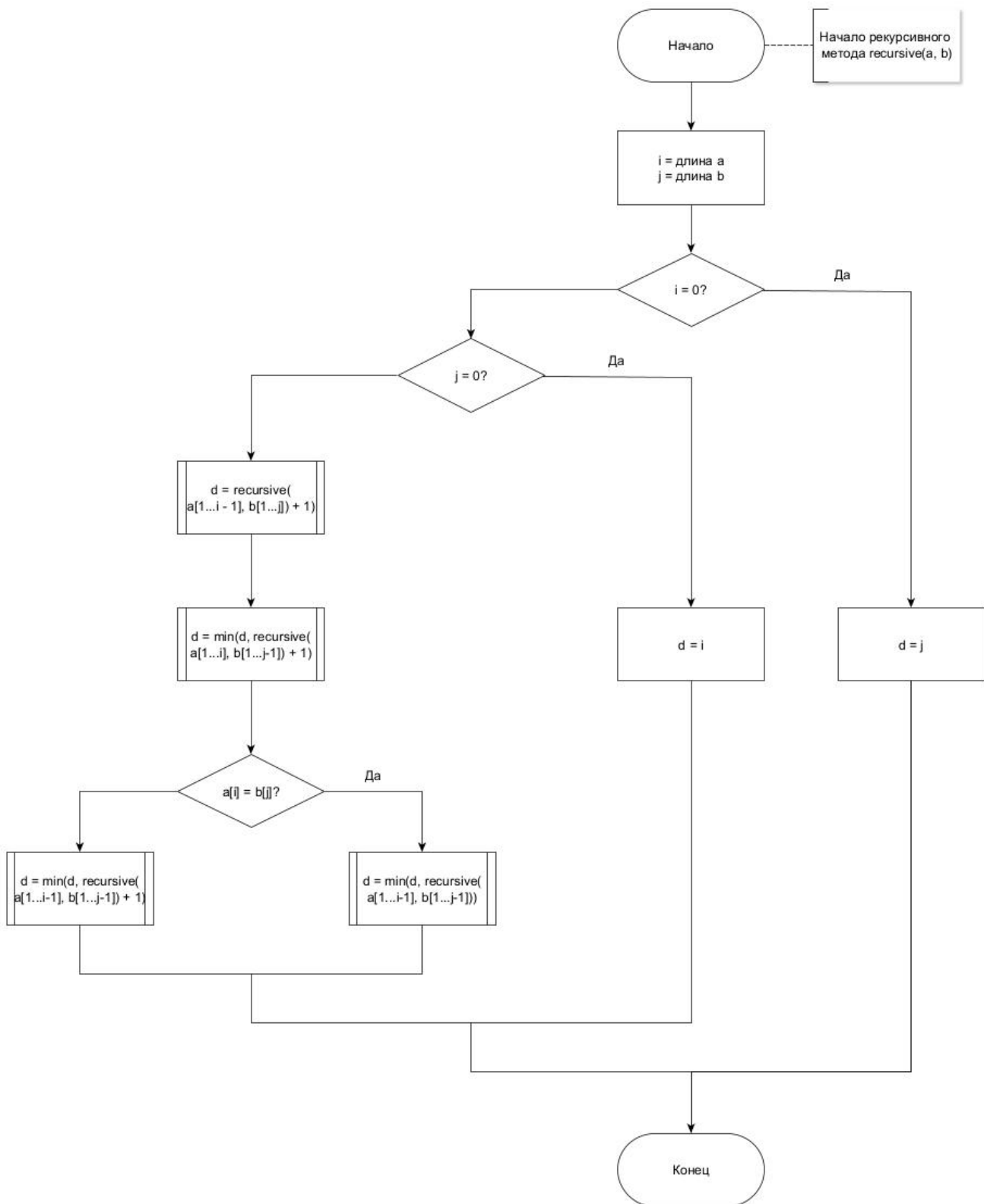


Рис. 2.1: Блок-схема рекурсивного алгоритма нахождения расстояния Левенштейна



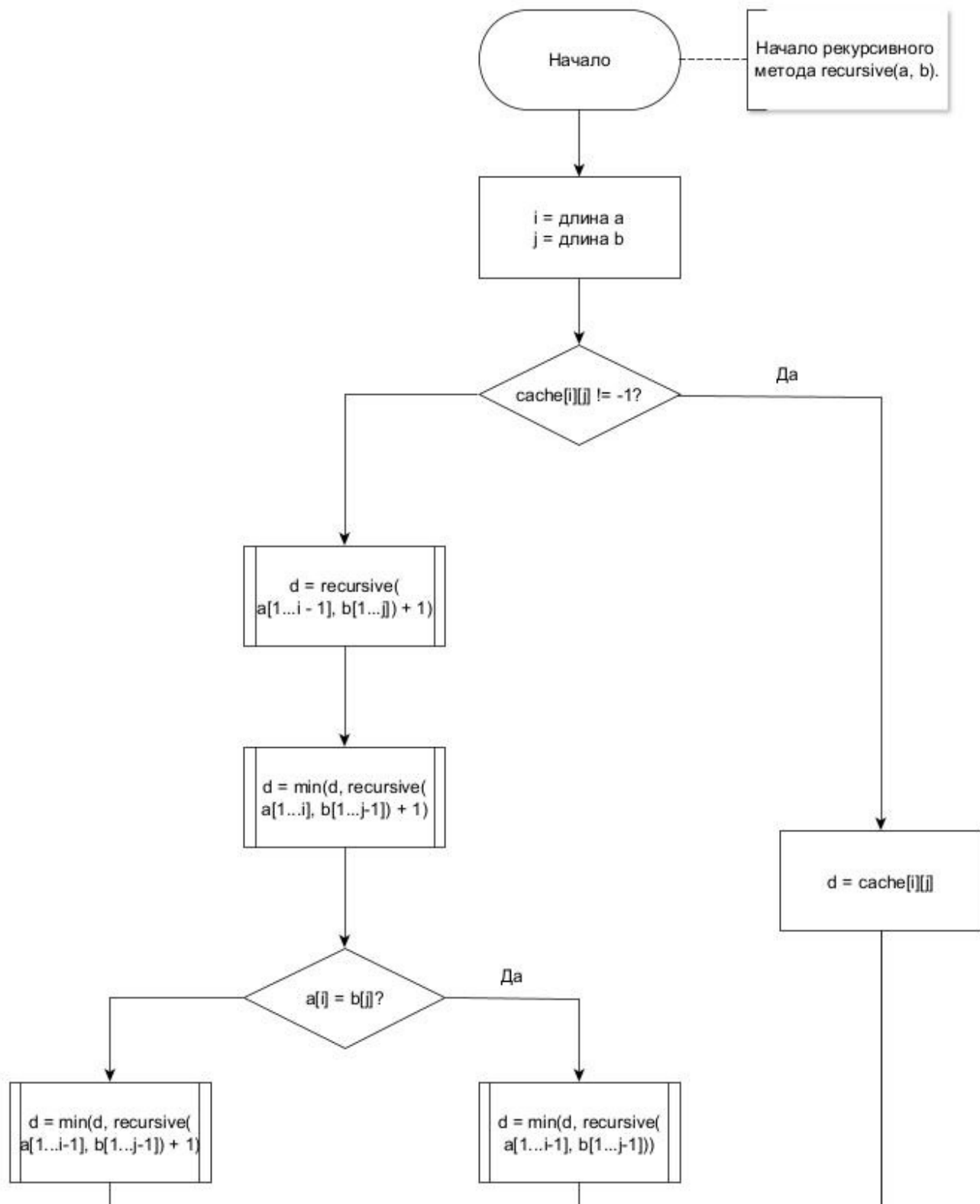


Рис. 2.2:

Блок-схема рекурсивного алгоритма нахождения расстояния Левенштейна с использованием матрицы ч.1

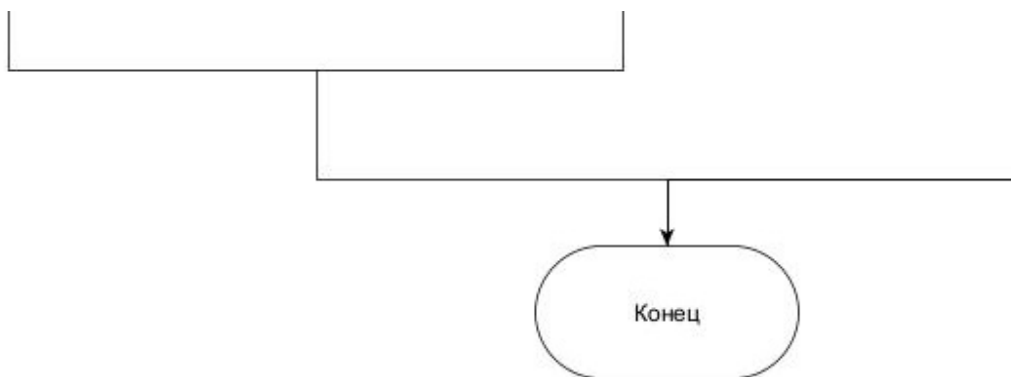


Рис. 2.3:

Блок-схема рекурсивного алгоритма нахождения расстояния Левенштейна с использованием матрицы ч.2

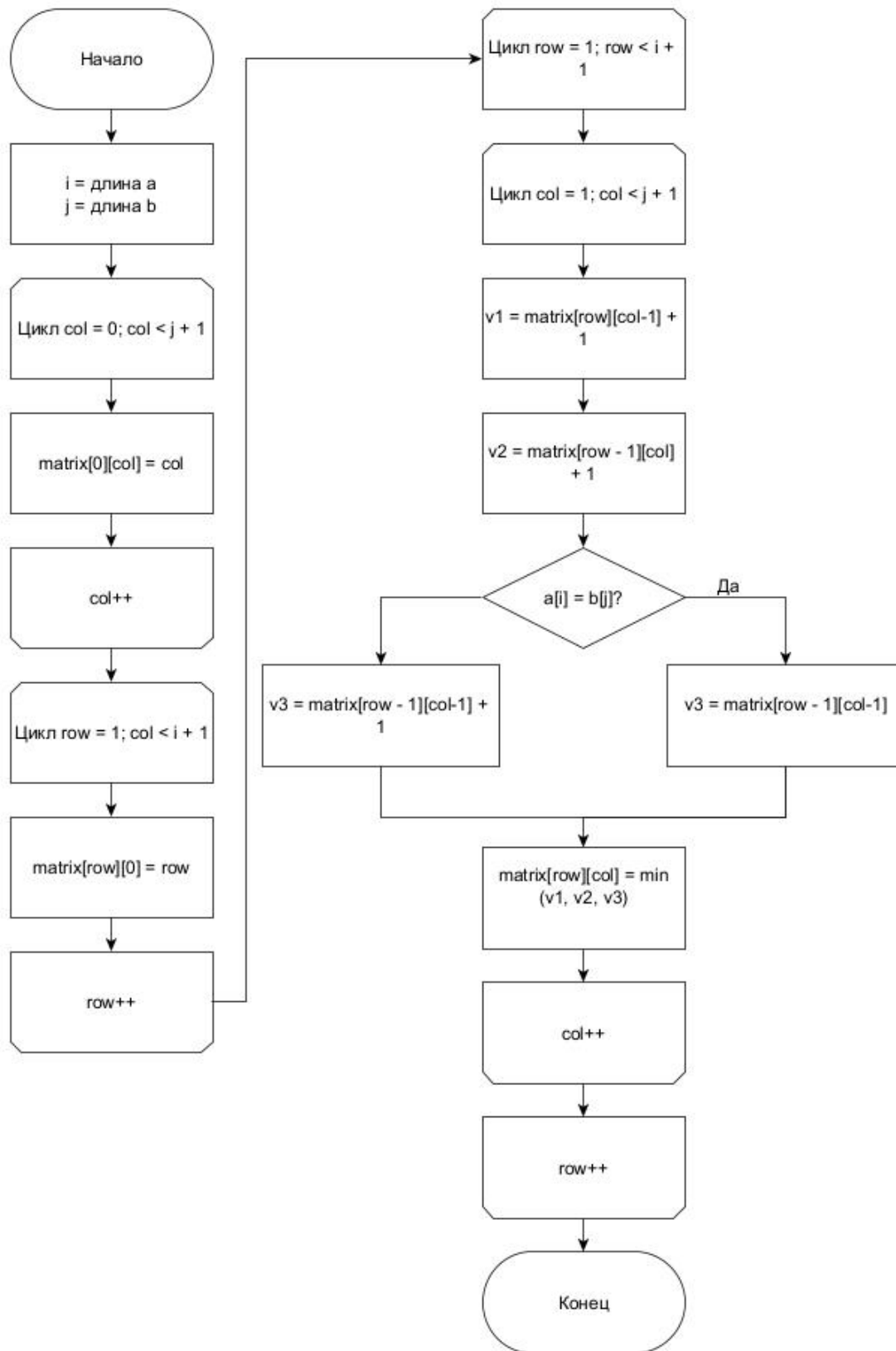


Рис. 2.4: Блок-схема итерационного алгоритма нахождения расстояния Левенштейна

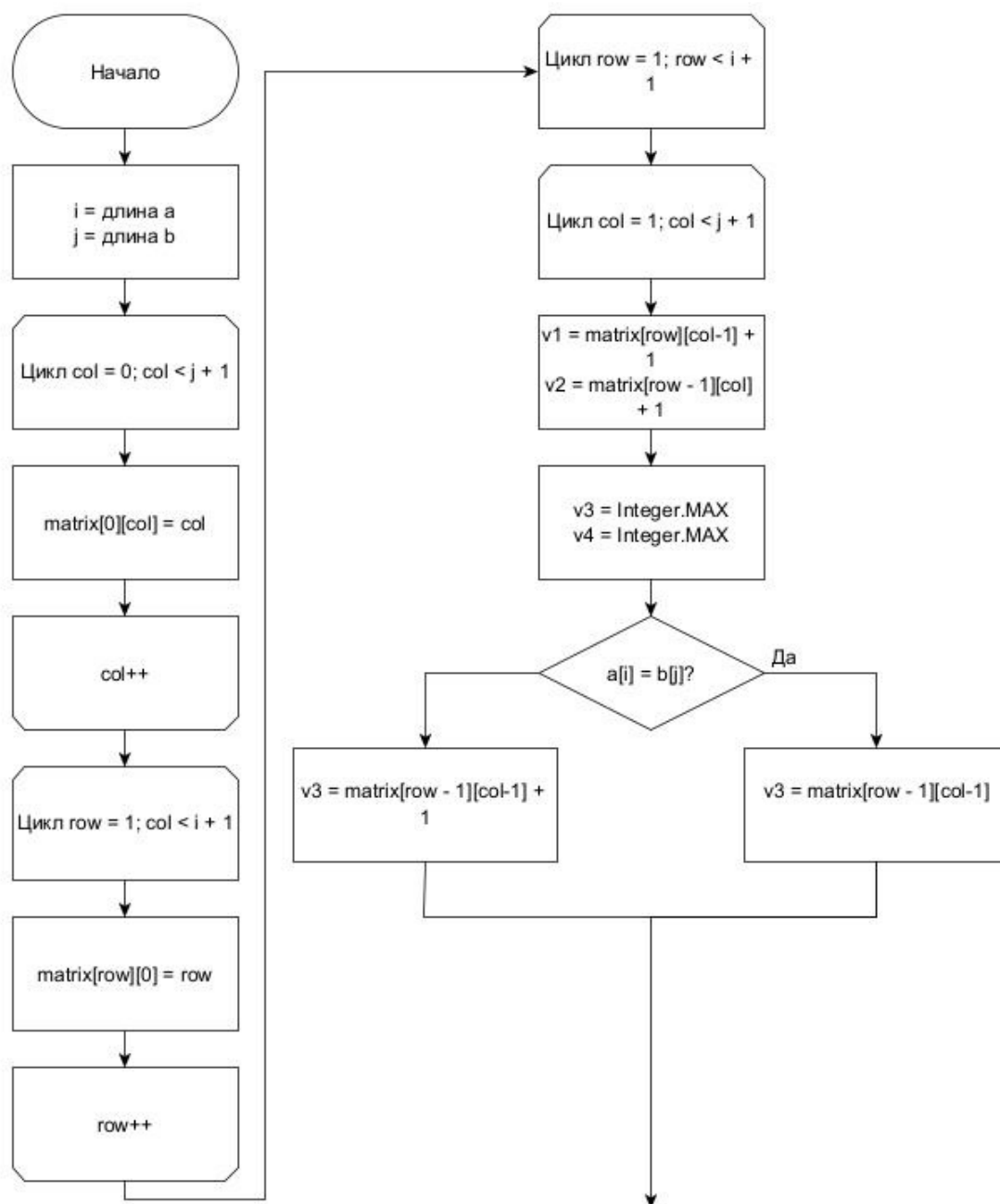


Рис. 2.5:

Блок-схема итерационного алгоритма нахождения расстояния Дameraу-Левенштейна ч.1

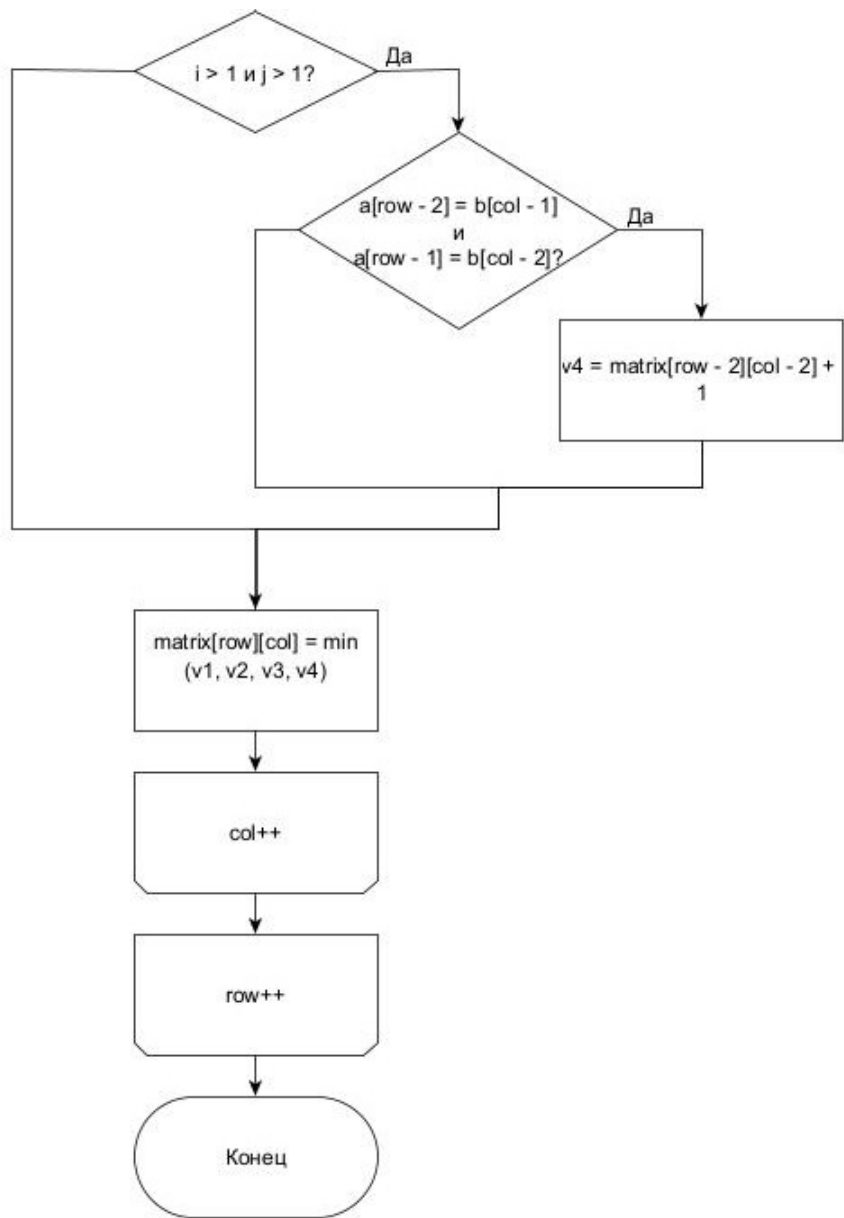


Рис. 2.6:

Блок-схема итерационного алгоритма нахождения расстояния Дamerau-Левенштейна ч.2

## 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

Ниже перечислены основные требования к разрабатываемому ПО.

1. Предусмотреть возможность строк для сравнения и вывода результата сравнения;
2. ПО должно выводить количество использованного процессорного времени;
3. ПО должно работать корректно.

### 3.2 Средства реализации

Для реализации программы нахождения расстояния Левенштейна и Дamerau-Левенштейна был выбран язык программирования Java [1]. Такой выбор обусловлен желанием расширить знания в области применения данного языка программирования.

### 3.3 Реализация алгоритмов

В листингах 3.1 - 3.6 приведена реализация алгоритмов нахождения расстояния Левенштейна и Дamerau-Левенштейна.

Листинг 3.1: Метод для нахождения расстояния Левенштейна рекурсивно

```
1 public class LevRecursion extends StringDistanceAlgorithm {
2     public LevRecursion(String a, String b) {
3         super(a, b);
4     }
5
6     public int findDifference() {
7         return findDifference(a, a.length(), b, b.length());
8     }
9
10    private int findDifference(String a, int aLen, String b, int bLen) {
11        if (aLen == 0) {
12            return bLen;
13        }
14        if (bLen == 0) {
15            return aLen;
16        }
```

```

17
18     int deletion = findDifference(a, aLen - 1, b, bLen) + 1;
19     int insertion = findDifference(a, aLen, b, bLen - 1) + 1;
20
21     int match = Objects.equals(a.charAt(aLen - 1), b.charAt(bLen - 1)) ?
22         0 : 1;
23     int substitution = findDifference(a, aLen - 1, b, bLen - 1) + match;
24
25     return Math.min(substitution, Math.min(deletion, insertion));
26 }

```

Листинг 3.2: Метод для нахождения расстояния Левенштейна рекурсивно с матрицей

```

1 public class LevRecursionCache extends StringDistanceAlgorithm {
2     int [][] cache;
3
4     public LevRecursionCache(String a, String b) {
5         super(a, b);
6     }
7
8     public int findDifference() {
9         cache = new int[a.length() + 1][b.length() + 1];
10        for (int[] a : cache) {
11            Arrays.fill(a, -1);
12        }
13
14        return findDifference(a, a.length(), b, b.length());
15    }
16
17    private int findDifference(String a, int aLen, String b, int bLen) {
18        if (cache[aLen][bLen] != -1) {
19            return cache[aLen][bLen];
20        }
21        if (aLen == 0) {
22            return bLen;
23        }
24        if (bLen == 0) {
25            return aLen;
26        }
27
28        int deletion = findDifference(a, aLen - 1, b, bLen) + 1;
29        int insertion = findDifference(a, aLen, b, bLen - 1) + 1;
30
31        int match = Objects.equals(a.charAt(aLen - 1), b.charAt(bLen - 1)) ?
32            0 : 1;
33        int substitution = findDifference(a, aLen - 1, b, bLen - 1) + match;
34
35        int min = Math.min(substitution, Math.min(deletion, insertion));
36        cache[aLen][bLen] = min;

```

```

36     return min;
37 }
38 }

```

Листинг 3.3: Метод для нахождения расстояния Левенштейна итерационно

```

1  public class LevCasualMatrix extends StringDistanceAlgorithm {
2      private final int [][] matrix;
3
4      public LevCasualMatrix(String a, String b) {
5          super(a, b);
6          matrix = new int[a.length() + 1][b.length() + 1];
7      }
8
9      public int findDifference() {
10         for (int col = 0; col < matrix[0].length; col++) {
11             matrix[0][col] = col;
12         }
13
14         for (int row = 1; row < matrix.length; row++) {
15             matrix[row][0] = row;
16         }
17
18         for (int row = 1; row < matrix.length; row++) {
19             for (int col = 1; col < matrix[0].length; col++) {
20                 int valueFromTheRight = matrix[row][col - 1] + 1;
21                 int valueFromTheUp = matrix[row - 1][col] + 1;
22
23                 int match = Objects.equals(a.charAt(row - 1), b.charAt(col -
24                     1)) ? 0 : 1;
25                 int valueFromDiagonal = matrix[row - 1][col - 1] + match;
26
27                 matrix[row][col] = Math.min(valueFromTheRight, Math.min(
28                     valueFromTheUp, valueFromDiagonal));
29             }
30         }
31
32         return matrix[a.length()][b.length()];
33     }
34 }

```

Листинг 3.4: Метод для нахождения расстояния Дамерау-Левенштейна рекурсивно

```

1  public class LevDamRecursion extends StringDistanceAlgorithm {
2      public LevDamRecursion(String a, String b) {
3          super(a, b);
4      }
5
6      public int findDifference() {
7          return findDifference(a, a.length(), b, b.length());
8      }
9  }

```



```

9
10 private int findDifference(String a, int aLen, String b, int bLen) {
11     if (aLen == 0) {
12         return bLen;
13     }
14     if (bLen == 0) {
15         return aLen;
16     }
17
18     int deletion = findDifference(a, aLen - 1, b, bLen) + 1;
19     int insertion = findDifference(a, aLen, b, bLen - 1) + 1;
20
21     int match = Objects.equals(a.charAt(aLen - 1), b.charAt(bLen - 1)) ?
22         0 : 1;
23     int substitution = findDifference(a, aLen - 1, b, bLen - 1) + match;
24
25     int swap = Integer.MAX_VALUE;
26     if (aLen > 1 && bLen > 1) {
27         if (Objects.equals(a.charAt(aLen - 1), b.charAt(bLen - 2)) &&
28             Objects.equals(a.charAt(aLen - 2), b.charAt(bLen - 1)))
29             {
30                 swap = findDifference(a, aLen - 2, b, bLen - 2) + 1;
31             }
32     }
33     return Math.min(Math.min(swap, substitution), Math.min(deletion,
34         insertion));
35 }

```

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

```

1 public class LevDamRecursionCache extends StringDistanceAlgorithm {
2     private int [][] cache;
3
4     public LevDamRecursionCache(String a, String b) {
5         super(a, b);
6     }
7
8     public int findDifference() {
9         cache = new int [a.length() + 1][b.length() + 1];
10        for (int [] a : cache) {
11            Arrays.fill(a, -1);
12        }
13
14        return findDifference(a, a.length(), b, b.length());
15    }
16
17    private int findDifference(String a, int aLen, String b, int bLen) {

```

```

18     if (cache[aLen][bLen] != -1) {
19         return cache[aLen][bLen];
20     }
21     if (aLen == 0) {
22         return bLen;
23     }
24     if (bLen == 0) {
25         return aLen;
26     }
27
28     int deletion = findDifference(a, aLen - 1, b, bLen) + 1;
29     int insertion = findDifference(a, aLen, b, bLen - 1) + 1;
30
31     int match = Objects.equals(a.charAt(aLen - 1), b.charAt(bLen - 1)) ?
32         0 : 1;
33     int substitution = findDifference(a, aLen - 1, b, bLen - 1) + match;
34
35     int swap = Integer.MAX_VALUE;
36     if (aLen > 1 && bLen > 1) {
37         if (Objects.equals(a.charAt(aLen - 1), b.charAt(bLen - 2)) &&
38             Objects.equals(a.charAt(aLen - 2), b.charAt(bLen - 1)))
39             {
40                 swap = findDifference(a, aLen - 2, b, bLen - 2) + 1;
41             }
42     }
43
44     int min = Math.min(Math.min(swap, substitution), Math.min(deletion,
45         insertion));
46     cache[aLen][bLen] = min;
47     return min;
48 }

```

Листинг 3.6: Метод для нахождения расстояния Дамерау-Левенштейна итерационно

```

1 public class LevDamCasualMatrix extends StringDistanceAlgorithm {
2     private int [][] matrix;
3
4     public LevDamCasualMatrix(String a, String b) {
5         super(a, b);
6         matrix = new int[a.length() + 1][b.length() + 1];
7     }
8
9     public int findDifference() {
10
11         for (int col = 0; col < matrix[0].length; col++) {
12             matrix[0][col] = col;
13         }
14
15         for (int row = 1; row < matrix.length; row++) {

```

```

16         matrix[row][0] = row;
17     }
18
19     for (int row = 1; row < matrix.length; row++) {
20         for (int col = 1; col < matrix[0].length; col++) {
21             int valueFromTheRight = matrix[row][col - 1] + 1;
22             int valueFromTheUp = matrix[row - 1][col] + 1;
23
24             int match = Objects.equals(a.charAt(row - 1), b.charAt(col -
25                 1)) ? 0 : 1;
26             int valueFromDiagonal = matrix[row - 1][col - 1] + match;
27
28             matrix[row][col] = Math.min(valueFromTheRight, Math.min(
29                 valueFromTheUp, valueFromDiagonal));
30
31             int swap = Integer.MAX_VALUE;
32             if (row > 1 && col > 1) {
33                 if (Objects.equals(a.charAt(row - 1), b.charAt(col - 2))
34                     &&
35                     Objects.equals(a.charAt(row - 2), b.charAt(col -
36                         1))) {
37                     swap = matrix[row - 2][col - 2] + 1;
38                 }
39             }
40
41             matrix[row][col] = Math.min(Math.min(valueFromTheRight,
42                 valueFromTheUp),
43                 Math.min(swap, valueFromDiagonal));
44         }
45     }
46
47     return matrix[a.length()][b.length()];
48 }

```

### 3.4 Тестовые данные

В таблице 3.1 приведены тестовые данные, на которых было протестировано разработанное ПО. Тестирование проводится по методу черного ящика.

Все тесты были успешно пройдены

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	3, 2	3, 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 - ткскт  
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 были проведены замеры времени.

- операционная система — Windows [2] 10 64-bit;
- оперативная память — 4 GB (для Java);

- процессор — Intel(R) Core(TM) i5-7600 CPU @ 3.50GHz [3].

### 4.3 Время выполнения алгоритма

Время выполнения алгоритмов замерялось с помощью Java-интерфейса для управления потоками виртуальной машины Java - ThreadMXBean [4], позволяющего вычислить процессорное время, затраченное на определенный процесс. Процессорное время находилось как среднее арифметическое из нескольких разовых замеров процессорного времени (1000000 замеров).

В таблице 4.1 представлены замеры времени работы для каждого из алгоритмов. Сокращенные названия алгоритмов приведены ниже.

- Л-И — Расстояние Левенштейна, итерационный алгоритм;
- Л-Р — Расстояние Левенштейна, рекурсивный алгоритм;
- Л-Р-М — Расстояние Левенштейна, рекурсивный алгоритм с использованием матрицы;
- ЛД-И — Расстояние Дамерау-Левенштейна, итерационный алгоритм;
- ЛД-Р — Расстояние Дамерау-Левенштейна, рекурсивный алгоритм;
- ЛД-Р-М — Расстояние Дамерау-Левенштейна, рекурсивный алгоритм с использованием матрицы.

Таблица 4.1: Таблица времени выполнения алгоритмов (в наносекундах)

Длина строк	Л-И	Л-Р	Л-Р-М	ЛД-И	ЛД-Р	ЛД-Р-М
10	750	41718750	2500	3570	112187500	6125
20	3156	—	8406	14906	—	22734
50	17500	—	45264	89218	—	139843
100	67656	—	193593	356562	—	546875

На графиках 4.2 и 4.3 отображено среднее время (в наносекундах), требуемое для нахождения расстояний Левенштейна и Дамерау-Левенштейна соответственно для строк длиной от 10 до 100 символов.

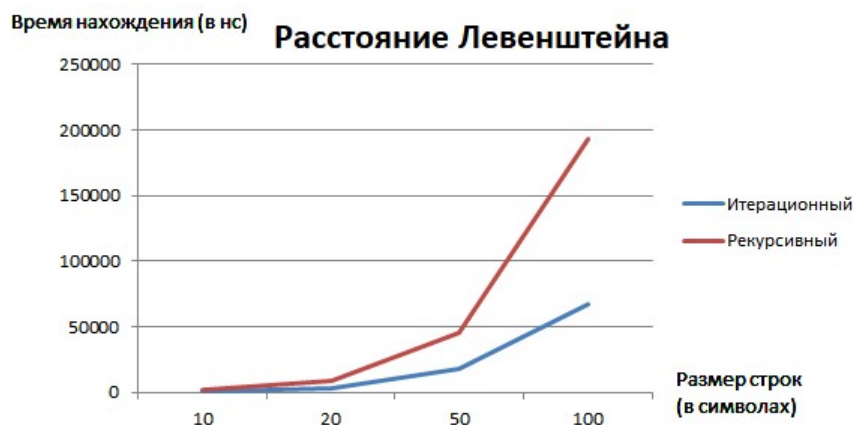


Рис. 4.2: График времени выполнения алгоритмов поиска расстояния Левенштейна

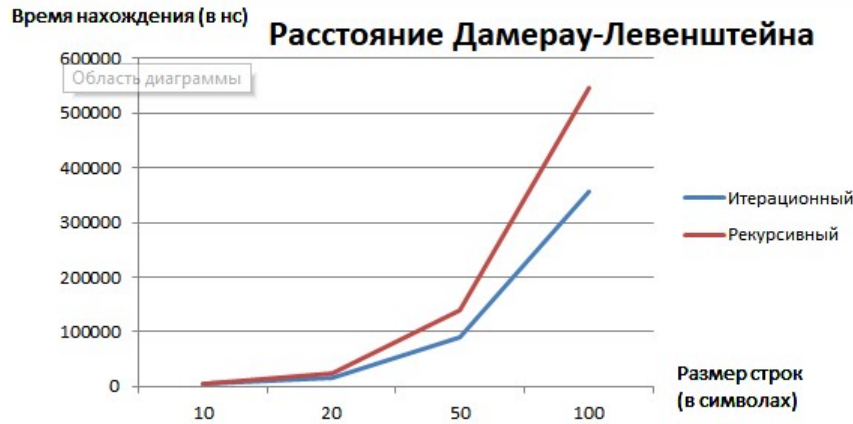


Рис. 4.3: График времени выполнения алгоритмов поиска расстояния Дamerau-Левенштейна

## 4.4 Использование памяти

Алгоритмы нахождения расстояний Левенштейна и Дamerau-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:

$$(\text{len}(\text{Str}_1) + \text{len}(\text{Str}_2)) \cdot (2 \cdot K(\text{String}) + 7 \cdot K(\text{int})), \quad (4.1)$$

$\text{len}$  — оператор вычисления длины строки,  $\text{Str}_1$  и  $\text{Str}_2$  — входные строки, где  $K$  — оператор вычисления размера,  $\text{String}$  — строковый тип,  $\text{int}$  — целочисленный тип.

Объем используемой памяти при итерационной реализации равен:

$$(\text{len}(\text{Str}_1) + 1) \cdot (\text{len}(\text{Str}_2) + 1) \cdot K(\text{int}) + 4 \cdot K(\text{int}) \quad (4.2)$$

## 4.5 Вывод

Рекурсивные реализации алгоритмов нахождения расстояний Левенштейна и Дamerau-Левенштейна без использования матрицы работают дольше их итерационных реализаций. Время работы алгоритмов увеличивается в геометрической прогрессии. Рекурсивный алгоритм с использованием матрицы работает лишь немногим медленнее итерационного.

# Заключение

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

Экспериментально подтверждено различие во временной эффективности рекурсивной и итерационной реализаций алгоритмов поиска расстояний Левенштейна и Дamerau-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 Дamerau-Левенштейна. Рекурсивный алгоритм с использованием матрицы использует больше всего памяти.



# Литература

- [1] Что такое технология Java и каково ее применение? [Электронный ресурс]. Режим доступа: [https://www.java.com/ru/download/help/whatis\\_java.html](https://www.java.com/ru/download/help/whatis_java.html). : 21.09.2021.
- [2] Сравните выпуски Windows 10 [Электронный ресурс]. Режим доступа: <https://www.microsoft.com/ru-ru/windows/compare-windows-10-home-vs-pro>. Дата обращения: 21.09.2021.
- [3] Процессор Intel® Core™ i5-3550 [Электронный ресурс]. Режим доступа: <https://ark.intel.com/content/www/ru/ru/ark/products/97150/intel-core-i57600-processor-6m-cache-up-to-4-10-ghz.html>. Дата обращения: 21.09.2021.
- [4] Interface ThreadMXBean [Электронный ресурс]. Режим доступа: <https://docs.oracle.com/javase/7/docs/api/java/lang/management/ThreadMXBean.html>. Дата обращения: 21.09.2021.