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 Тагилов А.М.

Группа ИУ7-55Б

Оценка (баллы) _____

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

Москва — 2021 г.

Содержание

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.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для:

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

Задача поиска редакционного расстояния в современном мире является наиболее актуальной, так как зачастую необходимо предоставить пользователю возможность делать ошибки в интернет-запросах [1], указаниях инструкций к помощниками и т.д. Данный подход повышает уровень взаимодействия с пользователем и, соответственно, влияет на качество итогового программного продукта, а, чтобы алгоритм работал быстро и качественно, необходим правильный его выбор. Целью данной работы является изучить алгоритмы поиска редакционного расстояния и реализовать некоторые из возможных алгоритмов с использованием методов динамического программирования [2].

Для достижения поставленной цели необходимо выполнить следующие задачи:

- рассмотреть существующие алгоритмы поиска редакционного расстояния;
- провести сравнение и выявить достоинства и недостатки рассмотренных алгоритмов;
- привести схемы выбранных алгоритмов;
- определить средства для реализации алгоритмов;
- подготовить классы данных и ожидаемый результат для эксперимента и описать эксперимент;
- оценить алгоритмы по времени и памяти.

1 Аналитическая часть

Расстояние Левенштейна [3] 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.
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 заданных 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, где:

- $|a|$ означает длину строки a;
- $a[i]$ - i-ый символ строки a.

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

Функция 1.2 позволяет сравнить два символа:

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

Рекурсивный алгоритм реализует формулу 1.1. Логика функции D состоит в следующем.

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:

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

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

В данном разделе выбранные алгоритмы будут формализованы в виде блок-схем.

2.1 Схемы алгоритмов

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

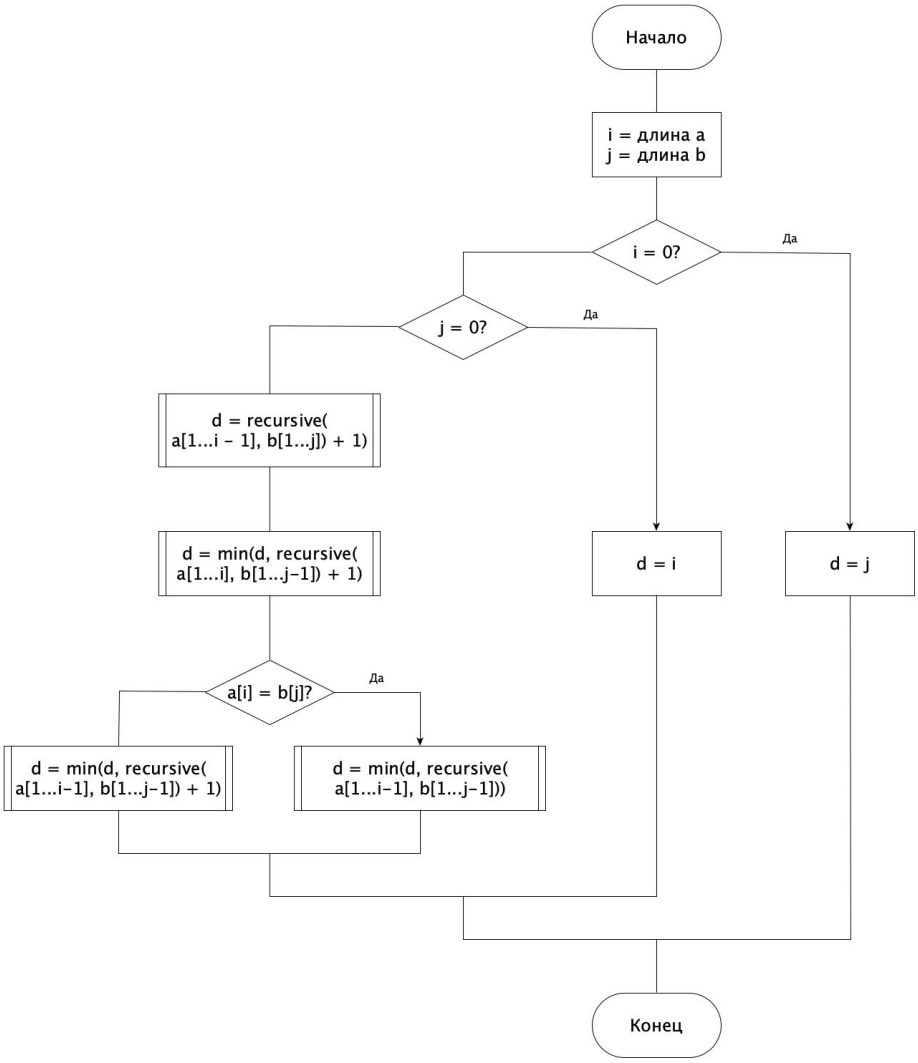

Рис. 2.1: Блок-схема рекурсивного агоритма нахождения расстояния Левенштейна

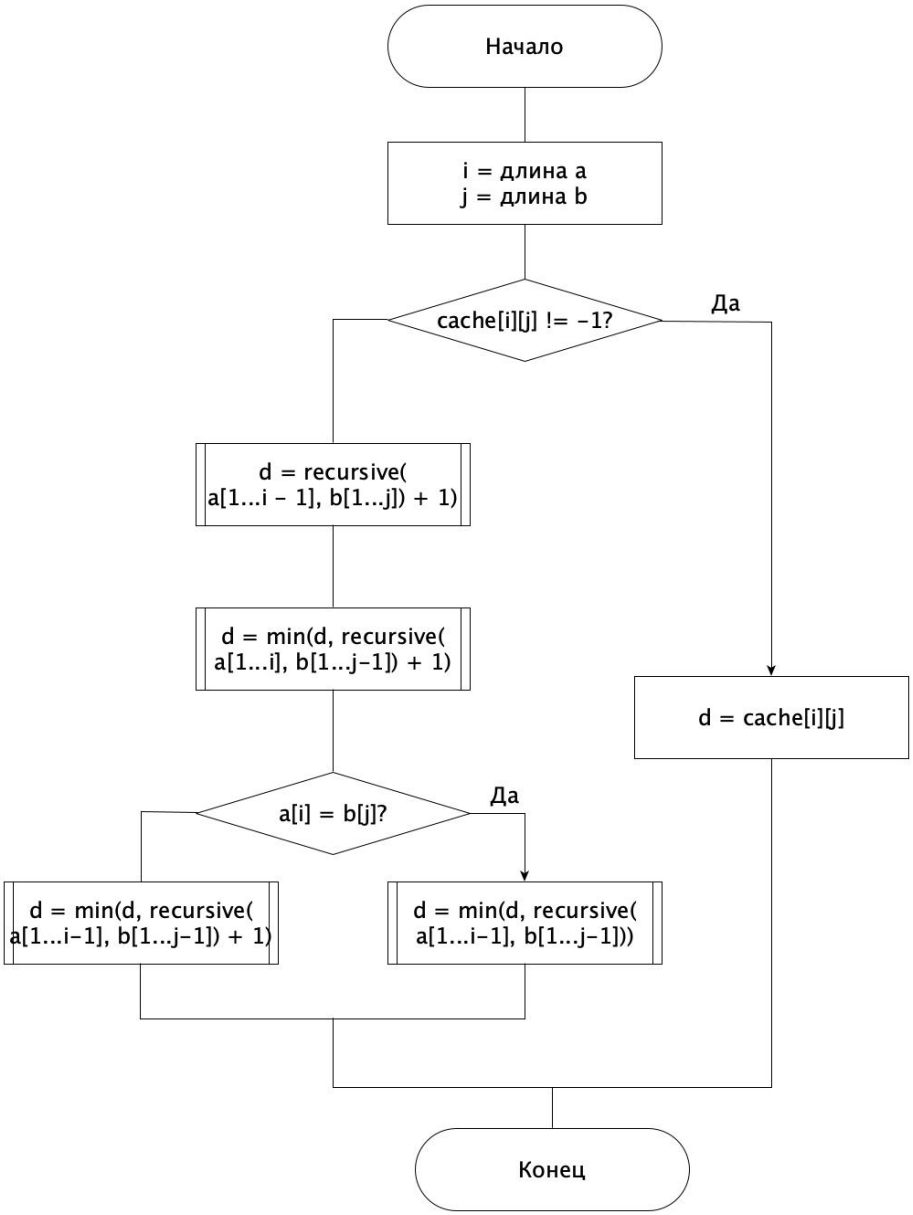

Рис. 2.2: Блок-схема рекурсивного агоритма нахождения расстояния Левенштейна с использованием матрицы

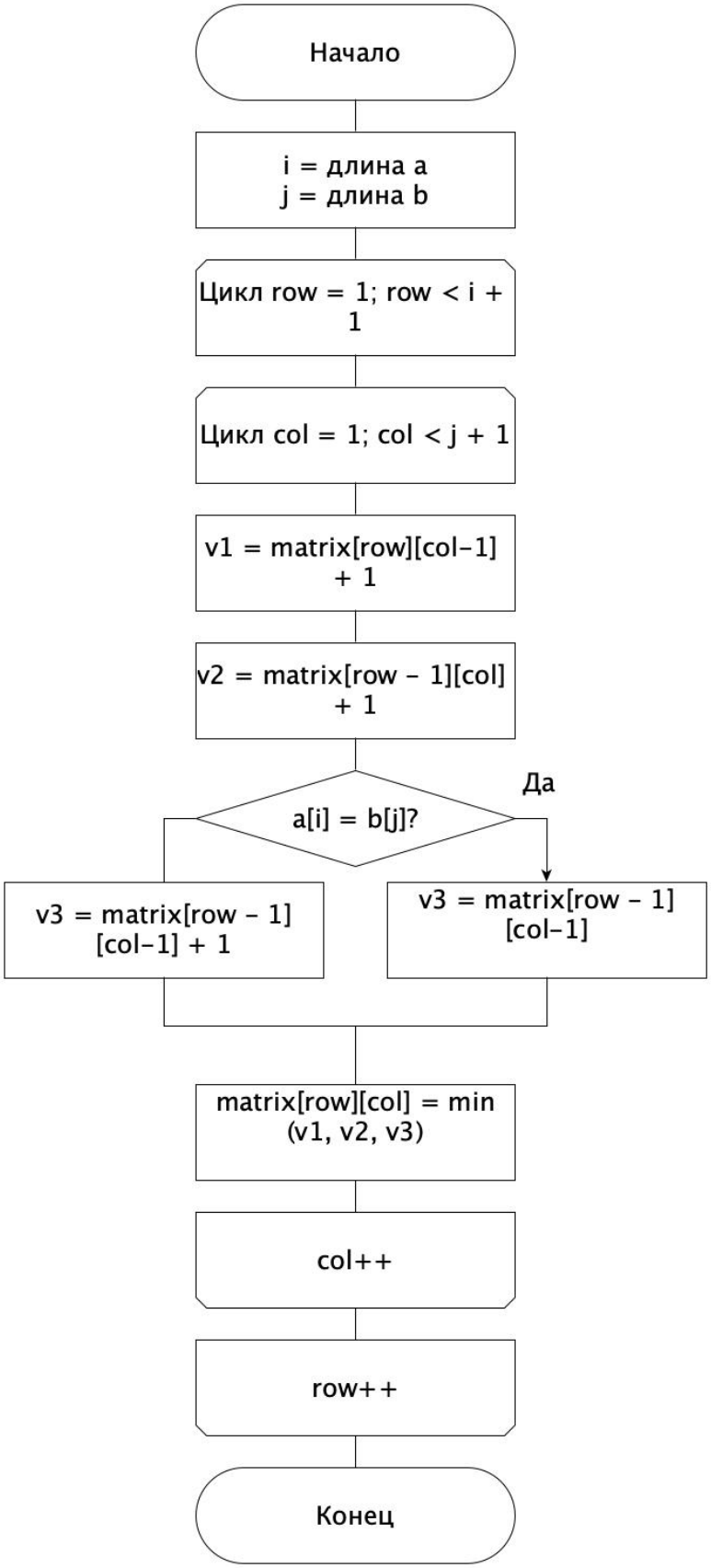

Рис. 2.3: Блок-схема итерационного агоритма нахождения расстояния Левенштейна

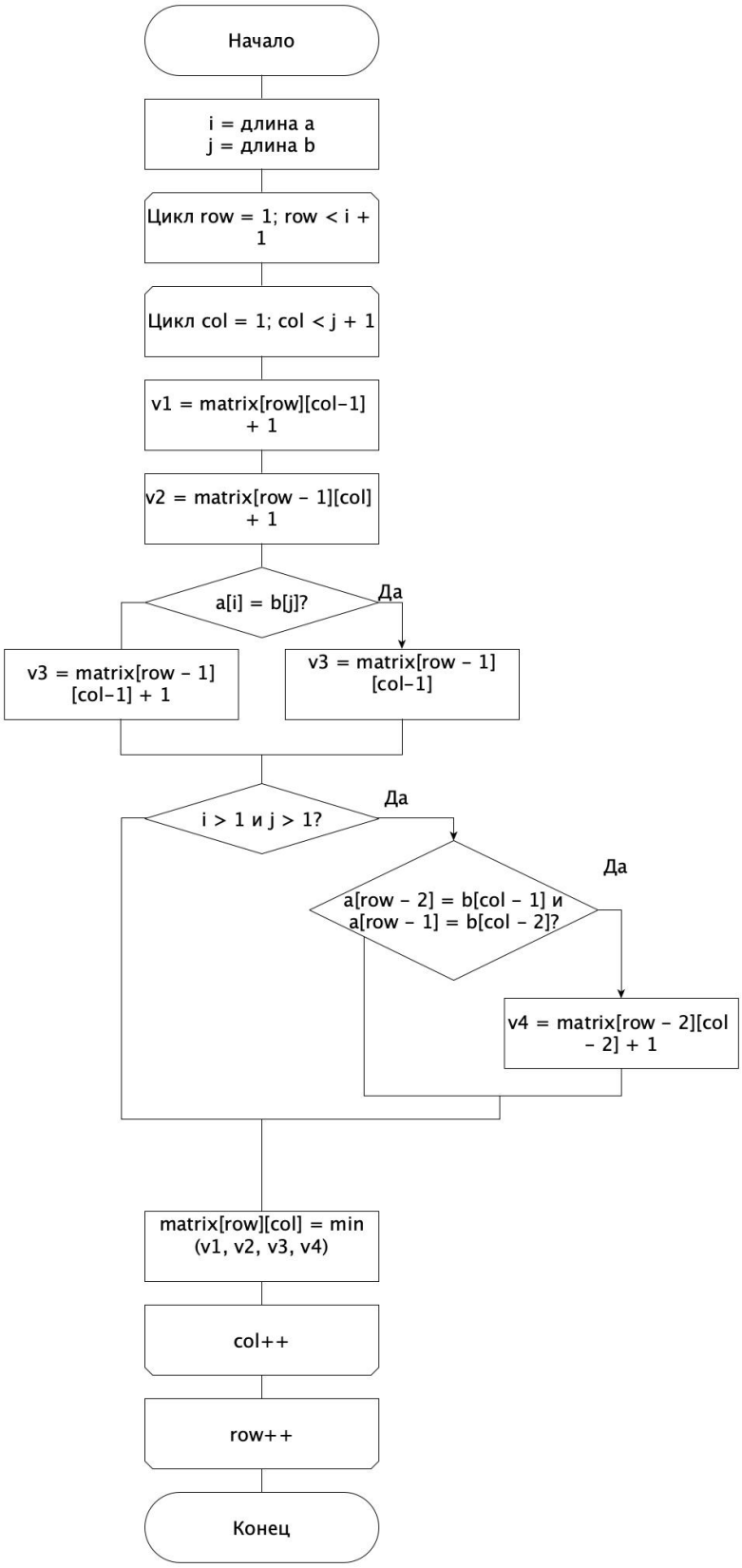

Рис. 2.4: Блок-схема итерационного агоритма нахождения расстояния Дамерау-Левенштейна

2.2 Вывод

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

3 | Технологическая часть

3.1 Требования к ПО

К разрабатываемому программному обеспечению предъявляются следующие требования:

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 Kotlin [4]. Такой выбор обусловлен широкими возможностями языка. Несмотря на то, что данный язык программирования только набирает популярность, его не коснулась проблема недостатка библиотек, как как он работает поверх Java Virtual Machine - можно использовать библиотеки из Java.

3.3 Реализация алгоритмов

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

Листинг 3.1: Базовый класс для всех алгоритмов

```
abstract class Algorithm() {
    open val name: String = "Abstract algorithm"
    protected var s1: String = ""
    protected var s2: String = ""
    public open val isDam: Boolean = false
    public abstract fun findDiff(): Int
    public fun setData(str1: String, str2: String) {
        s1 = str1
        s2 = str2
    }
}
```

Листинг 3.2: Метод для нахождения расстояния Левенштейна рекурсивно

```
class LevRecursionAlgorithm(): Algorithm() {
    override val name = "Levenshtein Recursive"
    override fun findDiff(): Int {
        return findDiff(s1.length, s2.length)
    }
    private fun findDiff(l1: Int, l2: Int): Int {
        if (l1 == 0) return l2
        if (l2 == 0) return l1
        val match: Int = if (s1[l1 - 1] == s2[l2 - 1]) 0 else 1
        val insertion: Int = findDiff(l1 - 1, l2) + 1
        val deletion: Int = findDiff(l1, l2 - 1) + 1
        val substitution: Int = findDiff(l1 - 1, l2 - 1) + match
        return minOf(insertion, deletion, substitution)
    }
}
```

Листинг 3.3: Метод для нахождения расстояния Левенштейна рекурсивно с матрицей

```
class LevMatrixAlgorithm(): Algorithm() {
    override val isDam = false
    override val name = "Levenshtein Recursive with cache"

    var array: Array<Array<Int>>? = null
    override fun findDiff(): Int {
        array = Array(s1.length + 1) { Array(s2.length + 1) { -1 } }
        return findDiff(s1.length, s2.length)
    }
    private fun findDiff(l1: Int, l2: Int): Int {
        if (array!![l1][l2] != -1)
            return array!![l1][l2]
        if (l1 == 0) return l2
        if (l2 == 0) return l1
        val match: Int = if (s1[l1 - 1] == s2[l2 - 1]) 0 else 1
        val insertion: Int = findDiff(l1 - 1, l2) + 1
        val deletion: Int = findDiff(l1, l2 - 1) + 1
        val substitution: Int = findDiff(l1 - 1, l2 - 1) + match
        val result = minOf(insertion, deletion, substitution)
        array!![l1][l2] = result
        return result
    }
}
```

Листинг 3.4: Метод для нахождения расстояния Левенштейна итерационно

```
class LevIterAlgorithm(): Algorithm() {
    override val name = "Levenshtein Iterable"
    override val isDam = false
    override fun findDiff(): Int {
        val array = Array(2) { Array(s2.length + 1) { -1 } }
        array[0].forEachIndexed { index, _ -> array[0][index] = index }
```

```
        for (index in 1..s1.length) {
            val i = 1
            if (index > 1) {
                array[0] = array[1]
                array[1] = Array(s2.length + 1) { -1 }
            }
            array[1][0] = index
            for (j in 1..s2.length) {
                val valueUp = array[i - 1][j] + 1
                val valueLeft = array[i][j - 1] + 1
                val diagMatch = if (s1[index - 1] == s2[j - 1]) 0 else 1
                val valueDiag = array[i - 1][j - 1] + diagMatch
                array[i][j] = minOf(valueUp, valueLeft, valueDiag)
            }
        }
        return array[1][s2.length]
    }
}
```

Листинг 3.5: Метод для нахождения расстояния Дамерау-Левенштейна рекурсивно

```
class DamRecursionAlgorithm(): Algorithm() {
    override val name = "Damerau-Levenshtein Recursive"
    override fun findDiff(): Int {
        return findDiff(s1.length, s2.length)
    }
    private fun findDiff(l1: Int, l2: Int): Int {
        if (l1 == 0) return l2
        if (l2 == 0) return l1
        val match: Int = if (s1[l1 - 1] == s2[l2 - 1]) 0 else 1
        val insertion: Int = findDiff(l1 - 1, l2) + 1
        val deletion: Int = findDiff(l1, l2 - 1) + 1
        var substitution: Int = findDiff(l1 - 1, l2 - 1) + match
        var result: Int = minOf(insertion, deletion, substitution)
        if (l1 > 1 && l2 > 1 && s1[l1 - 1] == s2[l2 - 2] && s1[l1 - 2] == s2
            [l2 - 1])
            result = minOf(findDiff(l1 - 2, l2 - 2) + 1, result)
        return result
    }
}
```

Листинг 3.6: Метод для нахождения расстояния Дамерау-Левенштейна рекурсивно с матрицей

```
class DamMatrixAlgorithm(): Algorithm() {
    var array: Array<Array<Int>>? = null
    override val name = "Damerau-Levenshtein Recursive with cache"
    override fun findDiff(): Int {
        array = Array(s1.length + 1) { Array(s2.length + 1) { -1 } }
        return findDiff(s1.length, s2.length)
    }
```

```
8     private fun findDiff(l1: Int, l2: Int): Int {
9         if (array!![l1][l2] != -1)
10            return array!![l1][l2]
11        if (l1 == 0) return l2
12        if (l2 == 0) return l1
13        val match: Int = if (s1[l1 - 1] == s2[l2 - 1]) 0 else 1
14        val insertion: Int = findDiff(l1 - 1, l2) + 1
15        val deletion: Int = findDiff(l1, l2 - 1) + 1
16        val substitutionOrSwap: Int = findDiff(l1 - 1, l2 - 1) + match
17        var result = minOf(insertion, deletion, substitutionOrSwap)
18        if (l1 > 1 && l2 > 1 && s1[l1 - 1] == s2[l2 - 2] && s1[l1 - 2] == s2
              [l2 - 1])
19            result = minOf(findDiff(l1 - 2, l2 - 2) + 1, substitutionOrSwap,
                  result)
20        array!![l1][l2] = result
21        return result
22    }
23 }
```

Листинг 3.7: Метод для нахождения расстояния Дамерау-Левенштейна итерационно

```
1
2  class DamIterAlgorithm(): Algorithm() {
3      override val name = "Damerau-Levenshtein Iterable"
4      override fun findDiff(): Int {
5          val array = Array(3) { Array(s2.length + 1) { -1 } }
6          array[0].forEachIndexed { index, _ -> array[0][index] = index }
7          for (index in 1..s1.length) {
8              var i = index
9              if (index > 2) {
10                 array[0] = array[1]
11                 array[1] = array[2]
12                 array[2] = Array(s2.length + 1) { -1 }
13                 i = 2
14             }
15             array[1][0] = index - 1
16             array[2][0] = index
17             for (j in 1..s2.length) {
18                 val valueUp = array[i - 1][j] + 1
19                 val valueLeft = array[i][j - 1] + 1
20                 val diagMatch = if (s1[index - 1] == s2[j - 1]) 0 else 1
21                 val valueDiag = array[i - 1][j - 1] + diagMatch
22                 var result = minOf(valueUp, valueLeft, valueDiag)
23                 if (index > 1 && j > 1 && s1[index - 2] == s2[j - 1] && s1[
                       index - 1] == s2[j - 2])
24                     result = minOf(result, array[0][j - 2] + 1)
25                 array[i][j] = result
26             }
27         }
28         if (s1.length > 2)
```

```
29                 return array[2][s2.length]
30             else
31                 return array[s1.length][s2.length]
32         }
33 }
```

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	кит	кот	1, 1	1, 1
2	собака	собачка	4, 2	4, 2
3	dija	djia	4, 4	4, 4
4			0, 0	0,0
5		абв	3, 3	3, 3
6	абв		3, 3	3, 3
7	увлечение	развлечения	4, 4	4, 4

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

```
Исходное слово - увлечение', конечное - 'развлечения'
Количество повторов - 1000
Рекурсивный алгоритм Левенштейна
    Avg CPU time: 27338653ns
    Ответ: 4
Алгоритм Левенштейна с матрицей
    Avg CPU time: 12125ns
    Ответ: 4
Итерационный алгоритм Левенштейна
    Avg CPU time: 12419ns
    Ответ: 4
Рекурсивный алгоритм Дамерау-Левенштейна
    Avg CPU time: 34489554ns
    Ответ: 4
Алгоритм Дамерау-Левенштейна с матрицей
    Avg CPU time: 14493ns
    Ответ: 4
Итерационный алгоритм Дамерау-Левенштейна
    Avg CPU time: 17765ns
    Ответ: 4
```

Рис. 4.1: Демонстрация работы программы

4.2 Технические характеристики

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

- операционная система — Mac OS Big Sur 11.4;

- процессор — Intel(R) Core(TM) i5-7267U CPU @ 2.90GHz [5];
- оперативная память — 8 GB.

4.3 Оценка времени алгоритмов

Kotlin работает поверх Java Virtual Machine, поэтому возможно использование Java библиотек [4]. Время выполнения алгоритмов замерялось с помощью Java-интерфейса для управления потоками виртуальной машины Java - ThreadMXBean [6], позволяющего вычислить процессорное время, затраченное на определенный процесс.

В таблице 4.1 представлены замеры времени работы для каждого из алгоритмов. Для всех алгоритмов время было усреднено по результатам 1000 замеров. Слова являлись случайным набором символов.

Таблица 4.1: Таблица времени выполнения алгоритмов (в наносекундах)

Длина строк	LevRec	LevRecMatrix	LevIter	LevDamRec	LevDamRecMatrix	LevDamIter
10	27432321	13730	20106	36903970	24900	27488
20	—	33711	45668	—	85596	69764
50	—	85698	94424	—	137873	163268
100	—	317519	213333	—	407036	248863

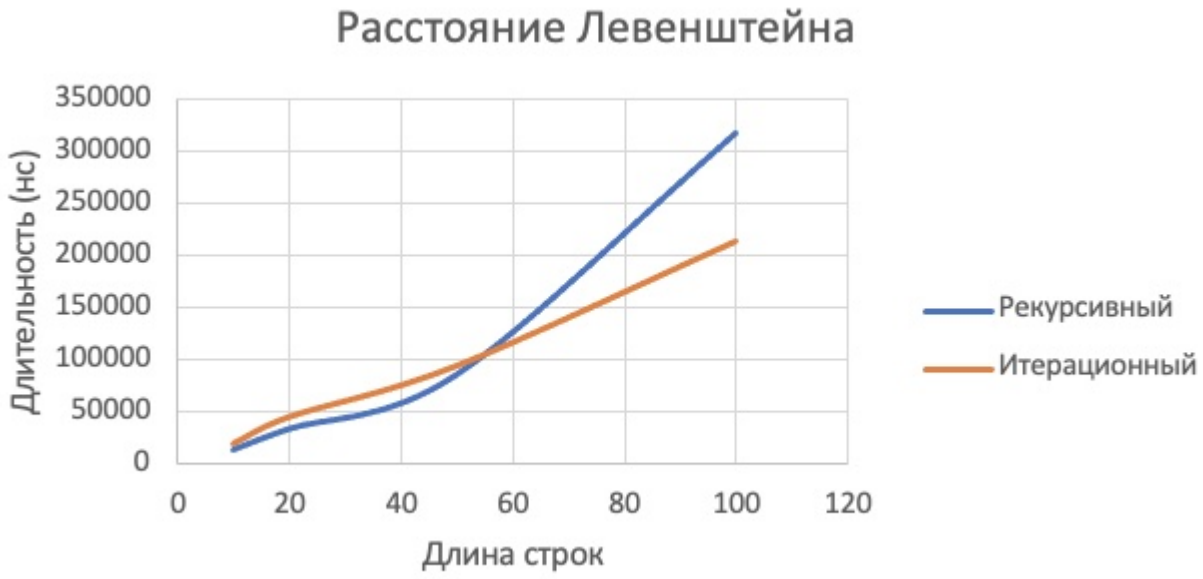

Рис. 4.2: График времени выполнения алгоритмов поиска расстояния Левенштейна

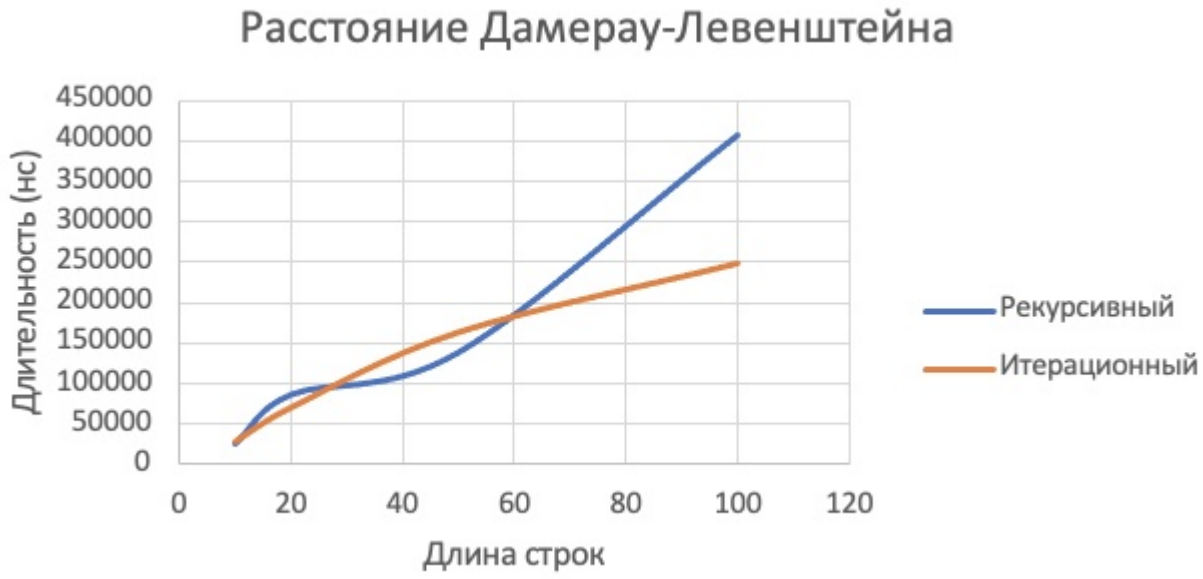

Рис. 4.3: График времени выполнения алгоритмов поиска расстояния Дамерау-Левенштейна

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 при длине слов от 60 символов: при длине в 100 символов итерационные алгоритмы справляются почти в два раза быстрее. Для меньшей длины разумно использовать рекурсивные алгоритмы. Время работы реализаций увеличивается в геометрической прогрессии. Рекурсивный алгоритм с матрицей работает немногим медленнее итерационного: для 100 символов разница составила порядка 30% для расстояния Левенштейна и порядка 40 % для Дамерау-Левенштейна.

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 оказались быстрее при длине слов до 60 символов, в остальных случаях итерационные показали уменьшение длительности работы до двух раз при длине слов в 100 символов.

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.

Литература

[1] Юрьевич Сапаров Алексеи. ПОИСК СОВПАДЕНИИ В ТЕКСТАХ ЗАПРОСОВ К БАЗАМ ДАННЫХ С ИСПОЛЬЗОВАНИЕМ РАССТОЯНИЯ ДАМЕРАУ-ЛЕВЕНШТЕИНА // Наука и просвещение.

[2] Калинин Петр. Динамическое программирование [Электронный ресурс]. Режим доступа: https://ejudge.lksh.ru/archive/2012/08/Aprime/texts/05$_d$*ynprog.pdf*. 2008. : 27.09.2021.

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

[4] Жемеров Д. Исакова С. Kotlin в действии. ДМК Пресс, 2018. Т. 402.

[5] 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: 27.09.2021.

[6] Interface ThreadMXBean [Электронный ресурс]. Режим доступа: https://docs.oracle.com/javase/7/docs/api/java/lang/management/ThreadMXBean.html. Дата обращения: 27.09.2021.