МГТУ им. Баумана

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

Расстояние Левенштейна

Работу выполнил: Ерохин Никита ИУ7-51Б

Преподаватели: Волкова Л.Л.

Москва, 2020

Оглавление

Введение

Расстояние Левенштейна - минимальное количество операций вставки, удаления или замены символа, необходимых для превращения одной строки в другую.

Поиск расстояний Левенштейна применяется в теории информации и компьютерной лингвистике для следующих задач:

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

Впервые задачу формализовал советский математик Владимир Левенштейн в 1965 году при изучении последовательностей. Более общую задачу для произвольного алфавита связали с его именем. Большой вклад в изучение вопроса внёс Дэн Гасфилд.

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

Глава 1

Аналитическая часть

1.1 Описание задачи

Расстояние Левинштейна между строками S_1 иS_2 - минимальное количество операций над этими строками, необходимых для преобразования одной строки в другую.
При нахождении расстояния Дамерау — Левенштейна добавляется операция транспозиции (перестановки соседних символов).

Операции обозначаются так:

1. I — вставить. Штраф - 1;
2. D — удалить. Штраф - 1;
3. R — заменить. Штраф - 1;
4. M - совпадение. Штраф - 0.

Пусть S_1 и S_2 — две строки длиной M и N соответственно над некоторым алфавитом. Тогда расстояние Левенштейна можно получить из следующей рекуррентной формулы:

$$D(i,j) = \begin{cases} 0, & i=0, j=0 \\ i, & j=0, i>0 \\ j, & i=0, j>0 \\ min(& \\ D(i,j-1)+1, & \\ D(i-1,j)+1, & j>0, i>0 \\ D(i-1,j-1)+m(S_1[i],S_2[j]) & \\ ), & \end{cases}$$

где $m(a,b)$ равна нулю при $a = b$, и единице в противном случае; $min\{\, a,b,c\}$ возвращает наименьший из аргументов.

Расстояние Дамерау-Левенштейна можно получить из следующей рекуррентной формулы:

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

1.2 Цель работы

Целью лабораторной работы является изучение и применение алгоритмов поиска расстояния Левенштейна и Дамрау-Левенштейна, а также получение практических навыков реализации следующих алгоритмов:

1. поиск расстояния Левенштейна рекурсивным способом;
2. поиск расстояния Левенштейна матричным способом;
3. поиск расстояния Левенштейна матрично-рекурсивным способом;

4. поиск расстояния Дамерау-Левенштейна табличным способом.

Глава 2

Конструкторская часть

Требования к программе:

- ввод двух строк;
- при введеных двух путых строках - корректный вывод, программа не должна завершиться аварийно;
- вывод редакционного расстояния;
- осуществление замера процессорного времени работы алгоритмов.

2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы алгоритмов посика расстояния Левенштейна:

- схема матричного алгоритма нахождения расстояния Левенштейна (рис. 2.1);
- схема рекурсивного алгоритма нахождения расстояния Левенштейна (рис. 2.3);
- схема матричного-рекурсивного алгоритма нахождения расстояния Левенштейна (рис. **??**);
- схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна изображена (рис. 2.4).

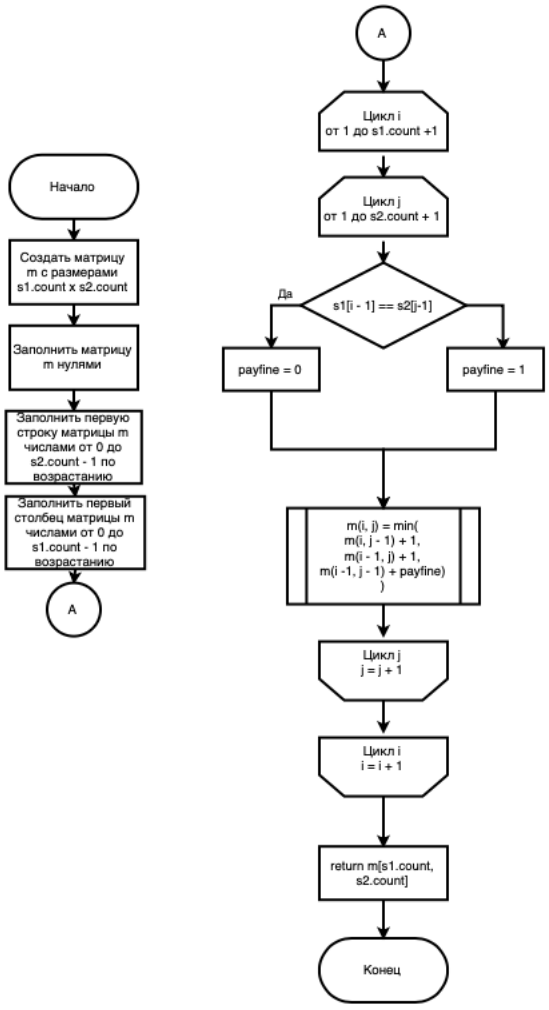

Рис. 2.1: Схема матричного алгоритма нахождения расстояния Левенштейна

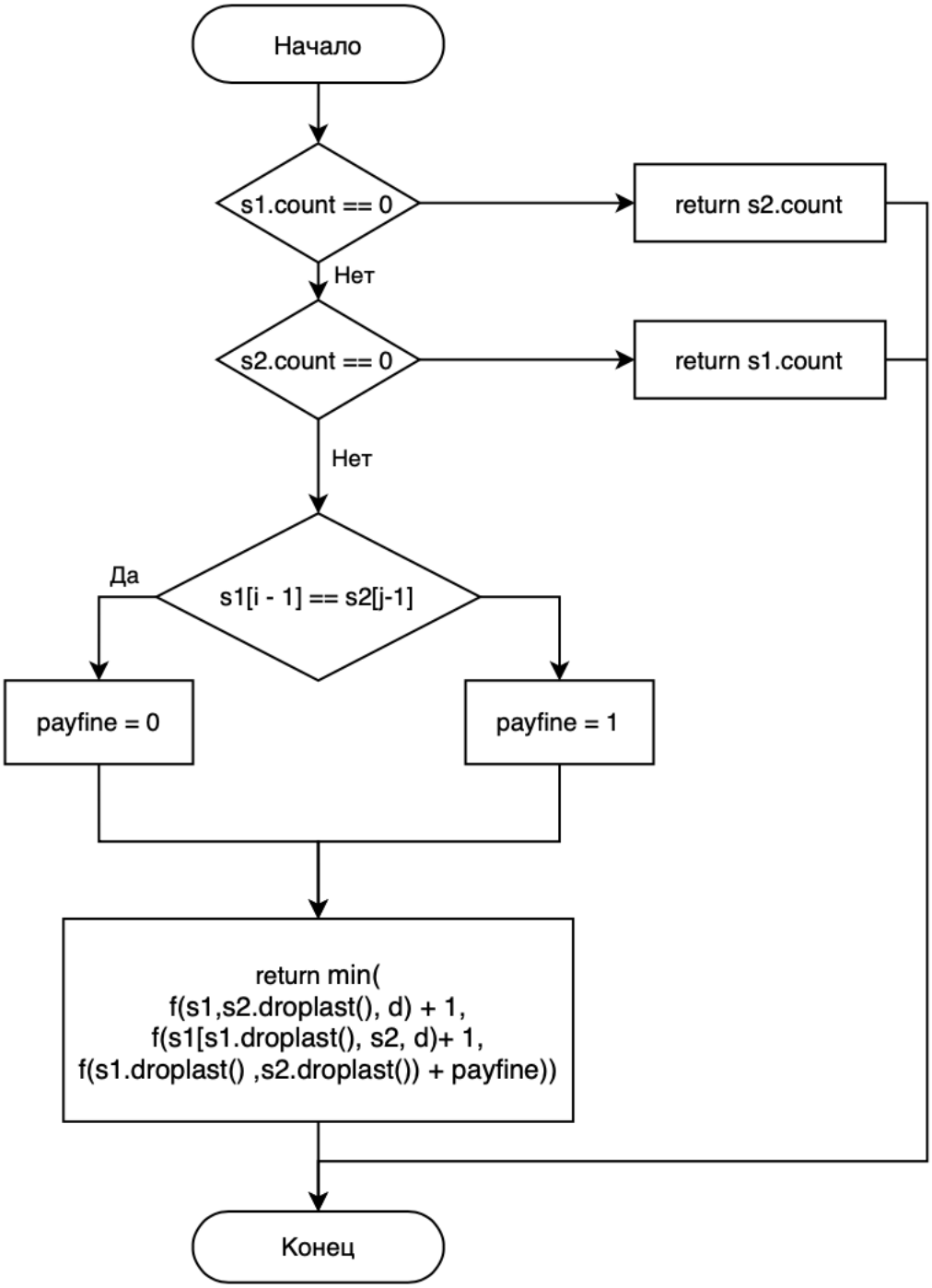

Рис. 2.2: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

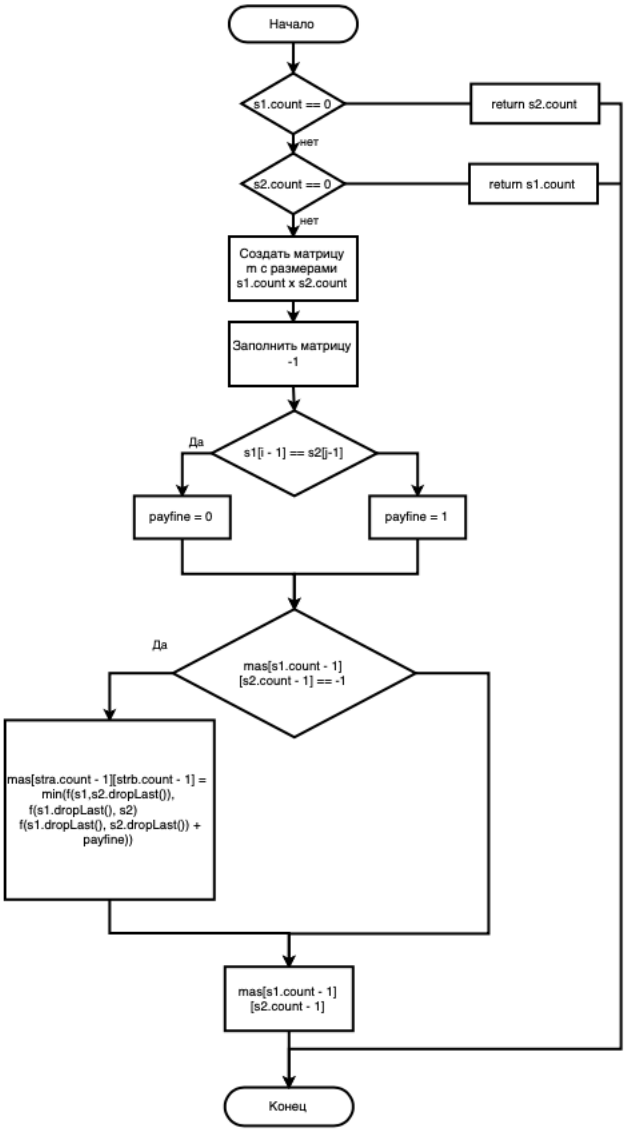

Рис. 2.3: Схема матричного-рекурсивного алгоритма нахождения расстояния Левенштейна

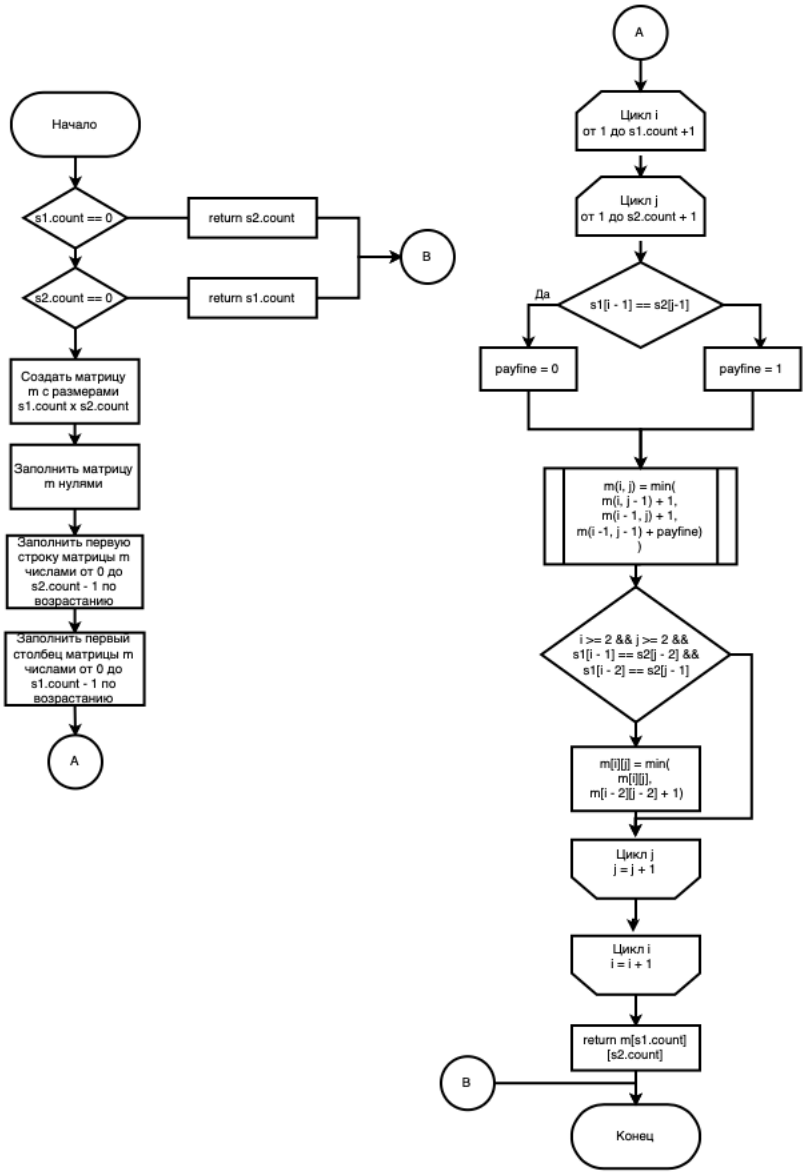

Рис. 2.4: Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

Глава 3

Технологическая часть

3.1 Выбор ЯП

Для реализации программы был выбран язык программирования swift, тк данный язык отличается скоростью компиляции, что значительно сократло время разработки.

3.2 Сведения о модулях программы

Программа состоит из:

- main.swift - тесты
- levenstein.swift - реализация представленных алгоритмов

Рекурсивный алгоритм нахождения расстояния Левенштейна изображен на листинге 3.1.

Листинг 3.1: Функция нахождения расстояния Левенштейна рекурсивно

```
1  func levensteinRecoursion(_ stra: String, _ strb: String)
       -> Int {
2      if (stra.count == 0) {
3          return strb.count
4      }
5      if (strb.count == 0) {
6          return stra.count
7      }
8      let payfine = stra.last != strb.last ? 1 : 0
9
10
11     return min(levensteinRecoursion(stra, String(strb.
           dropLast(1))) + 1, levensteinRecoursion(String(stra.
           dropLast(1)), strb) + 1,
12                levensteinRecoursion(String(stra.dropLast(1)),
                      String(strb.dropLast(1))) + payfine)
13 }
```

Матричный алгоритм нахождения расстояния Левенштейна изображен на листинге 3.2.

Листинг 3.2: Функция нахождения расстояния Левенштейна матрично

```
func levensteinMatrix(_ stra: String, _ strb: String) ->
    Int {
    if stra.count == 0 {
        return strb.count
    }
    if strb.count == 0 {
        return stra.count
    }

    var mas = getStartMas(stra.count, strb.count)

    for i in 1 ..< stra.count + 1 {
        for j in 1..<strb.count + 1 {
            let payfine = stra[i - 1] != strb[j - 1] ? 1 :
                0

            mas[i][j] = min(mas[i - 1][j] + 1,
                    mas[i][j - 1] + 1,
                    mas[i - 1][j - 1] + payfine)
        }
    }
    return mas[stra.count][strb.count]
}
```

Матрично-рекурсивный алгоритм нахождения расстояния Левенштейна изображен на листинге 3.3.

Листинг 3.3: Функция нахождения расстояния Левенштейна матрично-рекурсивно

```
func levensteinRecoursionMatrix(_ stra: String, _ strb:
    String, _ mas : inout [[Int]]) -> Int {
    if stra.count == 0 {
        return strb.count
    }
    if strb.count == 0 {
        return stra.count
    }

    if mas.count == 0 {
        mas = getStartMas(stra.count - 1, strb.count - 1,
            -1)
    }


    let payfine = stra[stra.count - 1] != strb[strb.count -
        1] ? 1 : 0

    if (mas[stra.count - 1][strb.count - 1] == -1) {
        mas[stra.count - 1][strb.count - 1]
            = min(levensteinRecoursionMatrix(stra, String(
                strb.dropLast(1)), &mas) + 1,
                    levensteinRecoursionMatrix(String(stra.
                        dropLast(1)), strb, &mas) + 1,
                    levensteinRecoursionMatrix(String(stra.
                        dropLast(1)), String(strb.dropLast(1))
                        , &mas) + payfine)
    }
    return mas[stra.count - 1][strb.count - 1]
}
```

Матричный алгоритм нахождения расстояния Дамерау-Левенштейна изображен на листинге 3.4.

Листинг 3.4: Функция матричного нахождения расстояния Дамерау-Левенштейна

```
func damerauLevenstein(_ stra: String, _ strb: String) ->
    Int {

    if stra.count == 0 {
        return strb.count
    }
    if strb.count == 0 {
        return stra.count
    }


    var mas = getStartMas(stra.count, strb.count)

    for i in 1..<stra.count + 1 {
        for j in 1..<strb.count + 1 {
            let payfine = stra[i - 1] != strb[j - 1] ? 1 :
                0
            mas[i][j] = min(mas[i - 1][j] + 1,
                    mas[i][j - 1] + 1,
                    mas[i - 1][j - 1] + payfine)
            if (i >= 2 && j >= 2 && stra[i - 1] == strb[j -
                2] && stra[i - 2] == strb[j - 1]) {
                mas[i][j] = min(mas[i][j], mas[i - 2][j -
                    2] + 1)
            }
        }
    }
    return mas[stra.count][strb.count]

}
```

3.3 Тестирование программы по методу черного ящика

В данном разделе будут приведены результаты тестирования программы.

На Рис. 3.1 показано поведение программы при вводе пустых строк.

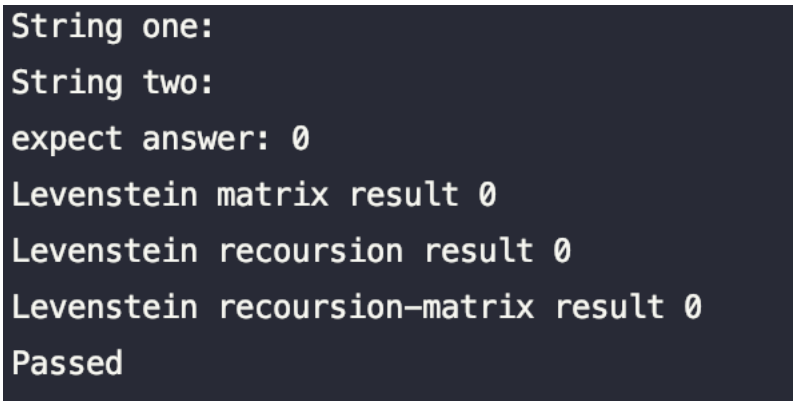

Рис. 3.1: Тест при строках размером 0

На Рис. 3.2 показано поведение программы одной пустой и одной непустой строке.

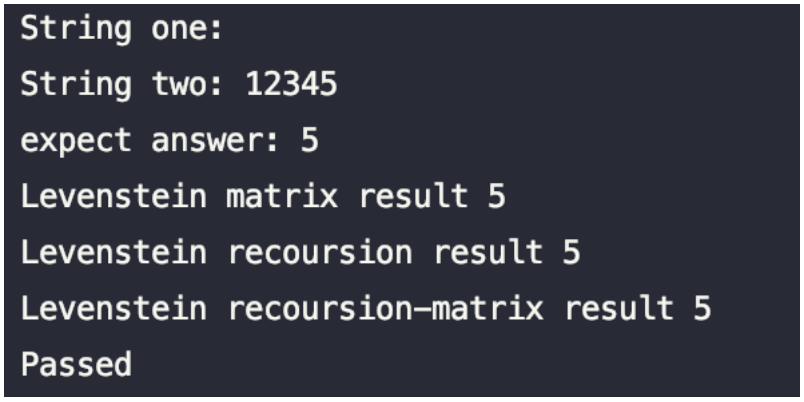

Рис. 3.2: Тест при одной строке длинны 0 и одной непустой строке

На Рис. 3.3 показано поведение программы строках "1234" и "1".

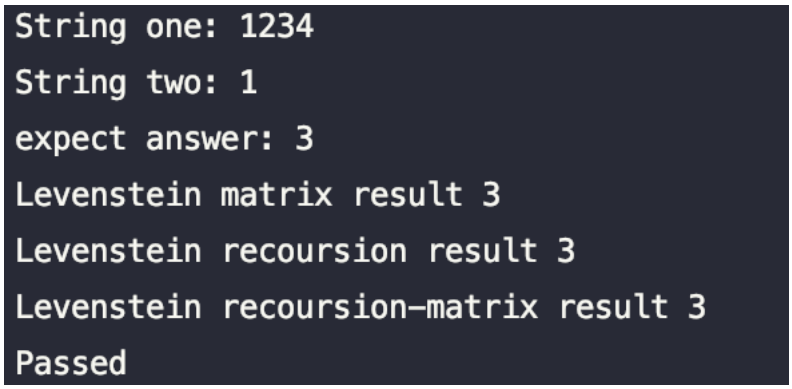

Рис. 3.3: Тест при введеных строках "1234" и "1"

На Рис. 3.4 показано поведение программы строках "*kitten*" и "*mittens*".

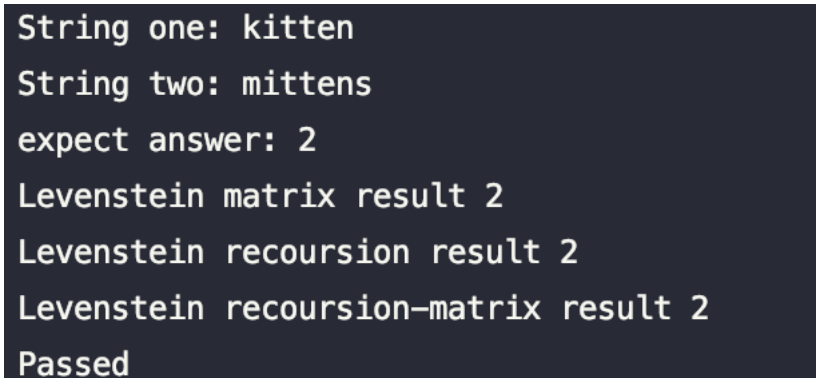

Рис. 3.4: Тест при введеных строках "*kitten*" и "*mittens*"

На Рис. 3.5 показано поведение тестирование функции поиска расстояния Дамерау-Левенштейна для строк "*iphone*" и "*iphnoe*".

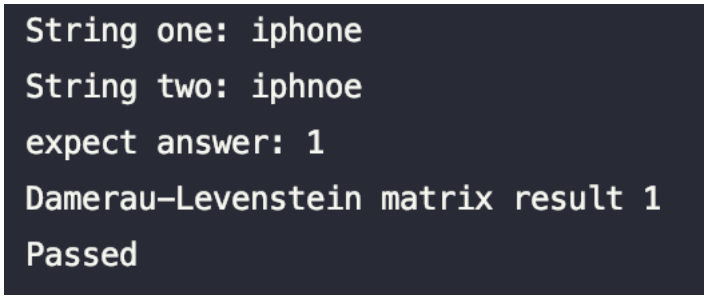

Рис. 3.5: Тестирование функции поиска расстояния Дамерау-Левенштейна для строк "*iphone*" и "*iphnoe*"

На Рис. 3.6 показано поведение тестирование функции поиска расстояния Дамерау-Левенштейна для строк "*supercat*" и "*uppercta*".

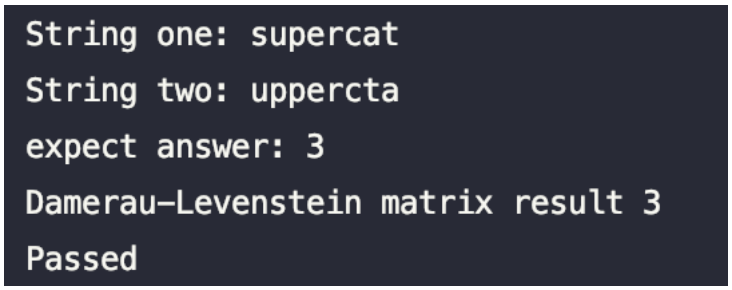

Рис. 3.6: Тестирование функции поиска расстояния Дамерау-Левенштейна для строк "*supercat*" и "*uppercta*"

Глава 4

Исследовательская часть

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Был произведен замер времени работы каждогой из реализаций алгоритмов (таблица 4.1).

Введем обозначения:

- lmatrix - матричный алгоритм поиска расстояния Левенштейна;
- lrnomatrix - рекурсивный алгоритм поиска расстояния Левенштейна;
- lrmatrix - матрично-рекурсивный алгоритм поиска расстояния Левенштейна;
- dameraulevenstein - матричный алгоритм поиска расстояния Дамерау-Левенштейна.

Таблица 4.1: Время работы алгоритмов (в секундах)

Количество символов	lmatrix	lrnomatrix	lrmatrix	damerauLevenstein
3	0.000023	0.000051	0.000038	0.000018
4	0.000019	0.000209	0.000052	0.000021
5	0.000026	0.001001	0.000082	0.000030
6	0.000035	0.004965	0.000106	0.000034
7	0.000048	0.026186	0.000141	0.000045

На основе таблицы был построен график(Рис. 4.1).

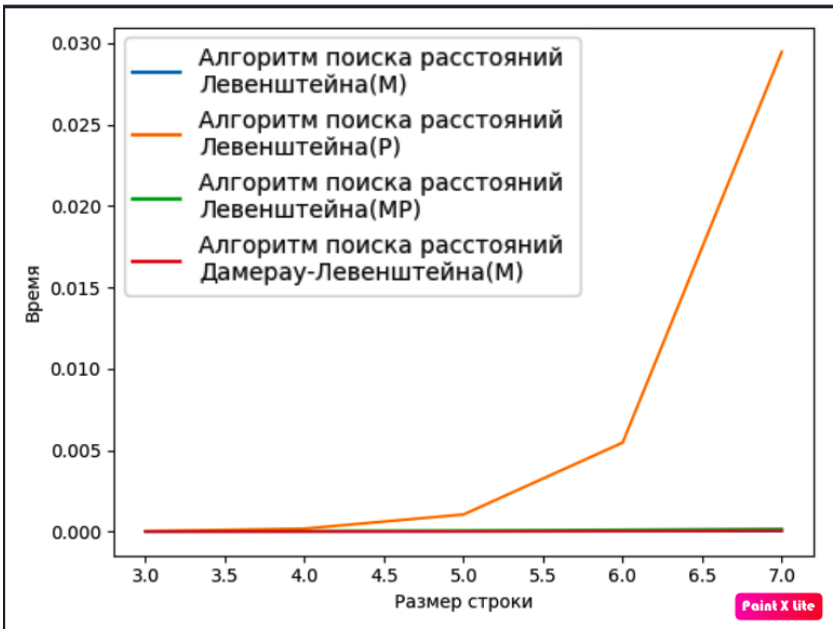

Рис. 4.1: Сравнение времени выполнение алгоритмов

Данный график наглядно демонстрирует недостатки рекурсивной реализации алоритма поиска расстояния Левенштейна. Однако масштаб не позволяет сравнить более эффективные алгоритмы. Далее, на (рис. 4.2) будут рассмотрены 3 графика из 4.

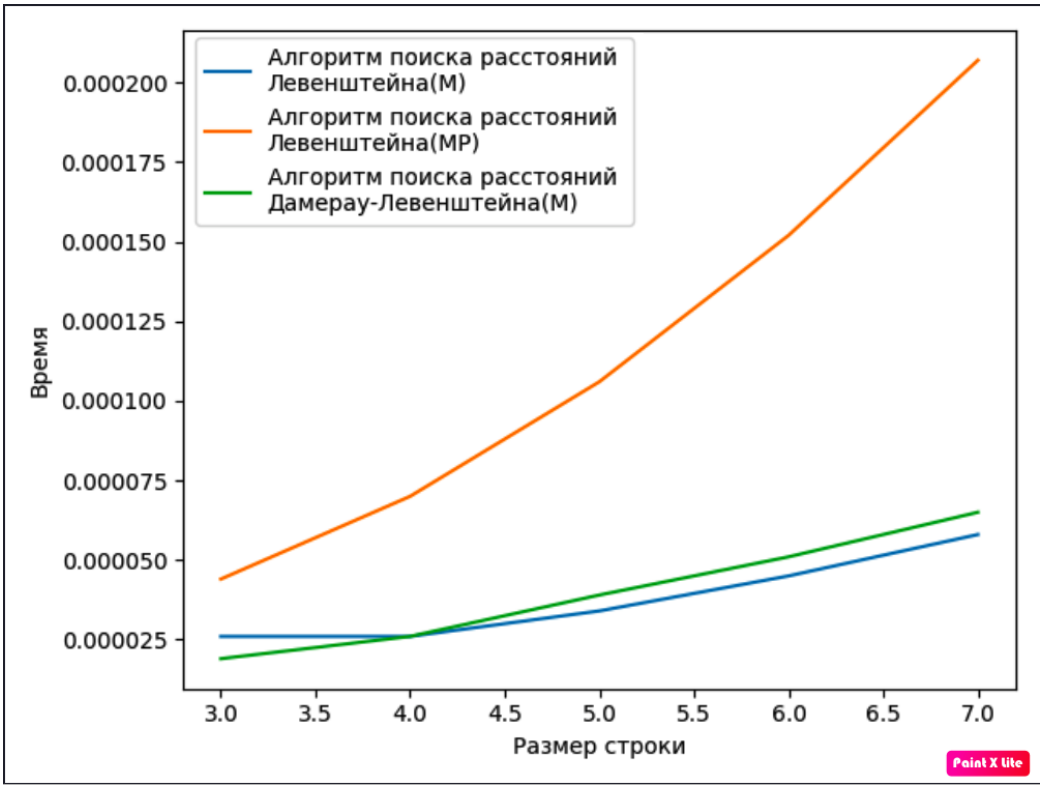

Рис. 4.2: Сравнение времени выполнение алгоритмов Левенштейна(Матричная реализация), Левенштейна(Матрично-рекурсивная реализация), Дамерау-Левенштейна(Матричная реализация)

Из полученных графиков можно сделать вывод, что рекурсивный алгоритм является самым медленным. Но можно уменьшить количество повторных вызовов в рекурсивном алгоритме Левенштейна, если использовать матрицу, в которой все поля изначально -1(бесконечность). Так как не будет повторного вызова. Самым быстрым алгоритмом является матричный алгоритм Левенштейна. Из полученных графиков можно сделать вывод, что рекурсивная реализация алгоритма поиска расстояния Левенштейна является самымой медленной. Ее возможно улучшить, уменьшить количество повторных вызовов в рекурсивном алгоритме. Для этого необходимо использовать матрицу, в которой все поля изначально -1. Самым быстрым алгоритмом является матричная реализация алгоритма поиска расстояния Левенштейна.

4.2 Сравнительный анализ ёмкостной характеристики алгоритмов

Проведем анализ ёмкостной характеристики у разных структур данных:

Емкностная характеристика у различный структур данных приведена в таблице 4.2.

Таблица 4.2: Емкностная характеристика у различный структур данных(в байтах)

Структура данных	Память(байты)
Пустая строка	49
Целое число	28
Пустой список	56
Список, хранящий указатель	64

Емкностная характеристика матричного алгоритма Дамерау-Левенштейна приведена в таблице 4.3.

Таблица 4.3: Емкностная характеристика матричного алгоритма Левенштейна

Структура данных	Занимаемая память(байты)
Матрица	56 + 28 * (n1 + 1) + (n1 + 1) *(56 + 28*(n2 + 1))
Счетчики	84
Передача параметров	2 * [49 + n]
Вспомогательные переменные	56

Емкностная характеристика матричного алгоритма поиска расстояния Дамерау-Левенштейна приведена в таблице 4.4.

Таблица 4.4: Емкностная характеристика матричного алгоритма Дамерау-Левенштейна

Структура данных	Занимаемая память(байты)
Матрица	56 + 28 * (n1 + 1) + (n1 + 1) *(56 + 28*(n2 + 1))
Счетчики	56
Передача параметров	2 * [49 + n]
Вспомогательные переменные	112

Емкностная характеристика рекурсивного алгоритма поиска растояния Левенштейна приведена в таблице 4.5.

Таблица 4.5: Емкностная характеристика рекурсивного алгоритма поиска растояния Левенштейна

Структура данных	Занимаемая память(байты)
Передача параметров + глубина рекурсии	2 * (49 + n) * (n + m)

Емкностная характеристика рекурсивного алгоритма поиска растояния Левенштейна приведена в таблице 4.6.

Таблица 4.6: Емкностная характеристика матрично-рекурсивного алгоритма поиска расстояния Левенштейна.

Структура данных	Занимаемая память(байты)
Матрица	56 + 28 * (n1 + 1) + (n1 + 1) *(56 + 28*(n2 + 1))
Передача параметров	2 * (49 + n) * (n + m)

Исходя из данных, привиденных выше, можно сделать вывод, что матричная реализация алгоритма поиска расстояния Левенштейна и Дамерау-Левенштейна имеют одинаковые затраты по памяти. Самым неэффекти-

вынм по памяти является матрично-рекурсивный способ Левенштейна, так как содержит в себе матрицу.

Заключение

В процессе выполнения данной лабораторной работы были реализованы и проанализированы различные реализации алгоритма поиска расстояний Левенштейна. Было проведено тестирование по методу "" и теоретический замер памяти.

Было подтверждено различие в эффективности различных реализаций(в частности, рекурсивной и нерекурсивной и матрично-рекурсивной) алгоритма определения расстояния Левенштейна между строками.

Список использованных источников

1. Дж. Макконнелл. Анализ Алгоритмов. Активный обучающий подход.-М.:Техносфера, 2009.
3. Расстояние Дамерау–Левенштейна [Электронный ресурс] Режим доступа: https://elementy.ru/problems/1068 (дата обращеия 01.10.20).
4. Вычисление редакционного расстояния [Электронный ресурс]. Режим доступа: https://habr.com/ru/post/117063/ (дата обращения 30.09.20).