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» (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.01 Информатика и вычислительная техника**

ОТЧЕТ

по лабораторной работе № 1

Название: Расстояние Левенштейна и Дамерау-Левенштейна

Дисциплина: Анализ алгоритмов

Студент ИУ7-52Б _____ Короткая В. М.
(Группа) (Подпись, дата) (И.О. Фамилия)

Преподаватель _____ Волкова Л.Л.
(Подпись, дата) (И.О. Фамилия)

Москва, 2021

Оглавление

Введение **3**

1 Аналитическая часть **5**

1.1 Расстояние Левенштейна, рекурсивный метод 5

1.2 Расстояние Левенштейна, матричный метод 6

1.3 Расстояние Левенштейна, рекурсивный метод с заполнением матрицы 6

1.4 Расстояние Дамерау-Левенштейна, матричный метод 7

2 Конструкторская часть **8**

2.1 Расстояние Левенштейна, рекурсивный метод 8

2.2 Расстояние Левенштейна, матричный метод 9

2.3 Расстояние Левенштейна, рекурсивный метод с заполнением матрицы 10

2.4 Расстояние Дамерау-Левенштейна, матричный 11

3 Технологическая часть **12**

3.1 Средства реализации 12

3.2 Требования к ПО 12

3.3 Сведения о модулях программы 12

3.4 Листинг кода 12

3.5 Тестирование 16

3.6 Вывод 16

4 Исследовательская часть **17**

4.1 Временные характеристики 17

4.2 Характеристики по памяти 18

4.3 Сравнительный анализ алгоритмов 19

4.4 Вывод 19

Заключение **20**

Список литературы **21**

Введение

Расстояние Левенштейна (также известное как редакционное расстояние) в теории информации и компьютерной лингвистике – мера различия двух последовательностей символов (строк) относительно минимального количества операций вставки, удаления и замены, необходимых для перевода одной строки в другую. Для одинаковых строк расстояние редактирования равно нулю.

В 1965 году советский математик Владимир Иосифович Левенштейн разработал алгоритм, который позволяет оценить, насколько похожа одна строка на другую. Алгоритм Левенштейна дает возможность получить именно численную оценку похожести строк.

Основная идея алгоритма состоит в том, чтобы посчитать минимальное количество операций удаления, вставки и замены, которые необходимо сделать над одной из строк, чтобы получить вторую.

В настоящее время алгоритм Левенштейна активно применяется в различных программных продуктах, в том числе грамматических приложениях, таких как в MS Office или подобных для решения следующих прикладных задач:

- в поисковых системах для нахождения объектов или записей по имени;
- в базах данных при поиске с неполно-заданным или неточно-заданным именем;
- для исправления ошибок при вводе текста;
- для исправления ошибок в результате автоматического распознавания отсканированного текста или записей речи;
- в приложениях, связанных с автоматической обработкой текстов.

Функция Левенштейна может играть роль фильтра, заведомо отбрасывающего неприемлемые варианты (у которых значение функции больше некоторой заданной константы).

Цель данной лабораторной работы заключается в реализации и последующем сравнении алгоритмов поиска расстояний Левенштейна и Дамерау-

Левенштейна.

Для достижения поставленной цели необходимо выполнить следующие задачи:

- математически описать расстояние Левенштейна и Домерау-Левенштейна;
- описать и реализовать алгоритмы поиска расстояний;
- замерить процессорное время работы алгоритмов при различных размерах строк;
- оценить наибольшую затрачиваемую память для каждого из алгоритмов;
- провести сравнительный анализ алгоритмов на основании проведённых экспериментов;

1. Аналитическая часть

В данном разделе будут рассмортенно формальное описание алгоритмов.

Поиск расстояний несёт в себе задачу нахождения такой последовательности операций, применение которых даст в результате минимальный суммарный штраф.

Существуют следующие штрафы:

- вставка (I, от англ. insert) - 1;
- замена (R, от англ. replace) - 1;
- удаление (D, от англ. delete) - 1;
- совпадение (M, от англ. match) - 0;
- транспозиция (T, от англ. transposition) - 1.

Рассмотрим алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна для строк *S1* и *S2* с длинами *l1* и *l2* соответственно.

1.1. Расстояние Левенштейна, рекурсивный метод

В методе используется рекурсивная формула нахождения *D(S1[1..i], S2[1..j])*, реализовать это можно с помощью рекурсивной функции. Функция будет принимать в качестве входных данных строки *S1* и *S2*, а также их длины i и j соответственно. Метод основан на последующем вызове той же функции для тех же строк, но для длин *(i - 1, j - 1)*, *(i - 1, j)*, *(i, j - 1)*, где возвращается минимальное из этих значений.

Данная проблема решается использованием рекуррентной формулы вычисления расстояний. Пусть *D(S1[1..i], S2[1..j])* - расстояние Левенштейна для подстроки *S1* и *S2* с длинами i и j соответственно.

Тогда, формула для вычисления D имеет следующий вид:

$$D(i,j) = \begin{cases} j, & \text{если i} = 0 \\ i, & \text{если j} = 0 \\ min(D(S1[1..i], S2[1..j-1]) + 1, \\ D(S1[1..i-1], S2[1..j]) + 1, \\ D(S1[1..i-1], S2[1..j-1] + \begin{cases} 0, \text{если S1[i] = S2[j]} \\ 1, \text{иначе} \end{cases}) \end{cases}$$

1.2. Расстояние Левенштейна, матричный метод

Данный матричный метод основан на использовании рекуррентной формулы. В начале работы создаётся целочисленная матрица с размерами *(l1 + 1)* на *(l2 + 1)*, затем заполняются первый столбец и первая строка, которые являются базой для рекуррентной формулы. Матрица заполняется построчно, в каждой ячейке *[i][j]* матрицы записывается значение *D(S1[1..i - 1], S2[1..j - 1])*. В том случае, когда $i = 1$ и $j = 1$ будет значить, что строки пусты. Итоговым результатом является значение в нижней правой ячейке матрицы, т.е. в ячейке с индексом *[l1 + 1][l2 + 1]*.

1.3. Расстояние Левенштейна, рекурсивный метод с заполнением матрицы

В данном методе создаётся матрица размерами *(l1 + 1)* на *(l2 + 1)*, все ячеки которой изначально заполнены значением бесконечности. В каждой клетке *[i][j]* этой матрицы будет записано значение *D(s1[1..i - 1], s2[1..j - 1])*.

Рекурсивная функция получает матрицу, индексы i, j положения в ней и две строки. Алгоритм начинает свою работу с ячейки *[1][1]*, которая заполняется значением *0*. Из положения *[i][j]* рассматривается переход в соседние ячейки *[i + 1][j + 1]*, *[i + 1][j]*, *[i][j + 1]*. В случае, если соседняя ячейка расположена в пределах матрицы и расстояние R при переходе из данной ячейки меньше ныне хранимого в ней значения, то значение соседней ячейки меняется на R, после чего функция запускается уже для соседней ячейки. После завершения работы всех функций, расстояние Левенштейна расположено в

ячейке *[l1 + 1][l2 + 1]*.

1.4. Расстояние Дамерау-Левенштейна, матричный метод

Метод является модифицированным методом подсчёта расстояния Левенштейна матричным методом. В матрице для ячейки *[i][j]*, где $i > 2$ и $j > 2$, учитывается также вариант перехода из клетки *[i - 2][j - 2]*, в случае когда *S1[i] = S2[j-1]* и *S1[i-1] = S[j]*. Результатом всё также будет правое нижнее значение ячейки *[l1 + 1][l2 + 1]*.

Аналогично и для Дамерау-Левенштейна:

$$D(i,j) = \begin{cases} j, \text{если i} = 0 \\ i, \text{если j} = 0 \\ min(D(S1[1..i], S2[1..j-1]) + 1, \\ D(S1[1..i-1], S2[1..j]) + 1, \\ D(S1[1..i-1], S2[1..j-1] + \begin{cases} 0, \text{если S1[i] = S2[j]} \\ 1, \text{иначе} \end{cases}, \\ \begin{cases} D(S1[1..i-2], S2[1..j-2]) + 1, \text{ если } \begin{cases} i > 1, j > 1 \\ S1[i] = S2[j-1] \\ S1[i-1] = S2[j] \end{cases} \\ +\infty, \text{иначе} \end{cases} \end{cases})$$

2. Конструкторская часть

В данном разделе представлены схемы алгоритмов.

2.1. Расстояние Левенштейна, рекурсивный метод

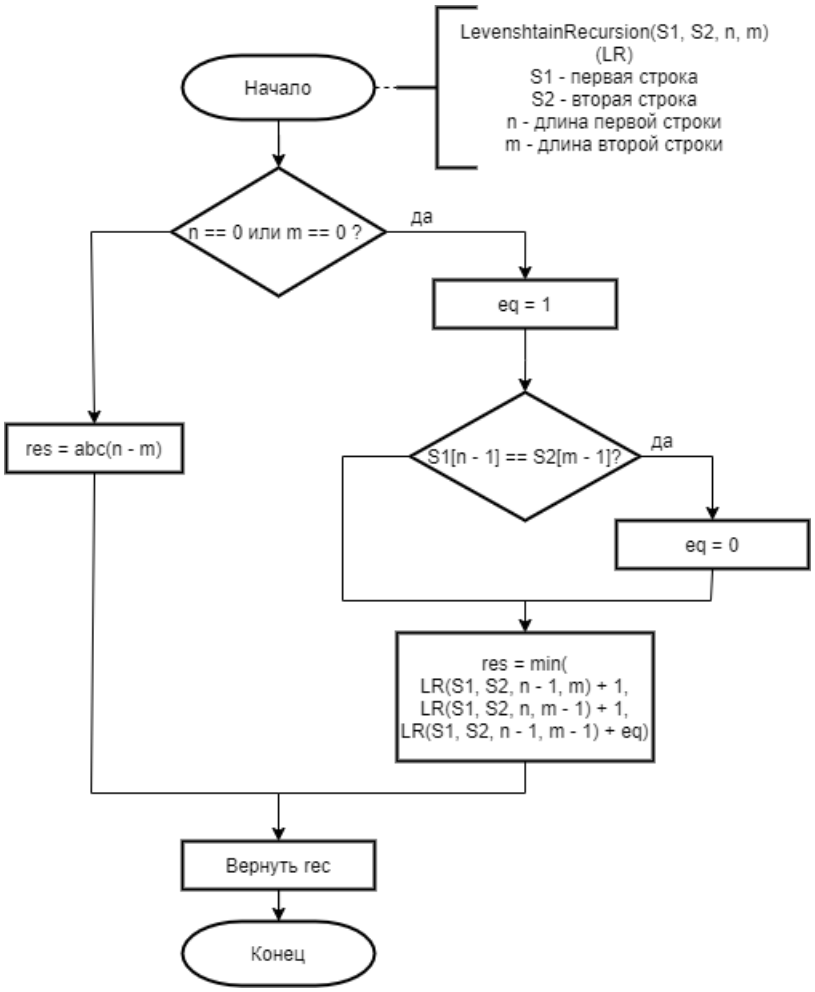

Рис. 2.1: Рекурсивный алгоритм Левенштейна

2.2. Расстояние Левенштейна, матричный метод

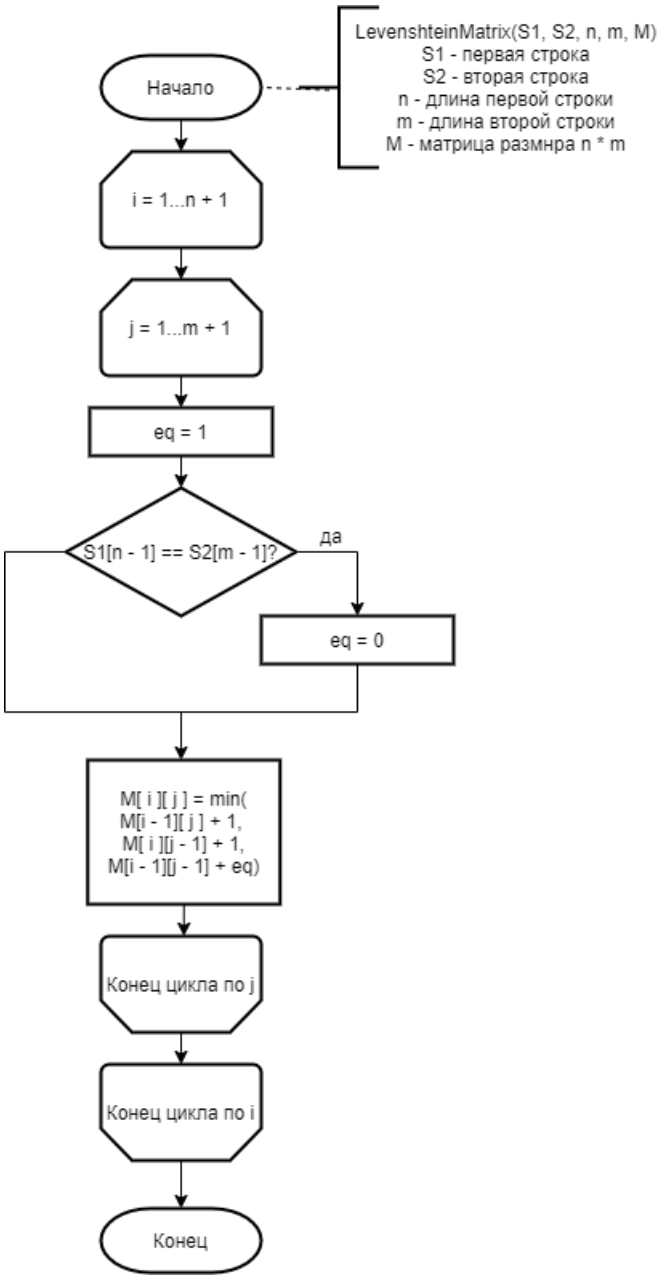

Рис. 2.2: Матричный алгоритм Левенштейна

2.3. Расстояние Левенштейна, рекурсивный метод с заполнением матрицы

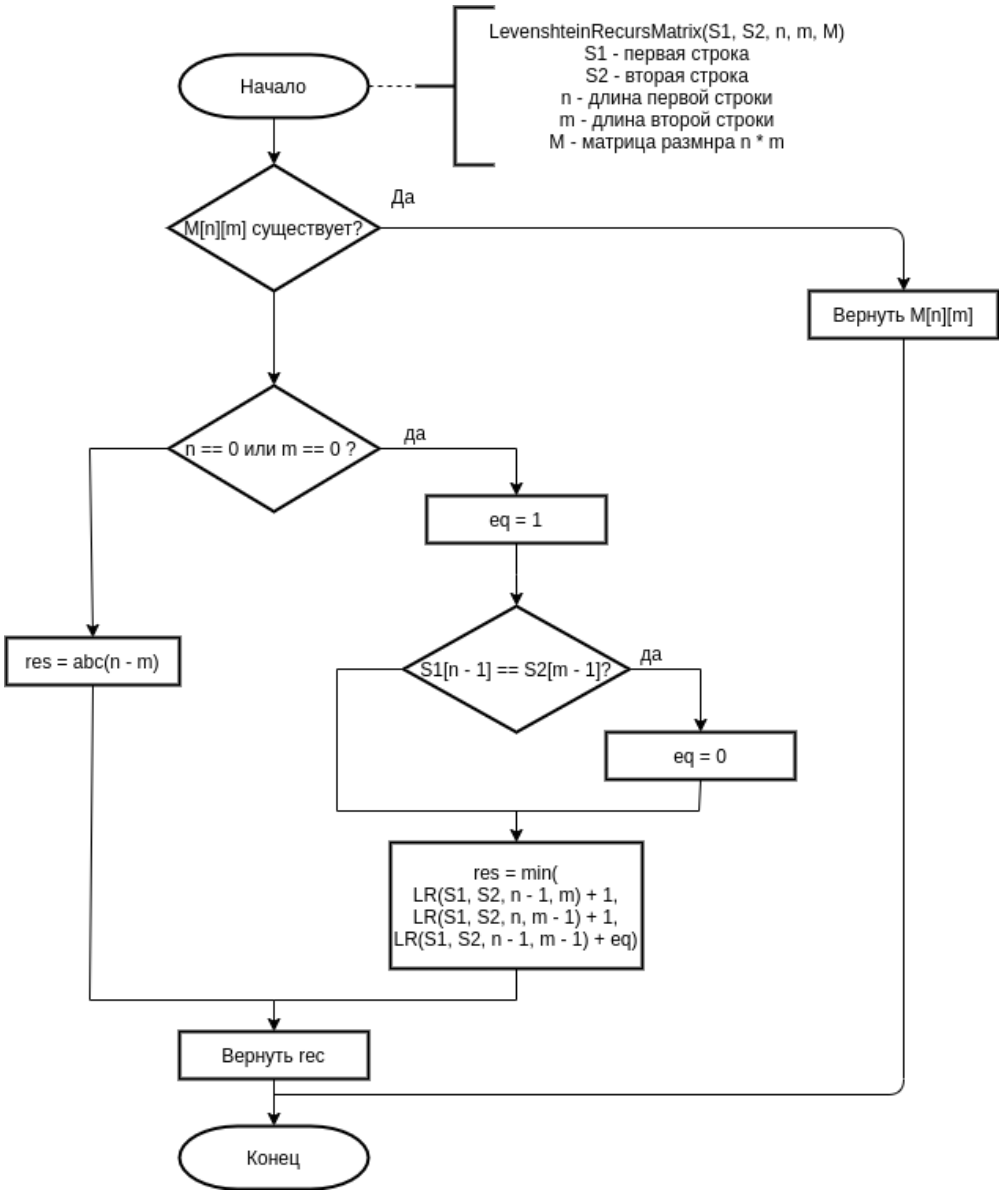

Рис. 2.3: Рекурсивный алгоритм Левенштейна, с заполнением матрицы

2.4. Расстояние Дамерау-Левенштейна, матричный

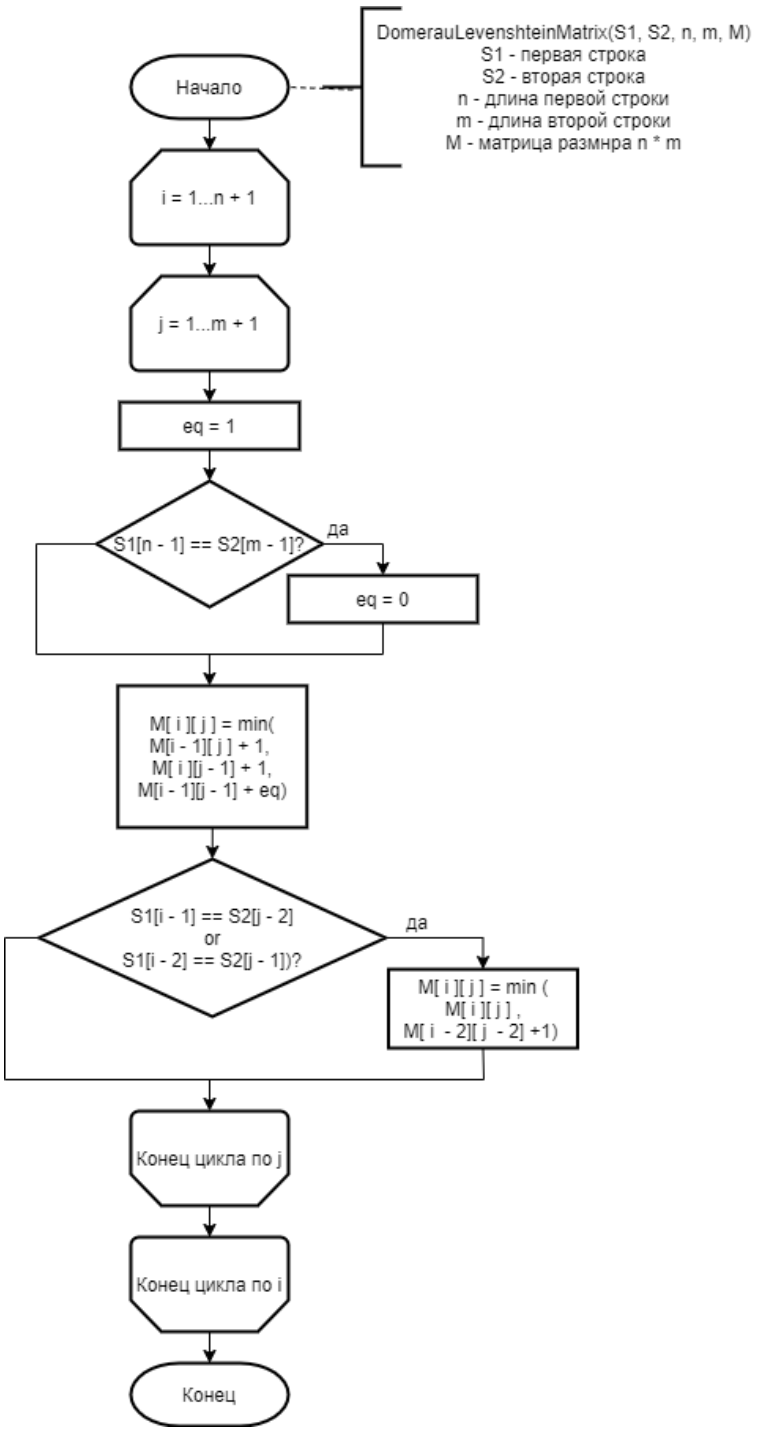

Рис. 2.4: Матричный алгоритм Дамерау-Левенштейна

3. Технологическая часть

3.1. Средства реализации

В качестве языка программирования был выбран c++. Данный язык знаком и предостовляет все необходимые ресурсы. В качестве среды разработки я использовала Visual Studio Code, т.к. считаю его достаточно удобным и легким. Visual Studio Code подходит не только для Windows, но и для Linux, это еще одна причина, по которой я выбрала VS code, т.к. у меня установлена ОС fedora 34.

3.2. Требования к ПО

К программе предъявляется требования:

- на вход подаются две строки;
- на выходе - искомое расстояние для четырех методов.

3.3. Сведения о модулях программы

Данная программа разбита на модули:

- main.cpp - Файл, содержащий точку входа в программу. В нем происходит общение с пользователем и вызов алгоритмов;
- levenshtain.cpp - Файл содержит непосредственно сами алгоритмы;

3.4. Листинг кода

Листинг 3.1: расстояние Левенштейна, рекурсивный метод

```
int Levenshtein::recursive()
{
        int i = n;
        int j = m;
        return getDistance(i, j);
}

int Levenshtein::getDistance(int i, int j)
{
        if (!i)
```

```
            return j;
            if (!j)
            return i;
            flag = 1;
            if (fString[i - 1] == sString[j - 1])
            flag = 0;
            return min(min(getDistance(i, j - 1) + 1,
            getDistance(i - 1, j) + 1),
            getDistance(i - 1, j - 1) + flag);
```

Листинг 3.2: расстояние Левенштейна, матричный метод

```
1  int Levenshtein::iterativeMatrix()
2  {
3    resetMatrix();
4    for (int i = 1; i < n + 1; i++)
5    {
6          for (int j = 1; j < m + 1; j++)
7          {
8            if (fString[i - 1] == sString[j - 1])
9                  flag = 0;
10           else
11                 flag = 1;
12           matrix[i][j] = min(matrix[i - 1][j] + 1,
13            min(matrix[i][j - 1] + 1,
14              matrix[i - 1][j - 1] + flag));
15         }
16   }
17   outputMatrix();
18   return matrix[n][m];
19 }
```

Листинг 3.3: расстояние Левенштейна, рекурсивный метод с заполнением матрицы

```
1  int Levenshtein::recursiveMatrix()
2  {
3          resetMatrix();
4          int i = n;
```

```
5           int j = m;
6           getDistanceRec(i, j);
7
8           outputMatrix();
9           return matrix[n][m];
10  }
11
12  int Levenshtein::getDistanceRec(int i, int j)
13  {
14          if (matrix[i][j] != -1)
15          return matrix[i][j];
16          if (!i)
17          {
18                  matrix[i][j] = j;
19                  return matrix[i][j];
20          }
21          if (!j)
22          {
23                  matrix[i][j] = i;
24                  return matrix[i][j];
25          }
26          flag = 1;
27          if (fString[i - 1] == sString[j - 1])
28          flag = 0;
29          matrix[i][j] = min(min(getDistance(i, j - 1) + 1,
30                  getDistance(i - 1, j) + 1),
31                          getDistance(i - 1, j - 1) + flag);
32          return matrix[i][j];
33  }
```

Листинг 3.4: расстояние Дамерау-Левенштейна, матричный метод

```
1   int Levenshtein::damerauLevenshtein()
2   {
3           resetMatrix();
4           for (int i = 1; i < n + 1; i++)
```

```
int Levenshtein::recursiveMatrix()
{
        resetMatrix();
        int i = n;
        int j = m;
        getDistanceRec(i, j);

        outputMatrix();
        return matrix[n][m];
}

int Levenshtein::getDistanceRec(int i, int j)
{
        if (matrix[i][j] != -1)
                return matrix[i][j];
        if (!i)
        {
                matrix[i][j] = j;
                return matrix[i][j];
        }
        if (!j)
        {
                matrix[i][j] = i;
                return matrix[i][j];
        }
        flag = 1;
        if (fString[i - 1] == sString[j - 1])
                flag = 0;
                matrix[i][j] = min(min(getDistance(i, j - 1) + 1,
                getDistance(i - 1, j) + 1),
                getDistance(i - 1, j - 1) + flag);
                        return matrix[i][j];
}
```

Рис. 3.1: Расстояние Левенштейна, рекурсивный метод с заполнением матрицы

```
5           {
6             for (int j = 1; j < m + 1; j++)
7             {
8                     if (fString[i - 1] == sString[j - 1])
9                             flag = 0;
10                    else
11                            flag = 1;
12                    matrix[i][j] = min(matrix[i - 1][j] + 1,
13                      min(matrix[i][j - 1] + 1,
14                              matrix[i - 1][j - 1] + flag));
15                    if ((i > 1) && (j > 1)  &&
16                      ((fString[i - 1] == sString[j - 2]) ||
```

```
17                                  (fString[i - 2] == sString[j - 1])))
18                          matrix[i][j] = min(matrix[i][j],
19                                  matrix[i - 2][j - 2] + 1);
20                      }
21                  }
22                  outputMatrix();
23                  return matrix[n][m];
24          }
```

3.5. Тестирование

В данном разделе будет приведена таблица 3.1, в которой четко отражено тестирование программы. Первый и второй столбец отвечают за введенные пользователем слова.

Таблица 3.1: Таблица тестов

Слово 1	Слово 2	Ожидаемый вывод	Вывод программы
сито	столб	3	3
exponential	polynomial	6	6
Vera	Vera	0	0
Vera	vera	1	1
ma	am	2 1	2 1
		0	0
abc	cab	2	2

Все тесты пройдены успешно.

3.6. Вывод

В данном разделе были рассмотрены листинги кода, обоснован выбор использованного в данной работе языка программирования и среды разработки, а также была произведена проверка корректной работы программы, благодаря таблице 3.1. Сравнив представленные листинги, можно сказать, что написание рекуррентных подпрограмм проще, чем матричных.

4. Исследовательская часть

В данном разделе сравним работу каждого алгоритма.

4.1. Временные характеристики

Сравним матричный алгоритм Левенштейна и Дамерау-Левенштейна. Для сравнения возьмем строки длинной [10, 20, 30, 50, 100, 200]. Воспользуемся усреднением массогово эксперимента. Для этого сложим результат работы n экспериментов(n >= 10). После чего поделим на n. Возьмем n = 500. Результат можно увидеть на рисунке . При короткой длинне строк разница почти не заметна, но при увелечении длины Дамерау-Левенштен уступает Левенштейну в силу дополнительной операции в Дамерау-Левенштейне.

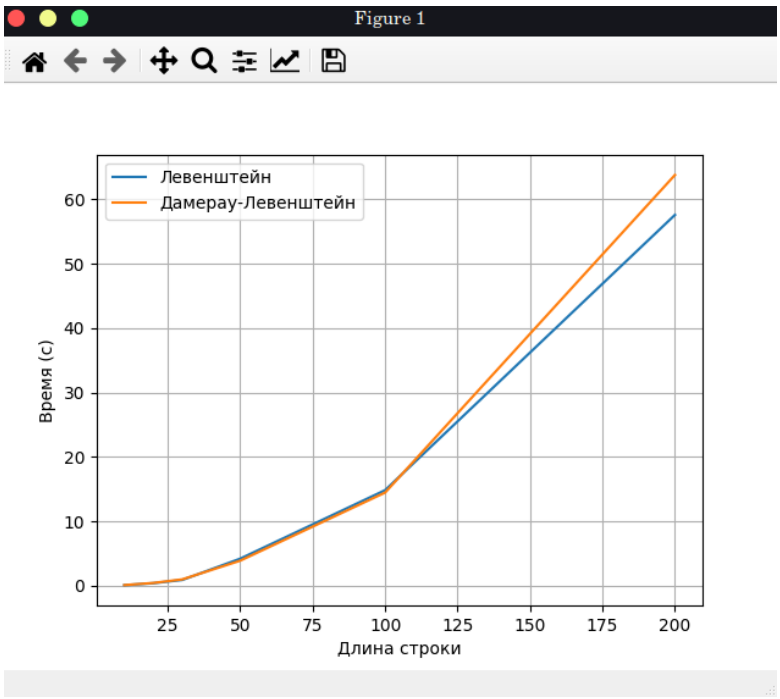

Рис. 4.1: Сравнение времени работы алгоритма поиска расстояния Левенштейна и Дамерау-Левенштейна

Далее проведем сравнительный анализ рекурсеонного и матричного Левенштейна. Вольмем строки длинной [2, 3, 4, 5, 6, 7], при n = 50. Результат на рисунке .Рост времени выполнения рекурсии обусловлен повторными вызовами с однотипнами параметрами.

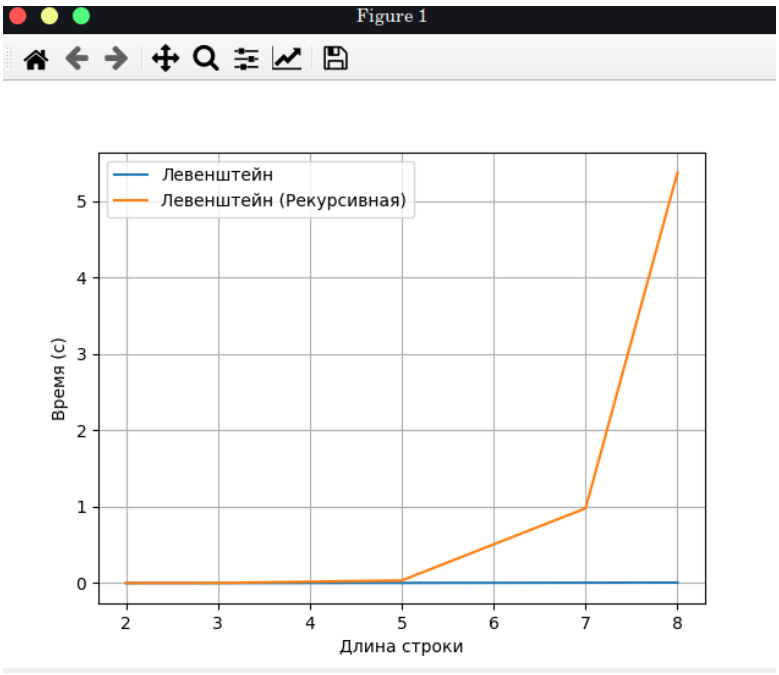

Рис. 4.2: Сравнение времени работы рекурсивной и матричной реализаций алгоритма Левенштейна.

4.2. Характеристики по памяти

На рисунке представлено дерево вызовов рекурсивного алгоритма Левенштейна. Видно, что на третьем уровне встречаются повторные вызовы. Чем больше будет уровень, тем чаще будут вызываться функции с однотипными аргументами, что может привести к превышению максимальной глубины рекурсии. При строках длиной 2 подпрограмма вызовется 18 раз. Каждый вызов задействует 32 мегабайт (замеры проведены с помощью библиотеки memory_profiler). В итоге нам потребуется 576 мегабайт для рекурсивных вызовов, в то время, когда в матричном алгоритме используется 42 мегабайта.

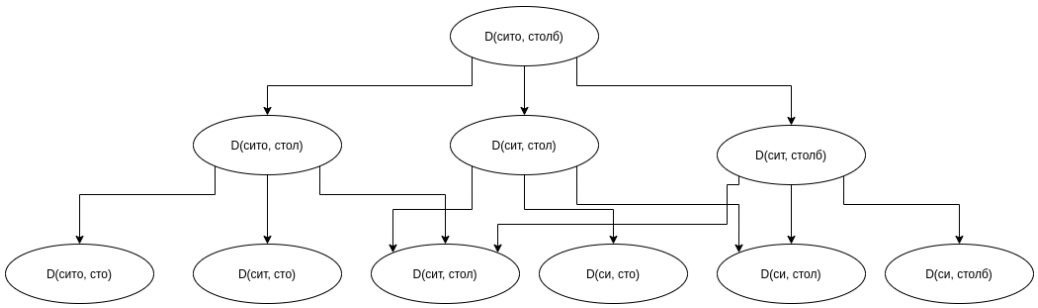

Рис. 4.3: Сравнение времени работы рекурсивной и матричной реализаций алгоритма Левенштейна.

4.3. Сравнительный анализ алгоритмов

Приведенные характеристики показывают нам, что рекурсивная реализация алгоритма очень сильно проигрывает по времени и по памяти.
Во время печати очень часто возникают ошибки связанные с транспозицией букв, поэтому алгоритм поиска расстояния Дамерау-Левенштейна предпочтительнее, не смотря на то, что он проигрывает по времени алгоритму Левенштейна.
По аналогии с первым абзацем можно сделать вывод о том, что рекуррентный алгоритм поиска расстояния Дамерау-Левенштейна будет более затратный, как по памяти, так и по времени по сравнению с матричной реализацией алгоритма поиска расстояния Дамерау-Левенштейна.

4.4. Вывод

В данном разделе было произведено сравнение количества затраченного времени и памяти вышеизложенных алгоритмов. Самым быстрым оказался матричный алгоритм нахождения расстояния Левенштейна.

Заключение

Алгоритмы поиска расстояний Левенштейна и ДамерауЛевенштейна являются самыми популярными алгоритмами, которые помогают найти редакторское расстояние.
В этой лабораторной работе мы познакомились с алгоритмами поиска расстояний Левенштейна (Формула **??**) и Дамерау-Левенштейна (Формула **??**). Построили схемы (Рисунок **??**, Рисунок **??**), соответствующие данным алгоритмам, также разобрали рекуррентные реализации (Рисунок **??**). Написали полностью готовый и протестированный (Таблица 3.1) программный продукт, который считает дистанцию 4 способами.

В рамках выполнения работы решены следующие задачи.

1. Изучены алгоритмы поиска расстояний Дамерау-Левенштейна и Левенштейна.

2. Реализованы изученные алгоритмы, а также матричную и рекурсивную реализации алгоритма.

3. Проиллюстрированы алгоритмы схемами.

4. Проведено сравнение временных характеристик, а также затраченной памяти.

Список литературы

1. Дж. Макконнел. Анализ алгоритмов. Активный обучающий подход. – М.: Техносфера, 2017. – 267с.

2. Карахтанов, Д. С. Программная реализация алгоритма Левенштейна для устранения опечаток в записях баз данных / Д. С. Карахтанов. — Текст : непосредственный // Молодой ученый. — 2010. — № 8 (19). — Т. 1. — С. 158-162. — URL: https://moluch.ru/archive/19/1966/ (дата обращения: 24.10.2021).

3. В. И. Левенштейн. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965. 163.4:845-848.

4. Основы программирования на языках Си и C++ для начинающих[Электронный ресурс]. Режим доступа: http://cppstudio.com/ (дата обращения 10.10.2021)

5. LINUX.ORG.RU - Русскоязычная информация о ОС Linux[Электронный ресурс] Режим доступа://www.linux.org.ru/(дата обращения 25.10.2021)