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»

Отчёт
по лабораторной работе № 1

Название: Расстояния Левенштейна и Дамерау-Левенштейна

Дисциплина: Анализ алгоритмов

Студент ИУ7-55Б _____ Д.О. Склифасовский
(Группа) (Подпись, дата) (И.О. Фамилия)

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

Москва, 2020

Оглавление

Введение **2**

1 Аналитическая часть **3**
1.1 Алгоритм Левенштейна 3
1.2 Алгоритм Дамерау-Левенштейна 4

2 Конструкторская часть **5**
2.1 Разработка алгоритмов 5
2.2 Сравнительный анализ памяти 9
2.2.1 Матричная реализация 9
2.2.2 Рекурсивная реализация 9
2.2.3 Вывод 9

3 Технологическая часть **10**
3.1 Общие требования 10
3.2 Средства реализации 11
3.3 Реализация алгоритмов 11

4 Экспериментальная часть **16**
4.1 Примеры работы программы 16
4.2 Результаты тестирования 18
4.2.1 Результаты работы программы 18
4.2.2 Сравнительный анализ времени работы алгоритмов 18

Заключение **20**

Литература **21**

Введение

Редакионное расстояние или **расстояние Левенштейна** - это минимальное количество редакторских операций, которые необходимы для преобразования одной строки в другую.

Расстояние Левенштейна и его обобщения активно применяются:

1) для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);
2) для сравнения текстовых файлов утилитой diff и ей подобными. Здесь роль «символов» играют строки, а роль «строк» — файлы;
3) в биоинформатике для сравнения генов, хромосом и белков.

Целью данной лабораторной работы является реализовать и сравнить по эффективности алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна.

Задачи данной лабораторной работы:

1) дать математическое описание расстояний;
2) разработать алгоритмы поиска расстояний;
3) реализовать алгоритмы поиска расстояний;
4) провести эксперименты по замеру времени работы реализации алгоритмов;
5) провести сравнительный анализ реализаций алгоритмов по затрачиваемому времени (и максимально затрачиваемой памяти);
6) дать теоретическую оценку максимально затрачиваемых по памяти реализациям алгоритмов.

1 Аналитическая часть

В данном разделе представлено математическое описание расстояний.

1.1 Алгоритм Левенштейна

Задача по нахождению расстояния Левенштейна заключается в поиске минимального количества операций вставки/удаления/замены для превращения одной строки в другую.

Редакционные операции:

1) вставки (I, insert) штраф 1;
2) удаления (D, delete) штраф 1;
3) замена (R, replace) штраф 1;
4) совпадение (M, match) штраф 0.

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по следующей формуле:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min(D(i,j-1)+1 & \\ D(i-1,j)+1, & \\ D(i-1,j-1)+m(S_1[i],S_2[j]) & \\ ), & j > 0, i > 0 \end{cases} \tag{1.1}$$

где $m(a,b)$ равна 1, если a не равно b и 0 иначе; $min(a,b,c)$ возвращает минимальный аргумент.

1.2 Алгоритм Дамерау-Левенштейна

В алгоритме Дамерау-Левенштейна добавлена еще одна операция перестановки символа (X, exchange), штраф 1.

$$D(i,j) = \begin{cases} 0, & i=0, j=0 \\ i, & j=0, i>0 \\ j, & i=0, j>0 \\ min(& \\ D(i,j-1)+1, & \\ D(i-1,j)+1, & j>0, i>0 \\ D(i-1,j-1)+m(S_1[i],S_2[j]) & \\ D(i-2,j-2)+1 & \text{if } i,j>1 \text{ and} \\ ), & a_i=b_{j-1}, a_{i-1}=b_j \end{cases} \tag{1.2}$$

2 Конструкторская часть

В данном разделе описаны схемы разработанных алгоритмов.

2.1 Разработка алгоритмов

На рисунке 1 изображена схема матричного алгоритма нахождения расстояний Левенштейна.

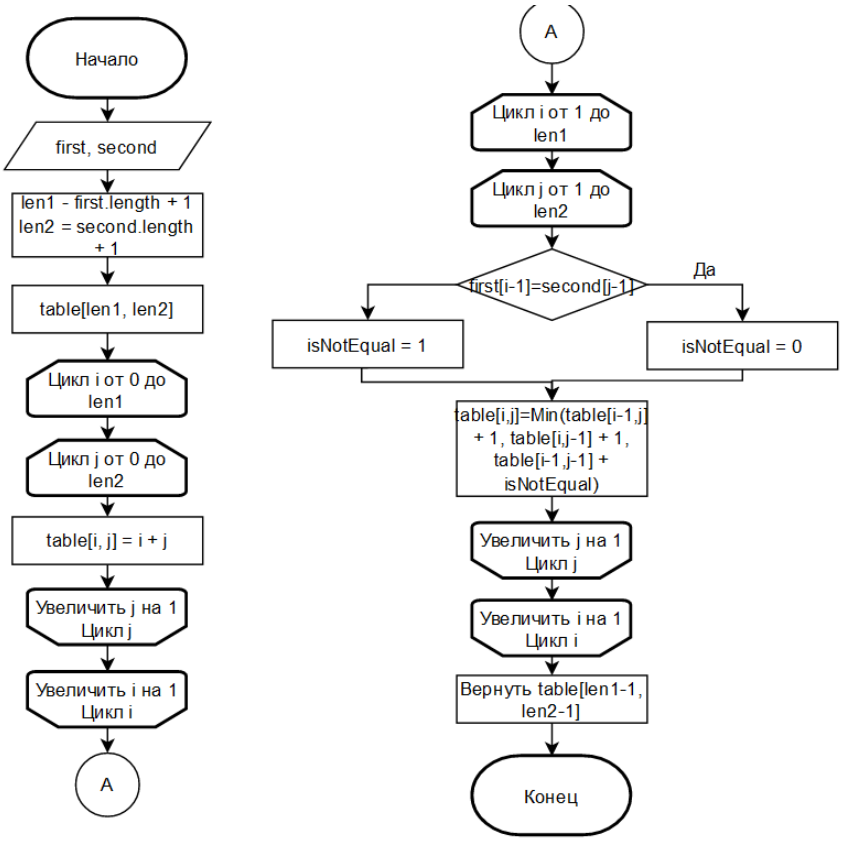

Рисунок 1. Схема матричного алгоритма Левенштейна

На рисунке 2 изображена схема рекурсивного алгоритма нахождения расстояний Левенштейна.

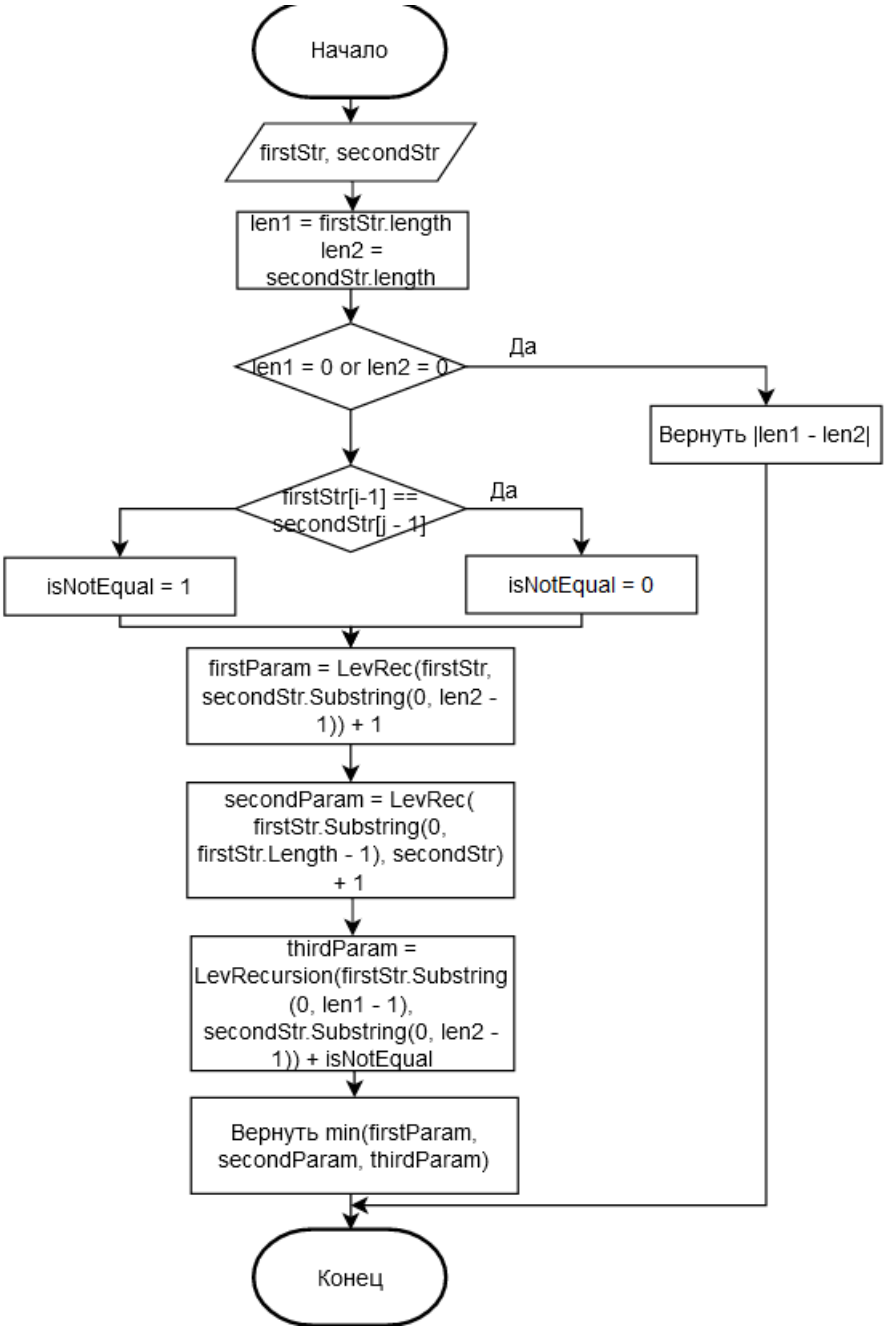

Рисунок 2. Схема рекурсивного алгоритма Левенштейна

На рисунке 3 изображена схема рекурсивного алгоритма с использованием матрицы нахождения расстояний Левенштейна.

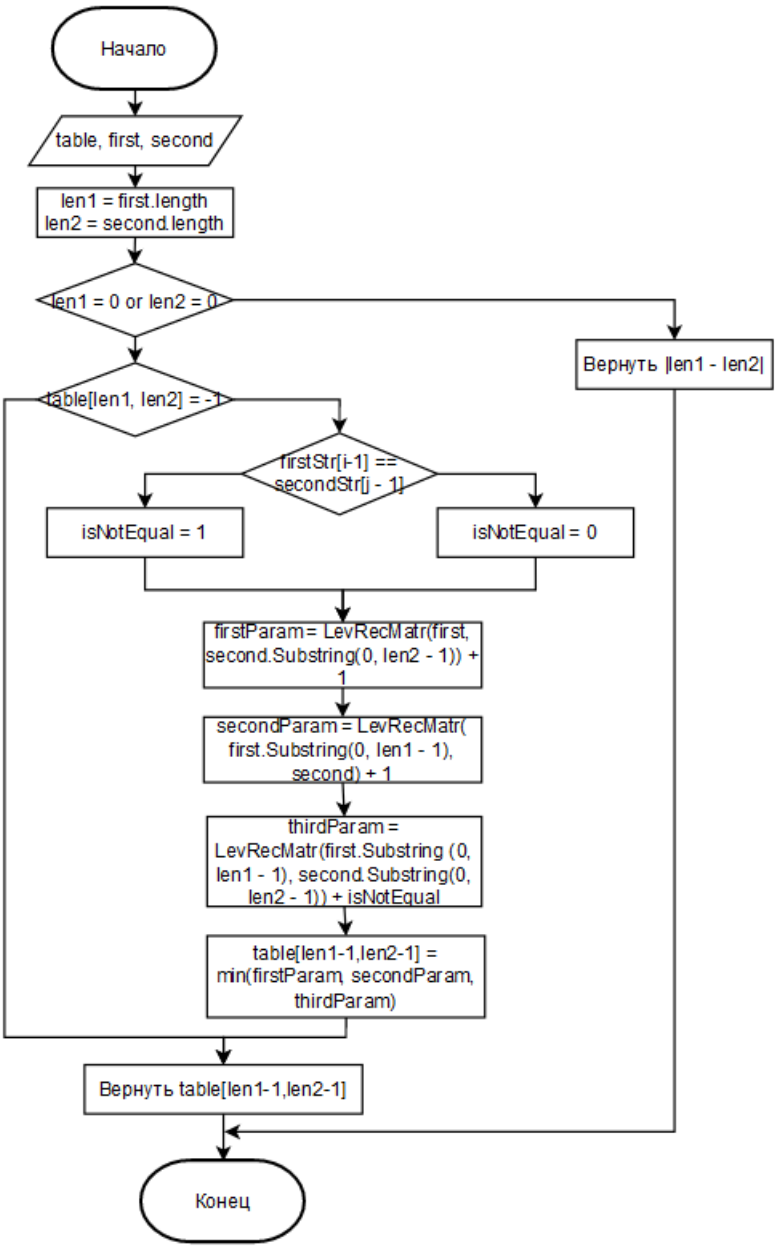

Рисунок 3. Схема рекурсивного алгоритма Левенштейна с использованием матрицы

На рисунке 4 изображена схема матричного алгоритма нахождения расстояний Дамерау-Левенштейна.

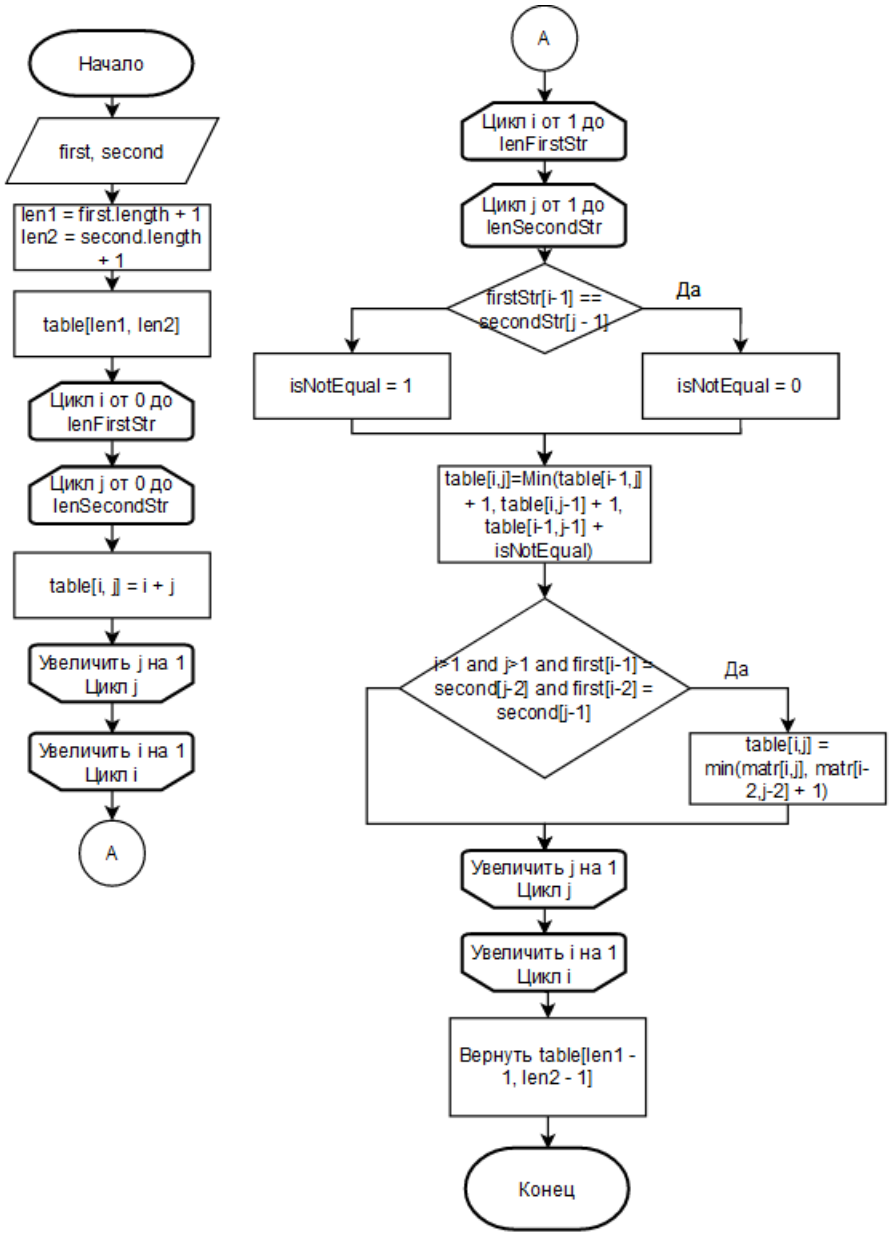

Рисунок 4. Схема матричного алгоритма Дамерау-Левенштейна

2.2 Сравнительный анализ памяти

2.2.1 Матричная реализация

В случае *матричной реализации* алгоритма нужно хранить:

1. матрица: $C_1 * (lenFirstStr + 1) + (lenFirstStr + 1) * C_1 * (lenSecondStr + 1)$;
2. переменная ($isNotEqual$): C_2;
3. счетчики: $2C_2$.

Также нужно передать параметры $C_2 * len$.
Таким образом, размер запрашиваемой памяти равен: $C_1 * (lenFirstStr + 1) + (lenFirstStr + 1) * C_1 * (lenSecondStr + 1) + C_2 + 2C_2 + C_2 * len$.

2.2.2 Рекурсивная реализация

В случае *рекурсивной реализации* алгоритма нужно хранить:

1) значения переменных: $4C_2$;
2) также нужно передать параметры $C_2 * len$.

Таким образом, размер запрашиваемой памяти равен: $C_2 + C_2 * len$

2.2.3 Вывод

Память матричного алгоритма растет пропорционально их произведения длин двух строк, а в случае рекурсивного алгоритма, она растет пропорционально сумме длин строк. Таким образом, на строках большей длины преимущество будет иметь рекурсивный алгоритм.

3 Технологическая часть

В данной части даны общие требования к программе, средства реализации и реализация алгоритмов

3.1 Общие требования

Требования к вводу:

1) выбирается какой алгоритм (из 4) будет использоваться:
   1.1) нерекурсивный алгоритм поиска расстояния Левенштейна с заполнением матрицы;
   1.2) рекурсивный алгоритм поиска расстояния Левенштейна без заполнения матрицы;
   1.3) рекурсивный алгоритм поиска расстояния Левенштейна с заполнением матрицы;
   1.4) нерекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с заполнением матрицы.
2) на вход подаются две строки;
3) на выходе программа выводит значение расстояния между ними;
4) в случае вызова матричного алгоритма программа выводит матрицу;
5) также необходимо предусмотреть выполнение замеров процессорного времени для каждого алгоритма.

Требования к программе:

1) две пустые строки являются корректным вводом;

2) uppercase и lowcase буквы считаются разными.

3.2 Средства реализации

В качестве языка программирования был выбран C#, так как он объектный, достаточно быстрый и в нем можно легко писать код, а требований по конкретнему языку не выдвигалось.[1]

Среда разработки - Visual Studio.[2]

Для замеров процессорного времени используется функция $Stopwatch$.[3]

3.3 Реализация алгоритмов

Листинг 3.1: Функция нахождения минимума из 3 переменных

```
1     private static int Min(int first, int second, int third)
2     {
3       int[] arr = { first, second, third };
4       return arr.Min();
5     }
```

Листинг 3.2: Функция нахождения расстояния Левенштейна используя матрицу

```
1     public static int LevTable(string firstStr, string secondStr)
2     {
3       int lenFirstStr = firstStr.Length + 1;
4       int lenSecondStr = secondStr.Length + 1;
5
6       int[,] table = new int[lenFirstStr, lenSecondStr];
7       for (int i = 0; i < lenFirstStr; i++)
8       for (int j = 0; j < lenSecondStr; j++) { table[i, j] = i + j; }
9
10      int isNotEqual = 1;
11      for (int i = 1; i < lenFirstStr; i++)
12      {
```

```
13          for (int j = 1; j < lenSecondStr; j++)
14          {
15              if (firstStr[i - 1] == secondStr[j - 1]) { isNotEqual = 0;
                    }
16              table[i, j] = Min(table[i - 1, j] + 1,
17              table[i, j - 1] + 1,
18              table[i - 1, j - 1] + isNotEqual);
19              isNotEqual = 1;
20          }
21      }
22      return table[lenFirstStr - 1, lenSecondStr - 1];
23  }
```

Листинг 3.3: Функция нахождения расстояния Левенштейна рекурсивно

```
1   public static int LevRecursion(string firstStr, string secondStr)
2   {
3       if (firstStr == "" || secondStr == "") { return Math.Abs(
            firstStr.Length - secondStr.Length); }
4       int isNotEqual = 1;
5       if (firstStr[firstStr.Length - 1] == secondStr[secondStr.Length
            - 1]) { isNotEqual = 0; }
6       return Min(LevRecursion(firstStr, secondStr.Substring(0,
            secondStr.Length - 1)) + 1,
7       LevRecursion(firstStr.Substring(0, firstStr.Length - 1),
            secondStr) + 1,
8       LevRecursion(firstStr.Substring(0, firstStr.Length - 1),
            secondStr.Substring(0, secondStr.Length - 1)) + isNotEqual);
9   }
```

Листинг 3.4: Функции нахождения расстояния Левенштейна рекурсивно с использованием матрицы

```
1   private static int GetMinValue(int[,] table, string first, string
        second, int lenFirst, int lenSecond)
2   {
3       int isNotEqual = (first[lenFirst - 1] == second[lenSecond - 1])
            ? 0 : 1;
4       int firstParam = FindLevDistance(table, first, second.Substring
            (0, lenSecond - 1)) + 1;
```

```
5         int secondParam = FindLevDistance(table, first.Substring(0,
              lenFirst - 1), second) + 1;
6         int thirdParam = FindLevDistance(table, first.Substring(0,
              lenFirst - 1), second.Substring(0, lenSecond - 1)) +
              isNotEqual;
7         return Min(firstParam, secondParam, thirdParam);
8       }
9
10      private static int FindLevDistance(int[,] table, string first,
            string second)
11      {
12        int lenFirst = first.Length;
13        int lenSecond = second.Length;
14
15        if (lenFirst == 0 || lenSecond == 0)
16        {
17          table[lenFirst, lenSecond] = Math.Abs(lenFirst - lenSecond);
18          return table[lenFirst, lenSecond];
19        }
20
21        if (table[lenFirst, lenSecond] == -1)
22        table[lenFirst, lenSecond] = GetMinValue(table, first, second,
              lenFirst, lenSecond);
23        return table[lenFirst, lenSecond];
24      }
25
26      public static int LevTableRec(string firstStr, string secondStr)
27      {
28        int lenFirstStr = firstStr.Length + 1;
29        int lenSecondStr = secondStr.Length + 1;
30        int[,] table = new int[lenFirstStr, lenSecondStr];
31        for (int i = 0; i < lenFirstStr; i++)
32        for (int j = 0; j < lenSecondStr; j++) { table[i, j] = -1; }
33
34        int res;
35        if (lenFirstStr - 1 == 0 || lenSecondStr - 1 == 0)
36        res = Math.Abs((lenFirstStr - 1) - (lenSecondStr - 1));
37        else
38        res = GetMinValue(table, firstStr, secondStr, lenFirstStr - 1,
```

```
            lenSecondStr - 1);
39
40          Console.WriteLine("RecTable");
41          for (int i = 0; i < lenFirstStr; i++)
42          {
43            for (int j = 0; j < lenSecondStr; j++) { Console.Write(String
                .Format("{0, 3}", table[i, j])); }
44            Console.WriteLine();
45          }
46
47          return res;
48        }
```

Листинг 3.5: Функция нахождения расстояния Дамерау-Левенштейна используя матрицу

```
1       public static int LevDamTable(string firstStr, string secondStr)
2       {
3         int lenFirstStr = firstStr.Length + 1;
4         int lenSecondStr = secondStr.Length + 1;
5
6         int[,] table = new int[lenFirstStr, lenSecondStr];
7         for (int i = 0; i < lenFirstStr; i++)
8         for (int j = 0; j < lenSecondStr; j++) { table[i, j] = i + j; }
9
10        int isNotEqual = 1;
11        for (int i = 1; i < lenFirstStr; i++)
12        {
13          for (int j = 1; j < lenSecondStr; j++)
14          {
15            if ((i > 1 && j > 1) && firstStr[i - 1] == secondStr[j - 2]
                  && firstStr[i - 2] == secondStr[j - 1])
16            table[i, j] = Math.Min(table[i, j], table[i - 2, j - 2] +
                1);
17            else
18            {
19              if (firstStr[i - 1] == secondStr[j - 1]) { isNotEqual =
                  0; }
20              table[i, j] = Min(table[i - 1, j] + 1,
21              table[i, j - 1] + 1,
```

```
22                table[i - 1, j - 1] + isNotEqual);
23            }
24            isNotEqual = 1;
25          }
26        }
27        return table[lenFirstStr - 1, lenSecondStr - 1];
28      }
```

4 Экспериментальная часть

В данном разделе описаны примеры работы программы, результаты работы программы и приведен сравнительный анализ времени работы каждого из алгоритмов.

4.1 Примеры работы программы

На рисунке 5 представлен пример вызова матричного алгоритма Левенштейна.

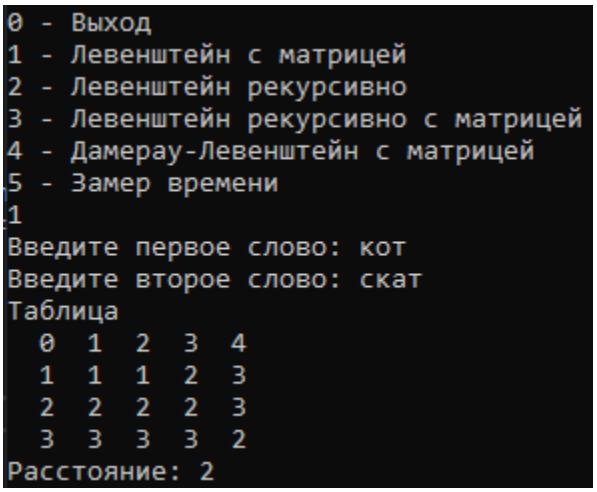

Рисунок 5. Результат работы матричного алгоритма Левенштейна

На рисунке 6 представлен пример вызова рекурсивного алгоритма Левенштейна.

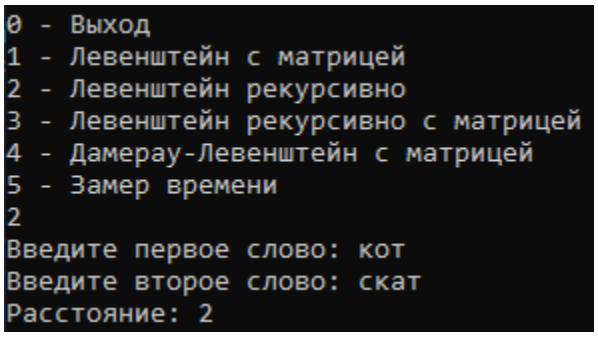

Рисунок 6. Результат работы рекурсивного алгоритма Левенштейна

На рисунке 7 представлен пример вызова рекурсивного алгоритма Левенштейна с использованием матрицы.

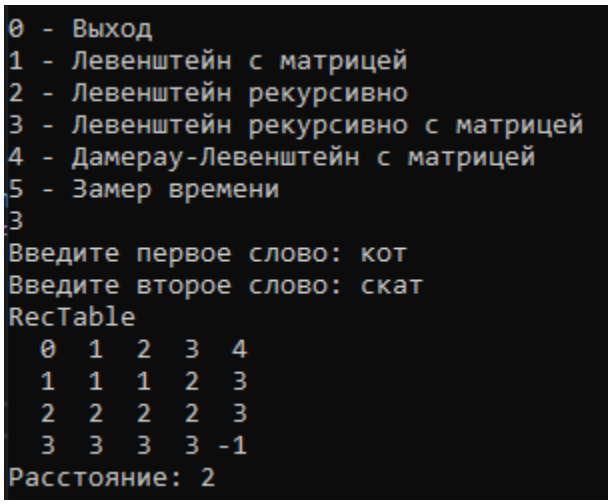

Рисунок 7. Результат работы рекурсивного алгоритма Левенштейна с использованием матрицы

На рисунке 8 представлен пример вызова матричного алгоритма Дамерау-Левенштейна.

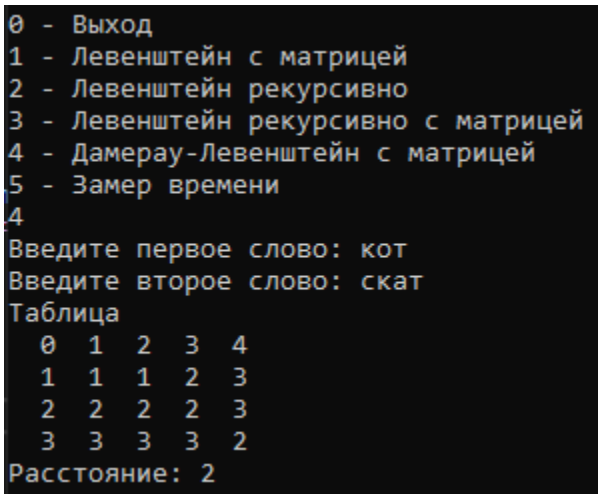

Рисунок 8. Результат работы матричного алгоритма Дамерау-Левенштейна

4.2 Результаты тестирования

4.2.1 Результаты работы программы

Результаты тестирования программы приведены в таблице 1.

Таблица 1. Результаты работы программы

Первое слово	Второе слово	Результат
kot	skat	2
kot	skot	1
-	-	0
smth	something	5
something	osemthing	3 | 2
something	something	0

4.2.2 Сравнительный анализ времени работы алгоритмов

Выполняется эксперимент. Берутся длина строк 2, 5, 10, 100, 200, 300, 500. Строки строятся рандомно. По рисунку 9 можно увидеть, что самым долгим алгоритмом

для строк большой длины является рекурсивный алгоритм без использования матрицы. Из-за того, что данный алгоритм выполняет много повторных операций, он намного дольше остальных. Самым быстрым является матричный алгоритм поиска расстояний Левенштейна. Длина строки влияет на работу данного алгоритма несущественно. Алгоритм Дамерау-Левенштейна немного медленнее, так как выполняется дополнительное сравнение в цикле. Рекурсивный алгоритм с использованием матрицы быстрее, чем обычный рекурсивный алгоритм, но медленнее матричных.

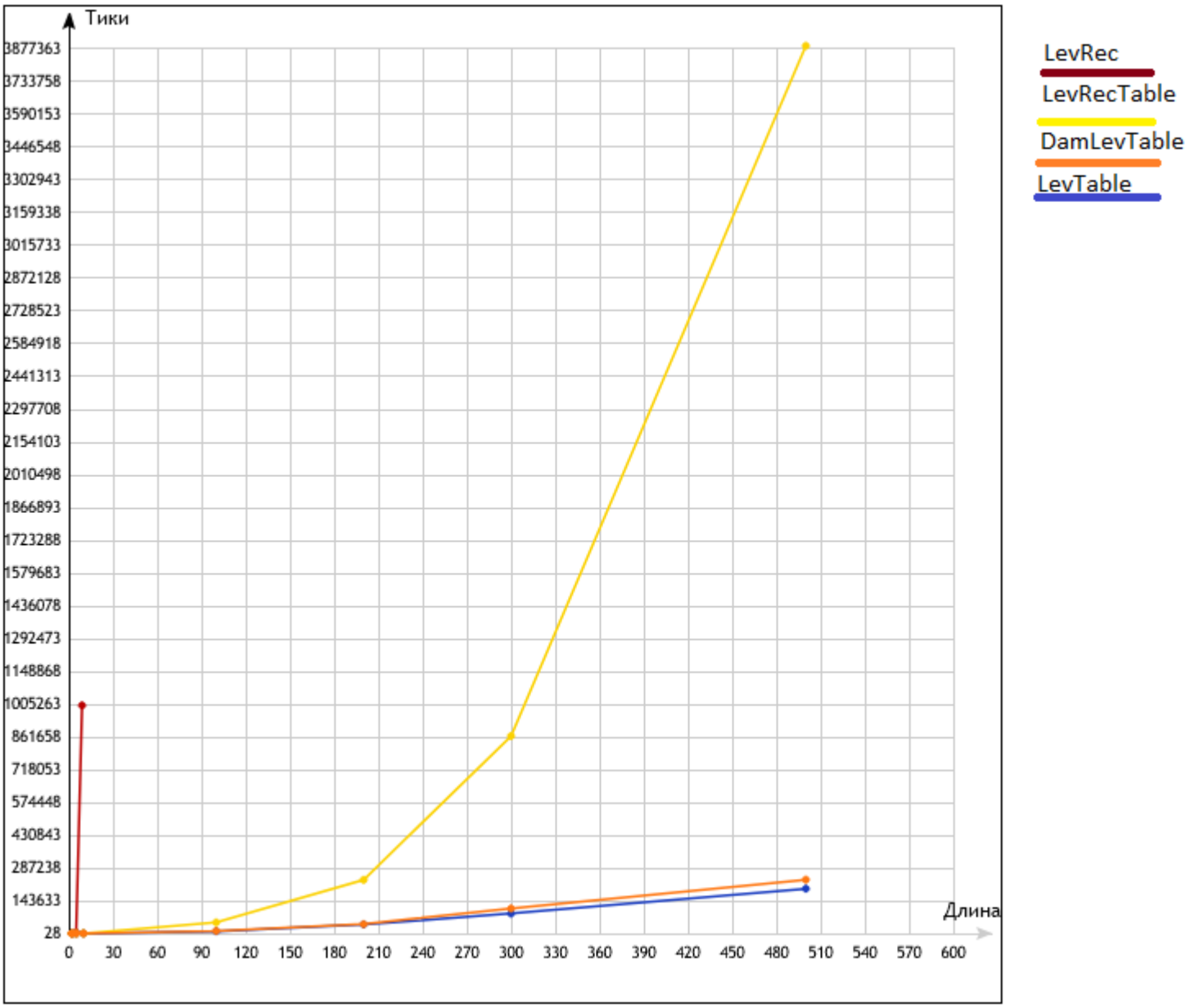

Рисунок 9. Результат замера времени работы алгоритмов

Заключение

В ходе лабораторной работы были разработаны и реализованы алгоритмы нахождения расстояния Левенштейна (матричный, рекурсивный, рекурсивный с использованием матрицы) и Дамерау-Левенштейна (матричный), а также был проведен анализ затрачиваемых ресурсов каждого из метода. В результате, после получения результата замера времени работы алгоритмов, было получено, что самым быстрым алгоритмом является матричный алгоритм нахождения расстояния Левенштейна, рекурсивный алгоритм оказался самым медленным, но рекурсивный алгоритм тратит меньше памяти, чем остальные.

Литература

1. Документация по C#. -URL: https://docs.microsoft.com/ru-ru/dotnet/csharp/ (дата обращения: 01.10.2020). -Текст: электронный.

2. Документация по семейству продуктов Visual Studio. -URL: https://docs.microsoft.com/ru-ru/visualstudio/?view=vs-2019 (дата обращения: 01.10.2020). -Текст: электронный.

3. Stopwatch Класс. -URL: https://goo.su/2e99 (дата обращения: 01.10.2020). -Текст: электронный.

4. Под капотом у Stopwatch. -URL: https://habr.com/ru/post/226279/ (дата обращения: 01.10.2020). Текст: электронный.

5. Вычисление редакционного расстояния. -URL: https://habr.com/ru/post/117063/ (дата обращения: 01.10.2020). Текст: электронный.