МГТУ им. Баумана

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

# Расстояние Левенштейна

Работу выполнил: Чернов Даниил, ИУ7-56Б

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

*Москва, 2019*

# Оглавление

# Введение

**Расстояние Левенштейна** - минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для:

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

Целью данной лабораторной работы является изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна.

Задачами данной лабораторной являются:

1. изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
2. применение метода динамического программирования для матричной реализации указанных алгоритмов;
3. получение практических навыков реализации указанных алгоритмов: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;
4. сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);

5. экспериментальное подтверждение различий во временно́й эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;

6. описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

# 1 Аналитическая часть

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

При нахождении расстояния Дамерау — Левенштейна добавляется операция транспозиции (перестановки соседних символов).

**Действия обозначаются так:**

1. D (англ. delete) — удалить,
2. I (англ. insert) — вставить,
3. R (replace) — заменить,
4. M(match) - совпадение.

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

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

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

Расстояние Дамерау-Левенштейна вычисляется по следующей рекурентной формуле:

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

### 1.0.1 Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который является модификаций первого, учитывающего возможность перестановки соседних символов.

# 2 | Конструкторская часть

**Требования к вводу:**

1. На вход подаются две строки
2. uppercase и lowercase буквы считаются разными

**Требования к программе:**

1. Две пустые строки - корректный ввод, программа не должна аварийно завершаться

## 2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы алгоритмов.

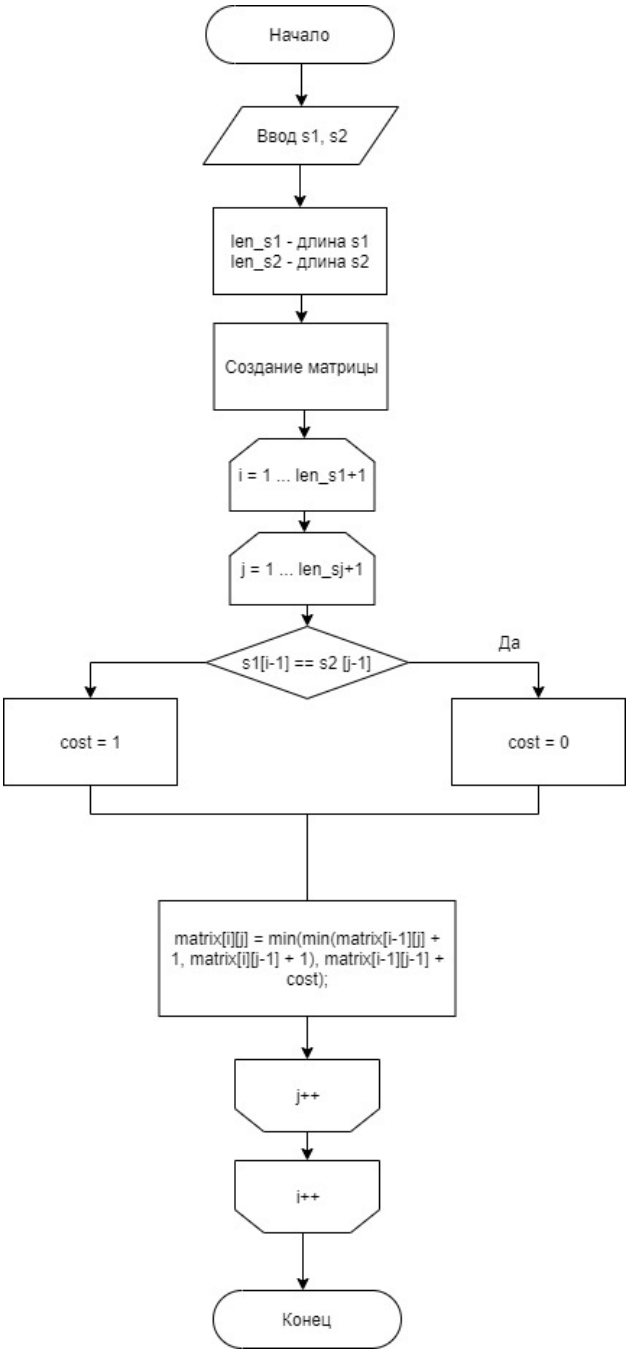



Рис. 2.1: Схема матричного алгоритма нахождения расстояния Левенштейна

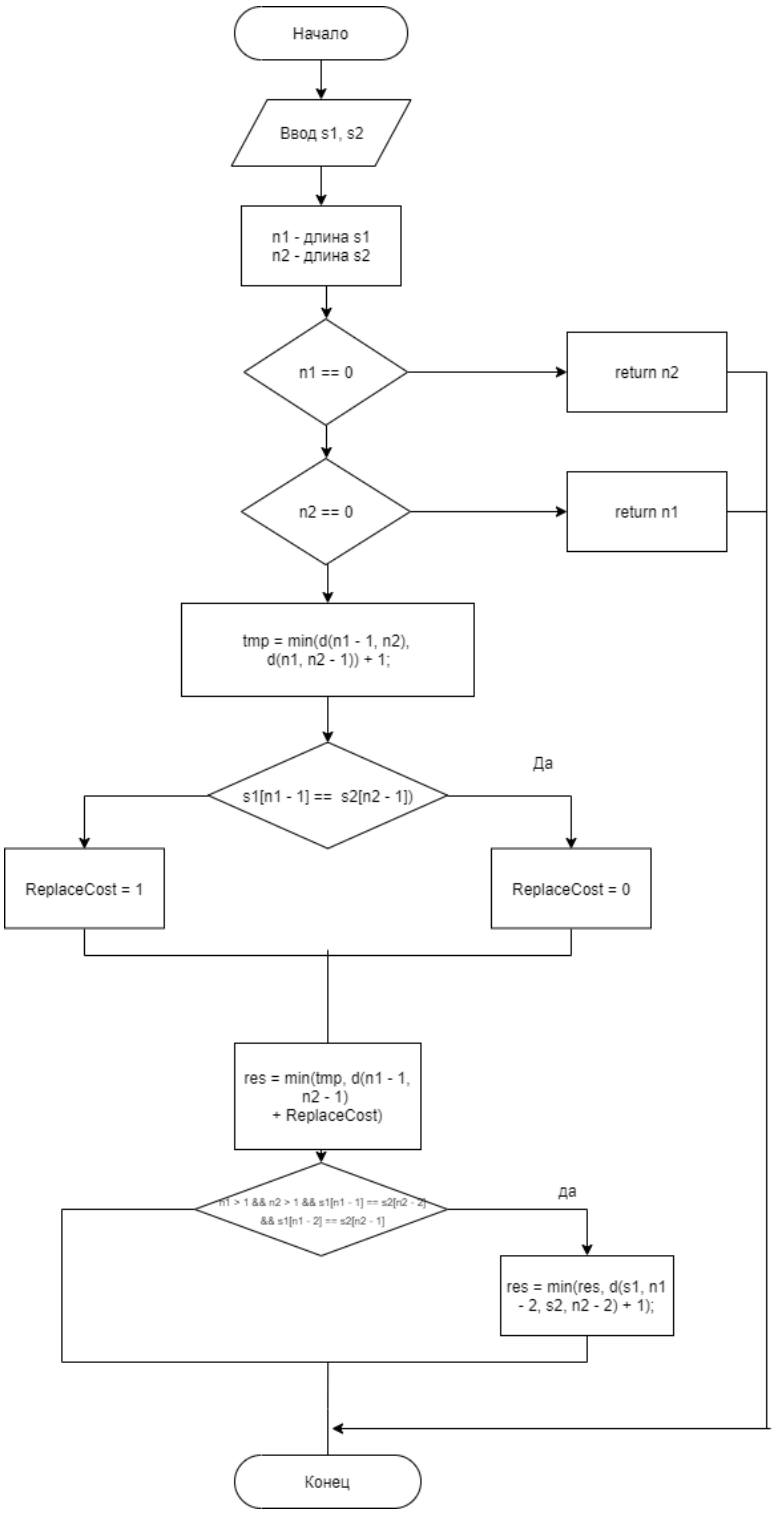





Рис. 2.2: Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

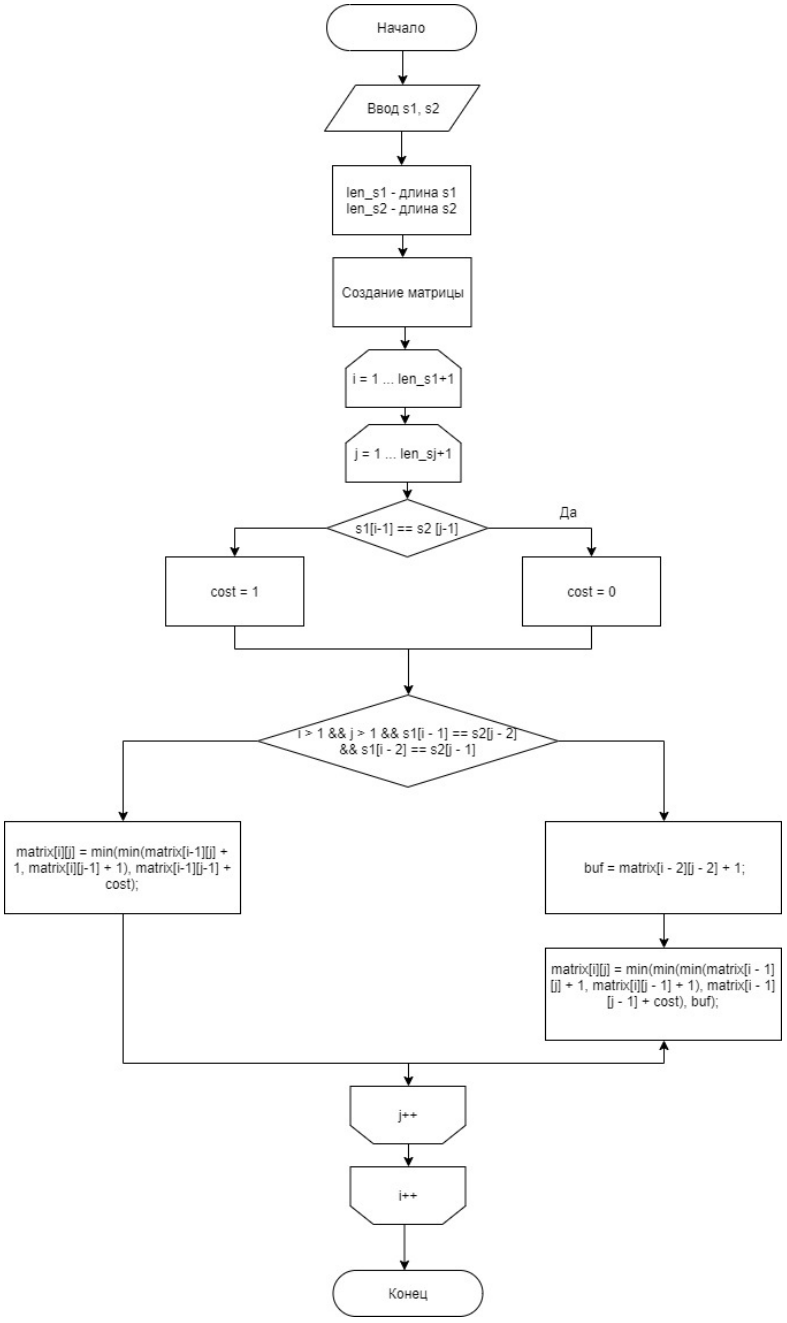



Рис. 2.3: Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

# 3 | Технологическая часть

## 3.1 Выбор ЯП

Для реализации программ я выбрала язык программирования C#, так имею большой опыт работы с ним. Среда разработки - Visual Studio. Программа была разделена на несколько классов:

1. ILowensteinDistance - абстрактный родительский класс для обобщения последующих реализаций алгоритма;
2. LowensteinDistance - класс, реализовующий поиск расстояния Левенштейна;
3. LowensteinDamerauDistance - класс, реализовующий поиск расстояния Дамерау-Левенштейна;
4. LowensteinDamerauRecursiveDistance - класс, реализовующий поиск рекурсивного расстояния Дамерау-Левенштейна;
5. MainWindow - класс пользовательского интерфейса, написанный с использованием WPF;
6. Tester - класс с тестированием методов, подсчитывающих расстояние Левенштейна.

Ниже представлены листинги кода для перечисленных реализаций классов.

## 3.2 Реализация алгоритма

Листинг 3.1: Абстрактный класс для последующих реализаций алгоритмов Левенштейна

```
1  namespace LevensteinDistanceAnalysis.Core
2  {
3      abstract class ILowensteinDistance
4      {
5          protected int distance;
6          protected string firstWord;
7          protected string secondWord;
8          public int GetDistance(string firstWord, string
               secondWord)
9          {
10             this.firstWord = firstWord;
11             this.secondWord = secondWord;
12
13             if (firstWord.Length == 0 && secondWord.Length
                   == 0)
14                 distance = 0;
15             else if (IfJustOneWordIsEmpty())
16                 distance = 1;
17             else
18                 distance = CountDistance(firstWord,
                       secondWord);
19
20             return distance;
21         }
22         protected bool IfJustOneWordIsEmpty()
23         {
24             if (firstWord.Length == 0 && secondWord.Length
                   != 0)
25                 return true;
26             if (firstWord.Length != 0 && secondWord.Length
                   == 0)
27                 return true;
28             return false;
29         }
30         abstract protected int CountDistance(string
               firstWord, string secondWord);
31         override public string ToString()
32         {
```

```
33              return $"The distance between \"{firstWord}\"
                    and \"{secondWord}\" is:  {distance}";
34          }
35      }
36  }
```

Листинг 3.2: Класс для нахождения расстояния Левенштейна

```
1   using System;
2
3   namespace LevensteinDistanceAnalysis.Core
4   {
5       class LowensteinDistance : ILowensteinDistance
6       {
7           protected int[,] matrix;
8           protected override int CountDistance(string
                firstWord, string secondWord)
9           {
10              matrix = new int[firstWord.Length + 1,
                    secondWord.Length + 1];
11
12              FillMatrixFirstColumnWithZeros();
13              FillMatrixFirstRowWithZeros();
14              FindingDistanceInMatrix();
15              distance = GetResultFromMatrix();
16
17              return distance;
18          }
19          protected void FindingDistanceInMatrix()
20          {
21              for (int i = 1; i <= firstWord.Length; i++)
22              {
23                  for (int j = 1; j <= secondWord.Length; j
                        ++)
24                  {
25                      GetNextElementInMatrix(i, j);
26                  }
27              }
28          }
29          protected virtual void GetNextElementInMatrix(int i
                , int j)
```

```
30          {
31              int cost = (firstWord[i - 1] == secondWord[j -
                    1]) ? 0 : 1;
32
33              matrix[i, j] = Math.Min(Math.Min(matrix[i - 1,
                    j] + 1,
34                                               matrix[i, j - 1] + 1),
35                                               matrix[i - 1, j - 1] +
                                                     cost);
36          }
37          protected void FillMatrixFirstColumnWithZeros()
38          {
39              for (int i = 0; i <= firstWord.Length; i++)
40                  matrix[i, 0] = i;
41          }
42          protected void FillMatrixFirstRowWithZeros()
43          {
44              for (int j = 0; j <= secondWord.Length; j++)
45                  matrix[0, j] = j;
46          }
47          protected int GetResultFromMatrix()
48          {
49              return matrix[firstWord.Length, secondWord.
                    Length];
50          }
51          public string MatrixToString()
52          {
53              string distanceMatrix = "";
54              for (int i = 0; i < matrix.GetLength(0); i++)
55              {
56                  for (int j = 0; j < matrix.GetLength(1); j
                        ++)
57                      distanceMatrix += matrix[i, j] + "\\t";
58                  distanceMatrix += "\\n";
59              }
60              return distanceMatrix;
61          }
62      }
63  }
```

Листинг 3.3: Класс нахождения расстояния Дамерау-Левенштейна рекурсивно

```
1  using System;
2
3  namespace LevensteinDistanceAnalysis.Core
4  {
5      class LowensteinDamerauRecursiveDistance :
           ILowensteinDistance
6      {
7          protected override int CountDistance(string
               firstWord, string secondWord)
8          {
9              int distance = RecursiveCountDistance(firstWord
                   , firstWord.Length, secondWord, secondWord.
                   Length);
10             return distance;
11         }
12         private int RecursiveCountDistance(string firstWord
               , int firstWordLen, string secondWord, int
               secondWordLen)
13         {
14             int cost;
15             if (firstWordLen == 0) return secondWordLen;
16             if (secondWordLen == 0) return firstWordLen;
17
18             if (firstWord[firstWordLen - 1] == secondWord[
                   secondWordLen - 1])
19                 cost = 0;
20             else
21                 cost = 1;
22
23             return Math.Min(Math.Min(RecursiveCountDistance
                   (firstWord, firstWordLen - 1, secondWord,
                   secondWordLen) + 1,
24                                 RecursiveCountDistance(
                                       firstWord, firstWordLen,
                                       secondWord, secondWordLen -
                                       1) + 1),
25                                 RecursiveCountDistance(
                                       firstWord, firstWordLen - 1,
```

```
                                        secondWord, secondWordLen -
                                        1) + cost);
26          }
27      }
28  }
```

Листинг 3.4: Класс нахождения расстояния Дамерау-Левенштейна матрично

```
1   using System;
2
3   namespace LevensteinDistanceAnalysis.Core
4   {
5       class LowensteinDamerauDistance : LowensteinDistance
6       {
7           protected override void GetNextElementInMatrix(int
                i, int j)
8           {
9               int cost = (firstWord[i - 1] == secondWord[j -
                    1]) ? 0 : 1;
10
11              matrix[i, j] = Math.Min(Math.Min(matrix[i - 1,
                    j] + 1,
12                                               matrix[i, j - 1] + 1),
13                                               matrix[i - 1, j - 1] +
                                                     cost);
14              if (IfAdjacentLettersEqual(i, j))
15                  matrix[i, j] = Math.Min(matrix[i, j],
                        matrix[i - 2, j - 2] + cost);
16          }
17          protected bool IfAdjacentLettersEqual(int i, int j)
18          {
19              if (i > 1 && j > 1 && firstWord[i - 1] ==
                    secondWord[j - 2] && firstWord[i - 2] ==
                    secondWord[j - 1])
20                  return true;
21              return false;
22          }
23      }
24  }
```

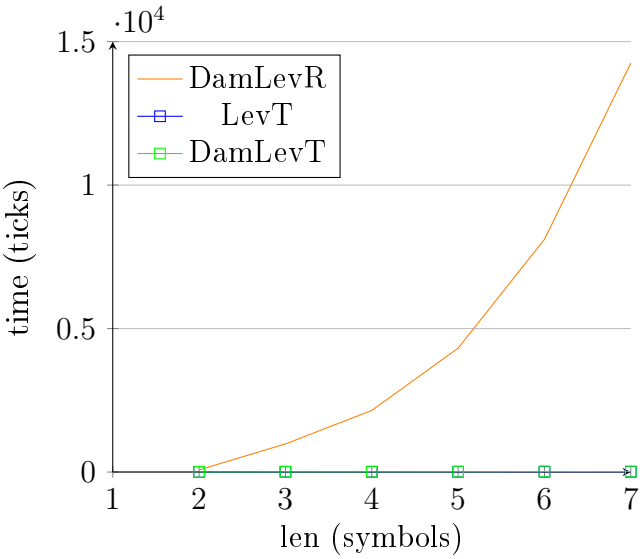
# 4 Исследовательская часть

## 4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведен замер времени работы каждого из алгоритмов с помощью класса StopWatch из библиотеки System.Diagnostics. Где единицей времени является ElapsedTick - версия процессорного Тика на платформе .NET. Результаты можно увидеть в следующей таблице.

Таблица 4.1: Время работы алгоритмов (в тиках)

| len | Lev(M) | DamLev(M) | DamLev(R) |
|---|---|---|---|
| 2 | 5 | 6 | 74 |
| 3 | 10 | 11 | 980 |
| 4 | 10 | 12 | 2147 |
| 5 | 11 | 14 | 4317 |
| 6 | 12 | 18 | 8111 |
| 7 | 12 | 17 | 14255 |




Наиболее эффективными по времени при маленькой длине слова являются рекурсивные реализации алгоритмов, но как только увеличивается длина слова, их эффективность резко снижается, что обусловлено большим количеством повторных рассчетов. Время работы алгоритма, использующего матрицу, намного меньше благодаря тому, что в нем требуется только (m + 1)*(n + 1) операций заполнения ячейки матрицы. Также установлено, что алгоритм ДамерауЛевенштейна работает немного дольше алгоритма Левенштейна, т.к. в нем добавлены дополнительные проверки, однако алгоритмы сравнимы по временной эффективности.

## 4.2 Сравнительный анализ на основе замеров потребляемой памяти алгоритмов

Для проведения анализа замерим потребляемую память у разных реализаций алгоритма. Все измерения представлены в байтах и представлены в следующих таблицах.

Таким образом, рекурсивный алгоритм занимает примерно одинаковое кол-во памяти при маленькой длине строк, и значительно выигрывает при строках большего размера.

Таблица 4.2: Потребляемая память структурами данных в алгоритме нахождения расстояния Левенштейна

| Структура данных | Длина 4 символа | Длина 1000 символов |
|---|---|---|
| Матрица | 480 | 8064096 |
| Две вспомогательные переменные (int) | 56 | 56 |
| Два счетчика (int) | 56 | 56 |
| Передача параметров | 106 | 2098 |
| Сумма данных | 698 | 8066306 |

Таблица 4.3: Потребляемая память структурами данных в алгоритме нахождения расстояния Дамерау-Левенштейна

| Структура данных | Длина 4 символа | Длина 1000 символов |
|---|---|---|
| Матрица | 480 | 8064096 |
| Три вспомогательные переменные (int) | 84 | 84 |
| Два счетчика (int) | 56 | 56 |
| Передача параметров | 106 | 2098 |
| Сумма данных | 726 | 8066334 |

Таблица 4.4: Потребляемая память структурами данных в рекурсивном алгоритме нахождения расстояния Дамерау-Левенштейна

| Структура данных | Длина 4 символа | Длина 1000 символов |
|---|---|---|
| Пять переменных для подсчета IDTR | 140 * 4 = 560 | 140 * 1000 = 140000 |
| Передача параметров | 106 * 4/2 = 212 | 2098 * 1000/2 = 1049000 |
| Сумма данных | 772 | 1189000 |

## 4.3 Тестирование

Тестирование проводилось в виде Unit Тестов, то есть тестов отдельных компонентов (методов) вычисляющих расстояние Левенштейна. Было написано 15 тестовых случаев по 5 различных случаев для каждой из трёх реализаций. Тестовые методы рассматривают такие случаи:

1. даны две пустые строки
2. дана одна пустая а вторая непустая строки

3. даны две одинаковые строки
4. даны две строки с одной различной буквой
5. даны две пустые строки, где необходимо сделать перестановку

Примеры тестов можно увидеть в следующей таблице.

Таблица 4.5: Таблица тестовых данных

| № | Первое слово | Второе слово | Ожидаемый результат | Полученный результат |
|---|---|---|---|---|
| 1 | | | 0 0 0 | 0 0 0 |
| 2 | kot | skat | 2 2 2 | 2 2 2 |
| 3 | kate | ktae | 2 1 1 | 2 1 1 |
| 4 | abacaba | aabcaab | 4 2 2 | 4 2 2 |
| 5 | sobaka | sboku | 3 3 3 | 3 3 3 |
| 6 | qwerty | queue | 4 4 4 | 4 4 4 |
| 7 | apple | aplpe | 2 1 1 | 2 1 1 |
| 8 | | cat | 3 3 3 | 3 3 3 |
| 9 | parallels | | 9 9 9 | 9 9 9 |
| 10 | bmstu | utsmb | 4 4 4 | 4 4 4 |

# Заключение

Был изучен метод динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна. Также изучены алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками, получены практические навыки раелизации указанных алгоритмов в матричной и рекурсивных версиях. Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработаного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк. В результате исследований я пришел к выводу, что матричная реализация данных алгоритмов заметно выигрывает по времени при росте длины строк, следовательно более применима в реальных проектах.