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 курсу ”Анализ алгоритмов”

Тема Расстояния Левенштейна и Дамерау-Левенштейна

Студент Сусликов Д.В.

Группа ИУ7-55

Оценка (баллы) _____

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

Москва — 2020 г.

Оглавление

Введение

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

Расстояние Левенштейна и его обобщения активно применяется:

- для автозамен ошибок в слове (например, в поисковых системах);
- в биоинформатике для сравнения генов, хромосом и белков;
- и в других областях.

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

Задачи:

1. Дать математическое описание расстояний Левенштейна и Дамерау-Левенштейна;
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

Задачей алгоритма Левештейна является нахождение минимального количества редакционных операций (вставок, удалений, замен) нужных для приведения одной строки символов к другой.

При нахождении расстояния Дамерау—Левенштейна добавляется операция перестановки двух соседних символов.

Действия обозначаются так:

- D (delete) — удалить;
- I (insert) — вставить;
- R (replace) — заменить;
- M (match) - совпадение;
- X (exchange) - перестановка (только в алгоритме Дамерау—Левенштейна).

Каждая операция, кроме перестановки, увеличивает релакционное расстояние на 1.

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

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

где $m(a,b)$ равно 0 при $a = b$, и 1 в противном случае;

$min\{\, a, b, c\}$ возвращает наименьший из аргументов.

Расстояние Дамерау-Левенштейна вычисляется по следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min(D(S_1[1..i], S_2[1..j-1]) + 1, & \\ D(S_1[1..i-1], S_2[1..j]) + 1, & j > 0, i > 0 \\ D(S_1[1..i-1], S_2[1..j-1]) + m(S_1[i], S_2[j])) & \\ D(S_1[1..i-2], S_2[1..j-2]) + 1 & \end{cases} \quad (1.2)$$

2 Конструкторская часть

Далее будут представлены схемы алгоритмов нахождения редакционного расстояния.

2.1 Разработка алгоритмов

Ниже на Рисунке 1 представлена схема матричного алгоритма нахождения расстояний Левенштейна.

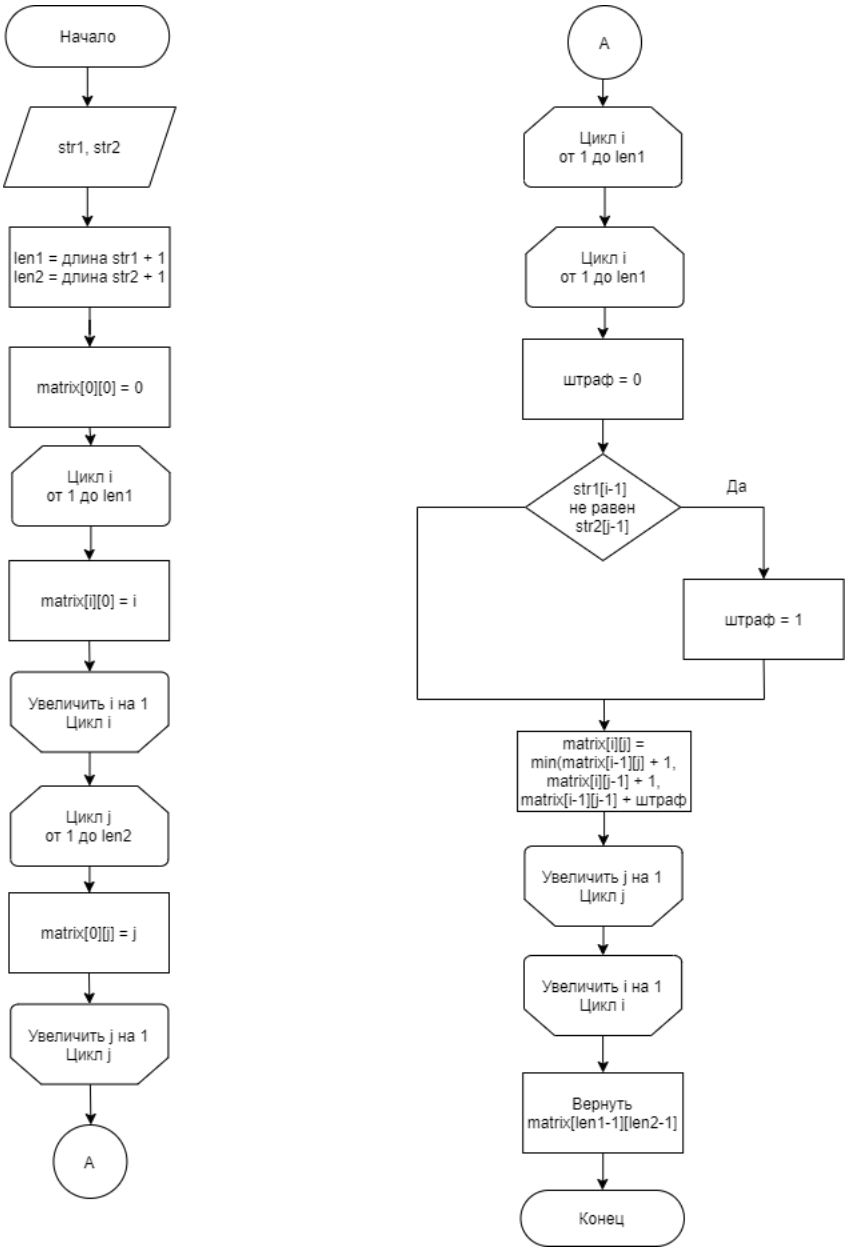

Рисунок 1 - Схема матричного алгоритма Левенштейна

Далее на Рисунке 2 можно увидеть схему рекурсивного алгоритма нахождения расстояния Левенштейна.

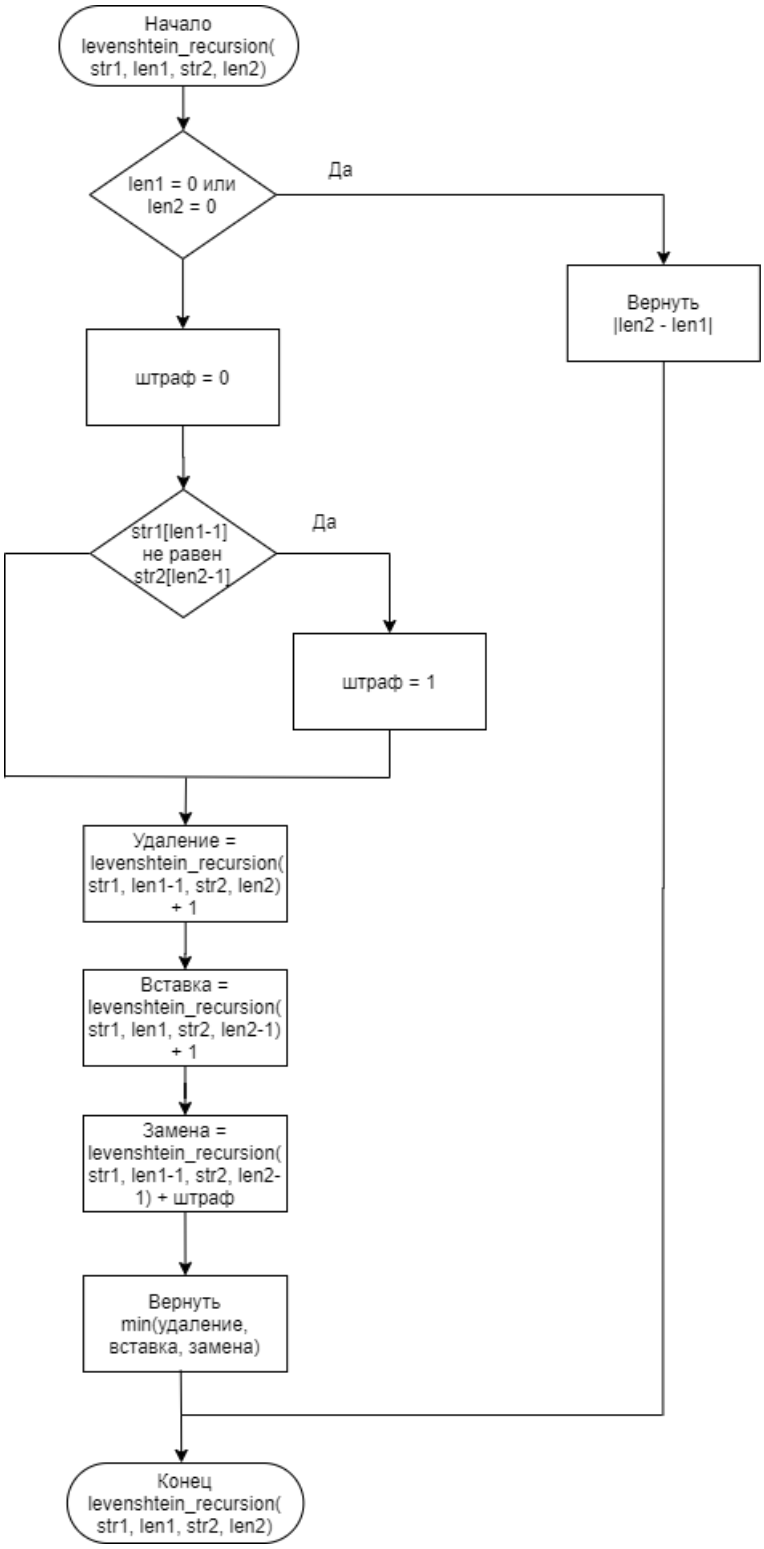

Рисунок 2 - Схема рекурсивного алгоритма Левенштейна

Ниже на Рисунке 3 изображена схема рекурсивного алгоритма с использованием матрицы для нахождения расстояний Левенштейна.

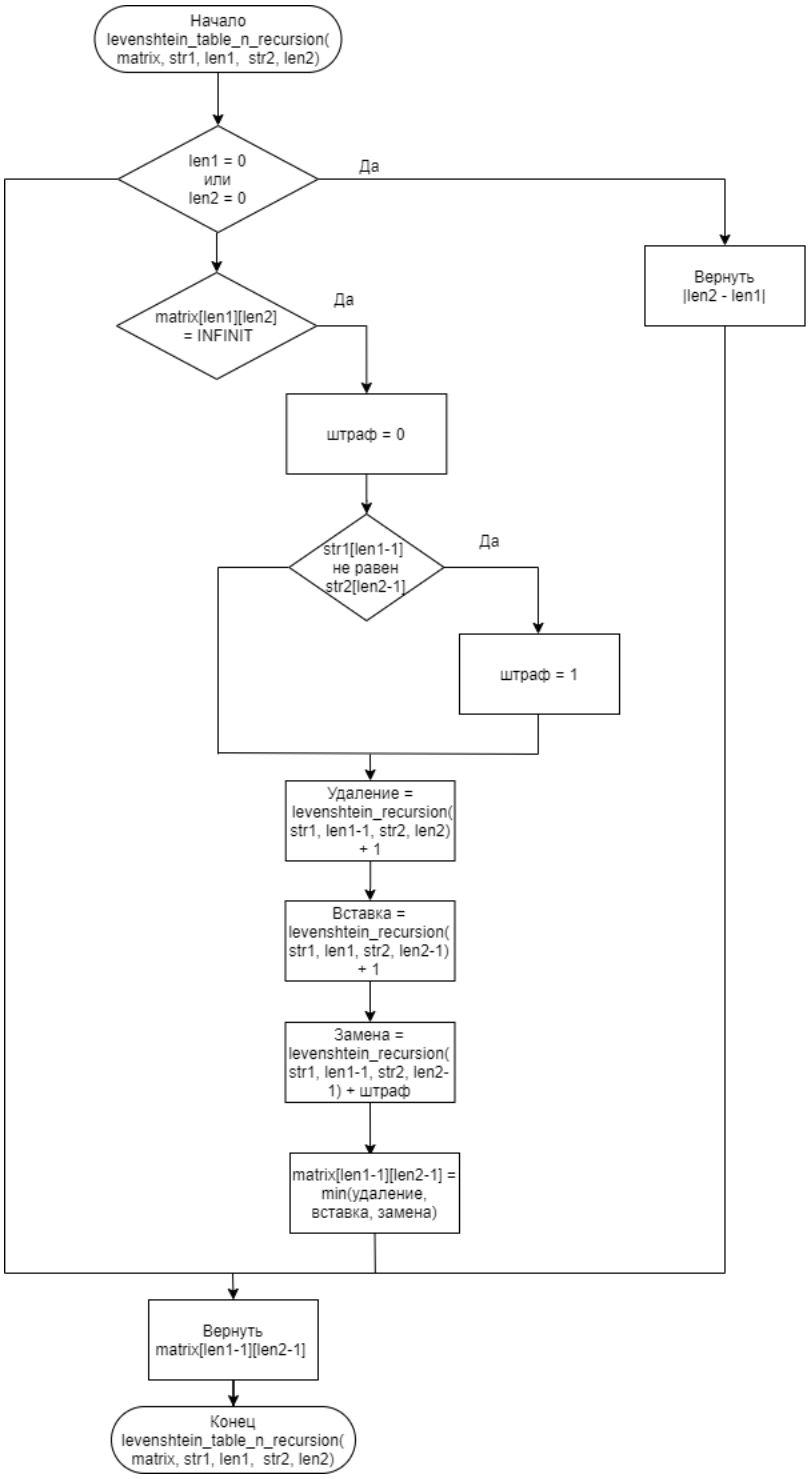

Рисунок 3 - Схема рекурсивного алгоритма Левенштейна с использованием матрицы

Далее на Рисунке 4 показана схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна.

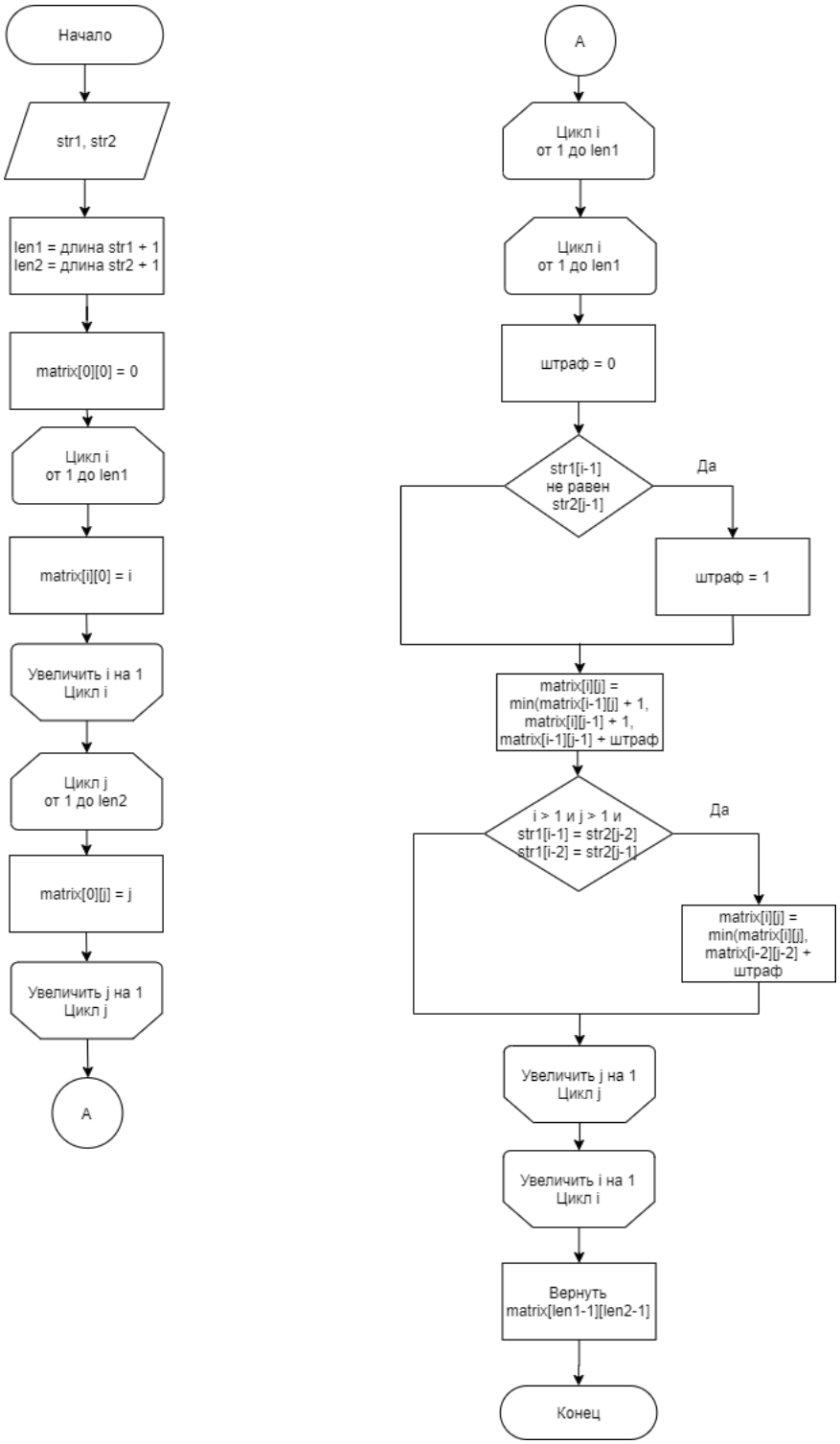

Рисунок 4 - Схема матричного алгоритма Дамерау-Левенштейна

2.2 Сравнительный анализ памяти

В случае матричной реализации алгоритма, требуется хранить

- динамическую матрицу размером C_1 * (len1 + 1) + (len1 + 1) * C_1 * (len2 + 1),
- значения двух счетчиков $2C_2$,
- значение вспомогательной переменной (штраф) C_2

и передавать параметры (C_2 * len).

В итоге, так будет высчитываться общий размер запрашиваемой памяти:

$$C_1 * (len1 + 1) + (len1 + 1) * C_1 * (len2 + 1) + 2C_2 + C_2 * len$$

В случае рекурсивной реализации при каждом вызове требуется хранить значение 4 вспомогательных переменных - $4C_2$ и передавать параметры (C_2 * len).

Причем, максимальная глубина рекурсивного вызова – максимальная длина двух строк.

Так как память рекурсивного алгоритма растет пропорционально сумме длин строк, а матричного – их произведения, то на строках большой длины рекурсивный алгоритм затрачивает меньше памяти по сравнению с матричным.

3 Технологическая часть

В данном разделе рассматривается выбранный язык программирования, среда разработки, требуемые инструменты для реализации и сама реализация.

3.1 Требования к программному обеспечению

1. Программа должна предусматривать ввод двух строк произвольной длины. Строки могут содержать произвольный набор символов.
2. Выбор применяемого алгоритма осуществляется пользователем из списка алгоритмов, предложенных в меню.
3. На выходе программа выводит матрицу (в случае выбора матричного алгоритма) и значение расстояния между введенными строками.
4. Также необходимо предусмотреть выполнение замеров процессорного времени для каждого из алгоритмов.

3.2 Средства реализации

В лабораторной работе быть использован язык $C++$, так как был ранее изучен и использован во многих предыдущих работах.

Среда разработки - Qt.

Для замеров процессорного времени была использована функция $clock()$.

3.3 Реализация алгоритмов

Листинг 1 - Функция нахождения расстояния матричного алгоритма Левенштейна

```
long int levenshtein_distance(const char* str1, const char* str2)
{
  long int len1 = strlen(str1) + 1;
  long int len2 = strlen(str2) + 1;

  int** matrix = create_matrix(len1, len2);

  matrix[0][0] = 0;
  for (long int i = 1; i < len1; i++)
    matrix[i][0] = i;

  for (long int j = 1; j < len2; j++)
    matrix[0][j] = j;

  for (long int i = 1; i < len1; i++)
  {
    for (long int j = 1; j < len2; j++)
    {
      int sub_cost = 0;
      if (str1[i - 1] != str2[j - 1])
        sub_cost = 1;

      matrix[i][j] = my_min(matrix[i - 1][j] + DELETION_COST,
      matrix[i][j - 1] + INSERTION_COST,
      matrix[i - 1][j - 1] + sub_cost);
    }
  }
  int answer = matrix[len1 - 1][len2 - 1];

  print_matrix(matrix, len1, len2);
  free_matrix(&matrix, len1);
  return  answer;
}
```

Листинг 2 - Функция рекурсивного алгоритма нахождения расстояний Левенштейна

```
long int levenshtein_recursion(const char* str1, long int len1, const
    char* str2, long int len2)
{
  if (len1 <= 0 || len2 <= 0)
    return abs((int)(len2 - len1));

  int sub_cost = 0;
  if (str1[len1 - 1] != str2[len2 - 1])
    sub_cost = 1;

  long int deletion = levenshtein_recursion(str1, len1 - 1, str2,
     len2) + DELETION_COST;
  long int insertion = levenshtein_recursion(str1, len1, str2, len2 -
      1) + INSERTION_COST;
  long int replacement = levenshtein_recursion(str1, len1 - 1, str2,
     len2 - 1) + sub_cost;

  return my_min(deletion, insertion, replacement);
}

long int levenshtein_recursion(const char* str1, const char* str2)
{
  long int len1 = strlen(str1);
  long int len2 = strlen(str2);

  if (len1 <= 0 || len2 <= 0)
    return abs((int)(len2 - len1));

  int sub_cost = 0;
  if (str1[len1 - 1] != str2[len2 - 1])
    sub_cost = 1;

  long int deletion = levenshtein_recursion(str1, len1 - 1, str2,
     len2) + DELETION_COST;
  long int insertion = levenshtein_recursion(str1, len1, str2, len2 -
      1) + INSERTION_COST;
```

```
31    long int replacement = levenshtein_recursion(str1, len1 - 1, str2,
          len2 - 1) + sub_cost;
32
33    return my_min(deletion, insertion, replacement);
34  }
```

Листинг 3 - Функция рекурсивного алгоритма с использованием матрицы нахождения расстояний Левенштейна

```
1  long int levenshtein_table_n_recursion(int** matrix, const char* str1
       ,
2  long int len1, const char* str2, long int len2)
3  {
4    if (len1 <= 0 || len2 <= 0)
5      return abs((int)(len2 - len1));
6
7    if (matrix[len1][len2] == INFINIT)
8    {
9      int sub_cost = 0;
10     if (str1[len1 - 1] != str2[len2 - 1])
11       sub_cost = 1;
12
13     long int deletion = levenshtein_table_n_recursion(matrix, str1,
           len1 - 1, str2, len2) + DELETION_COST;
14     long int insertion = levenshtein_table_n_recursion(matrix, str1,
           len1, str2, len2 - 1) + INSERTION_COST;
15     long int replacement = levenshtein_table_n_recursion(matrix, str1
           , len1 - 1, str2, len2 - 1) + sub_cost;
16
17     matrix[len1][len2] = my_min(deletion, insertion, replacement);
18   }
19
20   return matrix[len1][len2];
21 }
22
23 long int levenshtein_table_n_recursion(const char* str1, const char*
       str2)
24 {
25   long int len1 = strlen(str1) + 1;
26   long int len2 = strlen(str2) + 1;
```

```
27
28    int** matrix = (int**)create_matrix(len1, len2);
29
30    fill_matrix_with_infinity(matrix, len1, len2);
31
32    matrix[0][0] = 0;
33    for (long int i = 1; i < len1; i++)
34    matrix[i][0] = i;
35
36    for (long int j = 1; j < len2; j++)
37    matrix[0][j] = j;
38
39    len1--; len2--;
40
41    int sub_cost = 0;
42    if (str1[len1 - 1] != str2[len2 - 1])
43    sub_cost = 1;
44
45    long int deletion = levenshtein_table_n_recursion(matrix, str1,
          len1 - 1, str2, len2) + DELETION_COST;
46    long int insertion = levenshtein_table_n_recursion(matrix, str1,
          len1, str2, len2 - 1) + INSERTION_COST;
47    long int replacement = levenshtein_table_n_recursion(matrix, str1,
          len1 - 1, str2, len2 - 1) + sub_cost;
48
49    int answer = my_min(deletion, insertion, replacement);
50
51    print_matrix(matrix, len1, len2);
52
53    free_matrix(&matrix, len1);
54    return  answer;
55 }
```

Листинг 4 - Функция матричного алгоритма нахождения расстояний Дамерау-Левенштейна.

```
1   long int damerau_levenshtein_distance(const char* str1, const char*
        str2)
2   {
3     long int len1 = strlen(str1) + 1;
```

```
4       long int len2 = strlen(str2) + 1;
5
6       int** matrix = (int**)create_matrix(len1, len2);
7
8       matrix[0][0] = 0;
9       for (long int i = 1; i < len1; i++)
10      matrix[i][0] = i;
11
12      for (long int j = 1; j < len2; j++)
13      matrix[0][j] = j;
14
15      for (long int i = 1; i < len1; i++)
16      {
17        for (long int j = 1; j < len2; j++)
18        {
19          int sub_cost = 0;
20          if (str1[i - 1] != str2[j - 1])
21          sub_cost = 1;
22
23          matrix[i][j] = my_min(matrix[i - 1][j] + DELETION_COST,
24          matrix[i][j - 1] + INSERTION_COST,
25          matrix[i - 1][j - 1] + sub_cost);
26
27          if (i > 1 && j > 1 && str1[i - 1] == str2[j - 2]
28          && str1[i - 2] == str2[j - 1])
29          {
30            matrix[i][j] = std::min(matrix[i][j], matrix[i - 2][j - 2]
                + sub_cost);
31          }
32        }
33      }
34
35      int answer = matrix[len1 - 1][len2 - 1];
36      print_matrix(matrix, len1, len2);
37      free_matrix(&matrix, len1);
38
39      return  answer;
40    }
```

3.4 Описание тестирования

Тестирование осуществляется по принципу «черного ящика». Для проверки корректности программы необходимо предусмотреть наборы различных тестов, включающих в себя случаи одной и обеих пустых строк, случаи строки, состоящей из одного символа, случаи эквивалентных строк.

4 Экспериментальная часть

В данном разделе будут рассмотрены примеры работы программы, произведено тестирование, выполнены эксперименты по замеру времени, а также сделан сравнительный анализ полученных данных.

4.1 Примеры работы

Ниже на Рисунке 5 представлен пример работы программы при выборе матричного алгоритма Левенштейна для строк ”кит”и ”скат”.

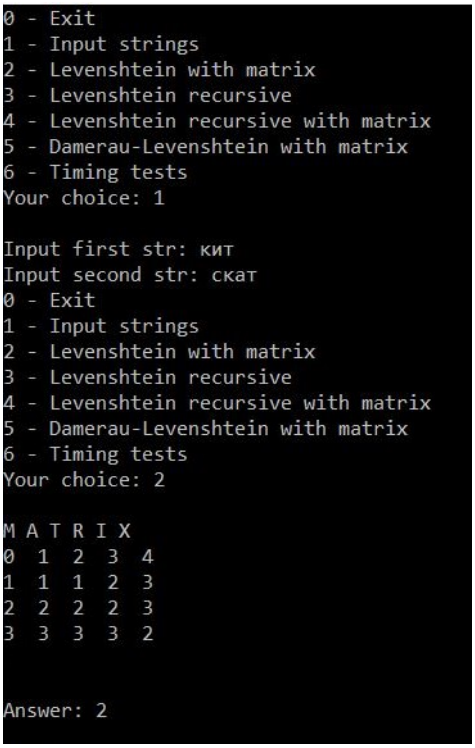

Рисунок 5 - Пример работы программы при выборе матричного алгоритма Левенштейна

Ниже на Рисунке 6 представлен пример работы программы при выборе рекурсивного алгоритма Левенштейна для строк ”кит”и ”скат”.

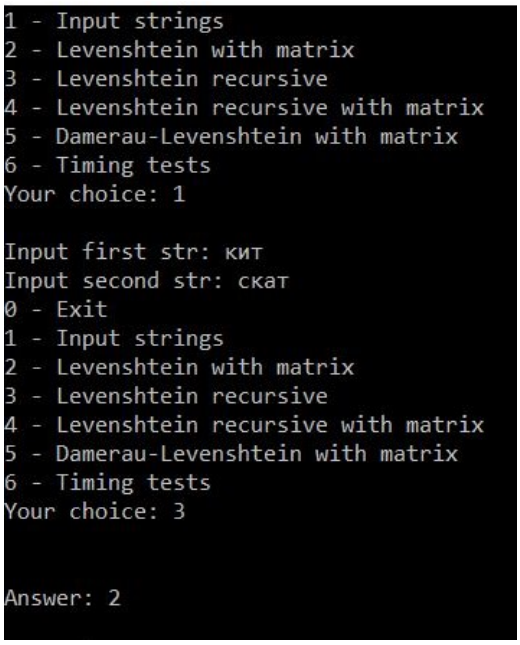

Рисунок 6 - Пример работы программы при выборе рекурсивного алгоритма Левенштейна

4.2 Результаты тестов

4.2.1 Результаты работы программы

В Таблице 1 показаны результаты работы программы при различных строках. Через ”/”показан результат алгоритма Дамерау-Левенштейна в случаях, когда он отличен от результата алгоритма Левенштейна.

Таблица 1 - Результаты работы программы

Первое слово	Второе слово	Результат
кит	скат	2
кит	кот	1
-	-	0
october	november	4
123	132	2 / 1
hotmm	hotmm	0

4.2.2 Сравнительный анализ времени работы алгоритмов

Проведем тестирование по замеру времени работы алгоритмов в зависимости от длины строк. Для этого подсчитаем время работы каждого алгоритма строках длиной 5, 10, 50, 100, 200 и 500 символов. Результаты проведенного эксперимента отображены на Рисунке 7.

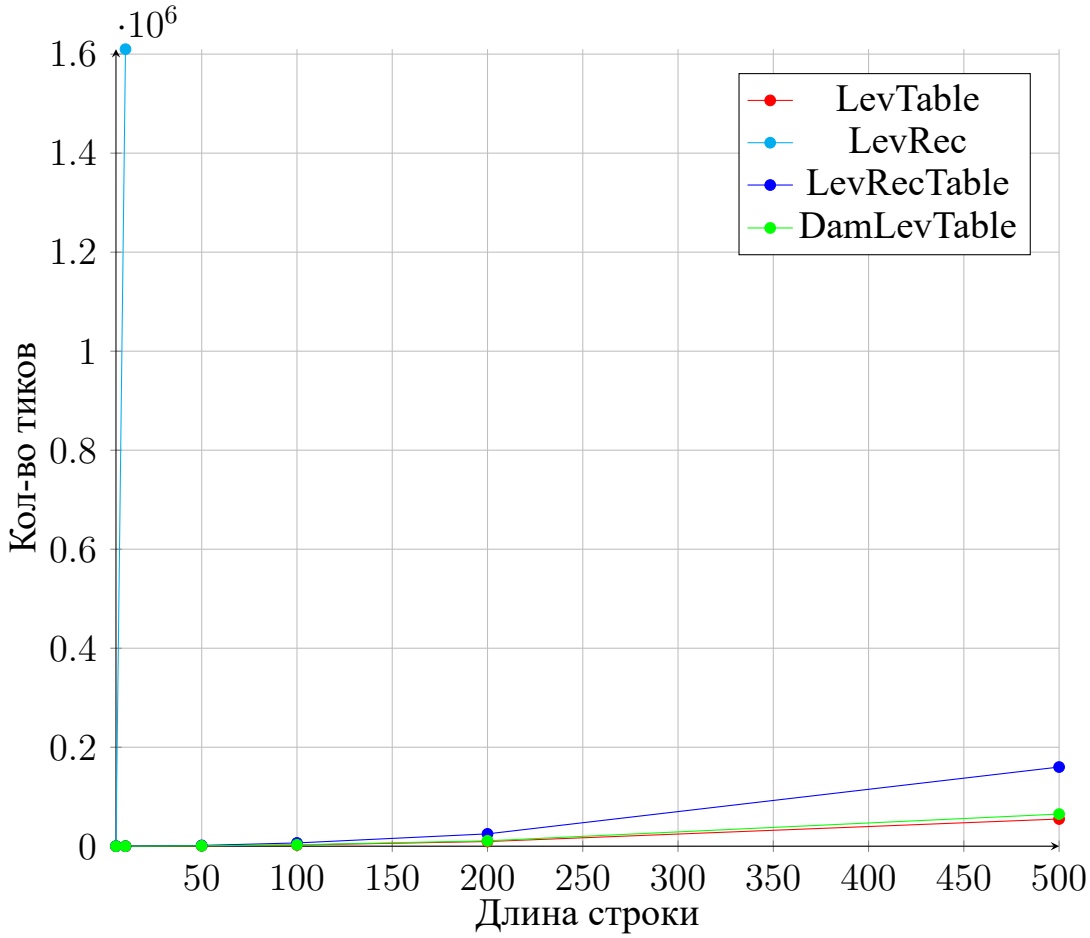

Рисунок 7 - Графики зависимости времени работы алгоритмов от длины строк

По результатам, отображенным на Рисунке 7, можно сделать вывод, что рекурсивный алгоритм Левенштейна - самый медленный из представленных алгоритмов. Это связано с большим количеством повторных операций. Самым быстрым алгоритмом оказался матричный алгоритм Левенштейна. Матричный алгоритм Дамерау-Левенштейна работает чуть медленее ранее названного алгоритма из-за операций сравнений, выполняющихся в цикле. Рекурсивный алгоритм Левенштейна с матрицей быстрее рекурсивного, но уступает по скорости выполнения матричным алгоритмам.

Заключение

В ходе лабораторной работы были разработаны и реализованы алгоритмы нахождения расстояний Левенштейна (матричный, рекурсивный, рекурсивный с использованием матрицы) и Дамерау-Левенштейна (матричный), а также проведен анализ затрачиваемых ресурсов каждого из метода. По результату анализу стало ясно, что матричные реализации алгоритмов быстрее, чем рекурсивные.

Литература