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.В. Сусликов
(Группа) (Подпись, дата) (И.О. Фамилия)

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

Москва, 2020 г.

Оглавление

Введение

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

Расстояние Левенштейна и его обобщения активно применяется:

1) для автозамен ошибок в слове (например, в поисковых системах);

2) в биоинформатике для сравнения генов, хромосом и белков;

3) 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:

1) D (delete) — удалить;

2) I (insert) — вставить;

3) R (replace) — заменить;

4) M (match) - совпадение;

5) 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 рекуррентной Формуле 1:

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

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 2 :

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

Вывод

Таким образом, для строк, где возможна перестановка букв местами, алгоритм Дамерау-Левенштейна будет выдавать редакционное расстояние меньше, чем алгоритм Левенштейна.

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.

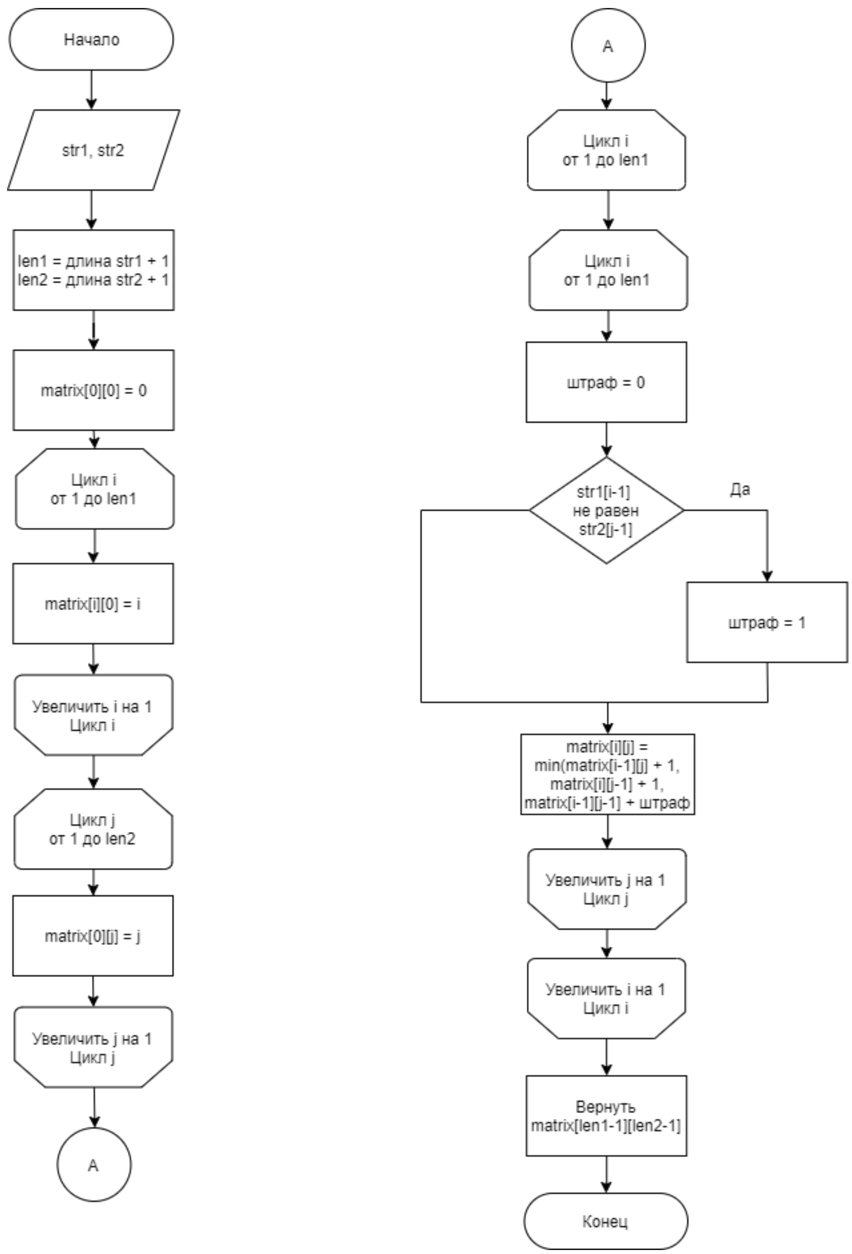

Рисунок 1 - Схема матричного алгоритма Левенштейна

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

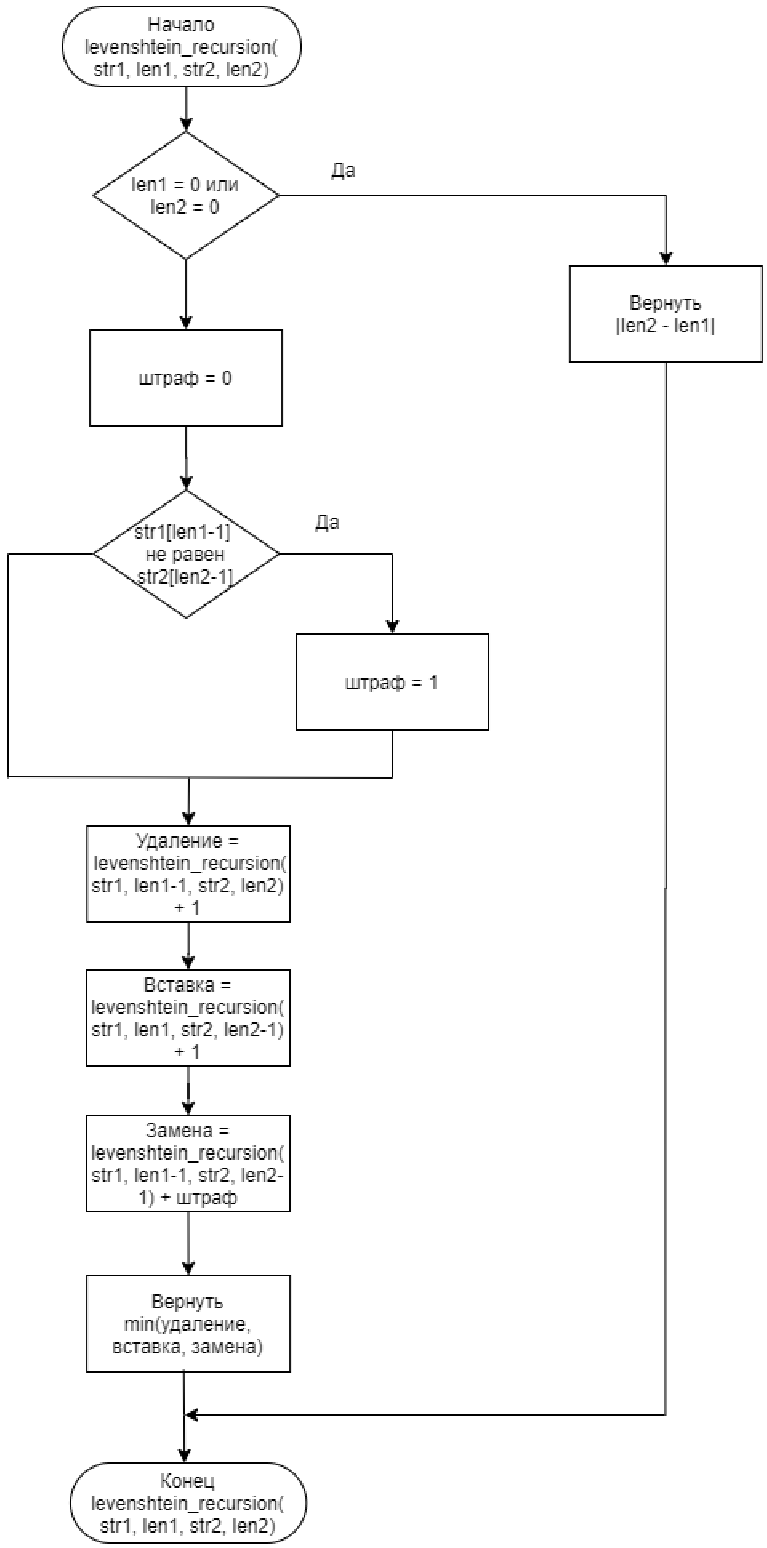

Рисунок 2 - Схема рекурсивного алгоритма Левенштейна

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

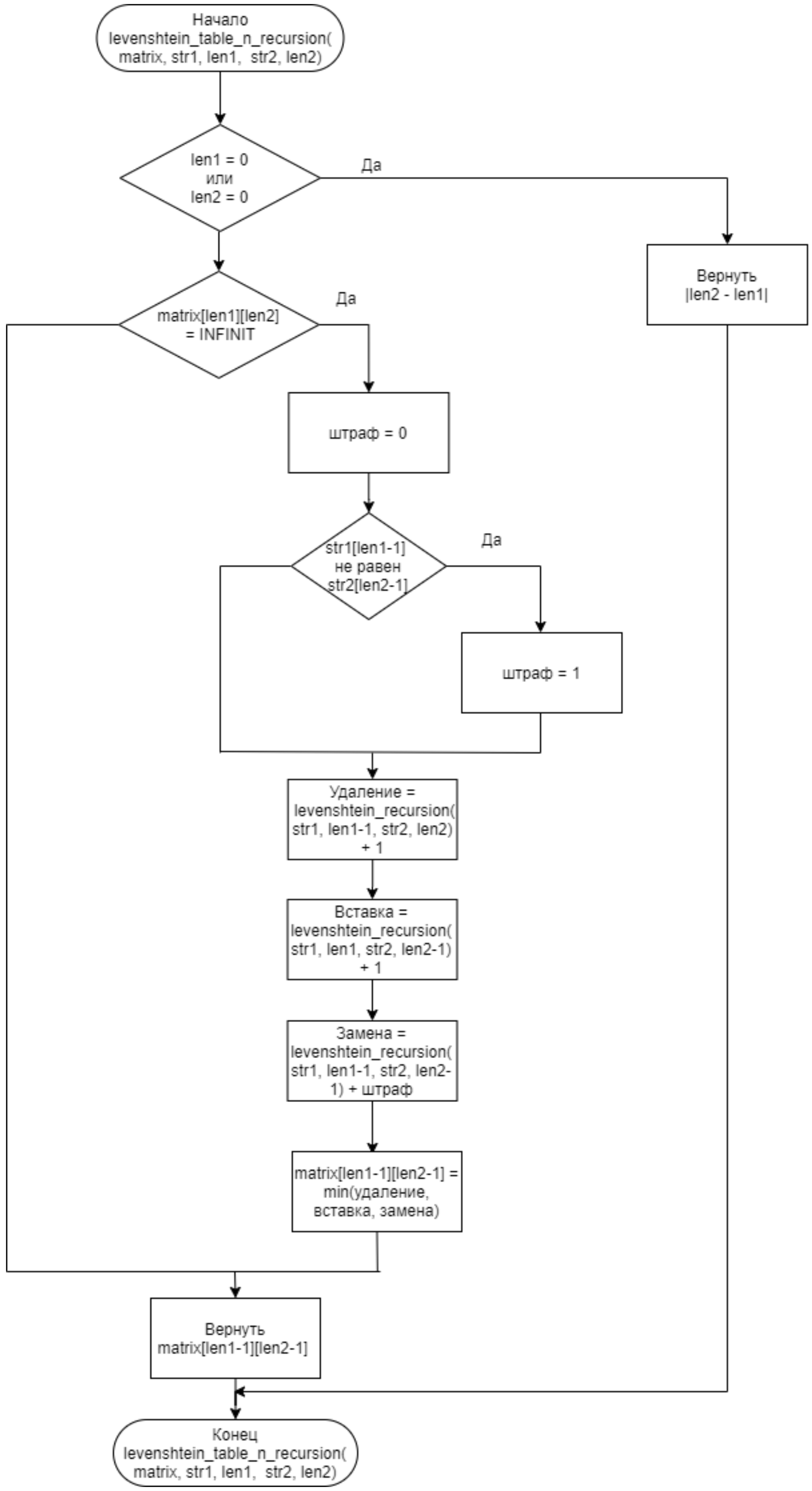

Рисунок 3 - Схема рекурсивного алгоритма Левенштейна с использованием матрицы

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

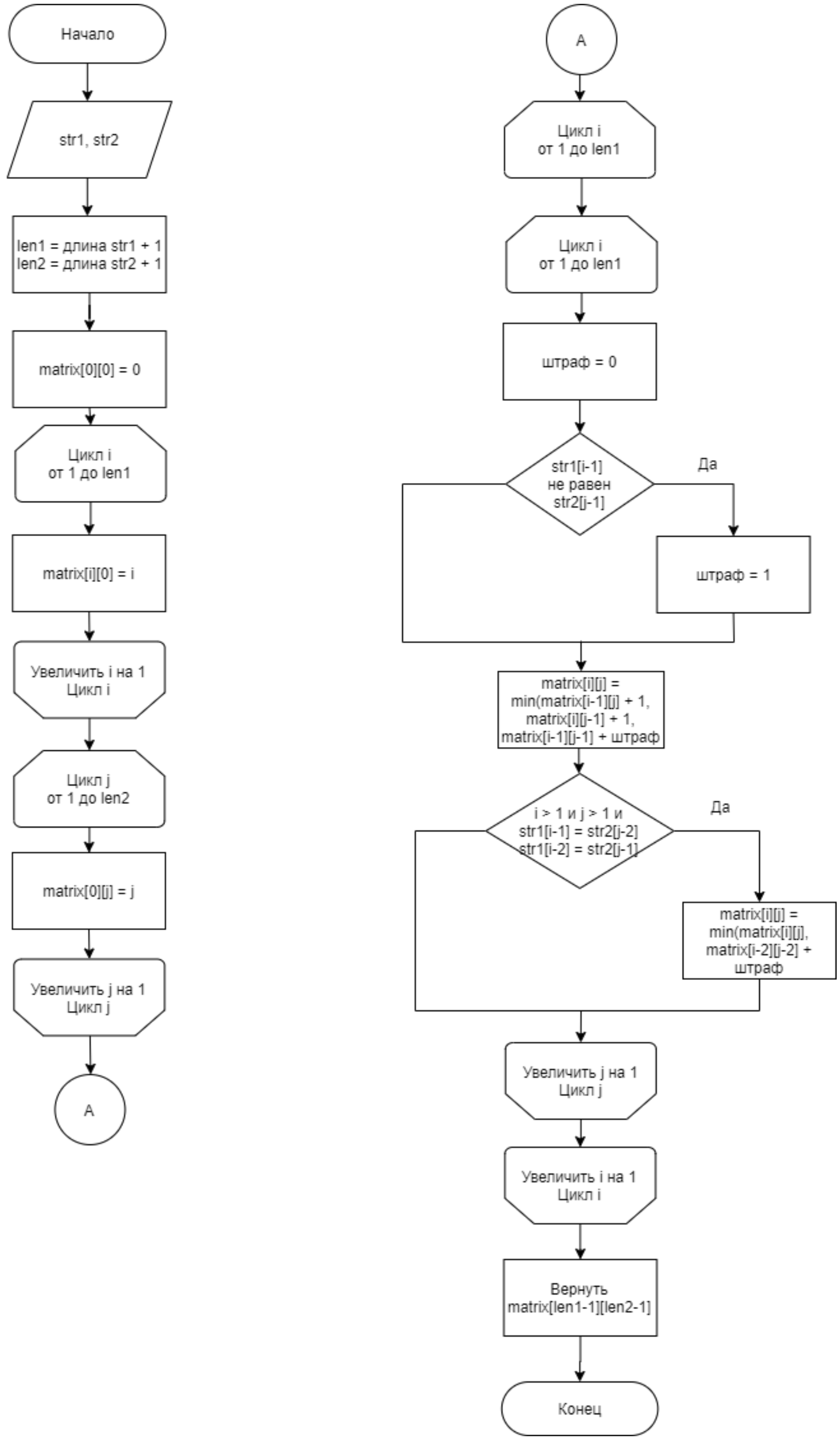

Рисунок 4 - Схема матричного алгоритма Дамерау-Левенштейна

2.2 Сравнение памяти

В случае матричной реализации алгоритма, требуется хранить

1) динамическую матрицу размером, расчитанным по Формуле 3
   $C_1 \cdot (len1 + 1) + (len1 + 1) \cdot C_1 \cdot (len2 + 1)$ (3),
2) значения двух счетчиков $2C_2$,
3) значение вспомогательной переменной (штраф) C_2

и передавать параметры ($C_2 \cdot len$).

В итоге, по Формуле 4 будет высчитываться общий размер запрашиваемой памяти:

$C_1 \cdot (len1 + 1) + (len1 + 1) \cdot C_1 \cdot (len2 + 1) + 2C_2 + C_2 \cdot len$ (4)

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

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

Вывод

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 описанных выше алгоритмов.

3.1 Требования к программному обеспечению

Программа должна соответствовать приведенным ниже требованиям:

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

3.2 Средства реализации

В лабораторной работе был использован язык C++[1], так как он известен, и на нём было написано множество предыдущих работ.

Среда разработки - Qt[2].

Для замеров процессорного времени была использована функция $clock()$[3].

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;
```

```
  long int replacement = levenshtein_recursion(str1, len1 - 1, str2,
      len2 - 1) + sub_cost;

  return my_min(deletion, insertion, replacement);
}
```

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

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

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

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

    matrix[len1][len2] = my_min(deletion, insertion, replacement);
  }

  return matrix[len1][len2];
}

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

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

  fill_matrix_with_infinity(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;

  len1--; len2--;

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

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

  int answer = my_min(deletion, insertion, replacement);

  print_matrix(matrix, len1, len2);

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

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

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

```
    long int len2 = strlen(str2) + 1;

    int** matrix = (int**)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);

        if (i > 1 && j > 1 && str1[i - 1] == str2[j - 2]
        && str1[i - 2] == str2[j - 1])
        {
          matrix[i][j] = std::min(matrix[i][j], matrix[i - 2][j - 2]
            + sub_cost);
        }
      }
    }

    int answer = matrix[len1 - 1][len2 - 1];
    print_matrix(matrix, len1, len2);
    free_matrix(&matrix, len1);

    return answer;
  }
```

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.

Вывод

По итогу, написанная программа соотвествует всем описанным выше требованиям, алгоритмы были реализованы на C++, так как данный язык известен, выполнено на нём много прошлых работ.

4 | Экспериментальная часть

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

4.1 Примеры работы

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

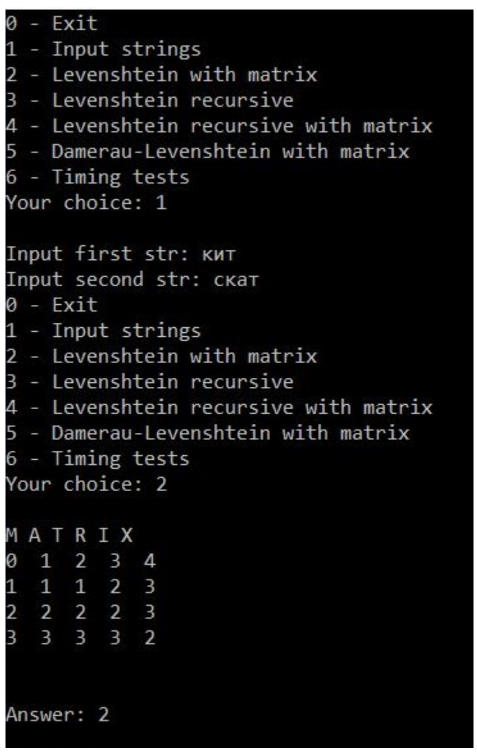

Рисунок 5 - Пример работы программы при выборе матричного алгоритма Левенштейна

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

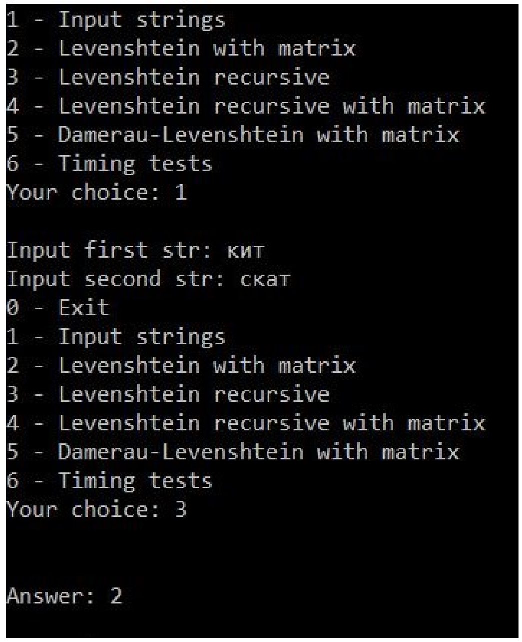

Рисунок 6 - Пример работы программы при выборе рекурсивного алгоритма Левенштейна

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 Сравнение 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.

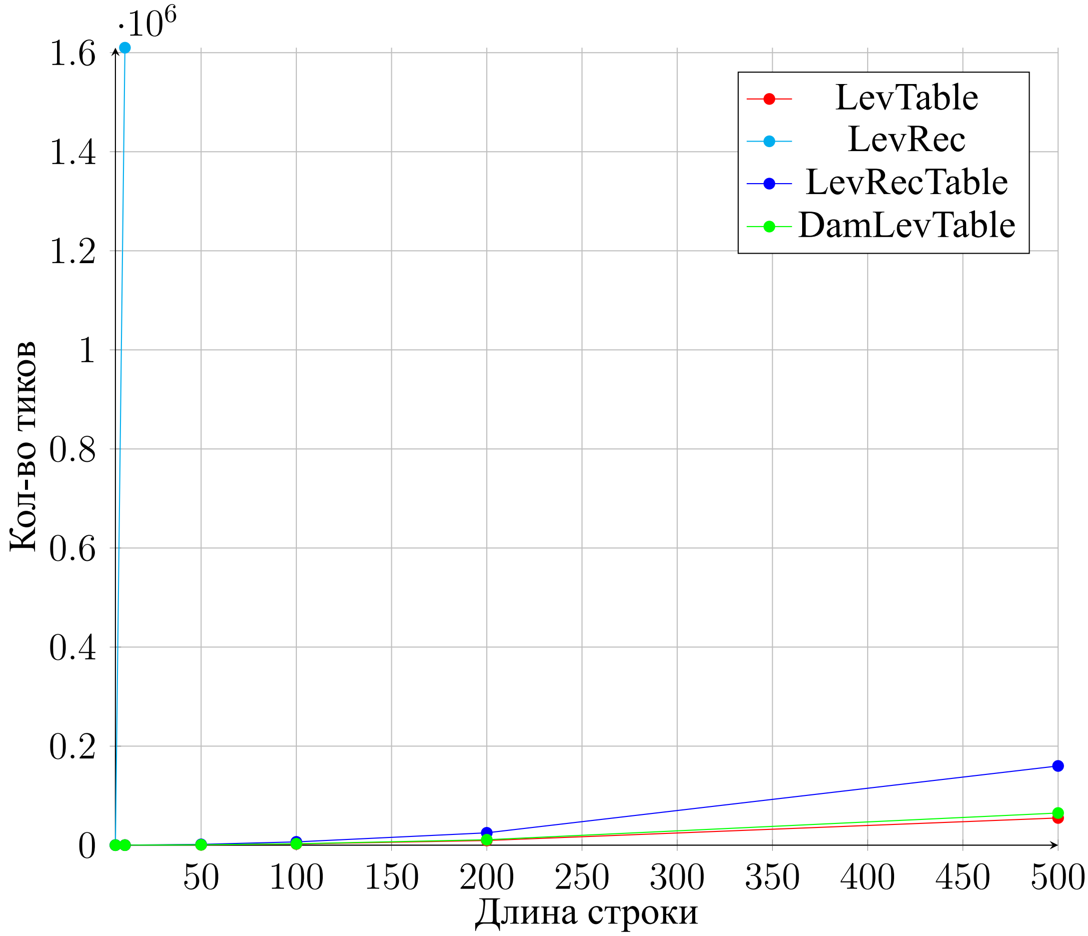

Рисунок 7 - Графики зависимости времени работы алгоритмов от длины строк

Вывод

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), были построены схемы данных алгоритмов, а также проведено сравнение затрачиваемых ресурсов каждого из методов.

По результату сравнения стало ясно, что матричные реализации алгоритмов быстрее, чем рекурсивные в более чем 10000 раз на строках длины от 10 и более символов. На строках длины до 10 символов скорости выполнения матричных алгоритмов примерно равны, однако на строках большой длины от 200 и более видна разница между алгоритмами. При строках длины в 500 символов алгоритм Левенштейна работает быстрее Дамерау-Левенштейна на 16% и быстрее матрично-рекурсивного на 66%.

Литература

1) Бьерн Страуструп. Язык программирования С++. -URL: https://codernet.ru/books/c_plus/bern_straustrup_yazyk_programmirovaniya_ c_specia (дата обращения: 01.10.2020). Текст: электронный.

2) Qt. -URL: https://www.qt.io/ (дата обращения: 01.10.2020). Текст: электронный.

3) Функция $clock$. -URL: https://docs.microsoft.com/ru-ru/cpp/c-runtime-library/reference/clock?view=vs-2019 (дата обращения: 01.10.2020). Текст: электронный.

4) Вычисление расстояний Левенштейна.-URL: https://foxford.ru/wiki/informatika/vychislenie-rasstoyania-levensteyna (дата обращения: 01.10.2020). Текст: электронный.