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

Дисциплина: Анализ алгоритмов

Тема: Расстояния Левенштейна и Дамерау-Левенштейна

Студент: Платонова Ольга

Группа: ИУ7-55Б

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

Москва.
2020 г.

Оглавление

Введение

В данной лабораторной работе требуется изучить и реализовать алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна, осуществить сравнительный анализ указанных алгоритмов по способу реализации: рекурсивно, матрично; и, как следствие, по затраченным ресурсам (времени и памяти).

Алгоритмы поиска расстояний Левенштейна

1. не рекурсивный с заполнением матрицы
2. рекурсивный без матрицы
3. рекурсивный с матрицей

Алгоритмы поиска расстояний Дамерау-Левенштейна

1. не рекурсивный с заполнением матрицы

1. Аналитическая часть

В данном разделе рассмотрим понятия, связанные с расстояниями Левенштейна и его модификацией - расстояниями Дамерау-Левенштейна, а также формулы, необходимые для поиска расстояний.

1.1. Формула Левенштейна

Редакционное расстояние, или расстояние Левенштейна – минимальное количеств редакционных операций, которые необходимы для преобразования одной строки в другую.

Применение расстояния Левенштейна:

1. автозамена (в поисковых системах, базах данных)
2. сравнение текстовых файлов утилитой diff и ей подобным
3. в биоинформатике (сравнение генов, хромосом, белков)

Введем штраф для каждой из редакторской операции:

$$\begin{cases} \text{вставка } (insert) \\ \text{удаление } (delete) \\ \text{замена } (replace) \end{cases} \text{ - штраф 1}$$

совпадение (match) – штраф 0.

Из определения следует, что поиск расстояния сводится к поиску минимального количества операций, вследствие чего, возникает проблема взаимного выравнивания строк для уменьшения числа операций. Решение проблемы взаимного выравнивания строк заключается в использовании рекуррентной формулы [1].

Тогда формула Левенштейна примет вид:

$$D(i,j) = \begin{cases} 0, i = 0, j = 0 \\ j, i = 0,\ j > 0 \\ i, i > 0, j = 0 \\ min(D(s1[1..i], s2[1..j-1]) + 1, \\ D(s1[1..i-1], s2[1..j]) + 1, \\ D(s1[1..i-1], s2[1..j-1]) + fine) \end{cases}, \text{где} \qquad (1)$$

s1[1..i] – подстрока длиной i

s2[1..j] – подстрока длиной j

$$fine = \begin{cases} 0, s1[i] = s2[j] \\ 1, \text{иначе} \end{cases}$$

1.2. Формула Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых выше, добавлена операция перестановка двух соседних символов (exchange) со штрафом 1.

Модификация была введена из-за частого количества ошибок пользователей, связанных с транспозицией символов.

Формула Дамерау-Левенштейна имеет следующий вид:

$$D(i,j) = \begin{cases} 0, i = 0, j = 0 \\ j, i = 0,\ j > 0 \\ i, i > 0, j = 0 \\ min1(D(s1[1..i], s2[1..j-1]) + 1, \\ D(s1[1..i-1], s2[1..j]) + 1, \\ D(s1[1..i-1], s2[1..j-1]) + fine) \\ min2(min1,\ D(s1[1..i-2], s2[1..j-2]) + 1) \end{cases} \quad \text{где} \qquad (2)$$

min2 верно при условии, что *i > 1, j > 1, s1[i] = s2[j-1], s1[i-1] = s2[j]*

иначе, верно *min1*

2. Конструкторская часть

В данном разделе рассмотрим схемы для каждого из алгоритмов.

2.1. Разработка алгоритмов

На рисунке 1 изображена схема *матричного алгоритма* нахождения расстояний Левенштейна.

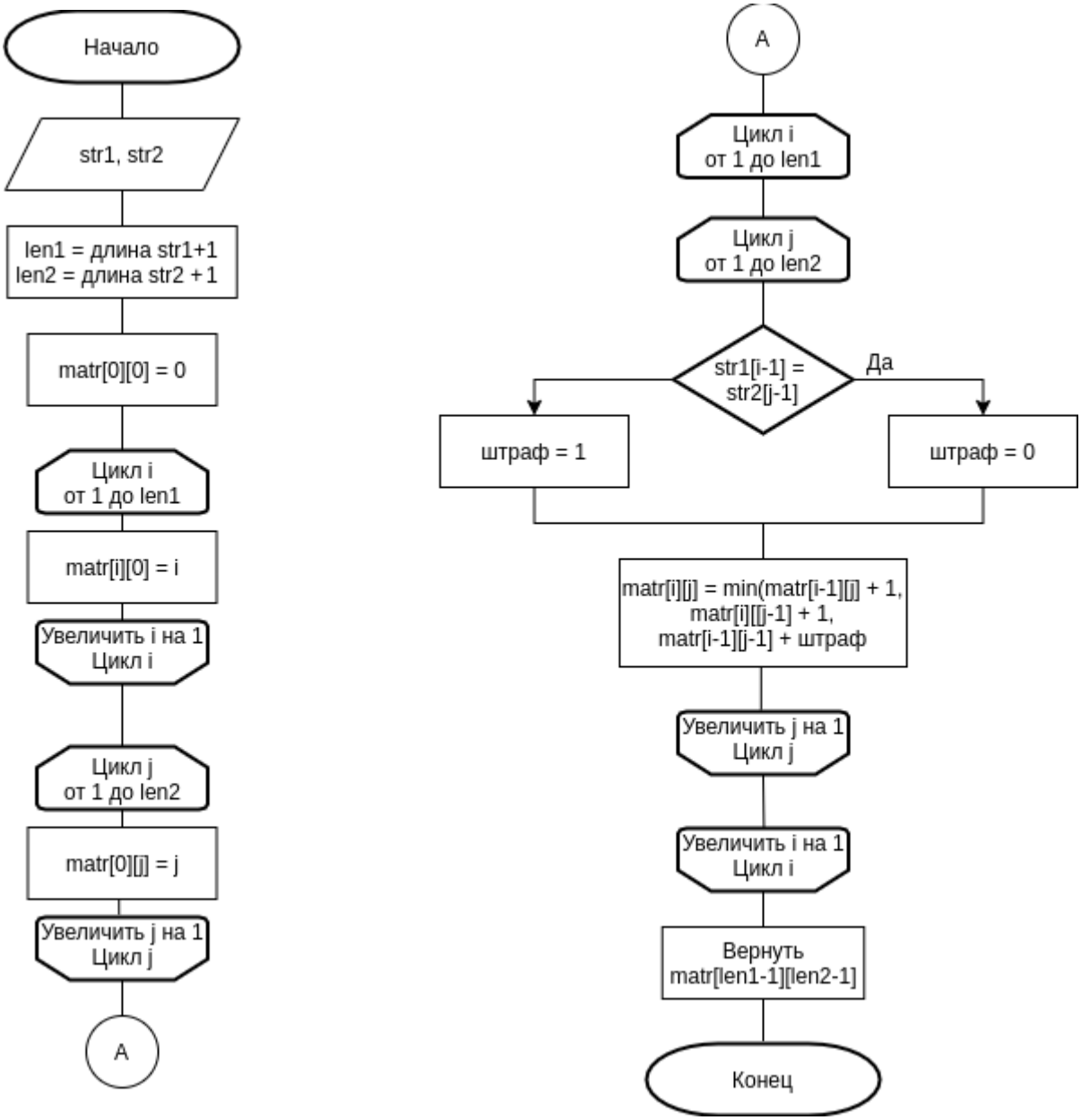


Рисунок 1. Схема матричного алгоритма нахождения расстояний Левенштейна.

На рисунке 2 изображена схема *рекурсивного алгоритма* нахождения расстояний Левенштейна.

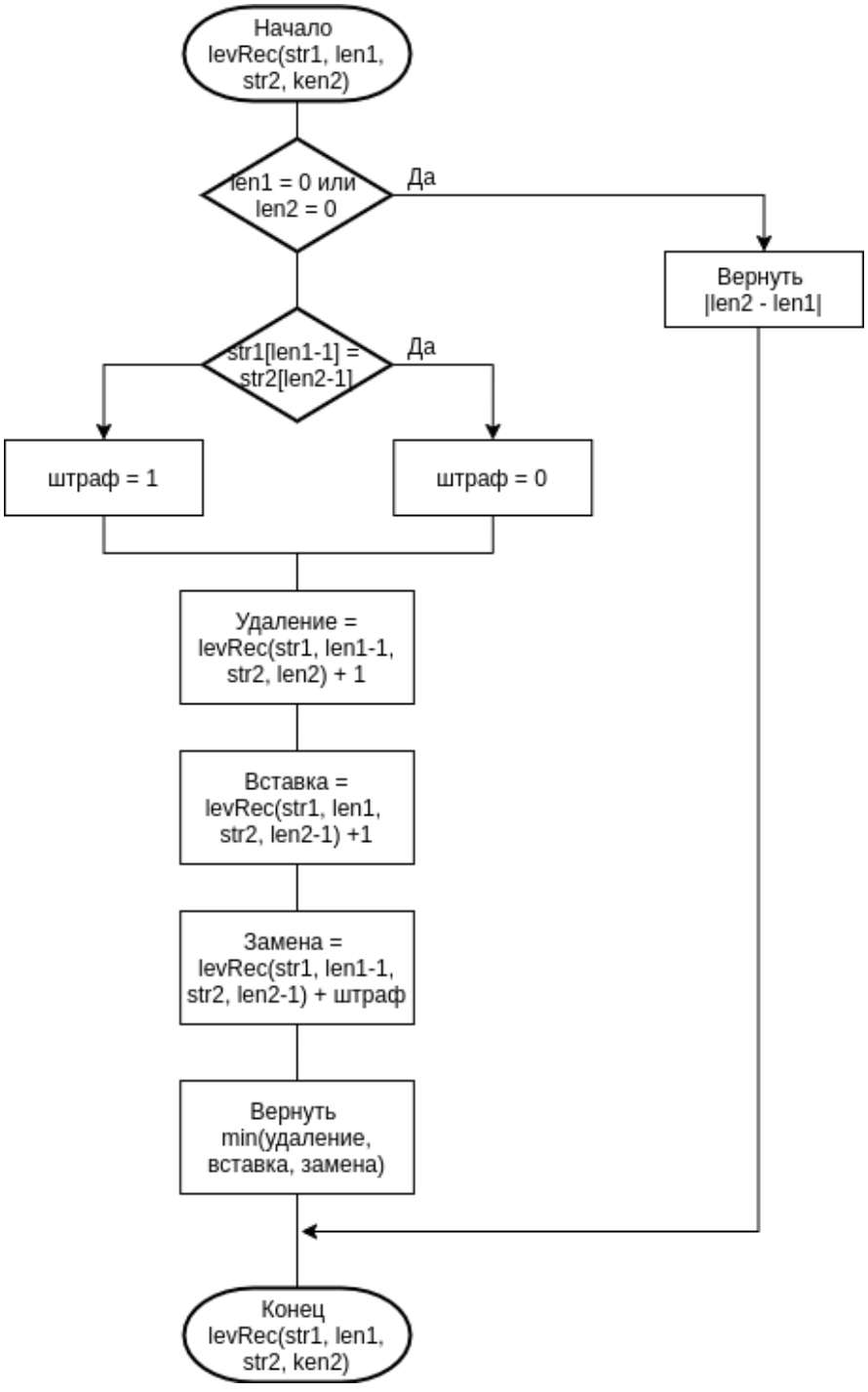


Рисунок 2. Схема рекурсивного алгоритма нахождения расстояний Левенштейна.

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

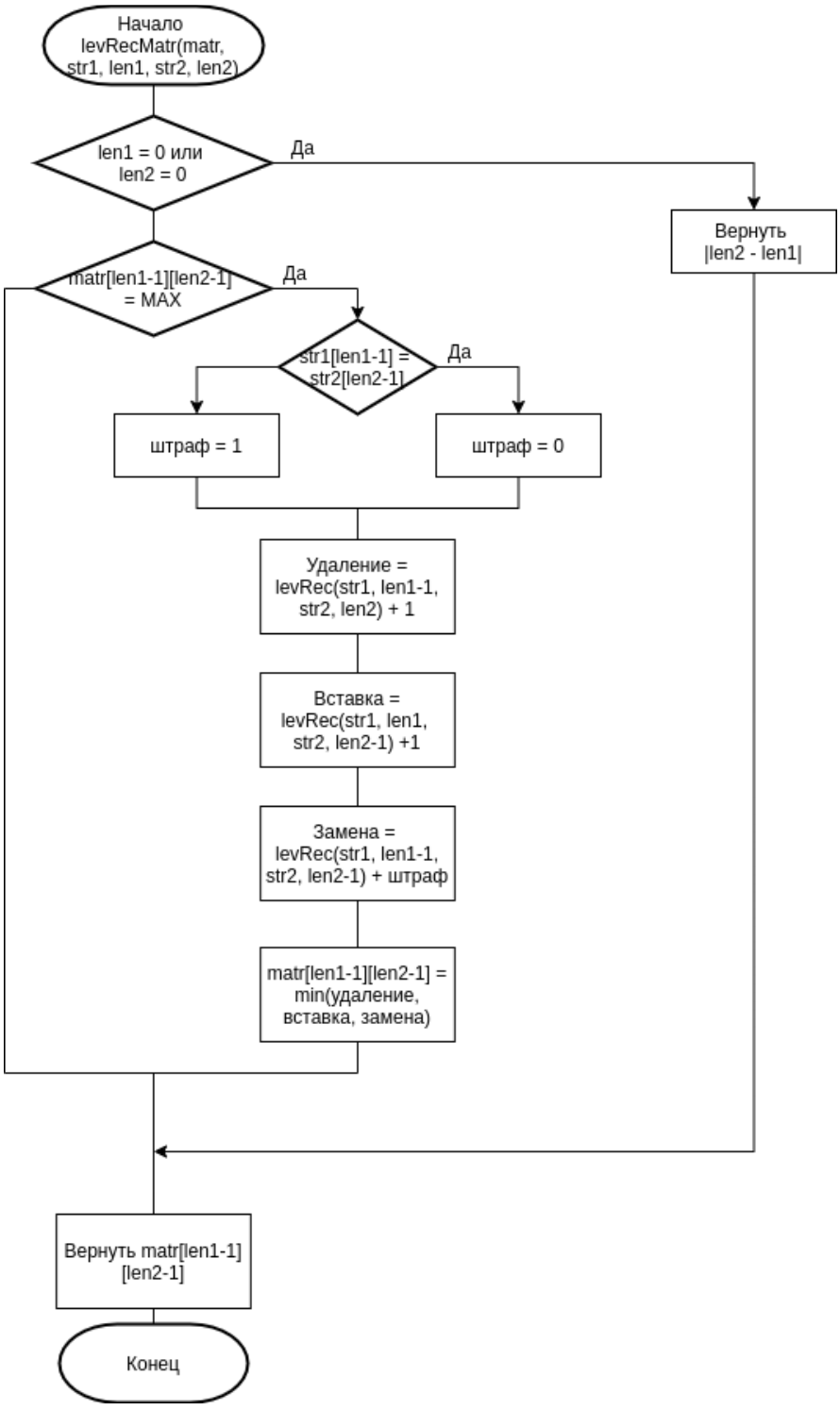


Рисунок 3. Схема рекурсивного алгоритма с использованием матрицы нахождения расстояний Левенштейна.

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

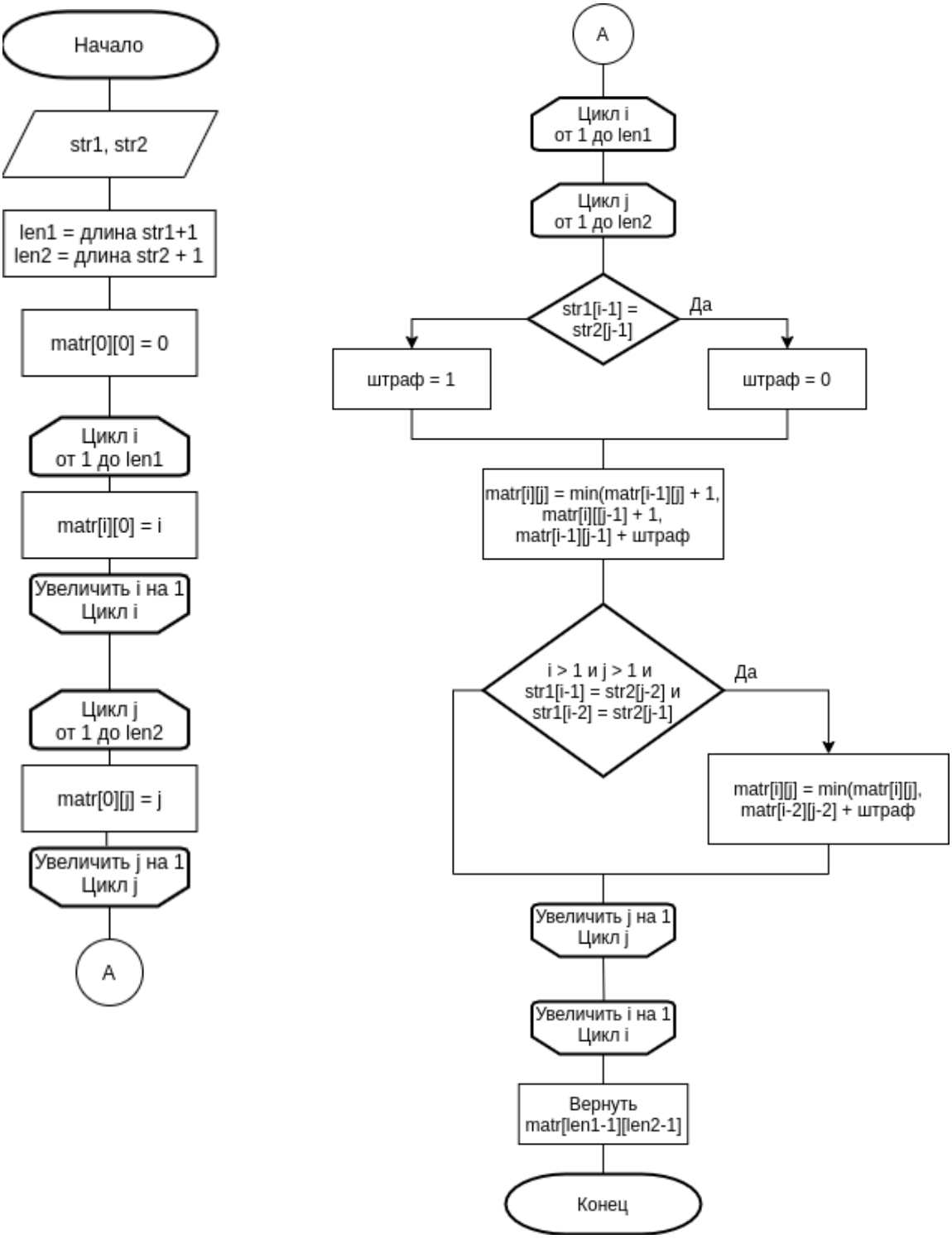


Рисунок 4. Схема матричного алгоритма нахождения расстояний Дамерау-Левенштейна.

2.2. Сравнительный анализ памяти

В случае *матричной* реализации алгоритма, требуется хранить

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

и передавать параметры (C_2 * len).

Таким образом, общий размер запрашиваемой памяти имеет вид:

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

В случае *рекурсивной* реализации при каждом вызове требуется хранить

1. значения 4 вспомогательных переменных $4C_2$

и передавать параметры (C_2 * len).

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

Поскольку, память рекурсивного алгоритма растет пропорционально сумме длин строк, а матричного – их произведения, то на строках большой длины можно добиться преимущества рекурсии по памяти [2].

3. Технологическая часть

В данном разделе рассмотрим язык программирования, среду разработки, требуемые инструменты для реализации. Также представим непосредственно реализацию.

3.1. Требования к программному обеспечению

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

3.2. Средства реализации

В данной работе используется язык программирования C++, из-за опыта работы на нем. Среда разработки – Qt.

Для замеров процессорного времени использовалась функция clock() [3].

3.3. Реализация алгоритмов

Листинг 1.

Матричный алгоритм нахождения расстояний Левенштейна.

```
int LevensteinMatr(string str1, string str2)
{
   int len1 = static_cast<int>(str1.length() + 1);
   int len2 = static_cast<int>(str2.length() + 1);

   int **matr = nullptr;
   allocateMatr(&matr, len1, len2);

   //Заполнение матрицы(тривиальные случаи)
   matr[0][0] = 0;
   for (int i = 1; i < len1; i++) {
      matr[i][0] = i;
   }
   for (int j = 1; j < len2; j++) {
      matr[0][j] = j;
   }

   //Нетривиальные - минимум
   for (int i = 1; i < len1; i++) {
      for (int j = 1; j < len2; j++) {
         int fine = (str1[i - 1] == str2[j - 1]) ? 0 : 1;
         matr[i][j] = min(matr[i - 1][j] + 1,              //Удаление
```

```
            matr[i][j - 1] + 1,              //Вставка
            matr[i - 1][j - 1] + fine);      //Замена

      }
    }

    int res = matr[len1 - 1][len2 - 1];
    outputMatr(matr, len1, len2);
    freeMatr(&matr, len1);

    return res;
}
```

Листинг 2.

Рекурсивный алгоритм нахождения расстояний Левенштейна.

```
int LevensteinRec(string str1, string str2)
{
    int len1 = static_cast<int>(str1.length());
    int len2 = static_cast<int>(str2.length());

    return levRec(str1, len1, str2, len2);
}

int levRec(string str1, int len1, string str2, int len2)
{
    if (len1 == 0 || len2 == 0) {
        return abs(len2 - len1);
    }

    int fine = (str1[len1 - 1] == str2[len2 - 1]) ? 0 : 1;

    int deletion = levRec(str1, len1 - 1, str2, len2) + 1;
    int insert = levRec(str1, len1, str2, len2 - 1) + 1;
    int replace = levRec(str1, len1 - 1, str2, len2 - 1) + fine;

    return min(deletion, insert, replace);
}
```

Листинг 3.

Рекурсивный алгоритм с использованием матрицы нахождения расстояний Левенштейна.

```
int LevensteinRecMatr(string str1, string str2) {
    int len1 = static_cast<int>(str1.length() + 1);
    int len2 = static_cast<int>(str2.length() + 1);

    int **matr = nullptr;
    createMatr(&matr, len1, len2, MAX);

    int res = levRecMatr(&matr, str1, len1-1, str2, len2-1);
    outputMatr(matr, len1, len2);
    freeMatr(&matr, len1);
```

```
    return res;
}

int levRecMatr(int ***matr, string str1, int len1, string str2, int len2) {
    if (len1 == 0 || len2 == 0) {
        return abs(len2 - len1);
    }

    if ((*matr)[len1 - 1][len2 - 1] == MAX) {

        int fine = (str1[len1 - 1] == str2[len2 - 1]) ? 0 : 1;

        int deletion = levRecMatr(matr, str1, len1 - 1, str2, len2) + 1;
        int insert = levRecMatr(matr, str1, len1, str2, len2 - 1) + 1;
        int replace = levRecMatr(matr, str1, len1 - 1, str2, len2 - 1) + fine;

        (*matr)[len1 - 1][len2 - 1] = min(deletion, insert, replace);
    }

    return (*matr)[len1 - 1][len2 - 1];

}
```

Листинг 4.

Матричный алгоритм нахождения расстояний Дамерау-Левенштейна.

```
int DemLevMatr(string str1, string str2)
{
    int len1 = static_cast<int>(str1.length() + 1);
    int len2 = static_cast<int>(str2.length() + 1);

    int **matr = nullptr;
    allocateMatr(&matr, len1, len2);

    //Заполнение матрицы(тривиальные случаи)
    matr[0][0] = 0;
    for (int i = 1; i < len1; i++) {
        matr[i][0] = i;
    }
    for (int j = 1; j < len2; j++) {
        matr[0][j] = j;
    }

    //Нетривиальные - минимум
    for (int i = 1; i < len1; i++) {
        for (int j = 1; j < len2; j++) {
            int fine = (str1[i - 1] == str2[j - 1]) ? 0 : 1;
            matr[i][j] = min(matr[i - 1][j] + 1,              //Удаление
                             matr[i][j - 1] + 1,              //Вставка
                             matr[i - 1][j - 1] + fine);      //Замена
            if (i > 1 && j > 1 &&
                str1[i - 1] == str2[j - 2] && str1[i - 2] == str2[j - 1]) {
                    matr[i][j] = min(matr[i][j], matr[i - 2][j - 2] + fine);
                }

        }
```

```
    }

    int res = matr[len1 - 1][len2 - 1];
    outputMatr(matr, len1, len2);
    freeMatr(&matr, len1);

    return res;
}
```

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 рассмотрим примеры работы программы, произведем тестирование, выполним эксперименты по замеру времени, а также выполним сравнительный анализ полученных данных.

4.1. Примеры работы программы

Пример работы программы и ее интерфейс представлен на рисунке 5.

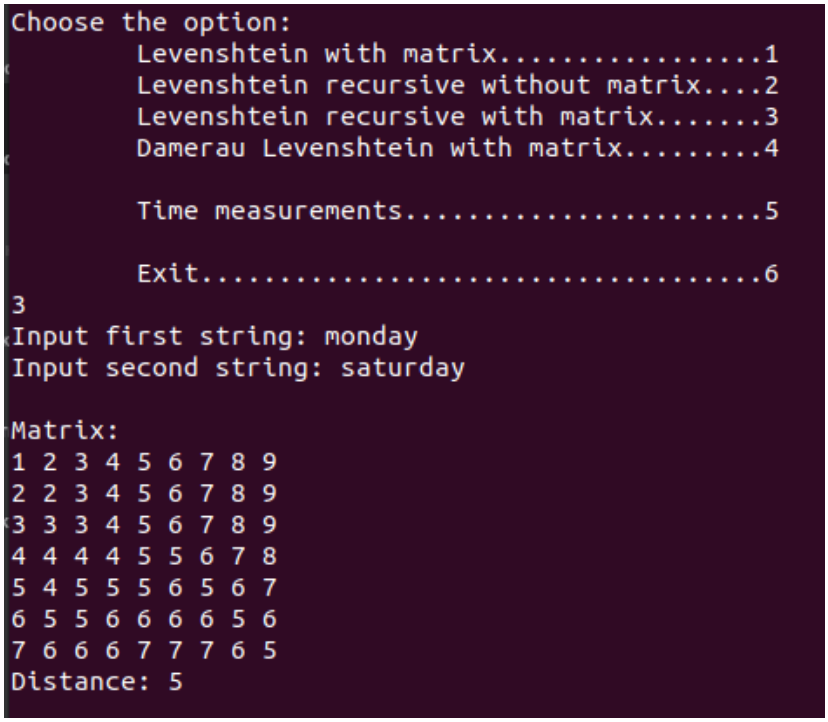


Рисунок 5. Интерфейс программного продукта.

4.2. Результаты тестирования

Результаты тестирования приведены в таблице 1.

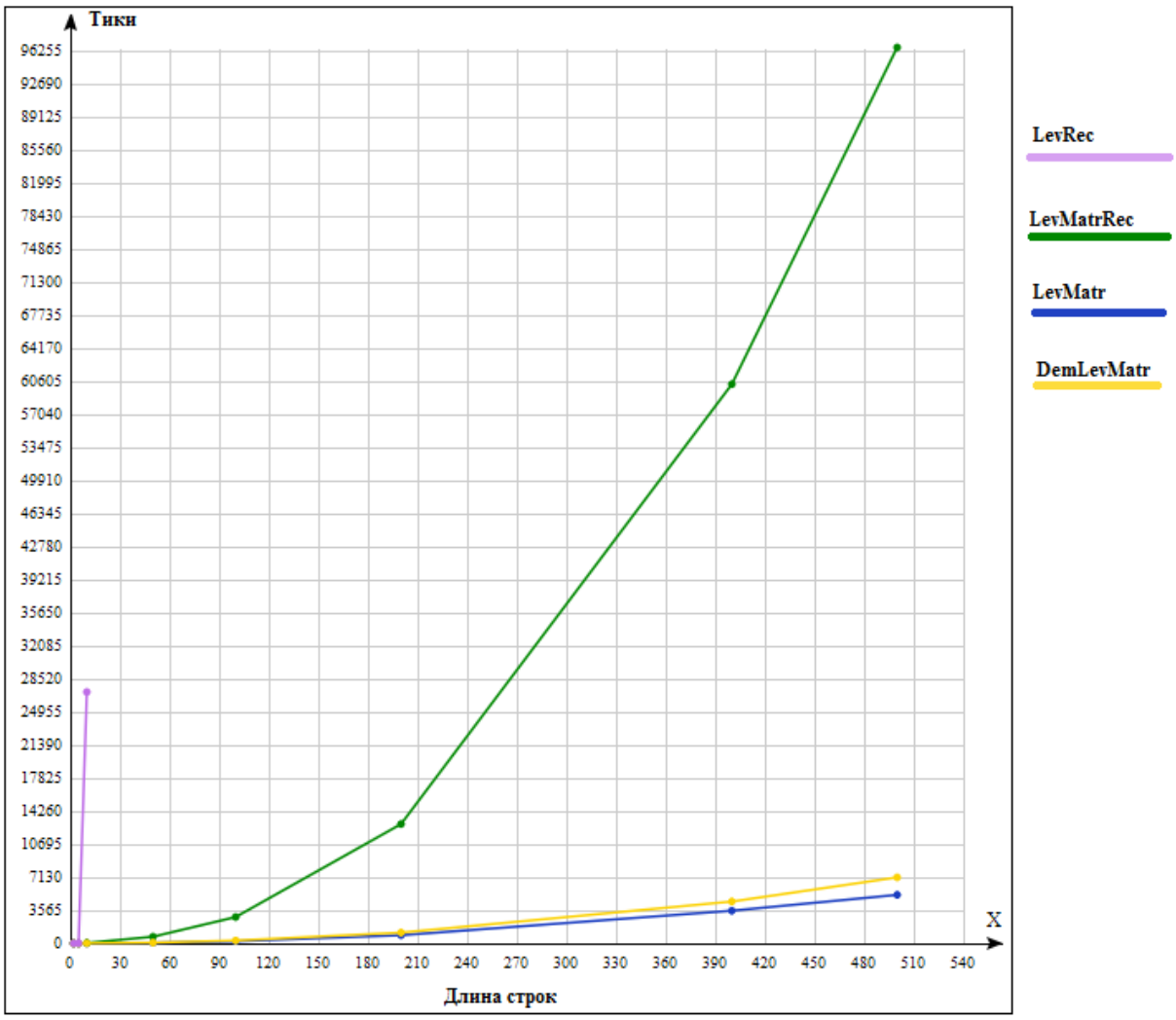
Таблица 1.

Входные параметры		Ожидаемый результат	Полученный результат
Первая строка	Вторая строка		
“milk”	“simka”	3	3
“”	“nothing”	7	7
“”	“”	0	0
“same”	“same”	0	0
“between”	“betwen”	1	1
“exchg”	“xechg”	2/1	2/1

4.3. Сравнительный анализ времени

Выполним эксперименты по замеру времени, использовав строки одинаковой длины. Эксперименты проводим на строках малой длины: 2, 5, 10 символов и большой длины: от 100 до 500 символов с шагом 100.

Результаты замеров времени приведены на графике.



Отметим, что самым долгим алгоритмом для строк большой длинны является рекурсивный алгоритм без использования матрицы. Такой резкий рост времени связан с большим количеством повторных операций. Однако на строках малой длины рекурсия не уступает матричной реализации. Самым быстрым является матричный алгоритм. Причем на скорость его выполнения длина строки влияет несущественно. Матричный алгоритм поиска расстояний Дамерау-Левенштейна работает медленнее из-за операций сравнения, выполняющихся в цикле. Рекурсивный алгоритм с использованием матрицы значительно быстрее обычной рекурсии, однако сильно уступает матричным реализациям.

Заключение

В ходе лабораторной работы был разработали и реализован алгоритмы нахождения расстояний Левенштейна (матричный, рекурсивный, рекурсивный с использованием матрицы) и Дамерау-Левенштейна (матричный), а также проведен анализ затрачиваемых ресурсов каждого из метода.

В результате анализа было установлено, что самую низкую скорость работы на строках малой длины (до 10 символов) имеет рекурсивный алгоритм с использованием матрицы. Рекурсивная реализация является самой быстрой на строках до 4 символов, а затем уступает в скорости матричной. На строках большой длины (100 символов) матричный алгоритм работает на порядок быстрее рекурсивного с матрицей. В то время, как память для матричной реализации требуется примерно в 6 раз больше.

Список литературы

1. Вычисление расстояний Левенштейна.-URL: https://foxford.ru/wiki/informatika/vychislenie-rasstoyania-levensteyna (дата обращения: 20.09.2020). Текст: электронный.
2. Алгоритмы во внешней памяти.-URL: https://neerc.ifmo.ru/wiki/index.php (дата обращения: 21.09.2020). Текст: электронный.
3. Техническая документация.-URL: https://docs.microsoft.com/ru-ru/cpp/c-runtime-library/reference/clock?view=vs-2019 (дата обращения: 20.09.2020). Текст: электронный.
4. Анализ поисковых алгоритмов.-URL: https://moluch.ru/archive/19/1968/ (дата обращения: 21.09.2020). Текст: непосредственный.