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

Группа ИУ7-51Б

Тема работы **Расстояние Левенштейна**

Студент Баранов Николай Алексеевич

Преподаватель Волкова Лилия Леонидовна

2024 г.

Содержание

Введение

В данной лабораторной работе будет рассмотрено hасстояние Левенштейна между двумя строками - это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения строки в другую[1].

Расстояние Левенштейна применяется для решения следующих задач:

— для исправления ошибок в слове поискового запроса;

— в формах заполнения информации на сайтах;

— для распознавания рукописных символов;

— в базах данных.

Расстояние Дамерау-Левенштейна отличается наличием операции транспозиции (перестановки двух соседних символов).

Цель: описание и исследование алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

Задачи:

1) Описать алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейн

2) Написать программу, реализующую несколько версий алгоритма поиска расстояния Левенштейна и одну версию алгоритма поиска расстояния Дамерау-Левенштейна;

3) Выбрать инструменты для замера процессорного времени выполнения реализаций алгоритмов;

4) Провести анализ затрат реализаций алгоритмов по времени.

1 Аналитическая часть

1.1 Расстояние Левенштейна

Рассмотрим 3 возможных реализации алгоритма - матричную, рекурсивную и рекурсивную с мемоизацией.

1.1.1 Рекурсивный алгоритм поиска расстояния Левенштейна

Пусть S_1 и S_2 - 2 строки (длиной n и m соответственно) над некоторым алфавитом [2]. Тогда расстояние Левенштейна $d(S_1, S_2)$ можно подсчитать по формуле 1.1.

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

Функция 1.2 показывает, была ли осуществлена замена символа.

$$m(a,b) = \begin{cases} 0 & \text{если a} = \text{b}, \\ 1 & \text{иначе} \end{cases}. \tag{1.2}$$

На основе формулы 1.1 получается рекурсивный алгоритм.

1.1.2 Матричный алгоритм поиска расстояния Левенштейна

При реализации функции с использованием формулы 1.1 может возникнуть ситуация, при которой будет возникать повторное вычисление значения для одних и тех же входных параметров. Для решения этой проблемы можно создать матрицу для сохранения промежуточных значений, после чего построчно заполнить её.

1.1.3 Рекурсивный алгоритм поиска расстояния Левенштейна с мемоизацией

Данный алгоритм получается путем объединении двух предыдущих алгоритмов. Он также использует рекурсию, но вычисленные значения сохраняются в таблице, что позволяет избежать повторного вычисления одних и тех же значений.

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна отличается от расстояния Левенштейна тем, что для него также определена операция перестановки двух соседних символов. Для его вычисления к результату, полученному с помощью формулы1.1 необходимо применить формулу1.3[**?**, levenshtein]

$$D[i,j] = \begin{cases} \min\{ & ; \text{Если выполняются условия} \\ \quad D[i,j] & ; \text{i} > 1, \text{j} > 1 \\ \quad D[i-2,j-2] + m(S_1[i], S_2[j]) & ; S_1[i] = S_2[j-1] \\ \} & ; S_1[i-1] = S_2[j] \\ D[i,j] & ; \text{Если не выполняются условия} \end{cases} \tag{1.3}$$

Здесь также могут быть реализованы 3 алгоритма - рекурсивный, матричный и рекурсивный с мемоизацией.

1.3 Вывод

В данном разделе были рассмотрены 3 метода для вычисления расстояний Левенштейна и Дамерау-Левенштейна.

2 Конструкторская часть

2.1 Функциональная схема работы программы

На рисунках 2.1 и 2.2 представлена функциональная схема работы программы.

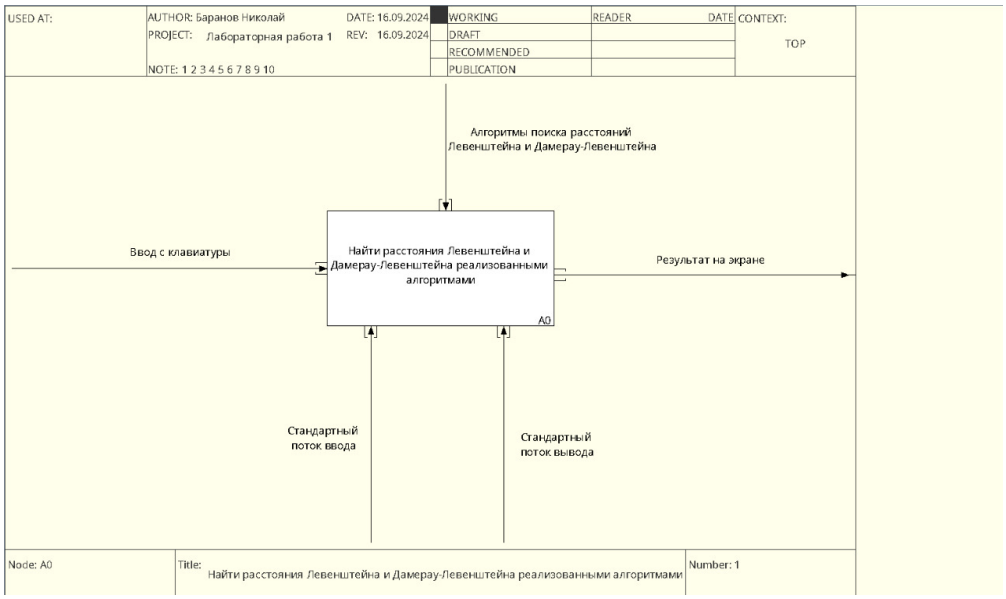

Рисунок 2.1 – Верхний уровень функциональной схемы

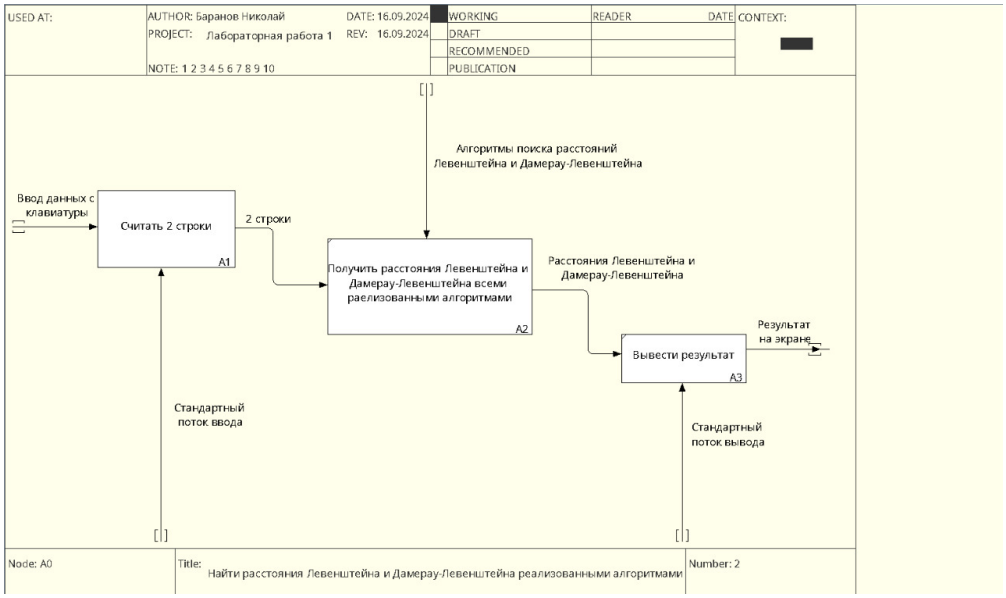

Рисунок 2.2 – Декомпозиция уровня A0

2.2 Схемы алгоритмов

На вход каждому из алгоритмов подаются строки first и second длиной n и m соответственно. Алгоритм с мемоизацией также будет получать матрицу cache размером n+1 на m+1.

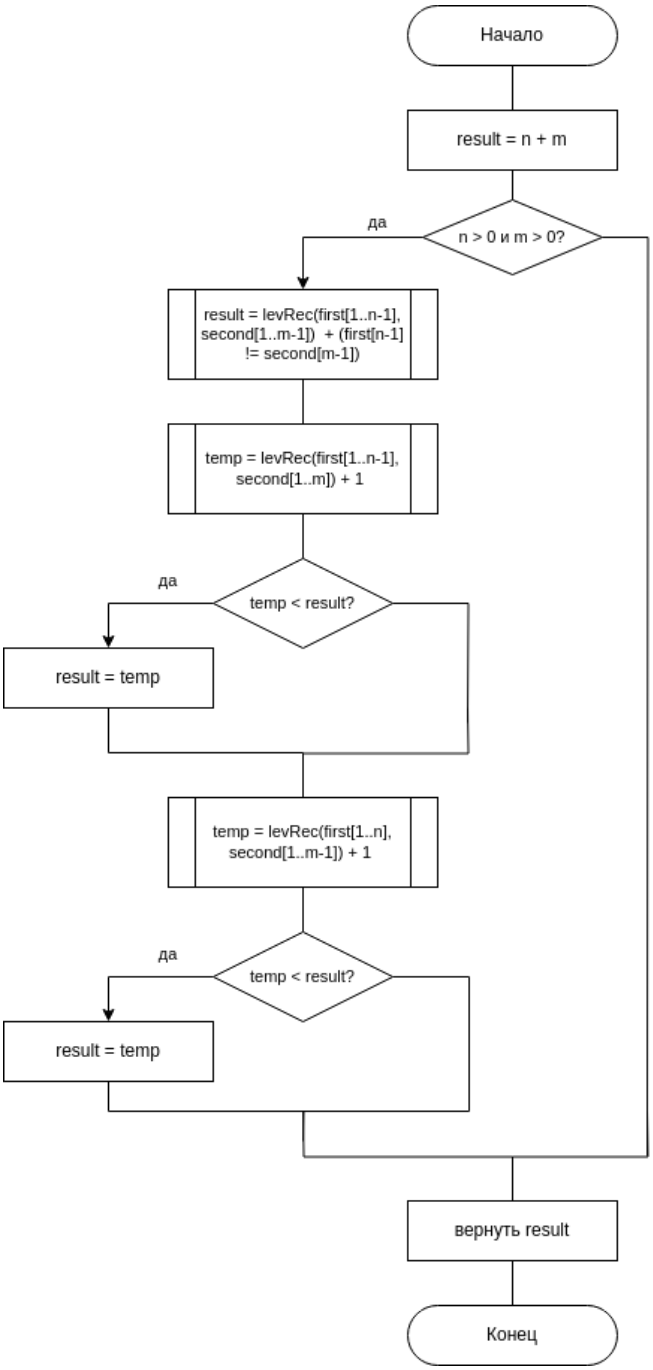

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна

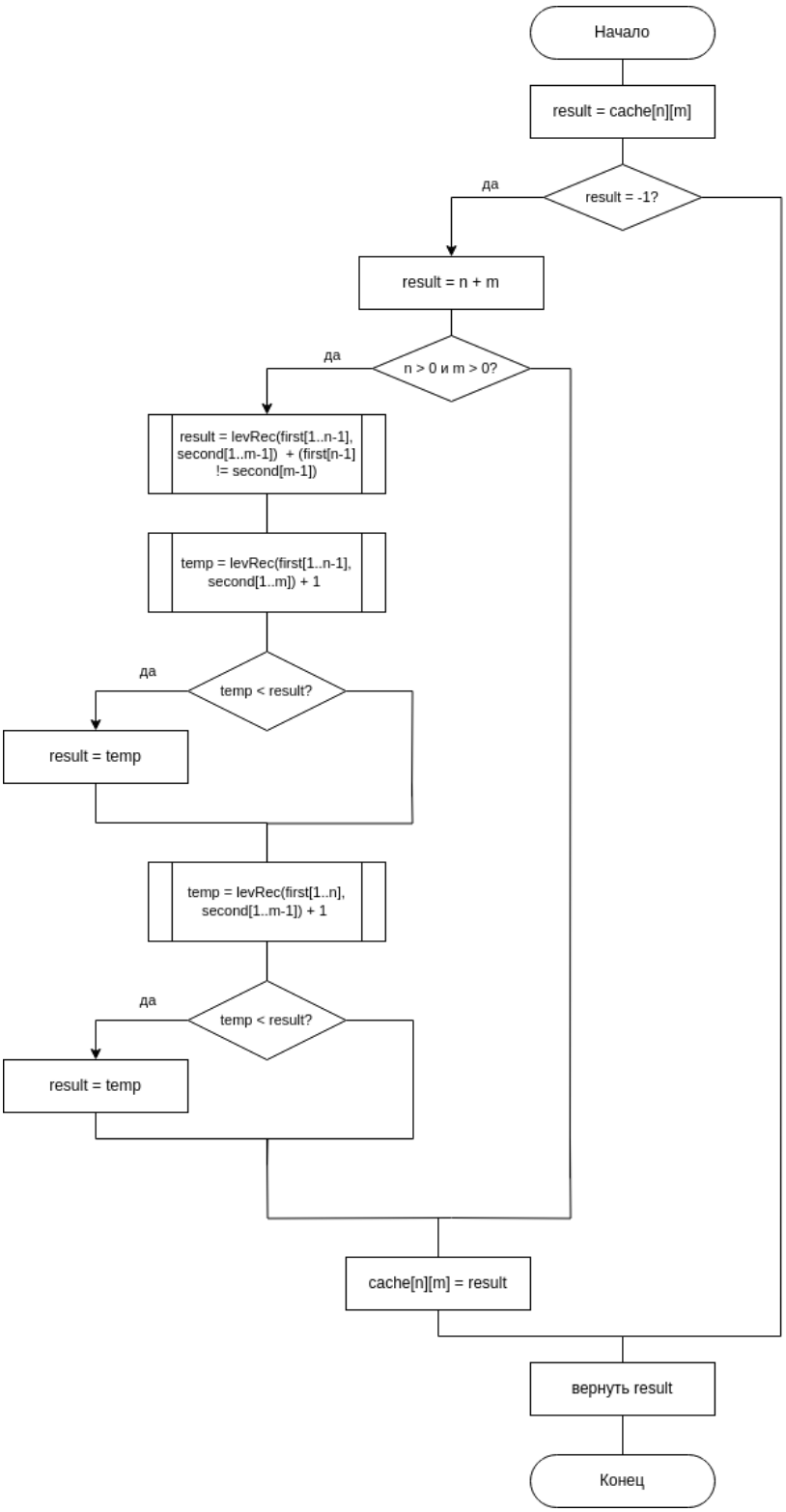

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна с мемоизацией

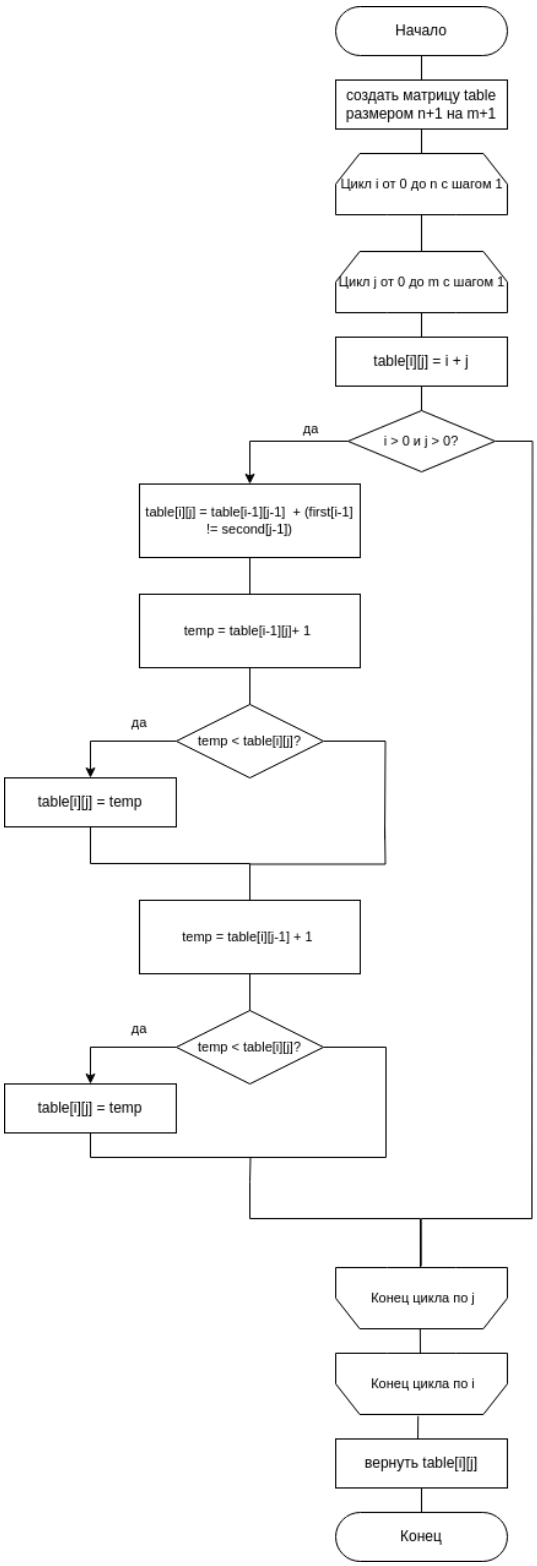

Рисунок 2.5 – Схема матричного алгоритма нахождения расстояния Левенштейна

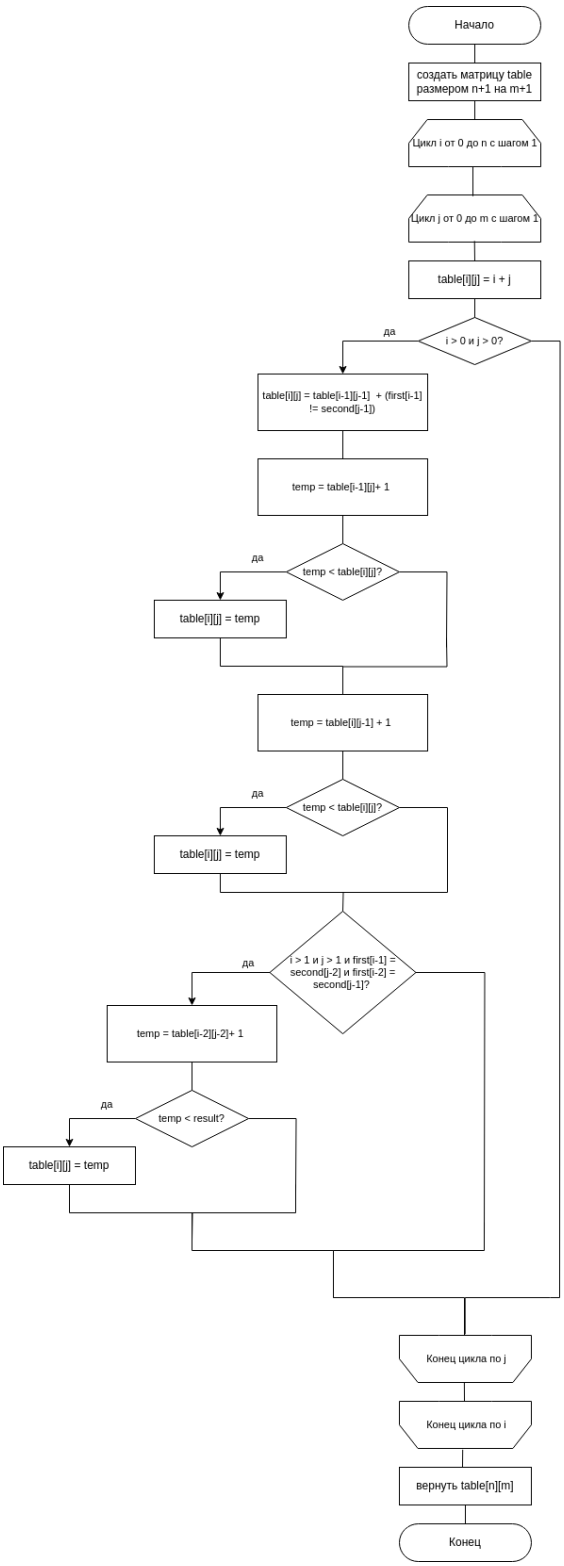

Рисунок 2.6 – Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

2.3 Используемые типы и структуры данных

При реализации алгоритмов будут использованы следующие структуры данных:

— строка — массив символов;

— длина строки — целое беззнаковое число;

— матрица — двумерный массив целых беззнаковых чисел.

2.4 Выводы

В данном разделе была представлена функциональная схема работы программы, а также были построены схемы алгоритмов и выбраны структуры данных.

3 Технологическая часть

3.1 Средства реализации

Для реализации данной лабораторной работы был выбран язык C++, так как, во-первых, уже имеется опыт написания программ на данном языке, а во-вторых, он содержит необходимые средства для реализации алгоритмов.

3.2 Реализация алгоритмов

В листингах представлены реализации алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Метод рекурсивного нахождения расстояния Левенштейна

```
1  int LevenshteinRecursiveAlgorithm::execute() {
2      return count(first.size(), second.size());
3  }
4
5  int LevenshteinRecursiveAlgorithm::count(const int s1, const
       int s2) {
6      auto result = s1 + s2;
7      if (s1 > 0 && s2 > 0) {
8          result = std::min({count(s1 - 1, s2) + 1, count(s1, s2
               - 1) + 1,
9              count(s1 - 1, s2 - 1) + (first[s1 - 1] != second[s2
                   - 1])});
10     }
11     return result;
12 }
```

Листинг 3.2 – Метод матричного нахождения расстояния Левенштейна

```
1  int LevenshteinTableAlgorithm::execute() {
2      auto s1 = first.size() + 1;
3      auto s2 = second.size() + 1;
4      table = initMatrix(s1, s2);
5      for (auto i = 0; i < s1; ++i) {
```

```
6           for (auto j = 0; j < s2; ++j) {
7               table[i][j] = i + j;
8               if (i > 0 && j > 0) {
9                   table[i][j] = std::min({table[i - 1][j] + 1,
                        table[i][j - 1] + 1,
10                      table[i - 1][j - 1] + (first[i - 1] !=
                            second[j - 1])});
11              }
12          }
13      }
14      if (isNeedPrintMatrix()) {
15          printMatrix(table);
16      }
17      return table[first.size()][second.size()];
18  }
```

Листинг 3.3 – Метод рекурсивного нахождения расстояния Левенштейна с мемоизацией

```
1   int LevenshteinMemoisedAlgorithm::execute() {
2       cache = initMatrix(first.size() + 1, second.size() + 1);
3       auto result = count(first.size(), second.size());
4       if (isNeedPrintMatrix()) {
5           printMatrix(cache);
6       }
7       return result;
8   }
9
10  int LevenshteinMemoisedAlgorithm::count(const int s1, const int
        s2) {
11      if (cache[s1][s2] == -1) {
12          cache[s1][s2] = s1 + s2;
13          if (s1 > 0 && s2 > 0) {
14              cache[s1][s2] = std::min({count(s1 - 1, s2) + 1,
                    count(s1, s2 - 1) + 1,
15                  count(s1 - 1, s2 - 1) + (first[s1 - 1] !=
                        second[s2 - 1])});
16          }
17      }
18      return cache[s1][s2];
19  }
```

Листинг 3.4 – Метод матричного нахождения расстояния Дамерау-Левенштейна

```
int DamerauLevenshteinAlgorithm::execute() {
    auto s1 = first.size() + 1;
    auto s2 = second.size() + 1;
    table = initMatrix(s1, s2);
    for (auto i = 0; i < s1; ++i) {
        for (auto j = 0; j < s2; ++j) {
            table[i][j] = i + j;
            if (i > 0 && j > 0) {
                table[i][j] = std::min({table[i - 1][j] + 1,
                    table[i][j - 1] + 1,
                    table[i - 1][j - 1] + (first[i - 1] !=
                        second[j - 1])});
                if (i > 1 && j > 1 && first[i - 1] == second[j
                    - 2] && first[i - 2] == second[j - 1]) {
                    table[i][j] = std::min(table[i][j], table[i
                        - 2][j - 2] + 1);
                }
            }
        }
    }
    if (isNeedPrintMatrix()) {
        printMatrix(table);
    }
    return table[first.size()][second.size()];
}
```

3.3 Функциональные тесты

В таблице 3.1 представлены функциональные тесты и результаты их выполнения. Для наглядности все числа помещены в 1 строку, а вывод таблиц не учитывался.

№	first	second	Ожидаемый результат	Полученный результат	Описание теста
1			0 0 0 0	0 0 0 0	Пустой ввод
2	Слово	Слово	0 0 0 0	0 0 0 0	Одинаковые слова
3	один	семь	4 4 4 4	4 4 4 4	Слова, не имеющие ни одной одинаковой буквы
4	бобер	ребро	4 4 4 4	4 4 4 4	Слова, имеющие одинаковые расстояния Левенштейна и Дамерау-Левенштейна
5	лоск	локс	2 2 2 1	2 2 2 1	Слова, имеющие различные расстояния Левенштейна и Дамерау-Левенштейна
6	белка	лак	4 4 4 3	4 4 4 3	Первое слово длиннее второго
7	лак	белка	4 4 4 3	4 4 4 3	Второе слово длиннее первого

Таблица 3.1 – Таблица тестовых данных

3.4 Вывод

В данном разделе был выбран язык программирования для написания программы, были рализованы все ранее описанные алгоритмы, а также были описаны тесты с ожидаемым и полученным результатом.

4 Исследовательская часть

4.1 Замеры времени работы

Для замеров времени работы функции запускались 10000 раз, на каждой итерации генерировались 2 строки одинаковой длины, состоящие из случайных строчных символов английского алфавита. Результаты измерений суммировались, после чего выводилось среднее значение. Время работы было замерено с помощью функции $clock\_gettime()$ со значением первого параметра $CLOCK\_PROCESS\_CPUTIME\_ID$ [3]. Все замеры проводились на ноутбуке Acer Swift 3x, процессор 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz. Во время замеров были отключены все программы, кроме Visual Studio Code.

При всех используемых вариантах замеры проводились на строках длины 2, 4, 6, 8, 10. Результаты представлены на рисунке 4.1.

Рисунок 4.1 – Результаты замера времени работы разных алгоритмов на строках длины 2, 4, 6, 8, 10.

Как видно из результатов замера, даже при небольшом размере строки рекурсивный алгоритм поиска расстояния Левенштейна работает на-

много дольше остальных и не позволяет нам сравнить их между собой.

При исключении рекурсивного варианта замеры проводились на строках длины 500, 1000, 1500, 2000, 2500. Результаты представлены на рисунке 4.2.

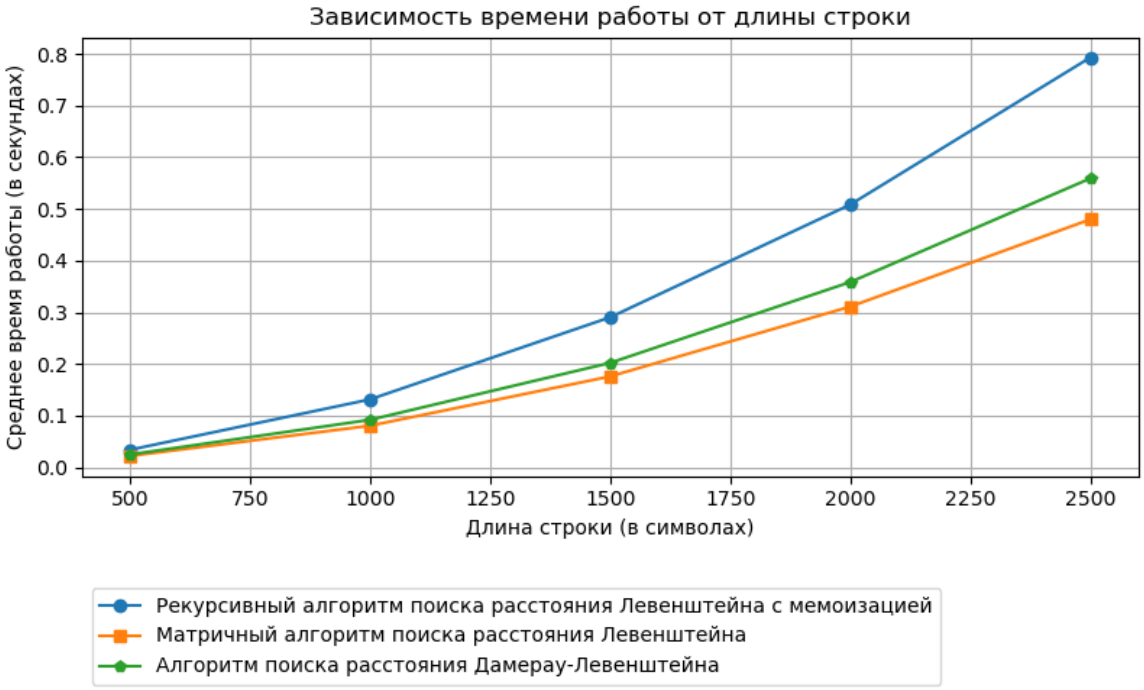

Рисунок 4.2 – Результаты замера времени работы разных алгоритмов (за исключением рекурсивного) на строках длины 500, 1000, 1500, 2000, 2500.

Наиболее эффективной по времени оказалась матричная реализация алгоритма поиска расстояния Левенштейна, а дольше всех среди перечисленных работает рекурсивный алгоритм с мемоизацией.

4.2 Вывод

В данном разделе были проведены замеры времени. Самыми эффективными оказались матричные реализации алгоритмов, причём расстояние Левенштейна считается быстрее, чем расстояние Дамерау-Левенштейна. Далее идёт рекурсивный алгоритм с мемоизацией, и самым медленной оказалась обычная рекурсивная реализация.

Заключение

В ходе выполнения лабораторной работы были рассмотрены рекурсивный, матричный и рекурсивный с мемоизацией способы нахождения расстояний Левенштейна, а также матричная реализация алгоритма поиска Дамерау-Левенштейна. Была написана программа, реализующая эти алгоритмы. Для замера процессорного времени работы была выбрана функция $clock\_gettime()$. Также было проведено сравнение эффективности работы разных реализаций по времени. Наиболее эффективными оказались матричные реализации алгоритмов, при этом функции поиска расстояния Левенштейна работали быстрее, чем функции поиска расстояния Дамерау-Левенштейна.

Список использованных источников

1 Погорелов Д. А. Тарзанов А. М. Волкова Л. Л. Сравнительный анализ алгоритмов редакционного расстояния Левенштейна и Дамерау-Левенштейна. Международный научный журнал «Синергия наук», 2019. — С. 803–811.

2 Черненький В. Е. Гапанюк Ю. Е. Методика идентификации пассажира по установочным данным. Вестник МГТУ им. Н. Э. Баумана, 2012. — С. 31–34.

3 *clock_gettime*(3) - Linux manual page [Электронный ресурс]. Режим доступа: `https://man7.org/linux/man-pages/man3/clock_gettime.3.html` (Дата обращения: 15.09.2024).