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 дисциплине "Анализ Алгоритмов"

Тема

Студент Козырнов А.Д.

Группа ИУ7-52Б

Преподаватель Волкова Л. Л., Строганов Д.В.

2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Расстояние Левенштейна и Дамерау-Левенштейна - это минимальное количество операций (вставка, удаление и замена символов), требуемых для преобразования одной строки в другую. Расстояние Левенштейна используется для исправления ошибок в словах, поиска дубликатов текстов, сравнения геномов и прочих полезных операций с символьными последовательностями.

Цель лабораторной работы — сравнение алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна. Для достижения поставленной цели необходимо выполнить следующие задачи:

- реализовать указанные алгоритмы поиска расстояний (один алгоритм с использованием рекурсии, два алгоритма с использованием динамического программирования);
- проанализировать рекурсивную и матричную реализации алгоритмов по затраченному процессорному времени и памяти на основе экспериментальных данных;
- описать и обосновать полученные результаты в отчете.

1 Аналитическая часть

В данном разделе будут рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна.

1.1 Описание алгоритмов

Расстояние Левенштейна — это минимальное число односимвольных преобразований (удаления, вставки или замены), необходимых для превращения одной строки в другую [1].

1.1.1 Расстояние Левенштейна

Для двух строк S_1 и S_2, представленных в виде списков символов, длинной M и N соответственно, расстояние Левенштейна рассчитывается по рекуррентной формуле 1.1:

$$D(S_1, S_2) = \begin{cases} len(S_1), & \text{если } len(S_2) = 0 \\ len(S_2), & \text{если } len(S_1) = 0 \\ D(tail(S_1), tail(S_2)), & \text{если } head(S_1) = head(S_2) \\ 1 + min \begin{cases} D(tail(S_1), S_2), \\ D(S_1, tail(S_2)), \\ D(tail(S_1), tail(S_2)), \end{cases} & \text{иначе} \end{cases} \tag{1.1}$$

где:

- $len(S)$ — длина списка S;
- $head(S)$ — первый элемент списка S;
- $tail(S)$ — список S без первого элемента;

1.1.2 Расстояние Дамерау-Левенштейна

В алгоритме поиска расстояния Дамерау-Левенштейна, помимо вставки, удаления, и замены присутствует операция перестановки символов. Расстояние Дамерау-Левенштейна может быть вычисленно по рекуррентной формуле 1.2:

$$D(S1, S2) = \begin{cases} len(S_1), & \text{если } len(S_2) = 0 \\ len(S_2), & \text{если } len(S_1) = 0 \\ D(tail(S_1), tail(S_2)), & \text{если } head(S_1) = head(S_2) \\ 1 + min \begin{cases} D(tail(S1), S2), \\ D(S1, tail(S2)), \\ D(tail(S1), tail(S2)), \end{cases} \\ 1 + min \begin{cases} D(tail(tail(S1)), tail(tail(S2))), & (*) \\ D(tail(S1), S2), \\ D(S1, tail(S2)), \\ D(tail(S1), tail(S2)), \end{cases} \end{cases} \tag{1.2}$$

(*): если head(S1) = head(tail(S2)) и head(S2) = head(tail(S1));

ВЫВОД

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна.

2 Конструкторская часть

В данном разделе будут приведены схемы алгоритмов нахождения расстояния Левенштейна и Дамерау-левенштейна.

2.1 Схемы алгоритмов

Алгоритмы на вход получают строчки S_1 и S_2, а на выходе возвращают целое число, которое показывает расстояние.

На рисунках 2.1 — 2.3 представлены схемы алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

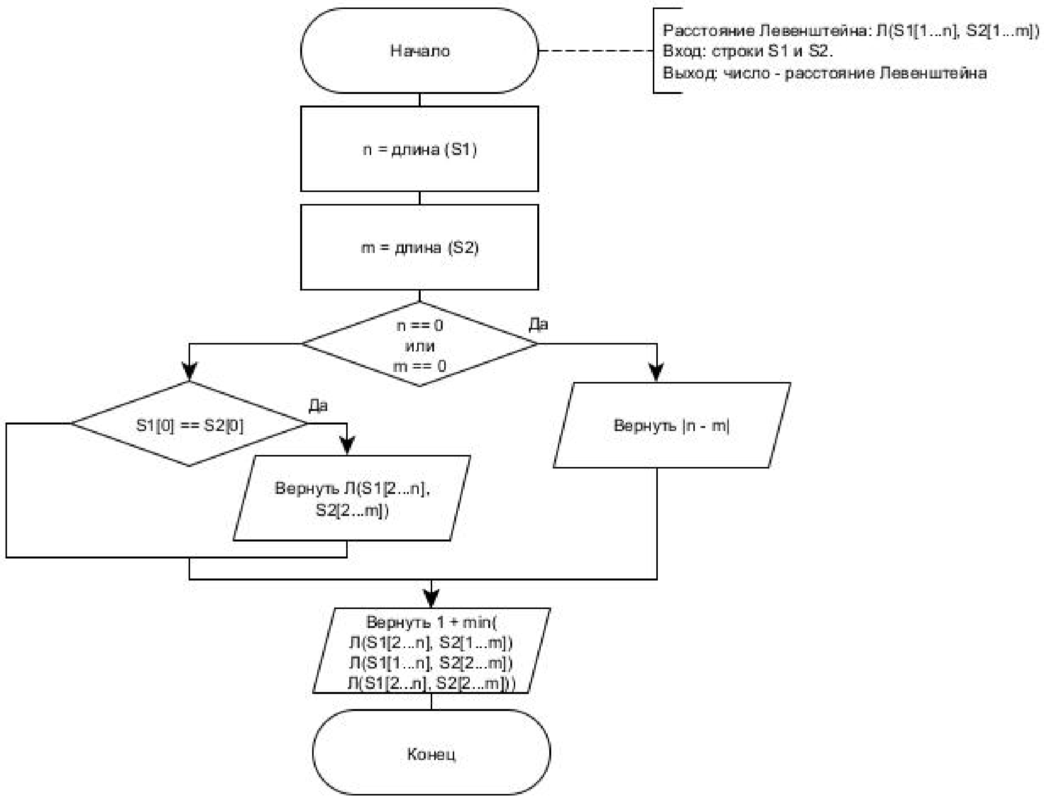

Рисунок 2.1 – Схема рекурсивного алгоритма расстояния Левенштейна

ВЫВОД

В данном разделе были представлены схемы алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

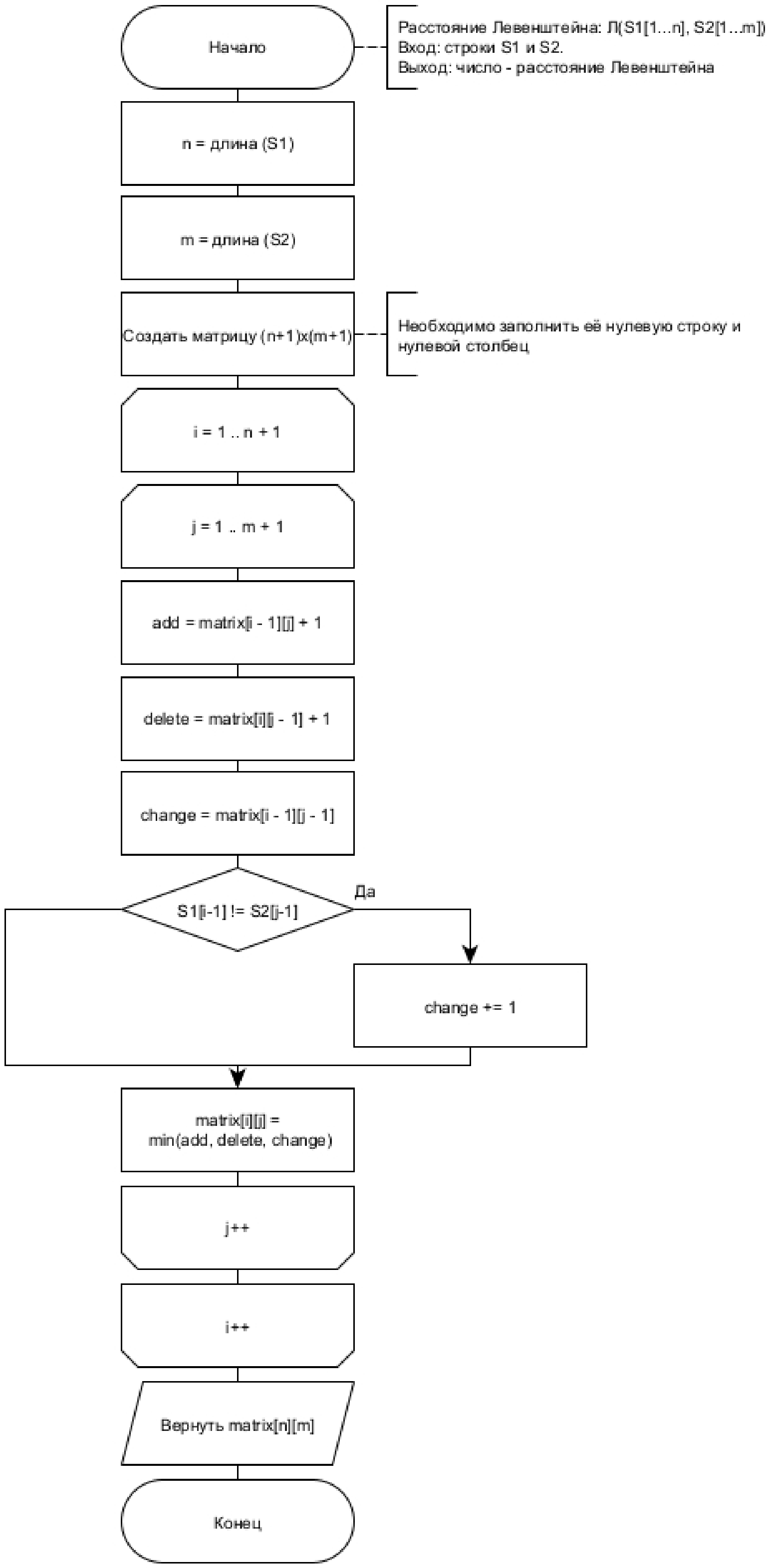

Рисунок 2.2 – Схема динамического алгоритма расстояния Левенштейна

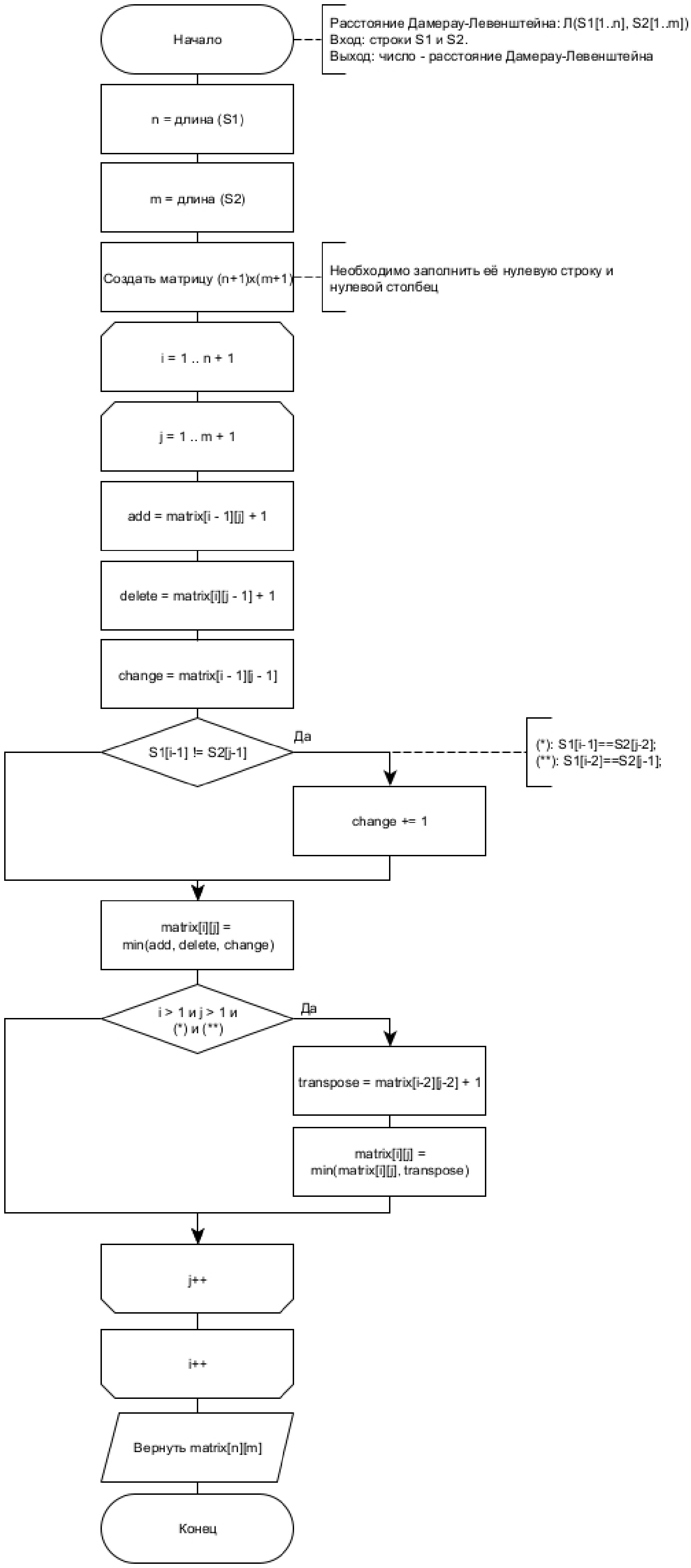

Рисунок 2.3 – Схема динамического алгоритма расстояния Дамерау-Левенштейна

3 Технологическая часть

В данном разделе будут приведены требования к ПО, реализация алгоритмов и средства реализации.

3.1 Требования к программному обеспечению

Входные данные: две строчки из латинских или кириллических символов;

Выходные данные: целое число, которое является искомым расстоянием между двумя строчками.

3.2 Средства реализации

Для реализации был выбран ЯП Python [2]. Выбор обсуловлен наличием функции вычисления процессорного времени в библиотеке time [3]. Время было замерено с помощью функции process_time().

3.3 Реализация алгоритмов

В листингах 3.1 — 3.3 представлены реализации алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Реализация рекурсивного алгоритма нахождения расстояния Левенштейна

```
def tail(string : str):
    return string[1:]

def head(string : str):
    return string[0]


def lvsh_recursive(string1 : str, string2 : str):
    if len(string1) == 0:
        return len(string2)
```

```
    if len(string2) == 0:
        return len(string1)

    if head(string1) == head(string2):
        return lvsh_recursive(tail(string1), tail(string2))
    else:
        a = lvsh_recursive(string1, tail(string2)) + 1
        b = lvsh_recursive(tail(string1), string2) + 1
        c = lvsh_recursive(tail(string1), tail(string2)) + 1

        return min(a, b, c)
```

Листинг 3.2 – Реализация динамического алгоритма нахождения расстояния Левенштейна

```
def lvsh_matrix_min(matrix, i, j):
    return min(matrix[i - 1][j], matrix[i][j - 1], matrix[i - 1][j - 1])

def lvsh_dynamic(string1 : str, string2 : str):
    lenstr1 = len(string1)
    lenstr2 = len(string2)
    if lenstr1 == 0:
        return lenstr2
    if lenstr2 == 0:
        return lenstr1

    matrix = [[0 for _ in range(lenstr2 + 1)] for _ in range(lenstr1 +
        1)]
    for i in range(1, lenstr1 + 1):
        matrix[i][0] = i
    for j in range(1, lenstr2 + 1):
        matrix[0][j] = j

    for i in range(lenstr1):
        for j in range(lenstr2):
            if string1[i] == string2[j]:
                matrix[i + 1][j + 1] = lvsh_matrix_min(matrix, i + 1, j
                    + 1)
            else:
                matrix[i + 1][j + 1] = lvsh_matrix_min(matrix, i + 1, j
                    + 1) + 1

```

```
    test = min(lenstr1, lenstr2)
    return matrix[test][test] + abs(lenstr1 - lenstr2)
```

Листинг 3.3 – Реализация динамического алгоритма нахождения расстояния Дамерау-Левенштейна

```
def dlvsh_dynamic(string1: str, string2: str):
    n = len(string1)
    m = len(string2)


    matrix = [[0 for _ in range(m + 1)] for _ in range(n + 1)]
    for i in range(1, n + 1):
        matrix[i][0] = i
    for j in range(1, m + 1):
        matrix[0][j] = j

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            A = matrix[i - 1][j] + 1
            B = matrix[i][j - 1] + 1
            C = matrix[i - 1][j - 1]

            if string1[i - 1] != string2[j - 1]:
                C += 1

            matrix[i][j] = min(A, B, C)

            if i > 1 and j > 1 and string1[i - 1] == string2[j - 2] and
                string1[i - 2] == string2[j - 1]:
                transpose = matrix[i - 2][j - 2] + 1
                matrix[i][j] = min(matrix[i][j], transpose)

    return matrix[n][m]
```

ВЫВОД

В данном разделе рассмотрены средства реализации, требования к ПО и реализации алгоритмов.

4 Исследовательская часть

4.1 Технические характеристики

Характеристики используемого оборудования:

- Операционная система — Linux [4]
- Оперативная память — 16 Гб.
- Процессор — AMD Ryzen 7 5800H with Radeon Graphics (16) @ 4.463GHz [5]

4.2 Описание используемых типов данных

Используемые типы данных:

- Строка - последовательность символов типа *str*
- Длина строки - целое число типа *int*
- Матрица - двумерный массив типа *int*

4.3 Оценка памяти

Рекурсивная версия алгоритма Левенштейна не использует явных структур данных для хранения промежуточных вычислений. Вместо этого каждый вызов функции обрабатывает небольшой фрагмент строк, и функция вызывает саму себя несколько раз. Глубина рекурсии в худшем случае составляет:

$$(len(S_1) + len(S_2)). \tag{4.1}$$

При этом каждый рекурсивный вызов требует хранения локальных переменных. 3 переменные типа int (a, b, c), 2 переменные типа str. В результате, максимальный расход памяти составляет:

$$(len(S_1) + len(S_2)) \cdot (3 \cdot size(\text{int}) + 2 \cdot size(\text{str})), \tag{4.2}$$

где $size$ - функция, вычисляющая размер параметра.

Алгоритм, основанный на динамическом программировании, использует двумерную матрицу размером $len(S_1{+}1){\times}len(S_2{+}1)$. Эта матрица хранит результаты всех промежуточных вычислений (расстояние Левенштейна для всех подстрок). Дополнительно хранятся локальные переменные: 3 переменных типа int, 2 переменные типа str. По итогу расход памяти в данном случае составляет:

$$(len(S_1 + 1) \cdot len(S_2 + 1) \cdot size(\mathrm{int})) + 3 \cdot size(\mathrm{int}) + 2 \cdot size(\mathrm{str})). \tag{4.3}$$

По расходу памяти алгоритм, использующий принцип динамического программирования, проигрывает рекурсивному: максимальный размер используемой памяти в них растёт как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк.

Алгоритм Дамерау-Левенштейна, также реализованный через динамическое программирование, аналогичен по своей структуре алгоритму Левенштейна. Основное отличие заключается в дополнительной проверке для учёта операций перестановки соседних символов. Для этого используется та же матрица размера $len(S_1 + 1) \times (len(S_2 + 1)$, что и в динамическом алгоритме Левенштейна.

Несмотря на добавление новой операции (перестановка), использование памяти остаётся также на уровне произведение длин строк, поскольку не требуется дополнительное пространство для хранения результатов перестановок. Как и в случае с Левенштейном, каждая клетка матрицы заполняется лишь однажды.

4.4 Время выполнения алгоритмов

Замер времени выполнения каждого из алгоритмов находится в таблице 4.1. Замеры для каждого из алгоритмов для одного и того же размера проводились 500 раз , и результаты замеров усреднялись.

Таблица 4.1 – Время работы алгоритмов (в секундах)

Длина строк	Лев рек.	Лев дин.	Дам-Лев дин.
1	1.23e-07	1.24e-07	5.23e-07
2	7.67e-07	1.14e-06	1.01e-06
3	4.13e-06	2.20e-06	2.57e-06
4	1.67e-05	3.97e-06	5.09e-06
5	9.79e-05	6.27e-06	6.37e-06
6	4.29e-04	9.34e-06	9.26e-06
7	2.12e-03	1.17e-05	1.27e-05
8	1.04e-02	1.54e-05	1.71e-05
9	5.62e-02	2.00e-05	2.21e-05

На рисунке 4.1 показаны графики зависимости времени от количества символов для динамических алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

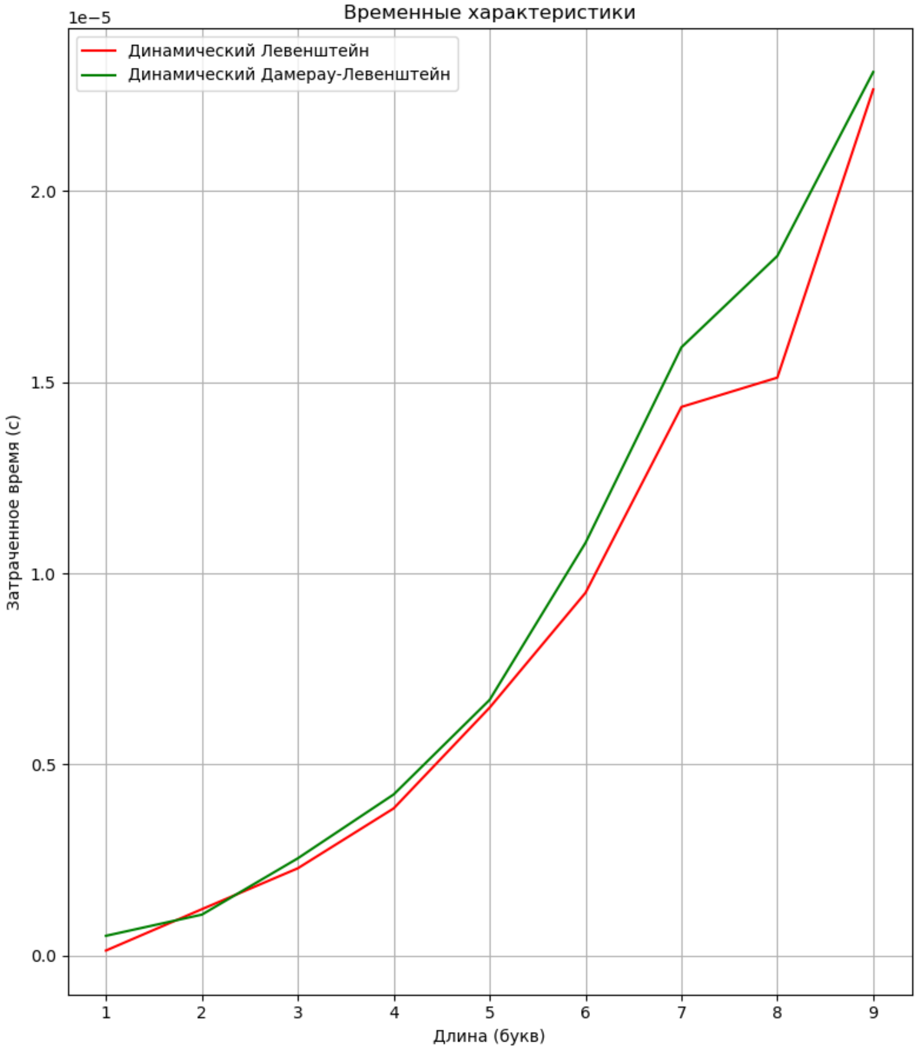

Рисунок 4.1 – Временные показатели динамических алгоритмов Левенштейна и Дамерау-левенштейна

На рисунке 4.2 показаны графики зависимости времени от количества символов для всех реализаций алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

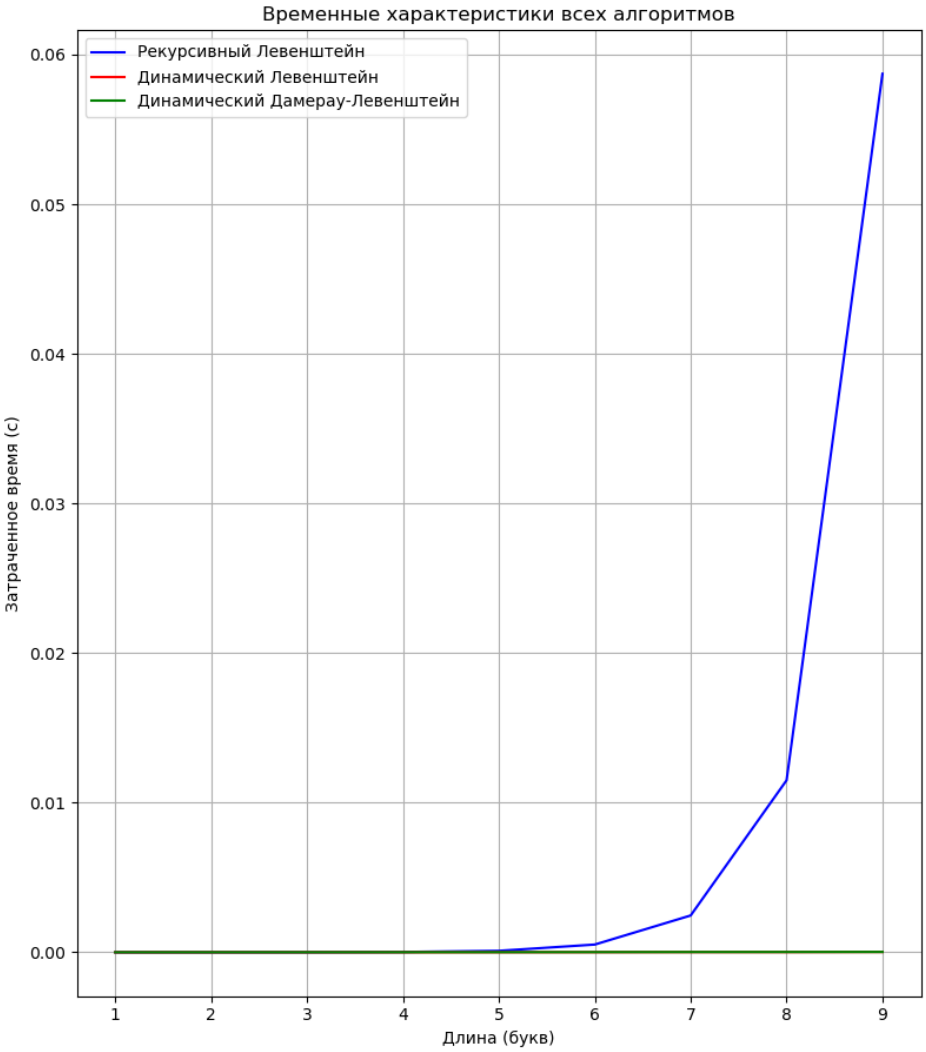

Рисунок 4.2 – Общие временные показатели

Наиболее эффективные реализации алгоритмы нахождения расстояния Левенштейна и Дамерау-левенштейна — это те алгоритмы, где используется динамический подход, так как в рекурсивном подходе идет повторный расчет.

4.5 Вывод

Рекурсивный алгоритм, вычисляющий расстояние Левенштейна, работает по времени на несколько порядков дольше, чем динамический вариант. Также стоит заметить, что динамические алгоритмы вычисления расстояний Левенштейна и Дамерау-Левенштейна сопостовимы между собой по времени выполнения и примерно равны.

Анализ расхода памяти показывает, что рекурсивный алгоритм имеет меньшие требования к памяти по сравнению с алгоритмом, использующим динамическое программирование. В случае динамического варианта алгоритма, считающего расстояние Дамерау-Левенштейна, несмотря на добавление операции перестановки, потребление памяти остается на уровне алгоритма, считающего расстояние Левенштейна, так как не требуется дополнительное пространство для хранения результатов перестановок.

ЗАКЛЮЧЕНИЕ

Было экспериментально подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранных алгоритмов нахождения расстояния между строками при помощи разработаного программного обеспечения на материале замеров процессорного времени выполнения реализаций на различных длинах строк.

В результате исследований можно сделать вывод о том, что матричная реализация данных алгоритмов, по сравнению с рекурсивной, заметно выигрывает по времени при росте длин строк, но проигрывает по количеству затрачиваемой памяти.

В ходе выполнения данной лабораторной работы были решены следующие задачи:

- реализованы указанные алгоритмы для нахождения расстояния Левенштейна (в рекурсивной и динамической вариации), Дамерау-Левенштейна (в динамической);
- проанализированы рекурсивная и динамическая реализации алгоритма Левенштейна, динамические реализации алгоритмов Левенштейна и Дамерау-Левенштейна по затрачиваемым ресурсам (времени и памяти);
- описаны и обоснованы полученные результаты в отчете.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Расстояние Левенштейна для чайников [Электронный ресурс]. URL: `https://habr.com/ru/articles/676858/` (дата обращения: 03.09.2024)

[2] Welcome to Python [Электронный ресурс]. URL: `https://www.python.org` (дата обращения: 07.09.2024).

[3] time — Time access and conversions [Электронный ресурс]. URL: `https://docs.python.org/3/library/time.html#functions` (дата обращения: 07.09.2024).

[4] Arch Linux [Электронный ресурс]. URL: `https://archlinux.org/` (дата обращения: 07.09.2024).

[5] AMD Ryzen 7 5800H Processor. Бенчмарки [Электронный ресурс] URL: `https://www.notebookcheck-ru.com/AMD-Ryzen-7-5800H.519526.0.html` (дата обращения: 07.09.2024).