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 Расстояние Левенштейна и Дамерау-Левенштейна

Студент Смирнов И.В.

Группа ИУ7-52Б

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

2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Расстояние Левенштейна и Дамерау-Левентшейна представляет собой минимальное количество операций (вставка, удаление и замена символов), требуемое 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 [1].

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 расстояний 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 [2].

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:

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

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

ВЫВОД

В данном разделе были рассмотрены два основных алгоритма для вычисления расстояний между строками: расстояние Левенштейна и расстояние Дамерау-Левенштейна.

2 Конструкторская часть

В данном разделе будут приведены схемы алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна, приведено описание используемых типов данных, оценки памяти, а также описана структура ПО.

2.1 Представление алгоритмов

На вход алгоритмов падаются строки S_1 и S_2, на выходе — единственное число, искомое расстояние.

На рис. 2.1 — 2.3 приведены схемы рекурсивных и матричных алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

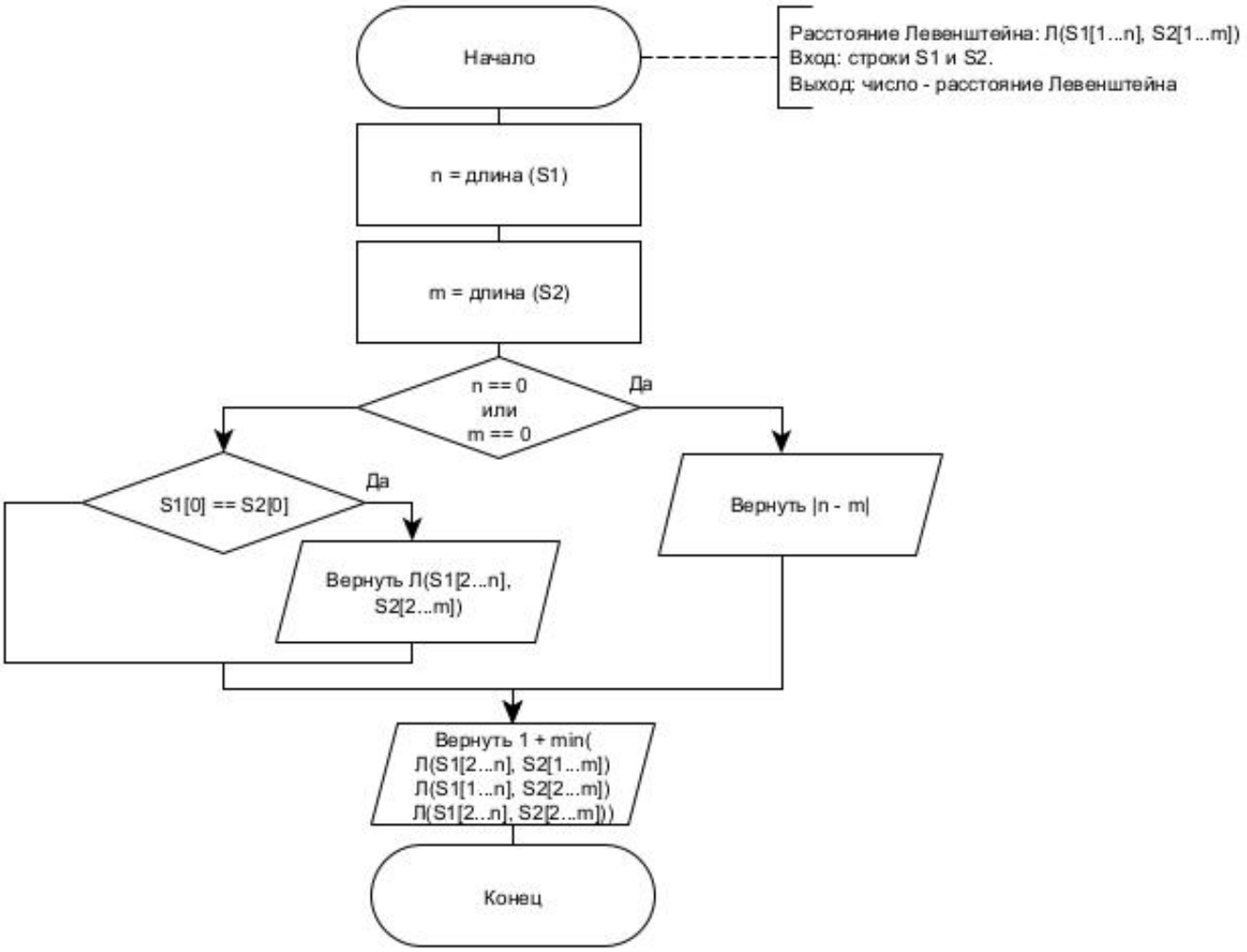

Рисунок 2.1 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна

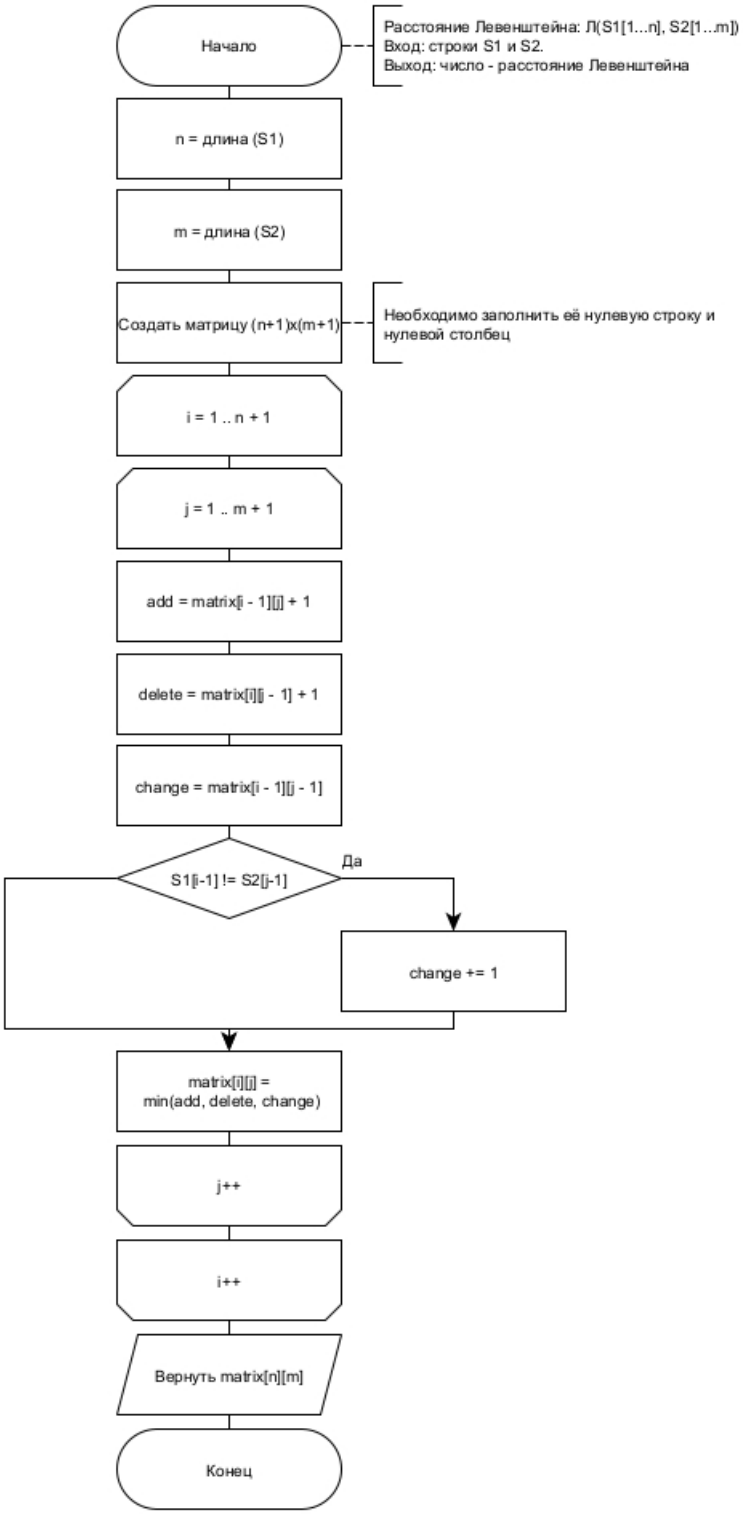

Рисунок 2.2 – Схема динамического алгоритма нахождения расстояния Левенштейна

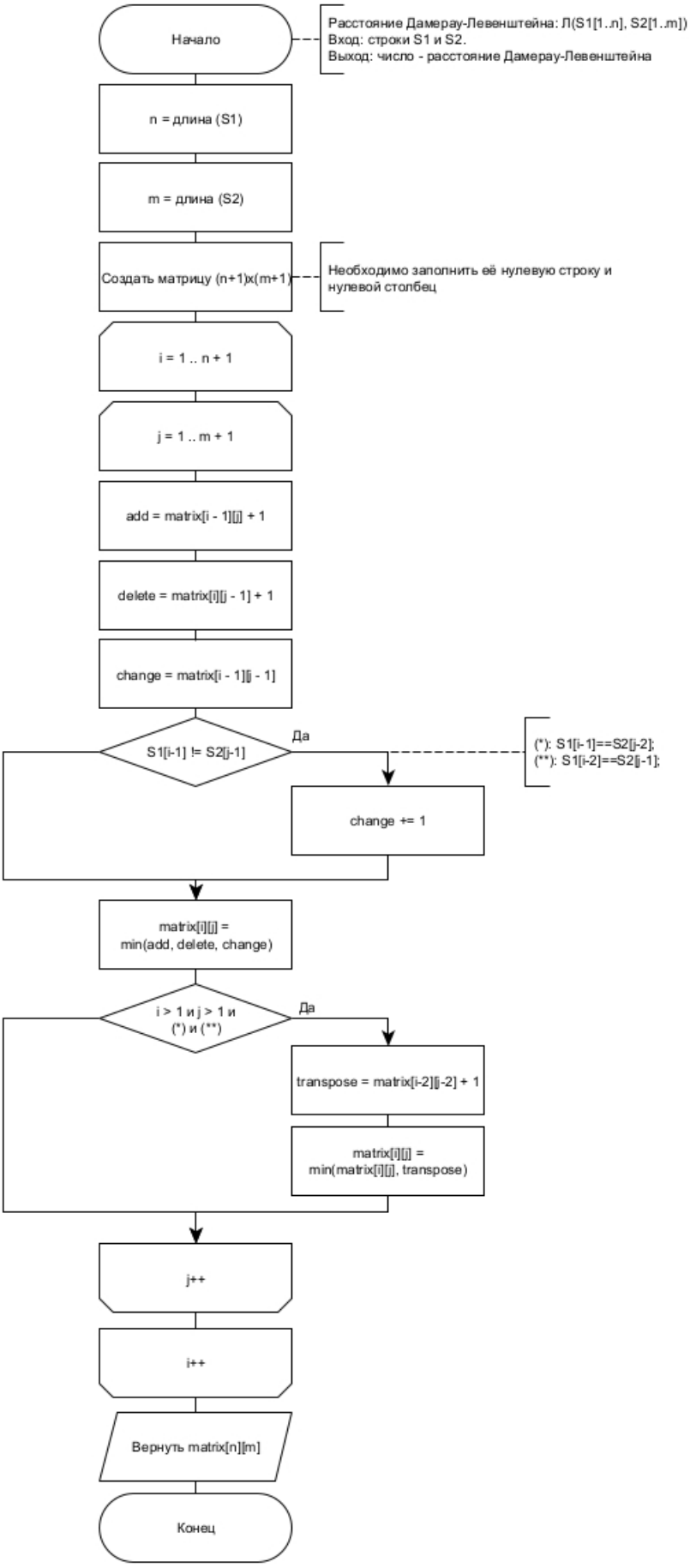

Рисунок 2.3 – Схема динамического алгоритма нахождения расстояния Дамерау-Левенштейна

ВЫВОД

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

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации, листинги кода.

3.1 Требования к программному обеспечению

Входные данные: две строки на русском или английском языке в любом регистре;

Выходные данные: искомое расстояние для выбранного метода и матрицы расстояний для матричных реализаций.

3.2 Средства реализации

В данной работе для реализации был выбран язык программирования *Python* [3]. Выбор обсуловлен наличием функции вычисления процессорного времени в библиотеке *time* [4]. Время было замерено с помощью функции *process_time()*.

3.3 Реализация алгоритмов

В листингах 3.1 - 3.3 представлены реализации алгоритмов нахождения расстояния Левенштейна и Дамерау–Левенштейна.

Листинг 3.1 – Функция нахождения расстояния Левенштейна рекурсивно

```
def levenstein_recursive(str1, str2, output = True):
    n = len(str1)
    m = len(str2)

    if ((n == 0) or (m == 0)):
        return abs(n - m)

    if (str1[0] == str2[0]):
        return levenstein_recursive(str1[1:], str2[1:])

    min_distance = 1 + min(levenstein_recursive(str1[1:], str2),
                    levenstein_recursive(str1, str2[1:]),
                    levenstein_recursive(str1[1:], str2[1:]))

    return min_distance
```

Листинг 3.2 – Функция нахождения расстояния Левенштейна динамически

```
def levenstein_dynamic(str1, str2, output = True):
    n = len(str1)
    m = len(str2)

    matrix = create_lev_matrix(n + 1, m + 1)

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            add = matrix[i - 1][j] + 1
            delete = matrix[i][j - 1] + 1
            change = matrix[i - 1][j - 1]

            if (str1[i - 1] != str2[j - 1]):
                change += 1

            matrix[i][j] = min(add, delete, change)

    if (output):
        print_lev_matrix(matrix, str1, str2)

    return matrix[n][m]
```

Листинг 3.3 – Функция нахождения расстояния Дамерау–Левенштейна динамически

```
def damerau_levenstein_dynamic(str1, str2, output=True):
    n = len(str1)
    m = len(str2)

    matrix = create_lev_matrix(n + 1, m + 1)

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            add = matrix[i - 1][j] + 1
            delete = matrix[i][j - 1] + 1
            change = matrix[i - 1][j - 1]

            if str1[i - 1] != str2[j - 1]:
                change += 1

            matrix[i][j] = min(add, delete, change)

            if i > 1 and j > 1 and str1[i - 1] == str2[j - 2] and
               str1[i - 2] == str2[j - 1]:
                transposition = matrix[i - 2][j - 2] + 1
                matrix[i][j] = min(matrix[i][j], transposition)

    if output:
        print_lev_matrix(matrix, str1, str2)

    return matrix[n][m]
```

ВЫВОД

В данном разделе были рассмотрены требования к программному обеспечению, используемые средства реализации, а также приведены листинги кода для вычисления расстояний Левенштейна (на основе рекурсивного и динамического алгоритмов) и Дамерау-Левенштейна (на основе динамического алгоритма).

4 Исследовательская часть

4.1 Технические характеристики

Характеристики используемого оборудования:

- Операционная система — Windows 11 Home [5]
- Память — 16 Гб.
- Процессор — Intel(R) Core(TM) i5-10300H CPU @ 2.50ГГц [6]

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: 2 переменные типа int, 2 переменные типа str. В результате, максимальный расход памяти составляет:

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

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: 5 переменных типа int, 2 переменные типа str. По итогу расход памяти в данном случае составляет:

$$(len(S_1 + 1) \cdot len(S_2 + 1) \cdot size(\text{int})) + 5 \cdot size(\text{int}) + 2 \cdot size(\text{str})). \tag{4.3}$$

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

Результаты замеров времени работы алгоритмов нахождения расстояний Левенштейна и Дамерау–Левенштейна приведены в таблице 4.1. На рисунке 4.1 приведены графики зависимости времени от количества букв для каждого из алгоритмов. Замеры времени проводились на строках одинаковой длины и усреднялись для каждого набора одинаковых экспериментов. Каждое значение получено путем взятия среднего из 100 измерений.

Таблица 4.1 – Время работы алгоритмов (в секундах)

Длина строк	Лев рек.	Лев дин.	Дам-Лев дин.
1	9.05e-06	7.97e-06	6.99e-06
2	3.36e-05	1.00e-05	1.06e-05
3	1.45e-04	1.42e-05	1.61e-05
4	6.62e-04	1.90e-05	2.30e-05
5	3.47e-03	2.53e-05	3.26e-05
6	2.08e-02	4.82e-05	5.38e-05
7	9.53e-02	4.31e-05	5.63e-05
8	4.85e-01	6.07e-05	8.28e-05

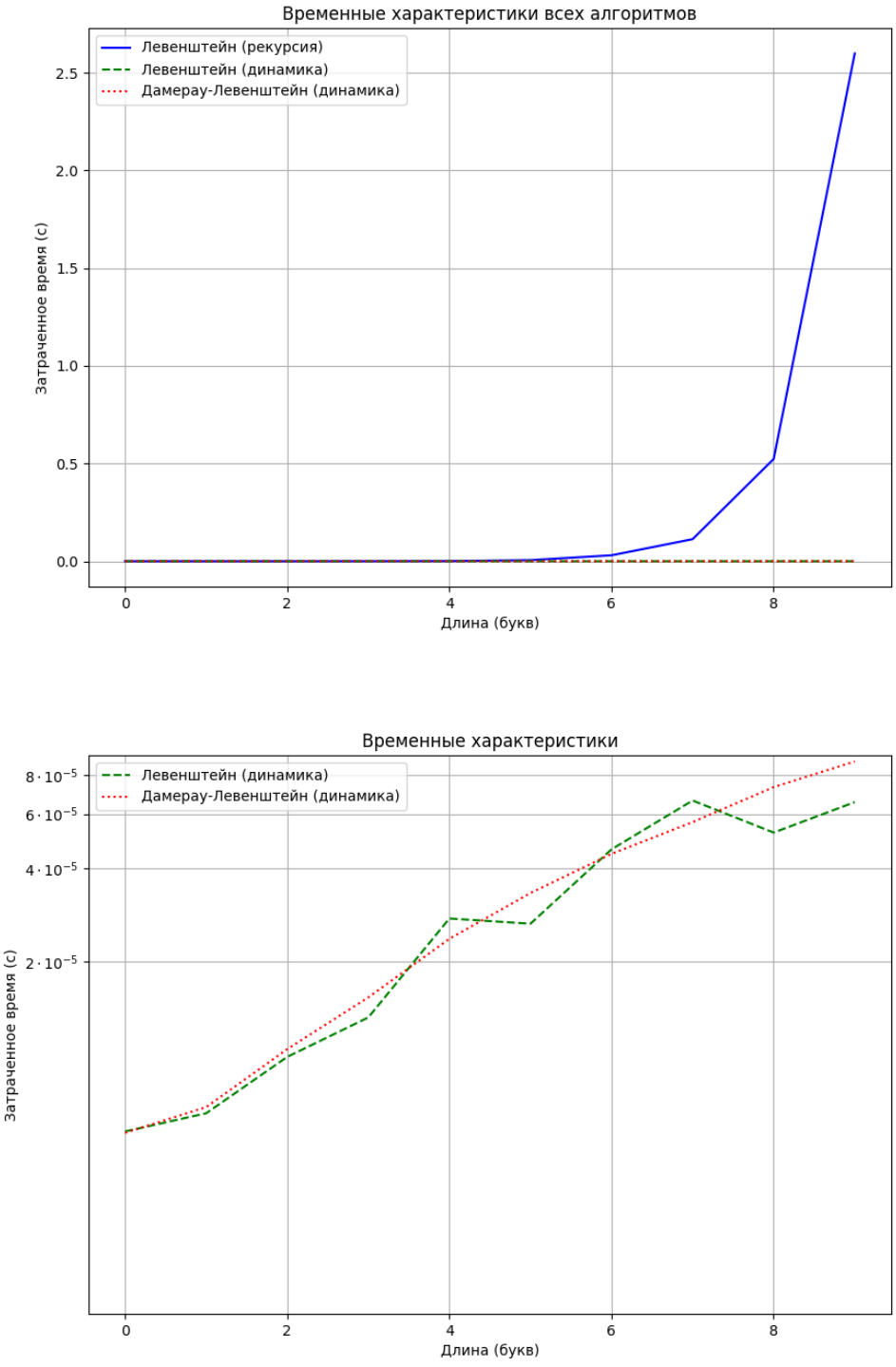

Рисунок 4.1 – Сравнение алгоритмов по времени

Наиболее эффективными являются алгоритмы, использующие динамический подход (матрицу), так как в рекурсивных алгоритмах большое количество повторных расчетов.

4.5 Вывод

Рекурсивный алгоритм, вычисляющий расстояние Левенштейна, работает по времени на несколько порядков дольше, чем динамический вариант (на 3-буквенных словах: в 10 раз; на 8-буквенных строках: в 10000 раз). 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] Levenshtein Distance — an overview [Электронный ресурс]. URL: https://www.sciencedirect.com/topics/computer-science/levenshtein-distance (дата обращения: 08.09.2024).

[2] Levenshtein Distance for Dummies [Электронный ресурс]. URL: https://medium.com/analytics-vidhya/levenshtein-distance-for-dummies-dd9eb83d3e09 (дата обращения: 08.09.2024).

[3] Welcome to Python [Электронный ресурс]. URL: https://www.python.org (дата обращения: 08.09.2024).

[4] time — Time access and conversions [Электронный ресурс]. URL: https://docs.python.org/3/library/time.html#functions (дата обращения: 08.09.2024).

[5] Windows technical documentation for developers and IT pros [Электронный ресурс]. URL: https://learn.microsoft.com/en-us/windows/ (дата обращения: 08.09.2024).

[6] Intel® Core™ i5-10300H Processor [Электронный ресурс]. URL: https://ark.intel.com/content/www/us/en/ark/products/201839/intel-core-i5-10300h-processor-8m-cache-up-to-4-50-ghz.html (дата обращения: 08.09.2024).