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: Расстояние Левенштейна и Дамерау-Левенштейна

Дисциплина: Анализ алгоритмов

Студент	ИУ7-54Б		Л.Е.Тартыков
	(Группа)	(Подпись, дата)	(И.О. Фамилия)
Преподаватель			Л.Л. Волкова
		(Подпись, дата)	(И.О. Фамилия)

Москва, 2021

Содержание

Введение

При наборе текста часто возникает проблема, связанная с опечатками. Необходимы средства, которые позволили бы быстро исправлять эти ошибки.

Для решения подобных задач в прикладной лингвистике выделяется такое направление, как компьютерная лингвистика, в которой разрабатываются и используются компьютерные программы, необходимые для исследования языка и моделирования функционирования языка в тех или иных условиях[1].

Одним из первых, кто занялся такой задачей, был советский ученый В.И.Левенштейн[2]. Алгоритм полученного решения связали с его именем. Расстояние Левенштейна - метрика, измеряющая разность двух строк, определяемая в количестве редакторских операций (вставка, удаление, замена), требуемых для преобразования одной последовательности символов в другую. Модификацией данного алгоритма является расстояние Дамерау-Левенштейна, которая добавляет транспозицию, обмен двух соседних символов, к редакторским операциям. Разработанные алгоритмы нашли применение не только в компьютерной лингвистике, но и в биоинформатике для определения схожести разных участков ДНК и РНК.

Целью лаборатоной работы является изучение и реализация алгоритмов нахождения расстояний Левенштейна и Дамерлау-Левештейна, а также получения навыка динамического программирования. Для её достижения необходимо выполнить следующие задачи:

- изучение алгоритмов Левенштейна и Дамерлау-Левештейна;
- разработать данные алгоритмы;
- применение методов динамического программирования для реализации алгоритмов;
- выполнить тестирование реализации алгоритмов методом черного ящика;

- провести сравнительный анализ этих алгоритмов по затратам памяти и процессорному выполнению времени на основе экспериментальных данных.

1 Аналитический раздел

1.1 Расстояние Левенштейна

Расстояние Левенштейна (редакторское расстояние) между двумя строками - минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую.

При преобразовании одной строки в другую используются следующие операции:

- I (insert) - вставка;
- D (delete) - удаление;
- R (replace) - замена.

Будем считать, что стоимость каждой из этих операции (штраф) равна единице.

Введем еще одну операцию M (match) - совпадение. Её стоимость будет равна нулю.

Необходимо найти последовательность замен с минимальным суммарным штрафом.

1.2 Реккурентный алгоритм

Расстояние между двумя строками s1 и s2 рассчитывается по реккуретной формуле (1.1).

$$D(s1[1..i], s2[1..j]) = \begin{cases} 0, & \text{i=0, j=0} \\ j, & \text{i=0, j>0} \\ i, & \text{i>0, j=0} \\ \min \begin{cases} D(s1[1..i], s2[1..j-1]) + 1 \\ D(s1[1..i-1], s2[1..j]) + 1 \\ D(s1[1..i-1], s2[1..j-1]) + \\ f(s1, s2) \end{cases} \end{cases} \tag{1.1}$$

Функция f(s1, s2) определяется по формуле (1.2).

$$f(s1, s2) = \begin{cases} 0, & \text{s1=s2} \\ 1, & \text{иначе} \end{cases} \tag{1.2}$$

1.3 Матрица расстояний

Реализация алгоритма по формуле (1.1) при больших значениях i,j, оказывается менее эффективной по времени ввиду того, что приходится вычислять промежуточные результаты неоднократно. Для оптимизации нахождения расстояния Левенштейна необходимо использовать матрицу стоимостей для хранения этих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы значениями D(i, j).

1.4 Использование двух строк

Модификацией использования матрицы является использование только двух строк этой матрицы, в которых хранятся промежуточные значения. После выполнения вычислений выполняется обмен значений этих двух строк. Алгоритм продолжает работать и перезаписывать значения только второй строки.

1.5 Рекурсивный алгоритм с кэшем в форме матрицы

При помощи матрицы можно выполнить оптимизацию рекурсивного алгоритма заполнения. Основная идея такого подхода заключается в том, что при каждом рекурсивном вызове алгоритма выполняется заполнение матрицы стоимостей. Главное отличие данного метода от того, что был описан в разделе 1.3 - начальная инициализация матрицы значением ∞. Если рекурсивный алгоритм выполняет вычисления для данных, которые не были обработаны, значение результата минимального расстояния для данного вызова заносится в матрицу. Если рекурсивный вызов уже обрабатывался (ячейка матрицы была заполнена), то алгоритм не выполняет вычислений, а сразу переходит к следующему шагу.

1.6 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна является модификацией расстояния Левенштейна, которая задействует еще одну редакторскую операцию - транспозицию T (transposition). Она выполняет обмен соседних символов в слове.

Дамерау показал, что 80% человеческих ошибок при наборе текстов является перестановка соседних символов, пропуск символа, добавление нового символа или ошибочный символ[3]. Таким образом, расстояние Дамерау-Левенштейна часто используется в редакторских программах для проверки правописания. Это расстояние может быть вычислено по следующей реккуретной формуле (1.3).

$$D(s1[1..i], s2[1..j]) = \begin{cases} 0, & \text{i=0, j=0} \\ j, & \text{i=0, j>0} \\ i, & \text{i>0, j=0} \\ \min \begin{cases} D(s1[1..i], s2[1..j-1]) + 1 \\ D(s1[1..i-1], s2[1..j]) + 1 \\ D(s1[1..i-1], s2[1..j-1]) + f(s1, s2) \\ D(s1[1..i-1], s2[1..j-1]) + 1, \\ \text{ i,j} > 1, a_i = b_{j-1}, a_{i-1} = b_j \\ \infty, \text{ иначе} \end{cases} \end{cases} \tag{1.3}$$

1.7 Вывод

В данном разделе были рассмотрены основные способы нахождения редакторского расстояния между двумя строками. Формулы для нахождения расстояния Левенштейна и Дамерау-Левенштейна задаются реккуретно, следовательно, алгоритмы могут быть реализованы как рекурсивно, так и итерационно.

2 Конструкторский раздел

2.1 Схемы алгоритмов Левенштейна

Ниже представлены следующие схемы алгоритмов:

- рис. 2.1 - схема алгоритма итеративного Левенштейна с использованием двух строк;

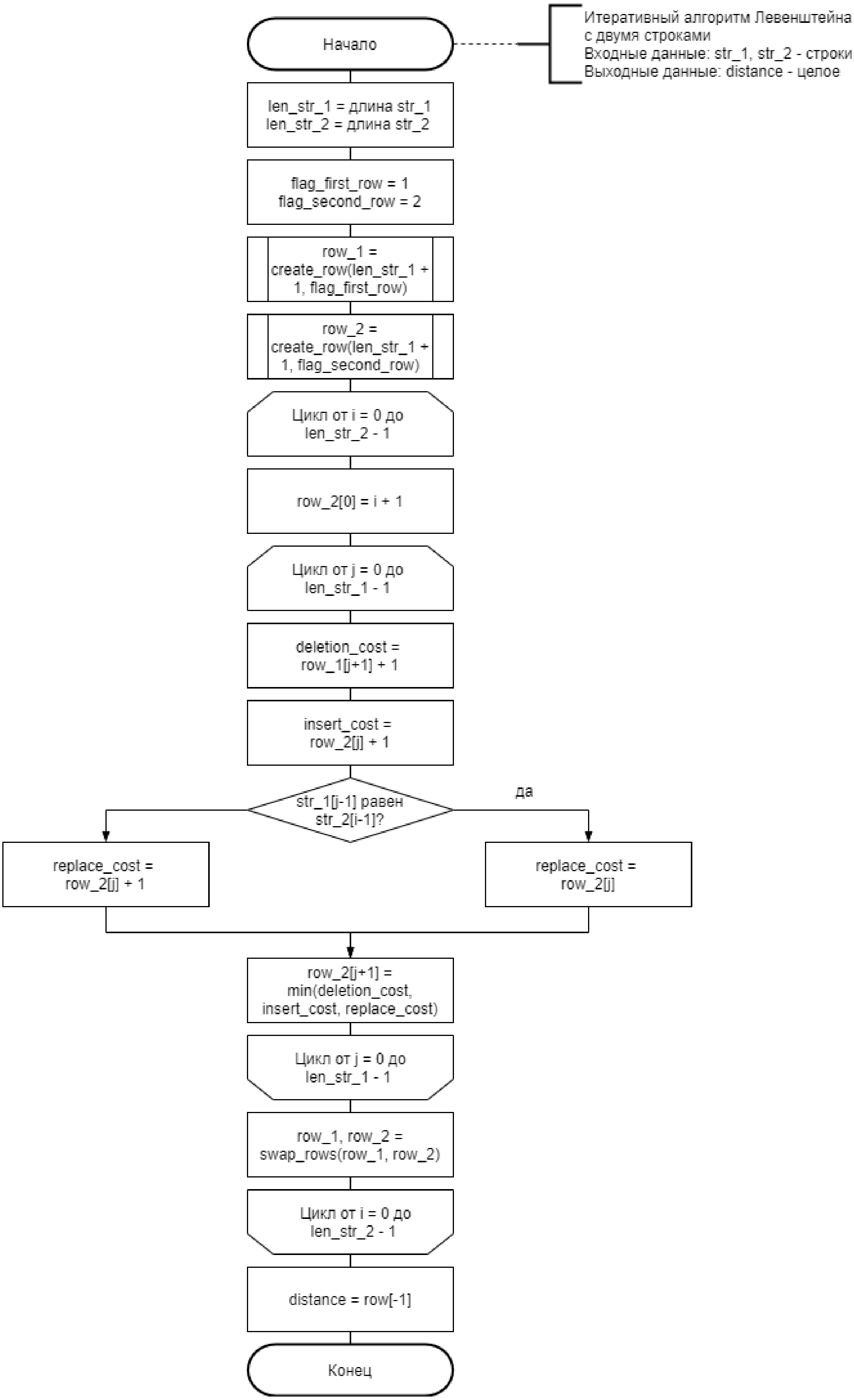

Рисунок 2.1 – Схема алгоритма итеративного Левенштейна с использованием двух строк.

- рис. 2.2 - схема алгоритма рекурсивного Левенштейна без кэша;

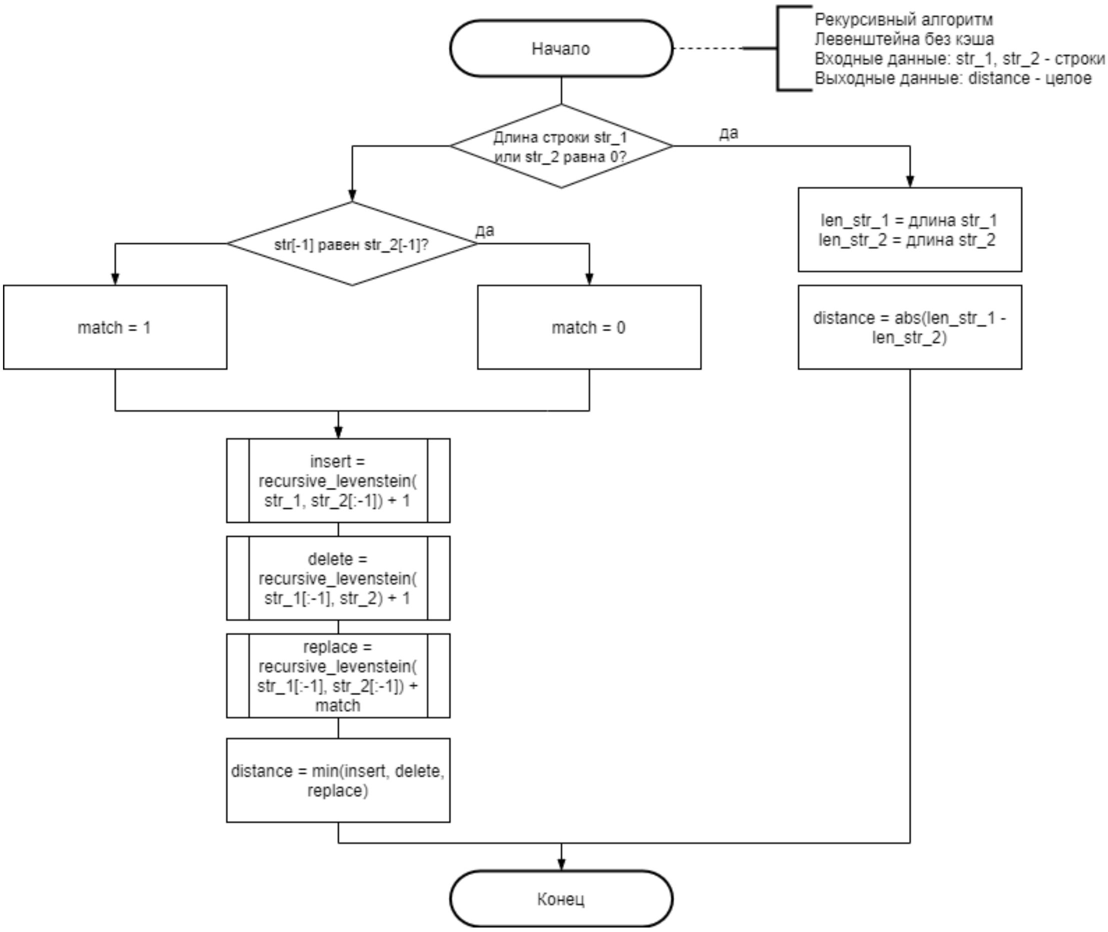

Рисунок 2.2 – Схема алгоритма рекурсивного Левенштейна без кэша.

- рис. 2.3 и рис. 2.4 - схема алгоритма рекурсивного Левенштейна с использованием матрицы;

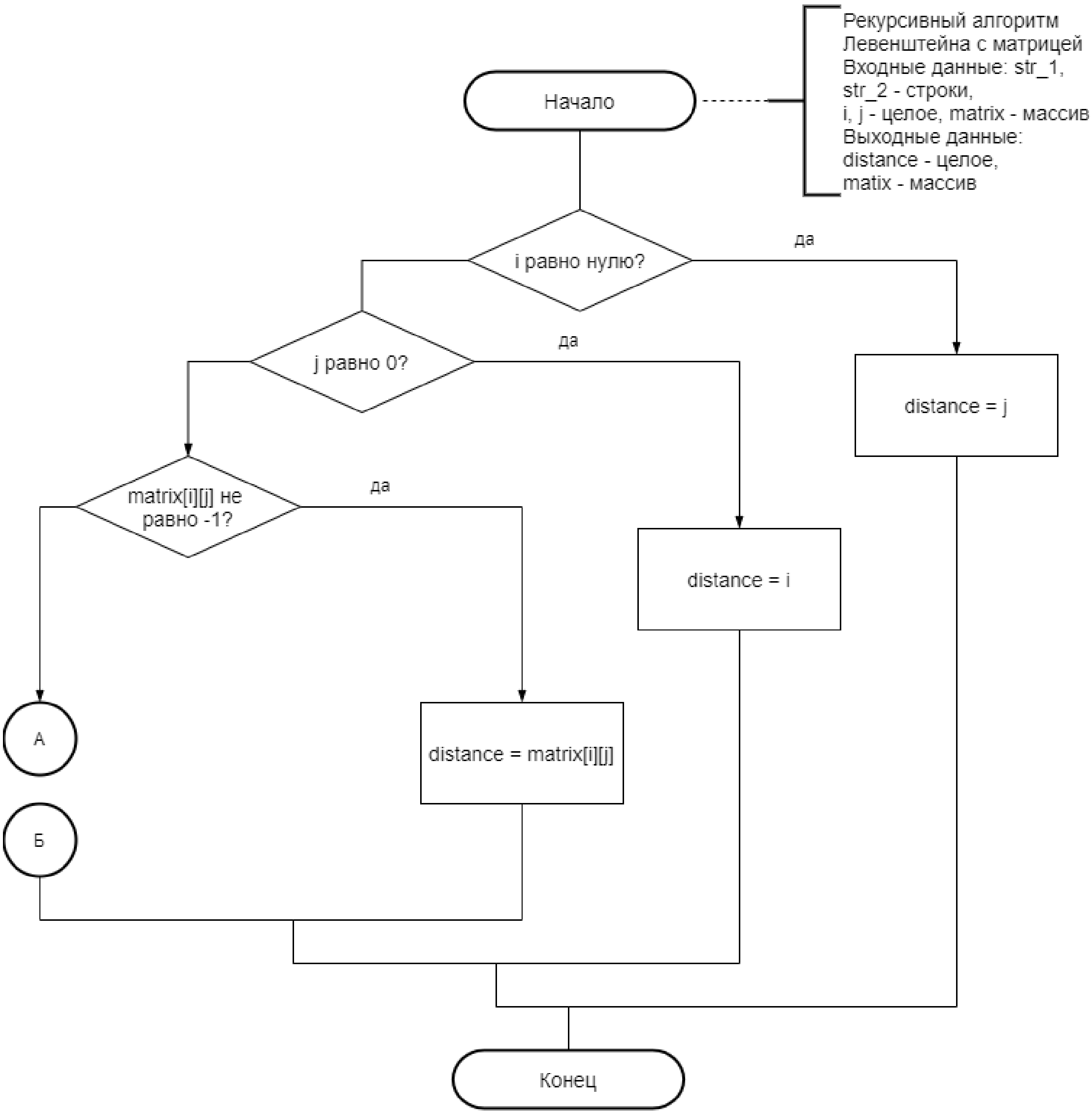

Рисунок 2.3 – Схема алгоритма рекурсивного Левенштейна с использованием матрицы.

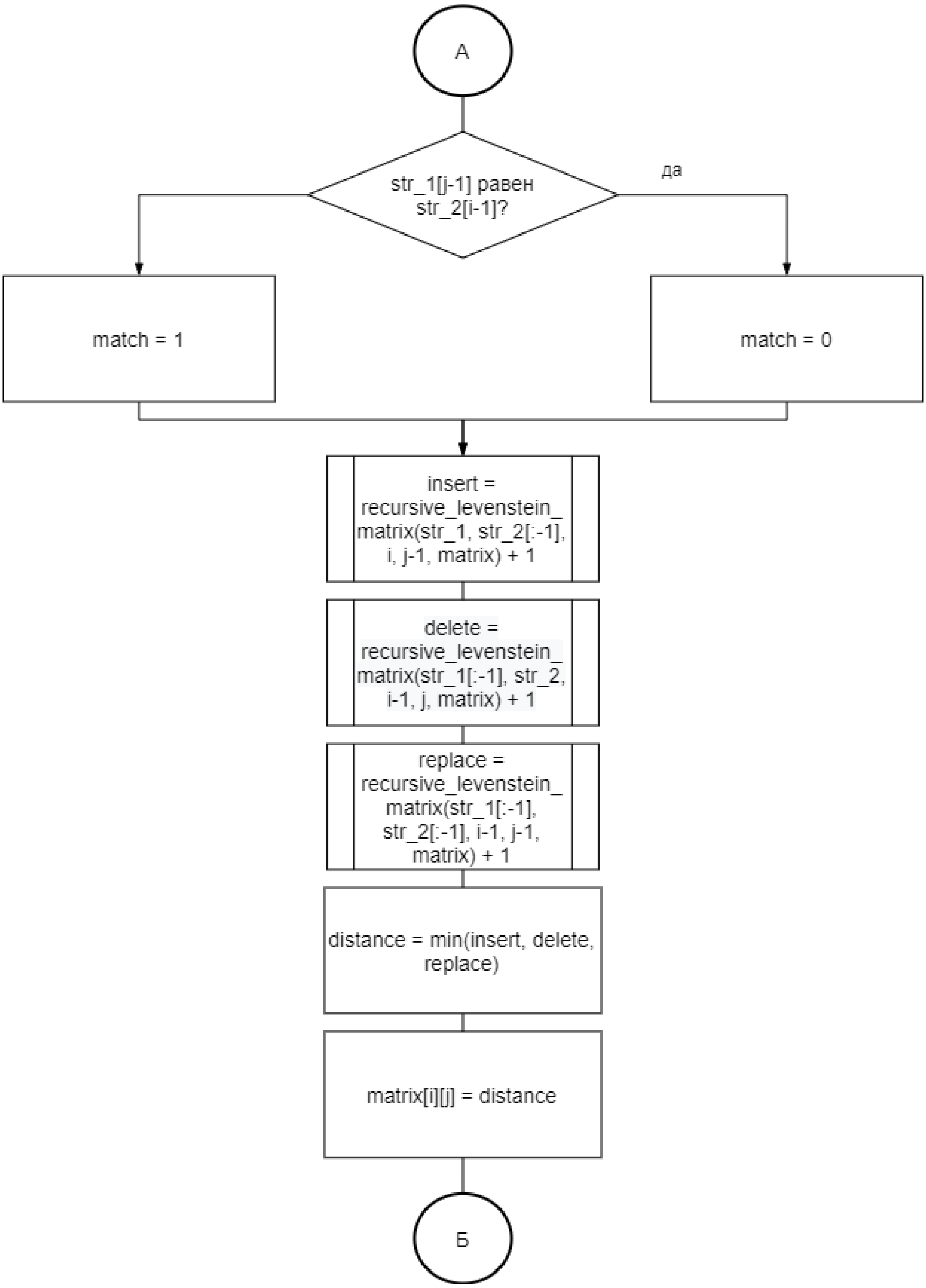

Рисунок 2.4 – Схема алгоритма рекурсивного Левенштейна с использованием матрицы.

- рис. 2.5 - схема алгоритма рекурсивного Дамерау-Левенштейна.

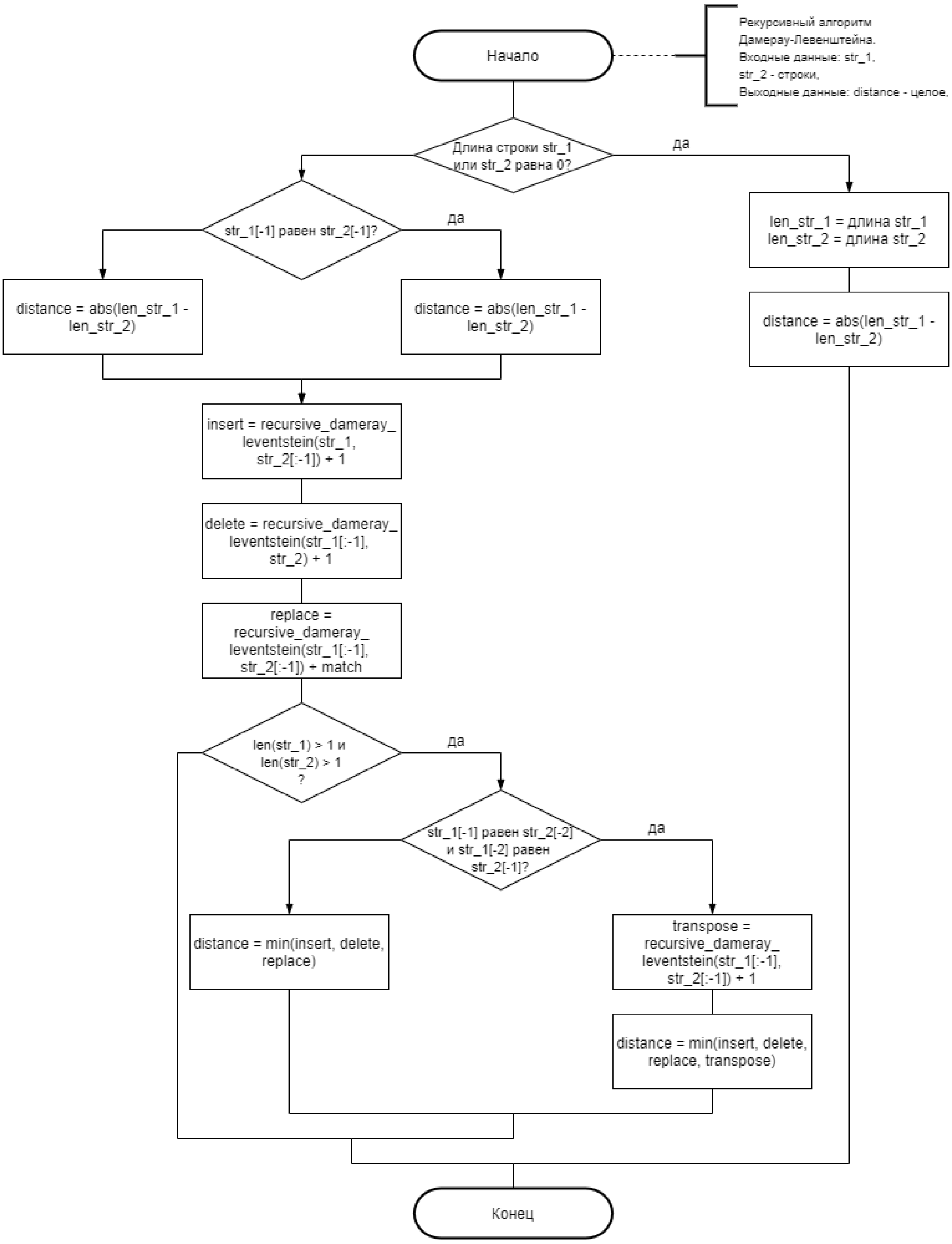

Рисунок 2.5 – Схема алгоритма рекурсивного Дамерау-Левенштейна.

2.2 Вывод

На основе теоретических данных, полученных в аналатическом разделе, были построены схемы нужных алгоритмов.

3 Технологический раздел

3.1 Требования к программному обеспечению

Программа должна отвечать следующим требованиям:

- на вход программе подаются две строки на русском или английском языке в любом регистре;
- на вход программе подаются корректные данные;
- осуществляется выбор алгоритма нахождения расстояния из меню;
- на выходе программа выдает результат - найденное расстояние между двумя строками выбранным пользователем алгоритмом.

3.2 Выбор средств реализации

Для реализации алгоритмов в данной лабораторной работе был выбран язык программирования Python 3.9.7[4]. Он является кроссплатформенным. Имеется опыт разработки на этом языке. В качестве среды разработки был использован Visual Studio Code[5], так как в нем можно работать как на операционной системе Windows, так и на дистрибутивах Linux. При замере процессорного времени был использован модуль time[6]. Замеры используемой памяти и число вызовов рекурсии проводились при помощи модуля cProfiler[7].

3.3 Листинги программ

Ниже представлены листинги разработанных алгоритмов Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Программный код нахождения расстояния Левенштейна итеративно с использованием двух строк

```
def iterative_levenstein_two_rows(str_1: str, str_2: str) -> int:
    len_str_1 = len(str_1); len_str_2 = len(str_2)
    flag_first_row = 1; flag_second_row = 2

    row_1 = create_row(len_str_1 + 1, flag_first_row)
    row_2 = create_row(len_str_1 + 1, flag_second_row)

    for i in range(len_str_2):
        row_2[0] = i + 1
        for j in range(len_str_1):
            deletion_cost = row_1[j + 1] + 1
            insert_cost = row_2[j] + 1
            replace_cost = row_1[j] if str_1[j - 1] == str_2[i - 1] \
                                   else row_1[j] + 1

            row_2[j + 1] = min(deletion_cost, insert_cost, replace_cost)
        row_1, row_2 = swap_rows(row_1, row_2)
    distance = row_1[-1]
    return distance
```

Листинг 3.2 – Программный код нахождения расстояния Левенштейна рекурсивно без использования кэша

```
def recursive_levenstein(str_1: str, str_2: str) -> int:
    if str_1 == '' or str_2 == '':
        return abs(len(str_1) - len(str_2))
    match = 0 if str_1[-1] == str_2[-1] else 1
    distance = min(recursive_levenstein(str_1, str_2[:-1]) + 1,
    recursive_levenstein(str_1[:-1], str_2) + 1,
    recursive_levenstein(str_1[:-1], str_2[:-1]) + match)
    return distance
```

Листинг 3.3 – Программный код нахождения расстояния Левенштейна рекурсивно с использованием матрицы

```
    def recursive_levenstein_matrix(str_1: str, str_2: str, i: int,
        j: int, matrix: list) -> Tuple[int, list[list[int]]]:
    if i == 0:
        return j, matrix
    if j == 0:
        return i, matrix
    if matrix[i][j] != -1:
        return matrix[i][j], matrix
    match = 0 if str_1[-1] == str_2[-1] else 1

    insert, matrix = recursive_levenstein_matrix(str_1, str_2[:-1], i,
                                                  j-1, matrix)
    delete, matrix = recursive_levenstein_matrix(str_1[:-1], str_2, i-1,
                                                    j, matrix)
    replace, matrix = recursive_levenstein_matrix(str_1[:-1], str_2
       [:-1],
                                                    -1, j-1, matrix)
    insert += 1; delete += 1; replace += match

    distance = min(insert, delete, replace)
    matrix[i][j] = distance
    return distance, matrix
```

Листинг 3.4 – Программный код нахождения расстояния Дамерау-Левенштейна рекурсивно

```
def recursive_dameray_levenstein(str_1: str, str_2: str) -> int:
    if str_1 == '' or str_2 == '':
        return abs(len(str_1) - len(str_2))
    match = 0 if str_1[-1] == str_2[-1] else 1
    insert = recursive_dameray_levenstein(str_1, str_2[:-1]) + 1
    delete = recursive_dameray_levenstein(str_1[:-1], str_2) + 1
    replace = recursive_dameray_levenstein(str_1[:-1], str_2[:-1]) + \
                                            match

    if len(str_1) > 1 and len(str_2) > 1 and str_1[-1] == str_2[-2] \
                                    and str_2[-1] == str_1[-2]:
        distance = min(insert, delete, replace,
                recursive_dameray_levenstein(str_1[:-2], str_2[:-2]) +
                   1)
    else:
        distance = min(insert, delete, replace)
    return distance
```

3.4 Вспомогательные функции

На листингах представлены программные модули, которые используются в данных функциях:

Листинг 3.5 – Программный код создания для кэша в виде строки

```
def create_row(len_row: int, flag_row: int) -> list[int]:
    row = list()
    if flag_row == 1:
        for i in range(len_row):
            row.append(i)
    else:
        for i in range(len_row):
            row.append(0)
    return row
```

Листинг 3.6 – Программный код обмена двух строк

```
def swap_rows(row_1: list[int], row_2: list[int]) \
                -> Tuple[list[int], list[int]]:
    temp_row = list()
    temp_row = deepcopy(row_1)
    row_1 = deepcopy(row_2)
    row_2 = deepcopy(temp_row)
    return row_1, row_2
```

3.5 Тестирование

Для тестирования используется метод черного ящика. В данном разделе приведена таблица 3.1, в которой указаны классы эквивалентностей тестов.

Таблица 3.1 – Таблица тестов

№	Описание теста	Слово 1	Слово 2	Алгоритм	
				Левенштейн	Дамерау-Левенштейн
1	Пустые строки	”	”	0	0
2	Нет повторяющихся символов	deepcopy	раздел	8	8
3	Инверсия строк	insert	tresni	6	6
4	Два соседних символа	heart	heatr	2	1
5	Одинаковые строки	таблица	таблица	0	0
6	Одна строка меньше другой	город	горо	1	1

3.6 Вывод

В данном разделе был выбран язык программирования, среда разработки. Реализованы функции, описанные в аналитическом разделе, и проведено их тестирование методом черного ящика по таблице 3.1.

4 Исследовательский раздел

4.1 Технические характеристики

Технические характеристии устройства, на котором выполнялось тестирование:

- операционная система: Windows 10 Pro;
- память: 8 GiB;
- процессор: Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz.

Тестирование проводилось на ноутбуке, который был подключен к сети питания. Во время проведения тестирования ноутбук был нагружен только встроенными приложениями окружения, самим окружением и системой тестирования.

4.2 Временные харастеристики выполнения

Ниже был проведен анализ времени работы алгоритмов. Исходными данными будут случайно сгенерированные строки длиной {3, 4, 5, 6, 7, 8}. Единичные замеры выдадут крайне маленький результат, поэтому проведем работу каждого алгоритма $n = 1000$ раз и поделим на число n. Получим среднее значение работы каждого из алгоритмов. Результат приведен на рис 4.1.

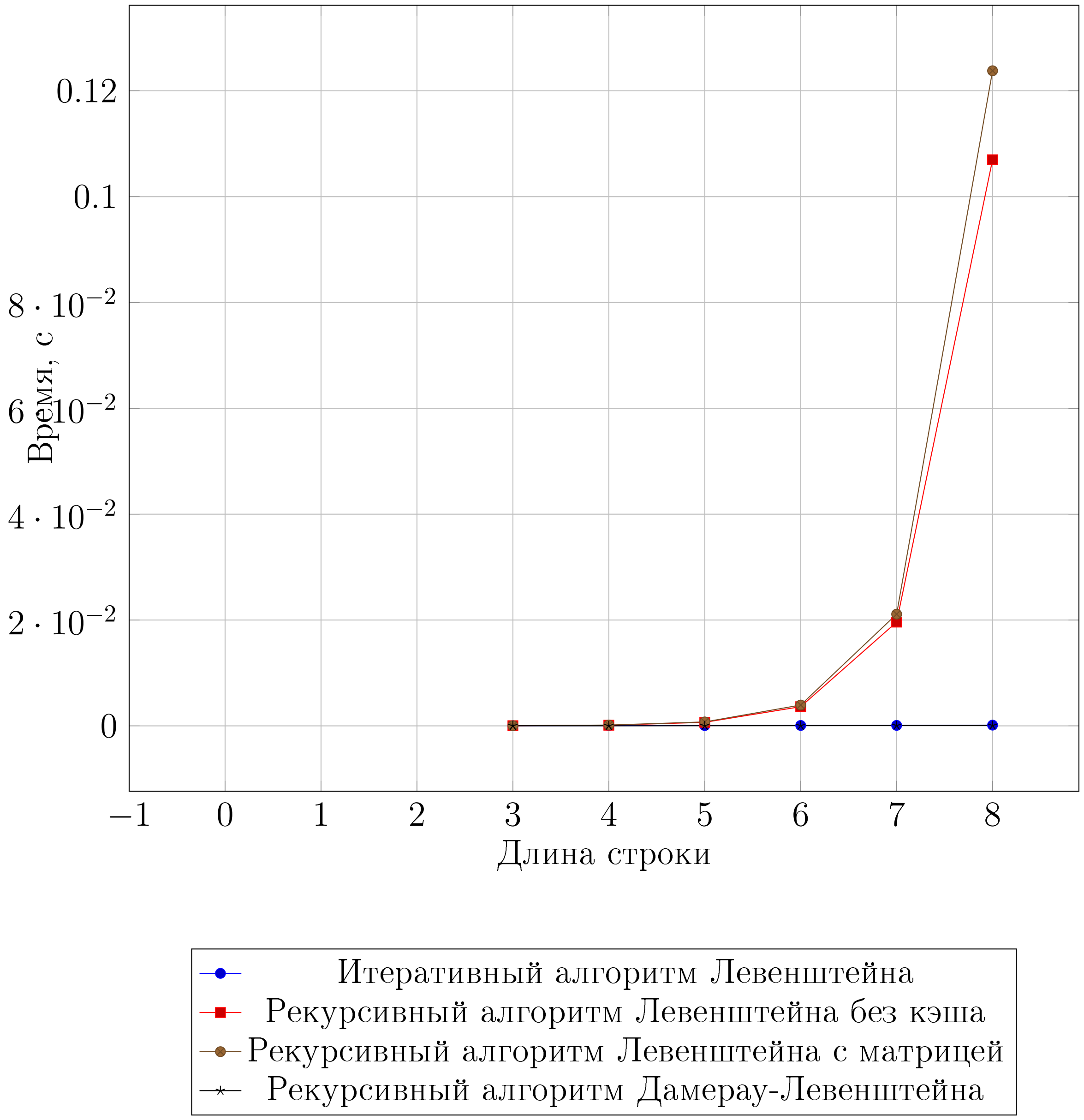

Рисунок 4.1 – График зависимости времени работы алгоритмов

Как видно из результатов, рекурсивный алгоритм Левенштейна без кэша и алгоритм Дамерау-Левенштейна уступают по скорости выполнения, начиная уже со строки длиной 7. Рекурсивный алгоритм Левенштейна выигрывает по времени выполнения у реализации с матрицей для длины строк 7, 8 на 313%, 1369% соответственно. Рекурсивный алгоритм Дамерау-Левенштейна выполняется быстрее реализации без кэша для длины строк 7, 8 на 337%, 1580% соответственно. Последний алгоритм задействует дополнительную операцию - транспозицию, которая тоже приводит к вызову рекурсии.

Выполнив анализ двух остальных алгоритмов на значения входных строк длиной {25, 50, 75, 100, 125, 150}, получим следующий результат, представленный на рис 4.2.

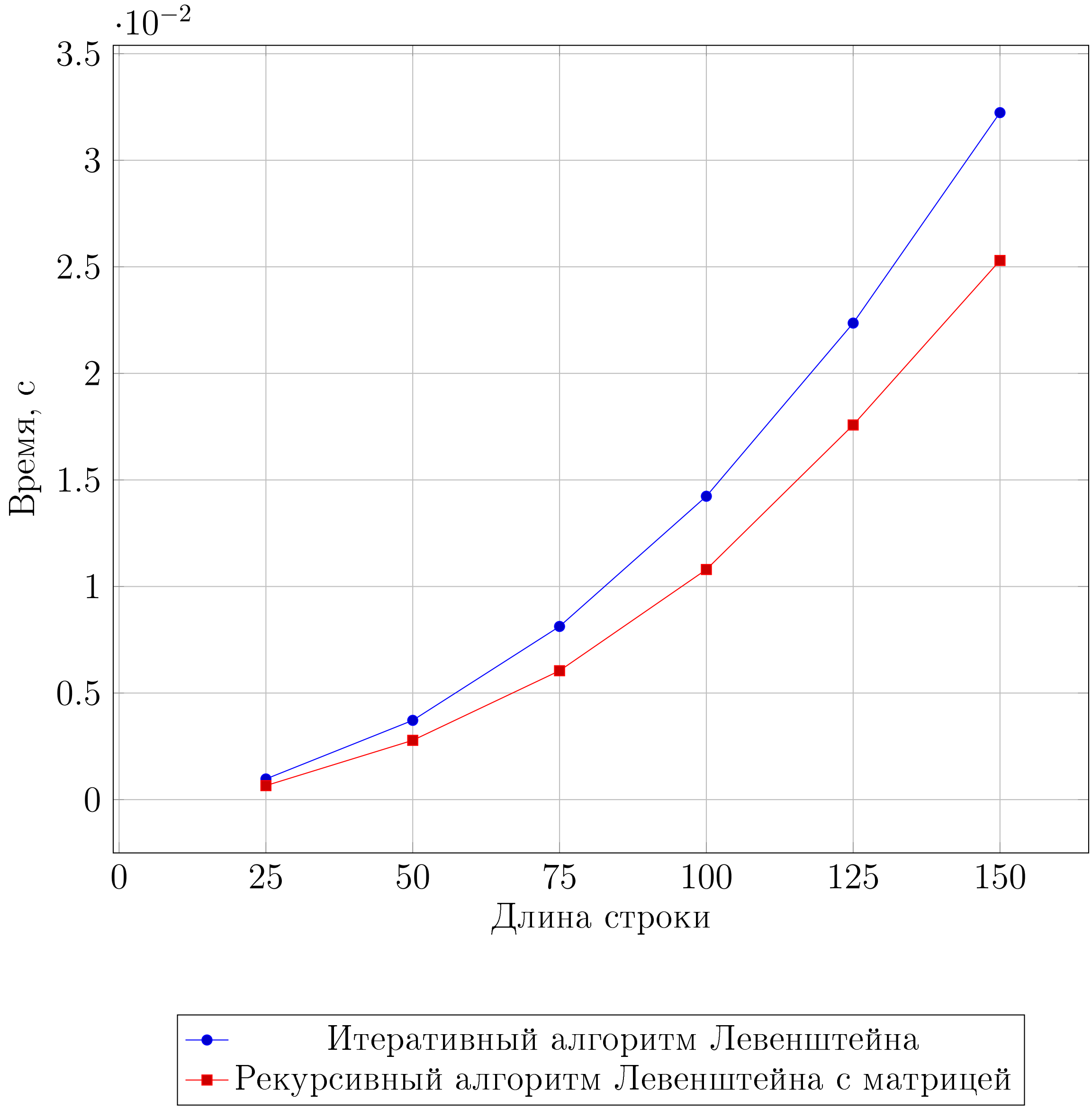

Рисунок 4.2 – График зависимости времени работы алгоритмов нахождения расстояния

Рекурсивный алгоритм Левенштейна с использованием матрицы выигрывает по скорости у итеративного метода на длинах строк 50, 100, 150 примерно на 27%, 31%, 33% соответсвенно. Это объясняется тем, что в итеративном случае выполняется дополнительная операция по обмену значений двух строк. На это требуется дополнительное время.

4.3 Объем потребляемой памяти

При исходных строках, длиною 3, требуется 52,8 Mb памяти. Результаты вызовов и объем потребляемой памяти приведены в таблице 4.1:

Таблица 4.1 – Число вызовов каждого алгоритма

Левенштейн			Дамерау-Левенштейн
Итеративный с двумя строками	Рекурсивный без кэша	Рекурсивный с матрицей	Рекурсивный
1	94	28	94

Общее значение потребляемой памяти складывается по формуле (4.1).

$$S = n_{calls} * V \tag{4.1}$$

где:

- n_{calls} - число вызовов функций;
- V - объем памяти, занимаемый одним вызовом функции.

По результатам исследования памяти алгоритмы Левенштейна и Дамерау-Левенштейна потребляют больше памяти при выполнении по сравнению с другими (отличается от итеративного способа в 94 раз, от рекурсивного с матрицей - приблизительно 3,35 раз).

4.4 Вывод

Рекурсивный вызов Левенштейна без кэша и Дамерау-Лвенштейна проигрывают как по скорости, так и по памяти итеративному. Причем рекурсивный алгоритм Левенштейна с матрицей выигрывает по скорости выполнения итеративному с двумя строками, но при этом проигрывает ему по памяти.

Сравнивая между собой рекурсивные вызовы алгоритмов Левенштейна и Дамерау-Левенштейна, можно сделать вывод о том, что рекурретный алгоритм поиска расстояния Левенштейна с матрицей выигрывает как по времени, так и по памяти у других реализаций этих алгоритмов, а рекуррентный Дамерау-Левенштейн проигрывает им по обоим параметрам. Однако, стоит отметить, что в системах автоматического исправления текста, где чаще всего встречаются ошибки, связанные с транспозицией двух символов, выполнение исправления ошибок выбор алгоритма Дамерау-Левенштейна будет оптимальным решением.

Заключение

В ходе выполнения лабораторной работы были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна. Были выполнено описание каждого из этих алгоритмов, приведены соотвествующие математические расчёты. Были получены навыки динамического программирования, а также реализованны данные алгоритмы. При тестировании каждого их них и анализе временных характеристик и объема потребляемой памяти можно сделать следующие выводы: выбор алгоритма Дамерау-Левенштейна является оптимальным решением ввиду того, что чаще всего необходимо исправлять ошибки, связанные с обменом двух соседних символов. В ином случае этот алгоритм является проигрышным как по времени, так и по памяти в сравнении с различными реализациями алгоритма Левенштейна. Рекурсивный алгоритм Левенштейн с кэшем в виде матрицы выигрывает по скорости выполнения у данной группы алгоритомв; но он проигрывает по использованию памяти за счет большого числа вызовов. Поэтому в иных ситуациях, не связанных с транспозицей, следует использовать итеративный алгоритм.

Список литературы

[1] КОМПЬЮТЕРНАЯ ЛИНГВИСТИКА Большая российская энциклопедия - электронная версия [Электронный ресурс]. Режим доступа: https://bigenc.ru/linguistics/text/2087783 (дата обращения: 20.09.2021).

[2] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. – М.: Доклады АН СССР, 1965. Т. 163. С. 845–848.

[3] Fred Damerau J. A technique for computer detection and correction of spelling errors. 1964. Т. 7. С. 171—176.

[4] Python. [Электронный ресурс]. Режим доступа: https://docs.python.org/3/ (дата обращения: 21.09.2021).

[5] Documentation for Visual Studio Code. [Электронный ресурс]. Режим доступа: https://code.visualstudio.com/docs (дата обращения: 21.09.2021).

[6] time — Time access and conversions — Python 3.9.7 documentation. [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html (дата обращения: 22.09.2021).

[7] The Python Profilers — Python 3.9.7 documentation. [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/profile.html (дата обращения: 22.09.2021).