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 курсу: "Анализ алгоритмов"

Студент Сукочева Алис
Группа ИУ7-43Б
Название предприятия МГТУ им. Н. Э. Баумана, каф. ИУ7
Тема Расстояние Левенштейна

Студент:	____________	Сукочева А.
	подпись, дата	Фамилия, И.О.
Преподаватель:	____________	Волкова Л.Л.
	подпись, дата	Фамилия, И. О.
Преподаватель:	____________	Строганов Ю.В.
	подпись, дата	Фамилия, И. О.

Москва — 2020 г.

Содержание

Введение

В данной лабораторной работе мы познакомимся с расстоянием Левенштейна. Данное расстояние показывает нам минимальное количество редакторских операций (вставки, замены и удаления), которые необходимы нам для перевода одной строки в другую. Это расстояние помогает определить схожесть двух строк.

Какие ошибки человек может допускать при написании какого-то текста? Например, его пальцы могут нажимать на нужные клавиши не в том порядке. С этой проблемой поможет нам справится расстояние Дамерау-Левенштейна. Данное расстояние задействует еще одну редакторскую операцию - транспозицию.

Практическое применение расстояние Левенштейна:

— Сравнение введенной строки со словарными словами в поисковой системе, такой как 'yandex' или 'google'

— Помогает найти разницу двух ДНК. Оценка мутации.

Целью данной работы является разбор и реализация алгоритма Дамерау-Левенштейна и Левенштейна.

В рамках выполнения работы необходимо решить следующие задачи:

а) Изучить алгоритмы Дамерау-Левенштейна и Левенштейна.

б) Реализовать изученные алгоритмы, а также матричную и рекурсивную реализацию алгоритма.

в) Подсчет времени поиска расстояния.

г) Сравнить временные характеристики, а также затраченную память.

д) Описать выбранную среду разработки и ЯП.

1 Аналитический раздел

1.1 Некоторые теоретические сведения

При преобразовании одного слова в другое мы можем использовать следующие операции:

а) D (*от англ. delete*) - удаление.

б) I (*от англ. insert*) - вставка.

в) R (*от англ. replace*) - замена.

Будем считать стоимость каждой вышеизложенной операции - 1.

Введем понятие совпадения - M (*от англ. match*). Его стоимость будет равна 0.

1.2 Расстояние Левенштейна

Имеем две строки S_1 и S_2, длинной M и N соответственно. Расстояние Левенштейна рассчитывается по рекуррентной формуле (1.1).

$$D(S_1[1...i],S_2[1...j]) = \begin{cases} \text{j, если i} == 0 \\ \text{i, если j} == 0 \\ min(\\ D(S_1[1...i],S_2[1...j-1]) + 1, \\ D(S_1[1...i-1],S_2[1...j]) + 1, & \text{j>0, i>0} \\ D(S_1[1...i-1],S_2[1...j-1]) + \\ \left[\begin{array}{c} \text{0, если } S_1\text{[i]} == S_2\text{[j]} \\ \text{1, иначе} \end{array} \right. \\ ) \end{cases} \quad (1.1)$$

1.3 Расстояние Дамерау-Левенштейна

Как было написано выше, в расстоянии Дамерау-Левенштейна задействует еще одну редакторскую операцию - транспозицию T (*от англ. transposition*). Расстояние Дамерау-Левенштейна рассчитывается по рекуррентной формуле. (1.2).

$$
D(S_1[1...i],S_2[1...j]) = \begin{cases}
\text{j, если i} == 0 \\
\text{i, если j} == 0 \\
min(\\
D(S_1[1...i],S_2[1...j-1]) + 1, \\
D(S_1[1...i-1],S_2[1...j]) + 1, & j > 0, i > 0 \\
D(S_1[1...i-1],S_2[1...j-1]) + \\
\left[\begin{array}{c} \text{0, если } S_1\text{[i]} == S_2\text{[j]} \\ \text{1, иначе} \end{array} \right. \\
D(S_1[1...i-2],S_2[1...j-2]) + 1, & \text{i, j>1, } a_i = b_{j-1}, b_j = a_{i-1} \\
)
\end{cases} \tag{1.2}
$$

1.4 Вывод

Мы познакомились с основополагающими материалами, которые в дальнейшем помогут нам при реализации алгоритмов Левенштейна и Дамерау-Левенштейна.

2 Констукторский раздел

В данном разделе мы рассмотрим схемы вышеизложенных алгоритмов.

2.1 Разработка алгоритмов

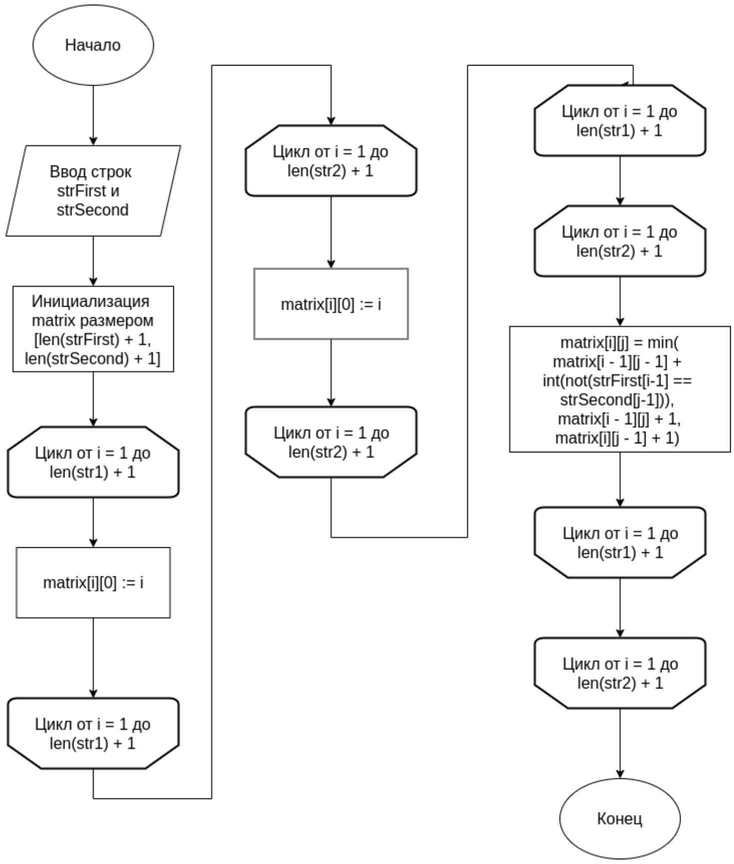

Рисунок 2.1 — Схема алгоритма Левенштейна

2.2 Вывод

В данном разделе мы рассмотрели схемы алгоритмов Левенштейна (2.1). и Дамерау-Левенштейна (2.2).

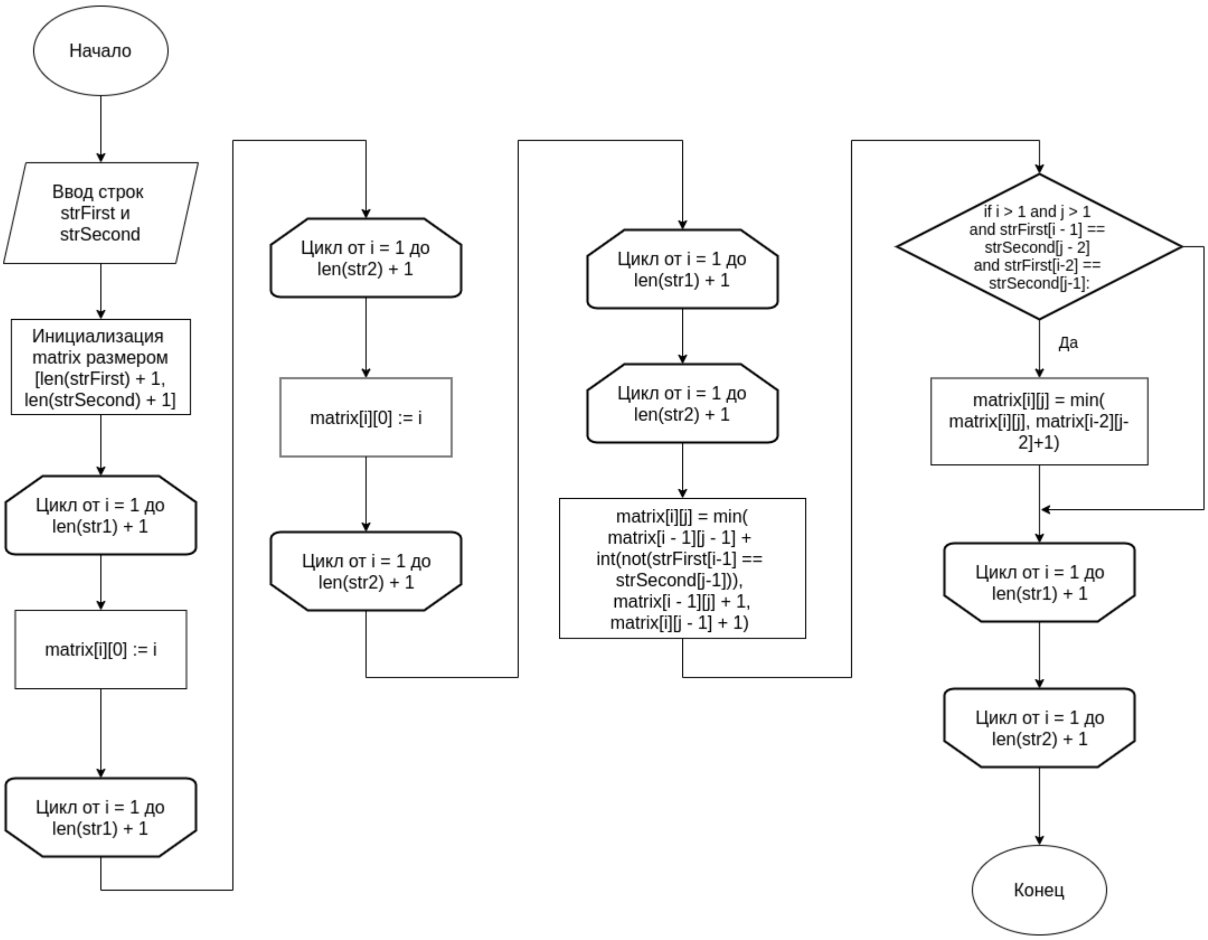

Рисунок 2.2 — Схема алгоритма Дамерау-Левенштейна

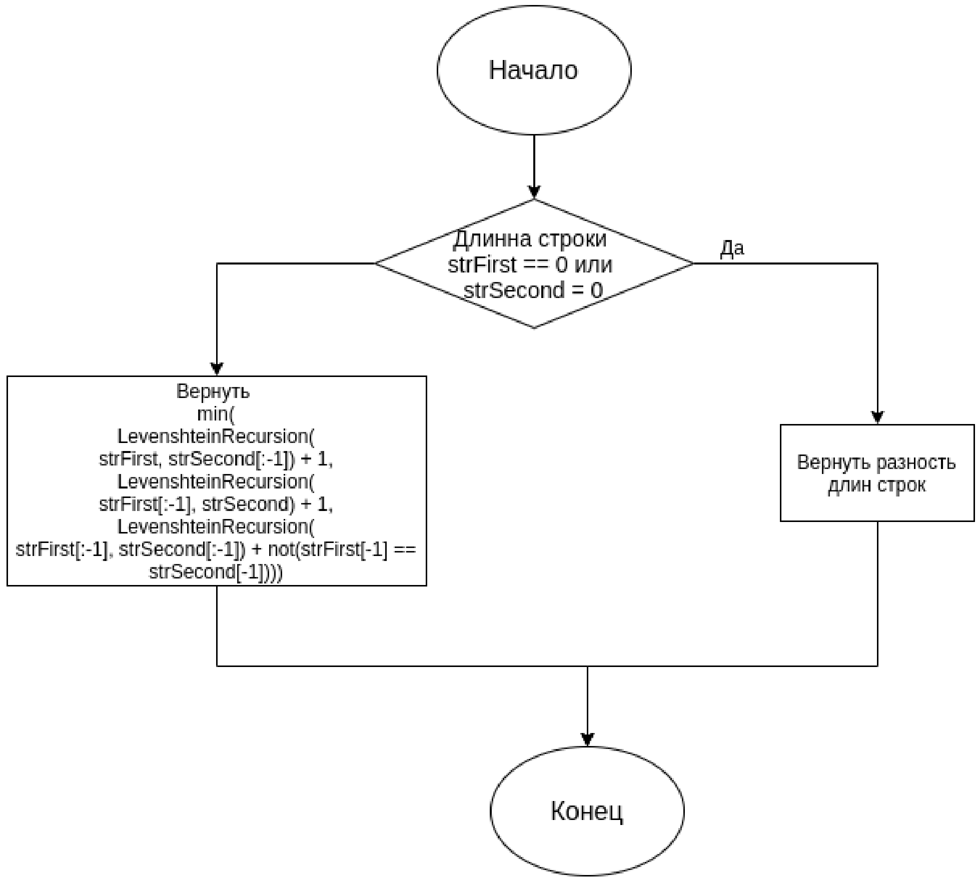

Рисунок 2.3 — Схема рекурсивного алгоритма Левенштейна

3 Технологический раздел

3.1 Выбор ЯП

В данной лабораторной работе использовался язык программирования - python [1]. Данный язык простой и понятный, также я знакома с ним. Поэтому данный язык был выбран. В качестве среды разработки я использовала Visual Studio Code [2], т.к. считаю его достаточно удобным и легким. Visual Studio Code подходит не только для Windows [3], но и для Linux [4], это еще одна причина, по которой я выбрала VS code, т.к. у меня установлена ОС Ubuntu 18.04.4 [5].

3.2 Требования к программному обеспечению

Входными данными являются две строки. Строки регистрозависимые. На выходе мы получаем матрицу и дистанцию, полученную алгоритмом Левенштейна. Результат алгоритма Дамерау-Левенштейна выводится только в том случае, когда его результат не совпадает с результатом алгоритма Левенштейна. Также требуется замерить время работы каждой реализации.

3.3 Сведения о модулях программы

Данная программа разбита на модули:

— main.py - Файл, содержащий точку входа в программу. В нем происходит общение с пользователем и вызов алгоритмов.

— algorithms.py - Файл содержит непосредственно сами алгоритмы.

— test_time.py - Файл с замером времени работы алгоритмов.

Листинг 3.1 — Главная функция main

```
def main():
    output("Enter the first string", GREEN)  # Column
    strFirst = input()
    output("Enter the second string:", GREEN)  # Row
    strSecond = input()
    distanceLev = Levenshtein(strFirst, strSecond, True)
    distanceDamLev = DamerauLevenshtein(strFirst, strSecond)
    output("distance Levenshtein: " + str(distanceLev), GREEN)
    if distanceLev != distanceDamLev:
        output("distance Damerau-Levenshtein: " + str(distanceDamLev),
            GREEN)
```

Листинг 3.2 — Функция нахождения расстояния Левенштейна матрично

```
def Levenshtein(strFirst, strSecond, flag=False):
    n, m = len(strFirst), len(strSecond)
```

```
    matrix = np.full((n + 1, m + 1), 0)  # math.inf)

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

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            matrix[i][j] = min(
            matrix[i - 1][j - 1] +
            int(not(strFirst[i-1] == strSecond[j-1])),  # R
            matrix[i - 1][j] + 1,                       # D
            matrix[i][j - 1] + 1)                       # I

    if flag:
        OutputMatrix(matrix, strFirst, strSecond)

    return matrix[-1][-1]
```

Листинг 3.3 — Рекурсивная функция нахождения расстояния Левенштейна

```
def LevenshteinRecursion(strFirst, strSecond):
    if (strFirst == "" or strSecond == ""):
        return abs(len(strFirst) - len(strSecond))

    temp = 0 if strFirst[-1] == strSecond[-1] else 1
    return min(
    LevenshteinRecursion(strFirst, strSecond[:-1]) + 1, # I
    LevenshteinRecursion(strFirst[:-1], strSecond) + 1, # D
    LevenshteinRecursion(strFirst[:-1], strSecond[:-1]) + temp # R
    )
```

Листинг 3.4 — Функция нахождения расстояния Дамерау-Левенштейна матрично

```
def DamerauLevenshtein(strFirst, strSecond, flag=False):
    n, m = len(strFirst), len(strSecond)
    matrix = np.full((n + 1, m + 1), 0)  # math.inf)

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

    for i in range(1, n + 1):
        for j in range(1, m + 1):
```

```
            matrix[i][j] = min(
            matrix[i - 1][j - 1] +
            int(not(strFirst[i-1] == strSecond[j-1])),   # R
            matrix[i - 1][j] + 1,                        # D
            matrix[i][j - 1] + 1)                        # I
            if i>1 and j>1 and strFirst[i-1] == strSecond[j-2] \
            and strFirst[i-2] == strSecond[j-1]:
            matrix[i][j] = min(matrix[i][j], matrix[i-2][j-2] + 1) # T

    if flag:
        OutputMatrix(matrix, strFirst, strSecond)
    return matrix[-1][-1]
```

Листинг 3.5 — Рекурсивная функция нахождения расстояния Дамерау-Левенштейна

```
def DamerauLevenshteinRecursion(strFirst, strSecond):
    if (strFirst == "" or strSecond == ""):
        return abs(len(strFirst) - len(strSecond))

    temp = 0 if strFirst[-1] == strSecond[-1] else 1
    result = min(
    DamerauLevenshteinRecursion(
    strFirst, strSecond[:-1]) + 1,              # I
    DamerauLevenshteinRecursion(
    strFirst[:-1], strSecond) + 1,              # D
    DamerauLevenshteinRecursion(
    strFirst[:-1], strSecond[:-1]) + temp       # R
    )

    if len(strFirst) > 1 and len(strSecond) > 1 and \
        strFirst[-1] == strSecond[-2] and strFirst[-2] == strSecond[-1]:
        result = min(result, DamerauLevenshteinRecursion(
        strFirst[:-2], strSecond[:-2]) + 1)  # T

    return result
```

Листинг 3.6 — функция замера времени

```
def TimeTest(strFirst, strSecond, countOperations):
    t1 = time.process_time()
    for _ in range(countOperations):
        Levenshtein(strFirst, strSecond)
    t2 = time.process_time()
    print("Levenshtein = ", t2 - t1)

    t1 = time.process_time()
```

```
    for _ in range(countOperations):
        LevenshteinRecursion(strFirst, strSecond)
    t2 = time.process_time()
    print("Levenshtein (Recursion)= ", t2 - t1)

    t1 = time.process_time()
    for _ in range(countOperations):
        DamerauLevenshtein(strFirst, strSecond)
    t2 = time.process_time()
    print("DamerauLevenshtein = ", t2 - t1)

    t1 = time.process_time()
    for _ in range(countOperations):
        DamerauLevenshteinRecursion(strFirst, strSecond)
    t2 = time.process_time()
    print("DamerauLevenshtein (Recursion)= ", t2 - t1)
```

3.4 Тестирование

В данном разделе будет приведена таблица 3.1, в которой четко отражено тестирование программы. Первый и второй столбец отвечают за введенные пользователем слова.

Слово 1	Слово 2	Ожидаемый вывод	Вывод программы
сито	столб	3	3
exponential	polynomial	6	6
Alice	Alice	0	0
Alice	alice	1	1
ma	am	2 1	2 1
		0	0
abc	cab	2	2

Таблица 3.1 — Таблица тестов

Все тесты пройдены успешно.

3.5 Вывод

В данном разделе мы рассмотрели листинги кода, обосновали выбор использованного в данной работе языка программирования и среды разработки, а также убедились в корректной работе программы, благодаря таблице 3.1. Сравнив представленные листинги можно сказать, что написание рекуррентных подпрограмм проще, чем матричных.

4 Экспериментальная часть

В данном разделе сравним работу каждого алгоритма.

4.1 Временные характеристики

Сравним матричный алгоритм Левенштейна и Дамерау-Левенштейна. Для сравнения возьмем строки длиной [10, 20, 30, 50, 100, 200]. Так как подсчет расстояния считается короткой задачей, мы воспользуемся усреднением массового эксперимента. Для этого сложим результат работы алгоритма n раз (n >= 10), после чего поделим на n. Тем самым мы получим достаточно точные характеристики времени. Сравнение произведем при n = 500. Результат можно увидеть на рис. 4.1. Как мы можем наблюдать при короткой длине разница по времени минимальна, при увеличении длины строки алгоритм Левенштейна с небольшим опережением вырывается вперед. Это обосновывается тем, что у алгоритма Дамерау-Левенштейна задействуется еще одна операция, которая замедляет алгоритм.

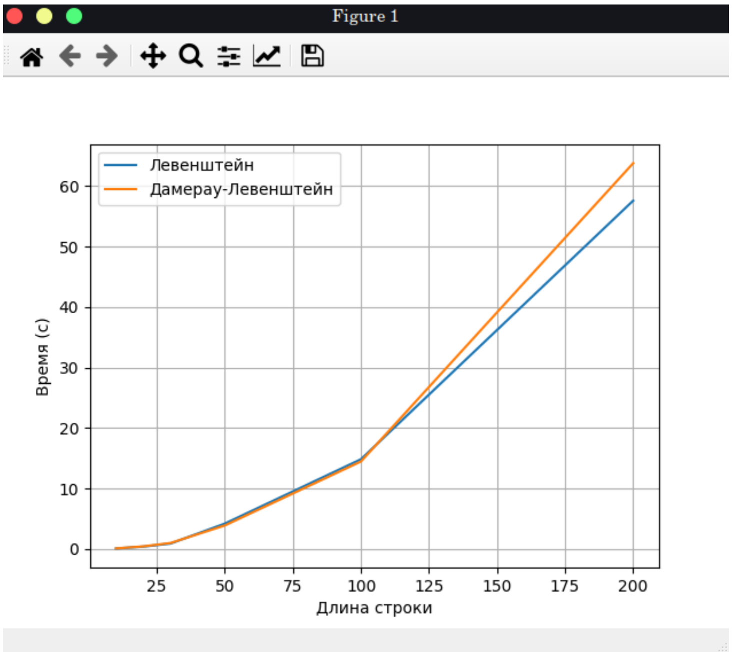

Рисунок 4.1 — Сравнение времени работы алгоритма Левенштейна и Дамерау-Левенштейна

Далее проведем сравнительный анализ временных характеристик рекурсивной и матричной реализаций алгоритма Левенштейна. Возьмем строки длиной [2, 3, 5, 7, 8], n положим равным 50. Результат можно увидеть на рис. 4.2. Такая большая разница во времени объясняется тем, что в рекурсивном алгоритме Левенштейна много рекурсивных вызовов с однотипными параметрами.

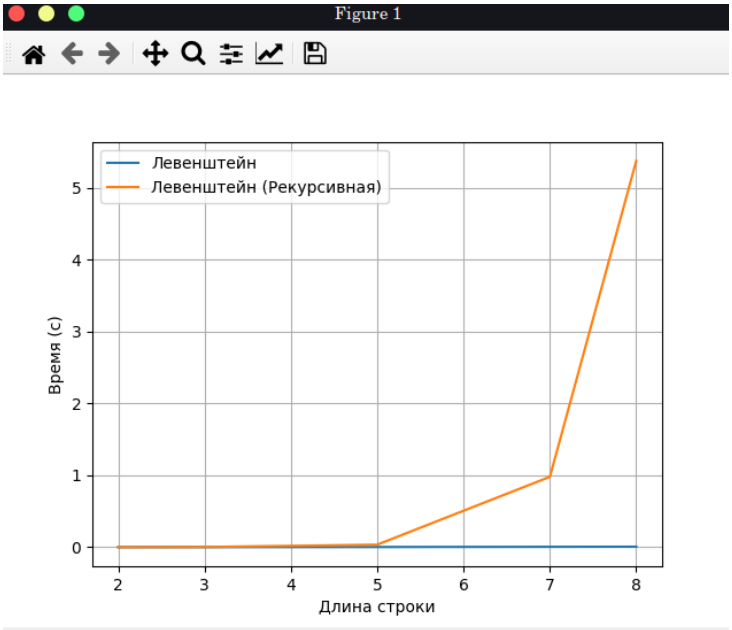

Рисунок 4.2 — Сравнение времени работы рекурсивной и матричной реализаций алгоритма Левенштейна.

4.2 Характеристики по памяти

На рисунке 4.3 представлено дерево вызовов рекурсивного алгоритма Левенштейна. Видно, что на третьем уровне встречаются повторные вызовы. Чем больше будет уровень, тем чаще будут вызываться функции с однотипными аргументами, что может привести к превышению максимальной глубины рекурсии. При строках длиной 2 подпрограмма вызовется 18 раз. Каждый вызов задействует 32 мегабайт (замеры проведены с помощью библиотеки memory_profiler [6]). В итоге нам потребуется 576 мегабайт для рекурсивных вызовов, в то время, когда в матричном алгоритме используется 42 мегабайта.

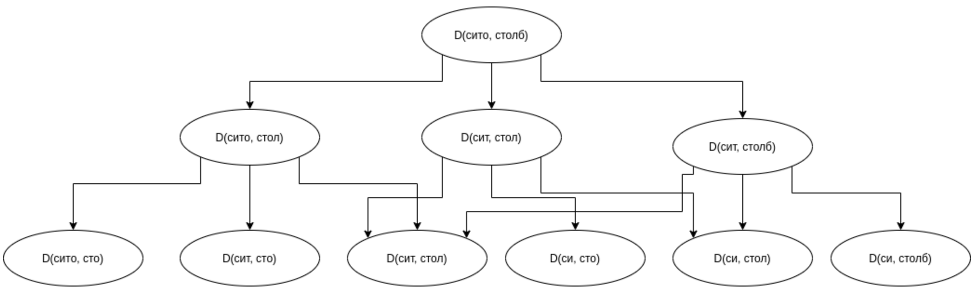

Рисунок 4.3 — Сравнение времени работы рекурсивной и матричной реализаций алгоритма Левенштейна.

4.3 Сравнительный анализ алгоритмов

Приведенные характеристики показывают нам, что рекурсивная реализация алгоритма очень сильно проигрывает по времени и по памяти.

Во время печати очень часто возникают ошибки связанные с транспозицией букв, поэтому алгоритм Дамерау-Левенштейна предпочтительнее, не смотря на то, что он проигрывает по времени алгоритму Левенштейна.
По аналогии с первым абзацем можно сделать вывод о том, что рекуррентный алгоритм Дамерау-Левенштейна будет более затратный, как по памяти, так и по времени по сравнению с матричной реализацией Дамерау-Левенштейна.

4.4 Вывод

В данном разделе мы сравнили количество затраченного времени и памяти вышеизложенных алгоритмов. Самым быстрым оказался матричный алгоритм нахождения расстояния Левенштейна.

Заключение

Алгоритмы Левенштейна и ДамерауЛевенштейна являются самыми популярными алгоритмами, которые помогают найти редакторское расстояние.
В этой лабораторной работе мы познакомились с алгоритмами Левенштейна (Формула 1.1) и Дамерау-Левенштейна (Формула 1.2). Построили схемы (Схема 2.1, Схема 2.2), соответствующие данным алгоритмам, также разобрали рекуррентные реализации (Схема 2.3). Написали полностью готовый и протестированный (Таблица 3.1) программный продукт, который считает дистанцию 4 способами.

В рамках выполнения работы мы решили следующие задачи:

а) Изучили алгоритмы Дамерау-Левенштейна и Левенштейна.

б) Реализовали изученные алгоритмы, а также матричную и рекурсивную реализации алгоритма.

в) Проиллюстрировали алгоритмы на схемах.

г) Описали выбранную среду разработки и ЯП.

д) Сравнили временные характеристики, а также затраченную память.

По окончании изучения данного материала можно смело идти и реализовывать алгоритмы нахождения редакционного расстояния!

Список использованных источников

1. *Майкл, Доусон.* Python Programming for the Absolute Beginner, 3rd Edition / Доусон Майкл. — Прогресс книга, 2019. — P. 416.
2. Visual Studio Code. — Microsoft, 2005. `https://code.visualstudio.com/`.
3. Windows. — Microsoft, 1985. `https://www.microsoft.com/ru-ru/windows`.
4. Linux. — 1991. `https://www.linux.org.ru/`.
5. Ubuntu 18.04. — 2018. `https://releases.ubuntu.com/18.04/`.
6. Замер памяти в Python3. — 2019. `https://pypi.org/project/memory-profiler/`.