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"

Тема Расстояние Левенштейна и Дамерау-Левенштейна

Студент Прянишников А. Н.

Группа ИУ7-55Б

Оценка (баллы) _____

Преподаватели Волкова Л.Л.

Москва — 2021 г.

Содержание

Введение

Расстояние Левенштейна[1] - мера, которая определяет, насколько отличаются друг от друга две строки. Фактически величина показывает, сколько нужно произвести односимвольных изменений, чтобы преобразовать одно слово к другому.

Расстояние Левенштейна получило широкое применение в компьютерной лингвистике. Активно его использует Яндекс[2] для поисковых систем. Мера используется для:

- Автоматического исправления ошибок в тексте
- Поиска возможных ошибок в поисковых запросах
- Расчёта изменений в различных версиях текста (утилита diff)

Также расстояние Левенштейна нашло применение в биоинформатике для нахождения разности последовательностей генов.[3]

Актуальность работы заключается в том, что нахождение расстояния Левенштейна должно выполняться за максимально короткое время. Целью данной работы является разработка программы, которая реализует четыре алгоритма поиска редакционного расстояния. Для достижения поставленной цели необходимо выполнить следующее:

- рассмотреть существующие алгоритмы поиска редакционного расстояния;
- привести схемы реализации рассматриваемых алгоритмов;
- определить средства реализации алгоритмов;
- реализовать рассматриваемые алгоритмы;
- провести модульное тестирование всех реализаций алгоритмов;
- оценить реализацию алгоритмов по времени и памяти.

1 Аналитический раздел

В этом разделе будут рассмотрены основные алгоритмы поиска редакционного расстояния, а также варианты их реализаций.

1.1 Расстояние Левенштейна

Расстояние Левенштейна[1] - минимально необходимое количество редакторских операций (удаление, вставка, замена) для преобразования одной строки в другую. Всего используется четыре возможные редакторские операции:

- I (англ. insert) - вставить;
- D (англ. delete) - удалить;
- R (англ. replace) - заменить;
- M (англ. match) - совпадение.

Операции I, D и R имеет цену, равную 1, а M - 0, так как в случае совпадения букв никаких действий не должно быть произведено.

1.2 Обобщённый алгоритм для определения расстояния Левенштейна

Пусть s_1 и s_2 - строки, для которых мы должны определить расстояние Левенштейна. Индикаторная функция m для двух символов a и b рассчитывается по формуле 1.1:

$$m(a,b) = \begin{cases} 1 & \text{, если a = b} \\ 0 & \text{иначе} \end{cases} \tag{1.1}$$

Для строк s_1 и s_2 ставится в соответствие функция D, которая принимает на вход два индекса и рассчитывает ответ по формуле 1.2:

$$D(i,j) = \begin{cases} 0 & \text{i} = 0, \text{j} = 0 \\ i & \text{j} = 0, \text{i} > 0 \\ j & \text{i} = 0, \text{j} > 0 \\ \min\{ & \\ \quad D(i, j-1) + 1 & \\ \quad D(i-1, j) + 1 & \text{i} > 0, \text{j} > 0 \\ \quad D(i-1, j-1) + m(a[i], b[j])\} & \end{cases}, \tag{1.2}$$

Функция D составлена из следующих соображений:

- для перевода из пустой строки в пустую требуется ноль операций;
- для перевода из пустой строки в строку a требуется $|a|$ операций;
- для перевода из строки a в пустую требуется $|a|$ операций.

Для перевода из строки a в строку b требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена) в некоторой последовательности. Последовательность проведения любых двух операций можно поменять, порядок проведения операций не имеет никакого значения. Полагая, что a', b' — строки a и b без последнего символа соответственно, цена преобразования из строки a в строку b может быть выражена как:

- сумма цены преобразования строки a в b и цены проведения операции удаления, которая необходима для преобразования a' в a;
- сумма цены преобразования строки a в b и цены проведения операции вставки, которая необходима для преобразования b' в b;
- сумма цены преобразования из a' в b' и операции замены, предполагая, что a и b оканчиваются разные символы;
- цена преобразования из a' в b', предполагая, что a и b оканчиваются на один и тот же символ.

Минимальной ценой преобразования будет минимальное значение приведенных вариантов.

Пусть $N_1 = |s_1|, N_2 = |s_2|$, тогда расстояние между строками s_1 и s_2 можно подсчитать по формуле 1.3:

$$\rho_л(S_1, S_2) = D(N_1, N_2) \tag{1.3}$$

Все алгоритмы подсчёта расстояния Левенштейна работают именно с этим алгоритмом, лишь используя разные типы данных.

1.3 Рекурсивный алгоритм для определения расстояния Левенштейна

Рекурсивный вариант напрямую реализует формулу 1.2, вызывая функцию D для величин, равных длине каждой из строк. Стоит отметить, что порядок рекурсивного вызова подпрограмм не имеет значения. База рекурсии - первые три условия, поэтому гарантировано, что на выходе будет однозначный ответ. Из недостатков сразу можно отметить, что одно и то же значение будет подсчитано несколько раз, поэтому будет происходить большое количество повторных операций.

1.4 Рекурсивный алгоритм для определения расстояния Левенштейна с использованием матрицы

В качестве оптимизации прошлого алгоритма можно подсчитанные значения сохранять в матрицу, в которой строкам i и j будет соответствовать значение функции $D(i, j)$. Каждый раз при подсчёте D программа будет проверять, было ли уже подсчитано значение для заданных аргументов - и использует при существовании уже готовый вариант.

1.5 Итеративный алгоритм для определения расстояния Левенштейна с использованием матрицы

Можно обойтись и без рекурсивных вызовов подпрограмм, сразу заполняя в матрицу значения функции $D(i, j)$.

Сразу можно заполнить первую строку матрицы и первый столбец матрицы: для них условие тривиально. После этого процесс продолжается вдоль каждой строки до того момента, как подсчитано значение для последней клетки - в ней и будет содержаться ответ. На рисунке 1.1 представлена работа итеративного алгоритма.

		А	Р	Е	С	Т	А	Н	Т
	0	1	2	3	4	5	6	7	8
Д	1	1	2	3	4	5	6	7	8
А	2	1	2	3	4	5	5	6	7
Г	3	2	2	3	4	5	6	6	7
Е	4	3	3	2	3	4	5	6	7
С	5	4	4	3	2	3	4	5	6
Т	6	5	5	4	3	2	3	4	5
А	7	6	6	5	4	3	2	3	4
Н	8	7	7	6	5	4	3	2	3

Рисунок. 1.1: Иллюстрация работы матричного алгоритма нахождения расстояния Левенштейна

1.6 Оптимизированный итерационный алгоритм для определения расстояния Левенштейна с хранением двух строк

В вычислениях используются только две строки матрицы, поэтому можно не хранить всю матрицу, а лишь прошлую строку и ту, для которой производятся вычисления. Это позволяет существенно оптимизировать затраты по памяти.

1.7 Расстояние Дамерау-Левенштейна

Зачастую пользователь допускает опечатку, перепутав местами соседние буквы. Обычное расстояние Левенштейна для таких строк равно двум, поэтому для оптимизации таких ситуации было создано расстояние Дамерау-Левенштейна[4].

К существующим операциям добавляется ещё одна - X (англ. Exchange). Штраф за неё также составляет один балл. Тогда функция D вычисляется по формуле 1.4:

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j) = 0, \\ \min\{ & \\ \quad d_{a,b}(i,j-1) + 1, & \\ \quad d_{a,b}(i-1,j) + 1, & \text{иначе} \\ \quad d_{a,b}(i-1,j-1) + m(a[i],b[j]), & \\ \quad \begin{cases} d_{a,b}(i-2,j-2) + 1, & \text{если } i,j > 1; \\ & a[i] = b[j-1]; \\ & b[j] = a[i-1] \\ \infty, & \text{иначе} \end{cases} & \\ \} & \end{cases}, \tag{1.4}$$

Для вычисления расстояния Дамерау-Левенштейна можно применять и итеративный, и рекурсивный метод вычислений. В работе будет использоваться рекурсивный вариант с заполнением матрицы для уже подсчитанных значений.

1.8 Вывод

В этом разделе был рассмотрен обобщённый алгоритм для нахождения расстояния Левенштейна, а также его три реализации: рекурсионная, рекурсионная с матрицей и итеративная. Также было рассмотрено расстояние Дамерау-Левенштейна. Для выполнения поставленных задач будет реализовано четыре алгоритма:

- оптимизированный итеративный алгоритм поиска расстояния Левенштейна;
- рекурсионный алгоритм поиска расстояния Левештейна;
- рекурсионный алгоритм поиска расстояния Левенштейна с хранением матрицы;
- рекурсионный алгоритм поиска расстояния Дамерау-Левенштейна с хранением матрицы.

2 Конструкторский раздел

В этом разделе будут приведены схемы алгоритмов, которые будут реализованы для выполнения поставленных задач.

2.1 Схемы алгоритмов

На рисунках 2.1, 2.2, 2.3 приведены схемы различных реализаций алгоритмов поиска расстояния Левенштейна, на рисунке 2.4 - реализация алгоритма нахождения расстояния Дамерау-Левенштейна.

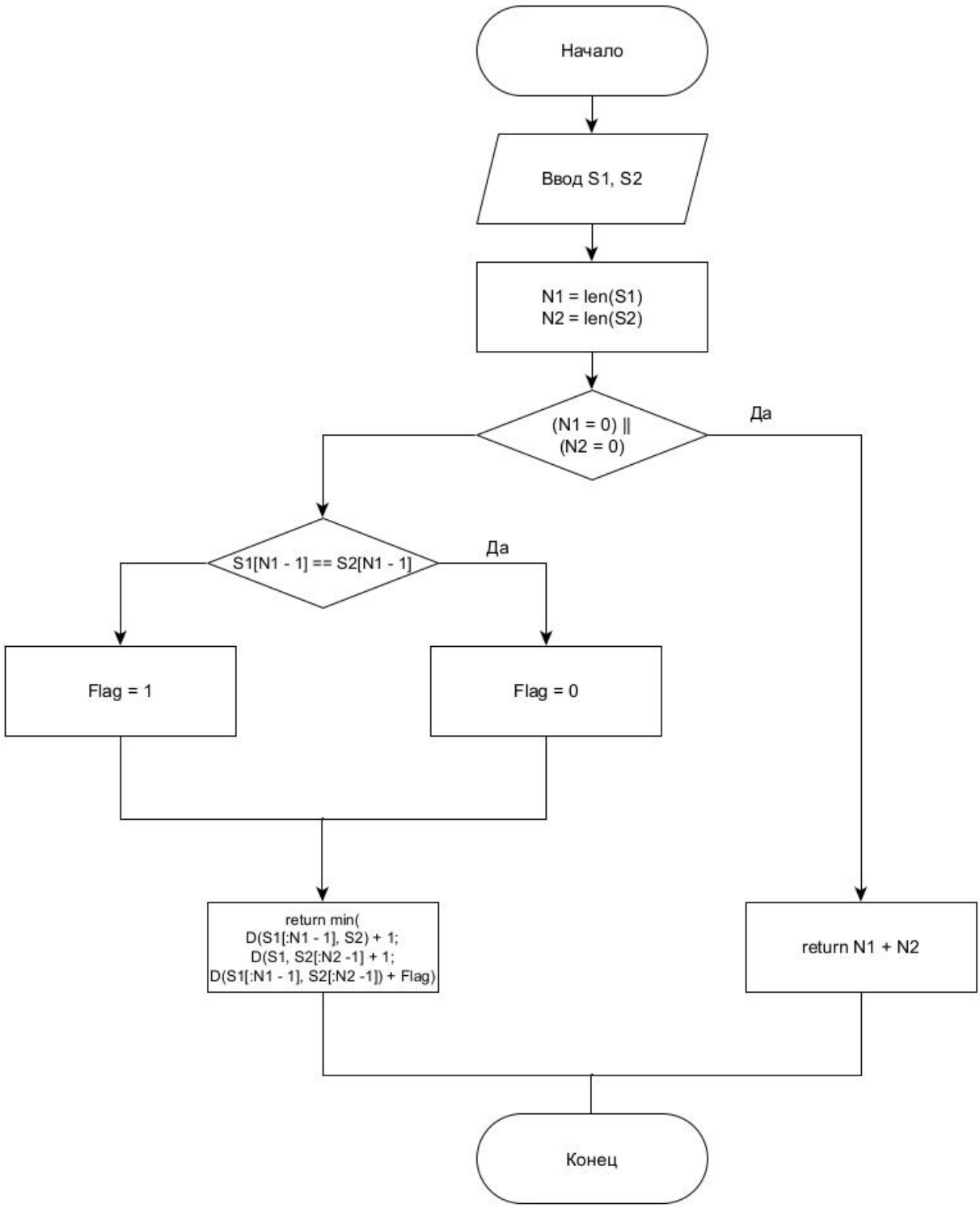

Рисунок. 2.1: Схема рекурсивного алгоритма Левенштейна без матрицы

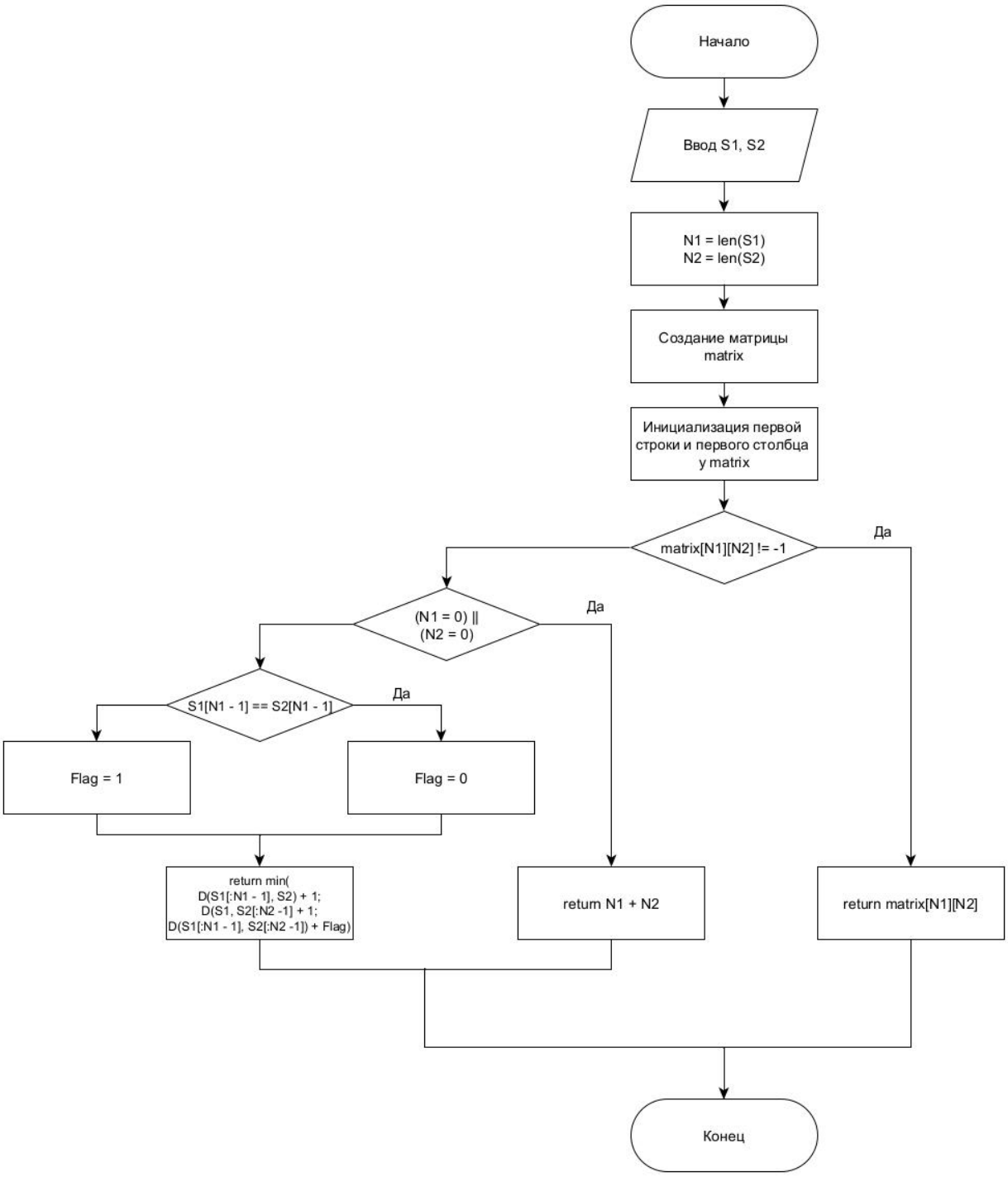

Рисунок. 2.2: Схема рекурсивного алгоритма Левенштейна с матрицей

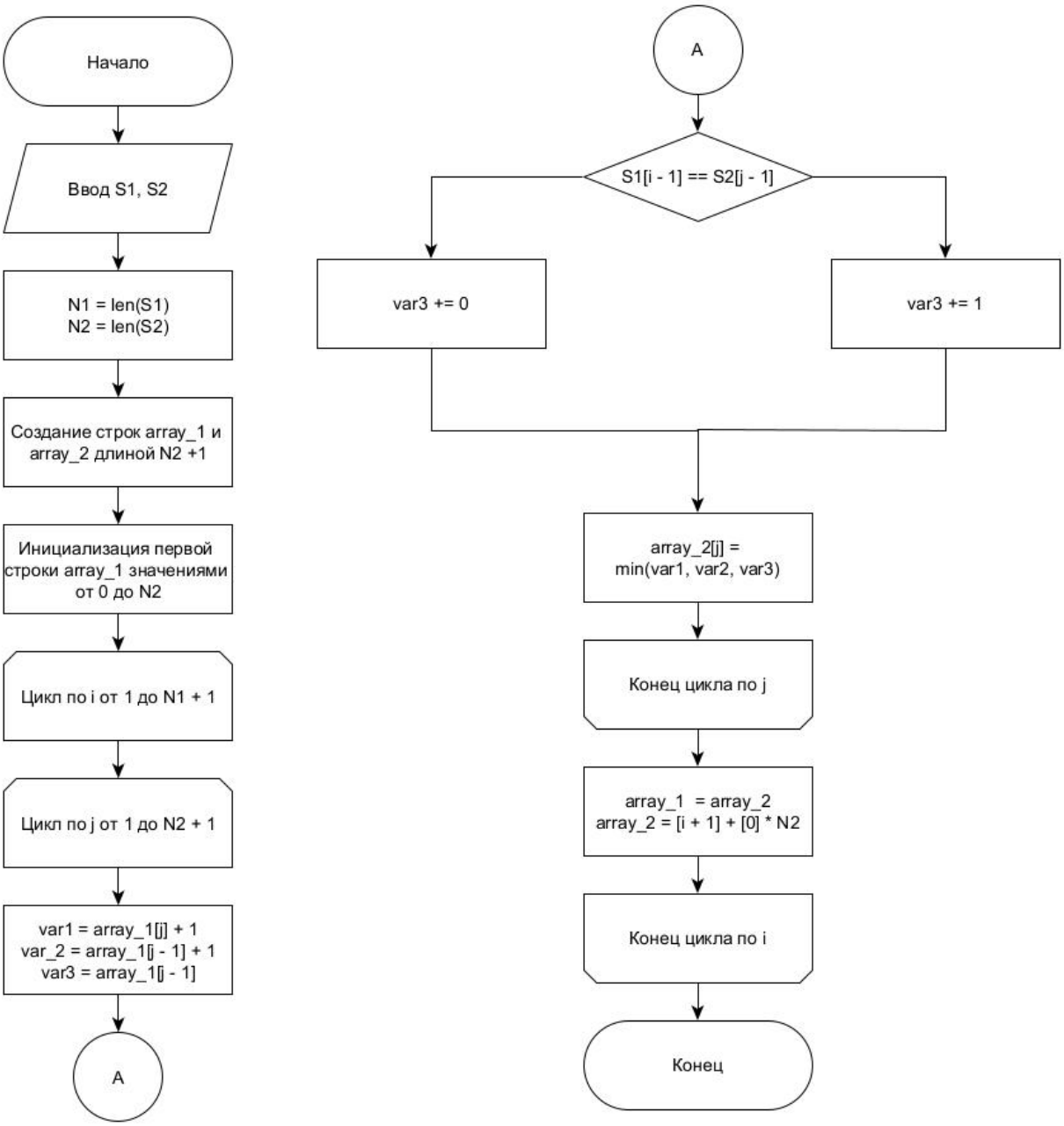

Рисунок. 2.3: Схема итеративного алгоритма Левенштейна с хранением двух строк

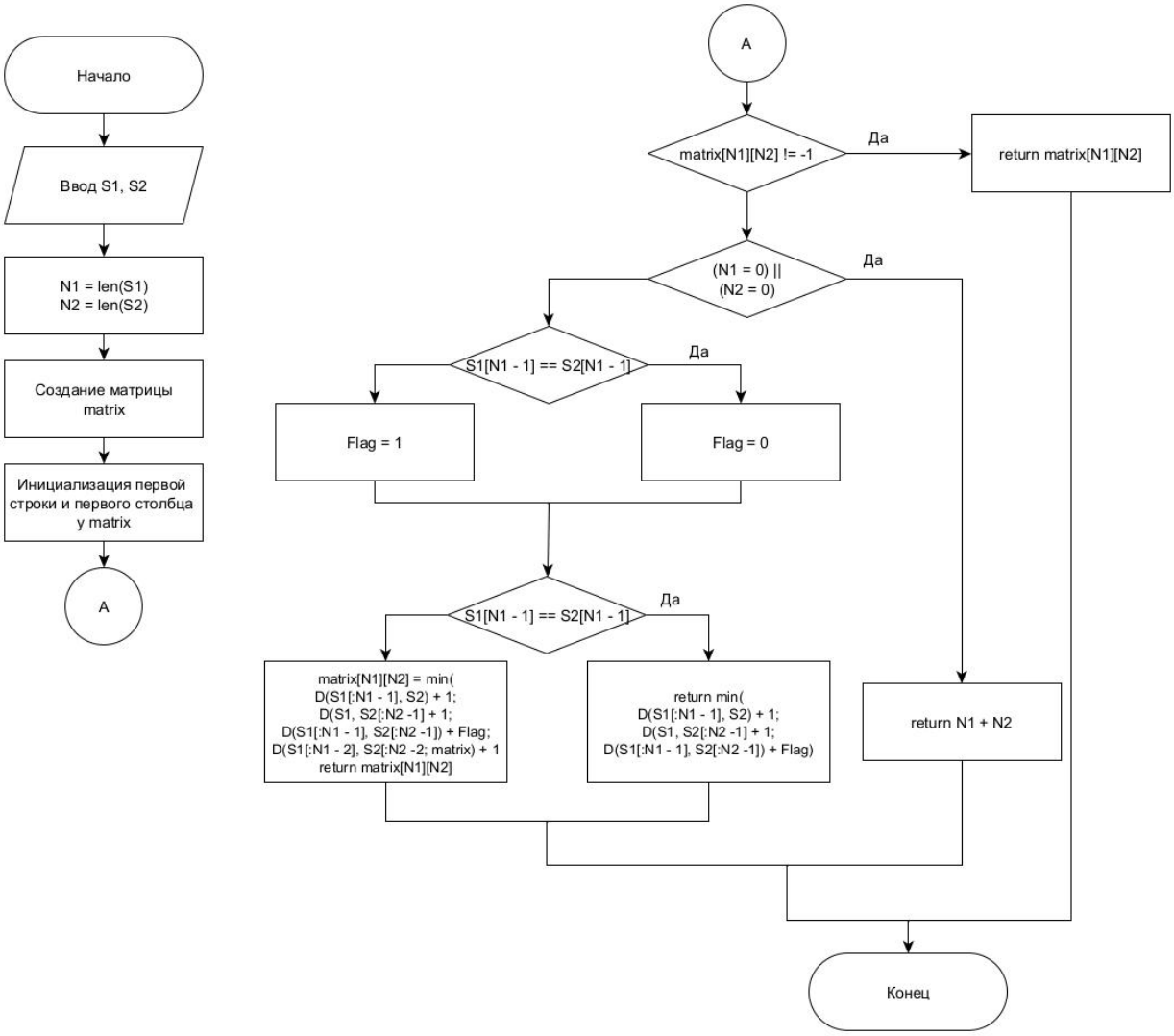

Рисунок. 2.4: Схема рекурсивного алгоритма Дамерау-Левенштейна (с матрицей)

2.2 Вывод

Были разработаны и протестированы спроектированные алгоритмы: вычисления расстояния Левенштейна рекурсивно, с заполнением матрицы и рекурсивно с заполнением матрицы, а также вычисления расстояния Дамерау — Левенштейна с заполнением матрицы.

3 Технологический раздел

В данном разделе приведены требования к ПО, выбранные средства реализации и листинги кода.

3.1 Средства реализации программного обеспечения

В качестве языка программирования выбран Python 3.9, так как имеется опыт разработки проектов на этом языке. Для замера процессорного времени используется функция process_time из библиотеки time. В её результат не включается время, когда процессор не выполняет задачу [5].

3.2 Требования к ПО

Ниже будет представлен список требований к разрабатываемому программному обеспечению.

Требования к входным данным:

- вводимые строки должны быть регистрозависимыми;
- язык строк должен быть русский или английский;
- строки могут быть пустыми – это не ошибочная ситуация.

Требования к выводу:

- программа должна выводить числовое значение (расстояние) для введенных строк по каждому алгоритму;
- программа должна выводить матрицу расстояний, если она использовалась в алгоритме.

3.3 Листинг кода

Листинг 3.1: Рекурсивная реализация алгоритма поиска расстояния Левенштейна

```
def levenshtein_rec(str_1, str_2):
    N1, N2 = len(str_1), len(str_2)
    if (N1 == 0) or (N2 == 0):
        return N1 + N2
    else:
        flag = int(str_1[N1 - 1] != str_2[N2 - 1])
        return min(
            levenshtein_rec(str_1[:N1 - 1], str_2)
            + 1,
            levenshtein_rec(str_1, str_2[:N2 - 1])
            + 1,
            levenshtein_rec(str_1[:N1 - 1]
            , str_2[:N2 - 1]) + flag)
```

Листинг 3.2: Рекурсивная реализация (с матрицей) алгоритма поиска расстояния Левенштейна

```
def levenshtein_rec_matrix(str_1, str_2):

    def recursion(str_1, str_2, matrix):
        N1, N2 = len(str_1), len(str_2)
        if matrix[N1][N2] == -1:
            if (N1 == 0) or (N2 == 0):
                matrix[N1][N2] = N1 + N2
            else:
                flag =int(str_1[N1 - 1] != str_2[N2 - 1])
                matrix[N1][N2] = min(
                            recursion(str_1[:N1-1],
                            str_2, matrix) + 1,
                            recursion(str_1,
                            str_2[:N2-1], matrix) + 1,
                            recursion(str_1[:N1-1],
                            str_2[:N2-1], matrix) + flag
                            )
        return matrix[N1][N2]
```

```
    N1, N2 = len(str_1), len(str_2)
    matrix = [[-1] * (N2 + 1) for i in range(N1 + 1)]
    result = recursion(str_1, str_2, matrix)
```

Листинг 3.3: Матричная реализация (с хранением двух строк) алгоритма поиска расстояния Левенштейна

```
def levenshtein_not_rec(str_1, str_2):
    N1, N2 = len(str_1), len(str_2)
    array_1 = [i for i in range(N2 + 1)]
    array_2 = [1] + [0] * N2

    for i in range(1, N1 + 1):
        for j in range(1, N2 + 1):
            var1 = array_1[j] + 1
            var2 = array_2[j - 1] + 1
            var3 = array_1[j - 1]

            if (str_1[i - 1] != str_2[j - 1]):
                var3 += 1
            array_2[j] = min(var1, var2, var3)

        array_1, array_2 = array_2, [i + 1] + [0] * N2
    return array_1[N2]
```

Листинг 3.4: Рекурсивная реализация (с матрицей) алгоритма поиска расстояния Дамерау-Левенштейна

```
def damerau_levenshtein(str_1, str_2):

    def recursion(str_1, str_2, matrix):
        N1, N2 = len(str_1), len(str_2)
        if matrix[N1][N2] == -1:
            if (N1 == 0) or (N2 == 0):
                matrix[N1][N2] = N1 + N2
            else:
                flag = int(str_1[N1 - 1] != str_2[N2 - 1])
                if (N1 > 1 and N2 > 1 and
                    str_1[N1 - 2] == str_2[N2 - 1]
                    and str_1[N1 - 1] == str_2[N2 - 2]):
                    matrix[N1][N2] = min(
                                    recursion(str_1[:N1-1],
                                    str_2, matrix) + 1,
```

```
                                    recursion(str_1,
                                    str_2[:N2-1], matrix) + 1,
                                    recursion(str_1[:N1-1],
                                    str_2[:N2-1], matrix)
                                    + flag,
                                    recursion(str_1[:N1-2],
                                    str_2[:N2-2], matrix) + 1)
            else:
                matrix[N1][N2] = min(
                                    recursion(str_1[:N1-1],
                                    str_2, matrix) + 1,
                                    recursion(str_1,
                                    str_2[:N2-1], matrix) + 1,
                                    recursion(str_1[:N1-1],
                                    str_2[:N2-1], matrix)
                                    + flag)
    return matrix[N1][N2]
```

3.4 Функциональные тесты

В таблице 3.1 приведены результаты функциональных тестов. Первые три числа в столбцах с ответами - результаты работы алгоритмов для нахождения расстояния Левенштейна, последний столбец - результат работы алгоритма нахождения расстояния Дамерау-Левенштайна

Таблица 3.1: Результаты функциональных тестов

Строка 1	Строка 2	Ожидаемый результат	Фактический результат
скат	кот	2 2 2 2	2 2 2 2
увлечения	развлечения	4 4 4 4	4 4 4 4
АААААА	БББББББ	6 6 6 6	6 6 6 6
хотдог	каток	4 4 4 4	4 4 4 4
		0 0 0 0	0 0 0 0
общага	общага	0 0 0 0	0 0 0 0
привет	Пока	6 6 6 6	6 6 6 6
привет	пока	5 5 5 5	5 5 5 5
привет	првиет	2 2 2 1	2 2 2 1
*общага	общага	1 1 1 1	1 1 1 1
1234	2143	3 3 3 2	3 3 3 2

* - первая буква в первой строке - на английской раскладке.

Все тесты пройдены успешно.

3.5 Вывод

Были разработаны и протестированы спроектированные алгоритмы: вычисления расстояния Левенштейна рекурсивно, с заполнением матрицы и рекурсивно с заполнением матрицы, а также вычисления расстояния Дамерау — Левенштейна с заполнением матрицы.

4 Экспериментальная часть часть

4.1 Демонстрация работы программы

На рисунке 4 представлена работы программы. В консоль выводится результат подсчёта расстояний для каждого из алгоритмов, а для алгоритмов, использующих матрицу - также выводится и матрица.

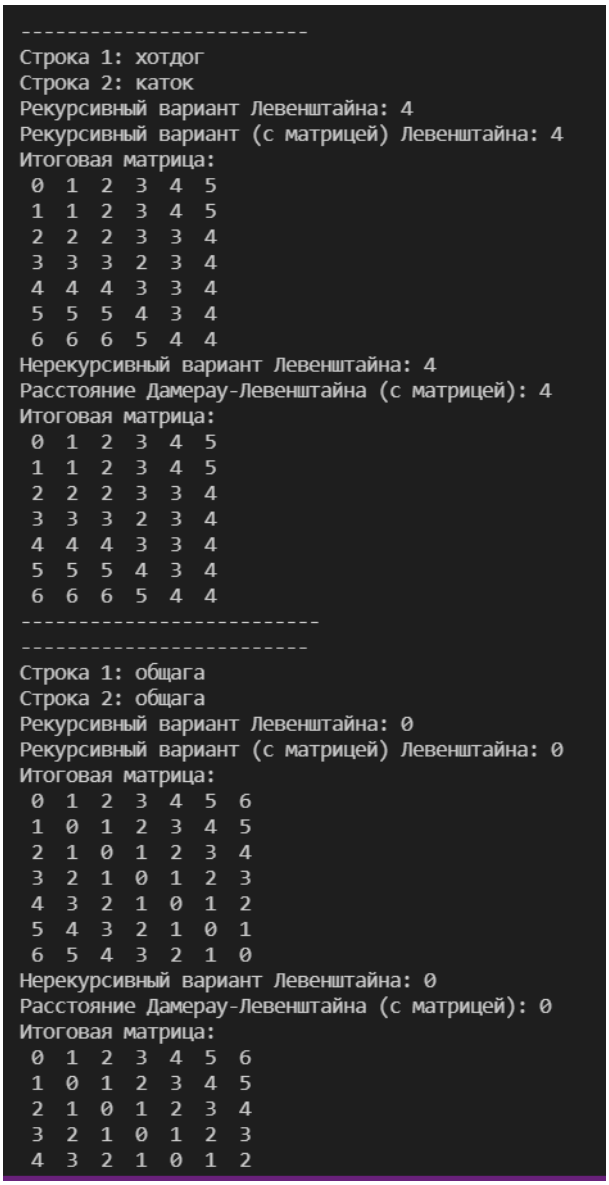

Рисунок. 4.1: Демонстрация работы алгоритма

4.2 Технические характеристики

Технические характеристики устройства, на котором производилось тестирование:

- операционная система: Windows 10;
- оперативная память: 8 ГБ;
- процессор: Intel Core i5-1135G7 @ 2.40GHz.

Тестирование производилось на ноутбуке при включённом режиме производительности. Во время тестирования ноутбук был загружен только системными процессами.

4.3 Время работы алгоритмов

Алгоритмы тестировались с помощью функции process_time_ns из библиотеки time. Чтобы время удалось отследить и результаты были достаточно точными, каждый тест проводится N=100 раз, после чего общее время делится на 100 - тогда получится среднее время работы каждого алгоритма по одному тесту. Для каждого теста генерировалась случайным образом строка из M символов.

Результаты представлены в таблице 4.1. Пропущенное значение означает, что для таких параметров тестирование не проводилось.

Таблица 4.1: Сравнительная таблица времени работы алгоритмов (время в нс)

Размер	Итер.	Рекурс.(с матрицей)	Рекурс.	Дамерау
0	3125	4687	1562	4687
1	6250	12500	6250	14062
2	10937	29687	28125	39062
3	15625	59375	125000	75000
4	23437	93312	620312	99375
5	28456	95750	2312500	103750
6	32812	167187	17723437	237500
7	56250	259375	93750000	421875
8	78125	421875	734375000	484375
9	95000	473500	12062500000	540605
10	109375	515625	14890625000	625000
15	312500	1562500	-	2187500
20	468750	2656250	-	3281250
25	647850	3593750	-	4843750
30	937500	5625000	-	7500000
35	1250000	6562500	-	9531250
40	1875000	8593750	-	12500000
45	2362500	10312500	-	15312500
50	2875000	12968750	-	24062500
60	3906250	21718750	-	31093750
70	4687500	28437500	-	42812500
80	7531250	53437500	-	55625000
90	10312500	69375000	-	69218750
100	14625000	77500000	-	92656250

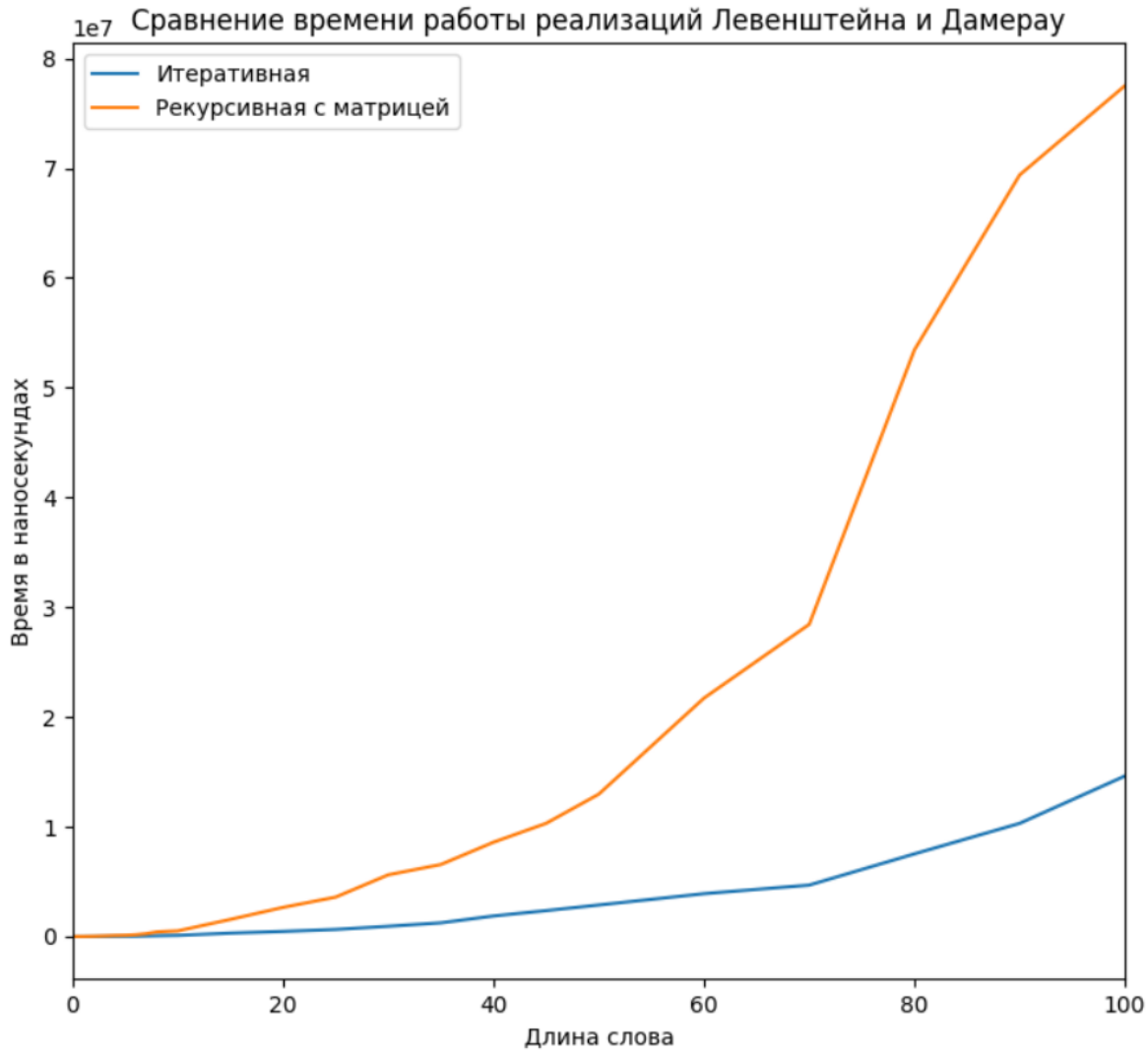

Рисунок. 4.2: Сравнительный график времени работы итерационного и рекурсивного алгоритмов нахождения расстояния Левенштейна

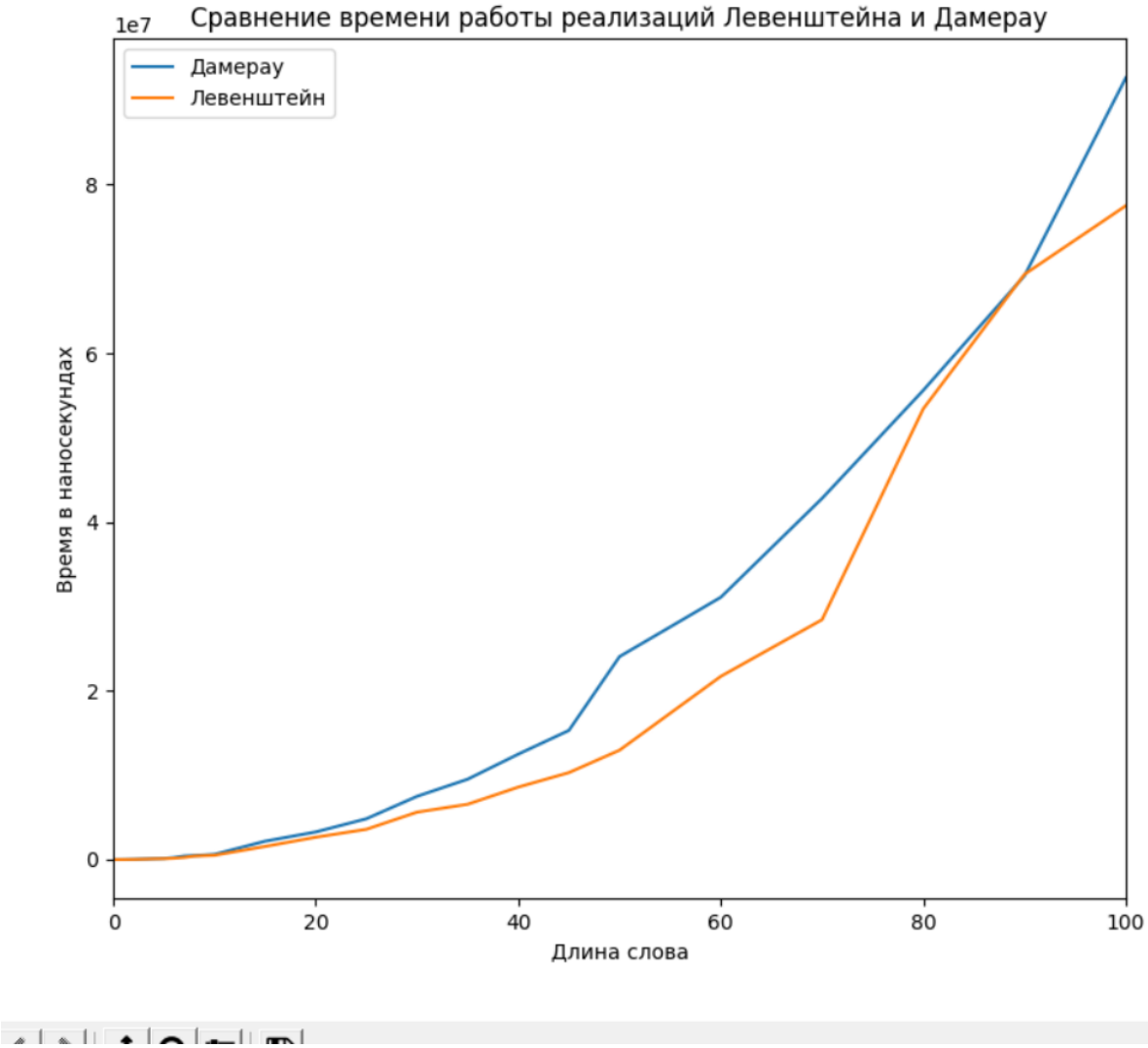

Рисунок. 4.3: Сравнительный график времени работы одинаковых алгоритмов поиска расстояния Левенштейна и расстояния Дамерау-Левенштейна

4.4 Использование памяти

Алгоритмы Левенштейна и Дамерау — Левенштейна не отличаются друг от друга с точки зрения использования памяти, следовательно, достаточно рассмотреть лишь разницу рекурсивной и матричной реализаций этих алгоритмов.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, соответственно, максимальный расход памяти (4.1)

$$(\mathcal{C}(S_1) + \mathcal{C}(S_2)) \cdot (2 \cdot \mathcal{C}(\text{string}) + 3 \cdot \mathcal{C}(\text{int})), \tag{4.1}$$

где $\mathcal{C}$ — оператор вычисления размера, S_1, S_2 — строки, int — целочисленный тип, string — строковый тип.

Использование памяти при итеративной реализации теоретически равно

$$(\mathcal{C}(S_1) + 1) \cdot (\mathcal{C}(S_2) + 1) \cdot \mathcal{C}(\text{int}) + 7 \cdot \mathcal{C}(\text{int}) + 2 \cdot \mathcal{C}(\text{string}). \tag{4.2}$$

Но в случае хранения двух строк количество потребляемой памяти резко падает:

$$(\mathcal{C}(S_2) + 1) \cdot 3 \cdot \mathcal{C}(\text{int}) + 7 \cdot \mathcal{C}(\text{int}) + 2 \cdot \mathcal{C}(\text{string}). \tag{4.3}$$

4.5 Вывод

Рекурсивный алгоритм Левенштейна работает на порядок дольше итеративных реализаций, время его работы увеличивается в геометрической прогрессии. На словах длиной 10 символов, матричная реализация алгоритма Левенштейна превосходит по времени работы рекурсивную в 160000 раз. Рекурсивный алгоритм с заполнением матрицы превосходит простой рекурсивный на аналогичных данных в 29000 раз. Алгоритм Дамерау — Левенштейна по времени выполнения выполняется медленнее, чем аналогичная реализация алгоритма Левенштейна. В нём добавлены дополнительные проверки, и по сути он является алгоритмом другого смыслового

уровня.

По расходу памяти классический итерационный алгоритм проигрывает рекурсионному, но в случае оптимизации итерационный алгоритм становится выигрышней по памяти, чем рекурсионные аналоги.

5 Заключение

В ходе выполнения работы были выполнены все поставленные задачи и изучены методы динамического программирования на основе алгоритмов вычисления расстояния Левенштейна.

Экспериментально были установлены различия в производительности различных алгоритмов вычисления расстояния Левенштейна. Для слов длины 10 рекурсивный алгоритм Левенштейна работает на несколько порядков медленнее (160000 раз) матричной реализации. Рекурсивный алгоритм с параллельным заполнением матрицы работает быстрее простого рекурсивного (в 29000 раз), но все еще медленнее матричного (в 5.5 раз). Если длина сравниваемых строк превышает 10, рекурсивный алгоритм становится неприемлимым для использования по времени выполнения программы. Матричная реализация алгоритма Дамерау — Левенштейна сопоставимо с алгоритмом Левенштейна. В ней добавлены дополнительные проверки, но, эти алгоритмы находятся в разных полях использования.

Теоретически было рассчитано использования памяти в каждом из алгоритмов вычисления расстояния Левенштейна. Обычные матричные алгоритмы потребляют намного больше памяти, чем рекурсивные, за счет дополнительного выделения памяти под матрицу. Оптимизация матричного варианта позволяет сократить объём потребляемой памяти, что делает его самым производительным вариантом.

Литература

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

[2] Умные алгоритмы обработки строк в ClickHouse. Режим доступа: https://habr.com/ru/company/yandex/blog/466183/ (дата обращения: 21.09.2021).

[3] Применение расстояний редактирования при биоинформационном анализе геномов для задач оценки состояния репродуктивной системы. Режим доступа: https://fundamental-research.ru/ru/article/view?id=38819 (дата обращения: 21.09.2021).

[4] Обзор методов нечеткого поиска текстовой информации.

[5] time - Time access and conversions. Режим доступа: https://docs.python.org/3/library/time.html (дата обращения: 21.09.2021).