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”

на тему:

«Редакционные расстояния»

Студент **Голикова С. М.**

Группа **ИУ7-55Б**

Оценка (баллы)

Преподаватели **Волкова Л. Л., Строганов Ю. В.**

Москва, 2023г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Расстояние Левенштейна – мера, которая определяет, насколько две строки различаются между собой. Фактически данная величина определяет, сколько односимвольных изменений (вставки, удаления, замены) требуется для преобразования одной последовательности символов к другой.

Модификацией расстояния Левенштейна является расстояние Дамерау-Левенштейна, в котором к операциям вставки, удаления и замены добавляется операция транспозиции (перестановки) символов. Данную меру чаще всего используют, когда текст набирается с клавиатуры и возрастает вероятность ошибки перестановки двух соседних симолов.

Расстояния Левенштейна и Дамерау-Левенштейна нашли широкое применение в следующих сферах:

- компьютерная лингвистика (автоматическое исправление ошибок в тексте, обнаружение возможных ошибок в поисковых запросах, расчет изменений в различных версиях текста);
- биоинформатика (сравнение последовательностей генов).

Целью работы является разработка, реализация и исследование алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

В рамках выполнения работы необходимо решить следующие задачи:

- изучить расстояния Левенштейна и Дамерау-Левенштейна;
- разработать и реализовать алгоритмы нахождения изученных расстояний;
- провести сравнительный анализ процессорного времени выполнения реализаций данных алгоритмов;
- провести сравнительный анализ максимальной затрачиваемой алгоритмами памяти.

1 Аналитический раздел

В данном разделе будут представлены описания редакционных расстояний Левенштейна и Дамерау-Левенштейна, а также варианты реализации поиска этих расстояний.

1.1 Расстояние Левенштейна

Расстояние Левенштейна [1] – это минимальное количество редакторских операций, необходимых для преобразования одной строки в другую.

Используются следующие редакторские операции:

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

Операциям I, D и R назначают цену (штраф) 1. Также существует обозначение M (англ. match) – совпадение символов. Штраф M составляет 0, т.к. в случае совпадения символов никаких действий не производится.

Задача нахождения расстояния Левенштейна сводится к поиску последовательности действий, минимизирующих суммарный штраф.

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом. Тогда расстояние Левенштейна можно подсчитать по следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0 & \text{i = 0, j = 0} \\ i & \text{j = 0, i > 0} \\ j & \text{i = 0, j > 0} \\ \min\{ & \\ \quad D(i,j-1)+1, & \\ \quad D(i-1,j)+1, & \text{i > 0, j > 0} \\ \quad D(i-1,j-1)+m(S_1[i],S_2[j]). & \\ \} & \end{cases} \tag{1.1}$$

Функция m определена как:

$$m(a,b) = \begin{cases} 0, & \text{a = b,} \\ 1, & \text{иначе.} \end{cases} \tag{1.2}$$

1.1.1 Нерекурсивный алгоритм для определения расстояния Левенштейна (с использованием матрицы расстояний)

Введем матрицу размером $(length(S1)+1)$ x $((length(S2)+1)$, где $length(S)$ — длина строки S. Значение в ячейке $[i,j]$ равно значению $D(S1[1...i], S2[1...j])$. Первая строка и первый столбец тривиальны.

Всю таблицу (за исключением первого столбца и первой строки) заполняем в соответствии с формулой 1.3.

$$A[i][j] = min \begin{cases} A[i-1][j]+1, \\ A[i][j-1]+1, \\ A[i-1][j-1]+m(S1[i],S2[j]). \end{cases} \tag{1.3}$$

Функция m определена как:

$$m(S1[i],S2[j]) = \begin{cases} 0, & \text{если } S1[i] = S2[j], \\ 1, & \text{иначе.} \end{cases} \tag{1.4}$$

В результате расстоянием Левенштейна будет ячейка матрицы с индексами $i = length(S1)$ и $j = length(S2)$.

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна является модификацией алгоритма Левенштейна и чаще всего используется при наборе текста с клавиатуры, т.к. в этом случае пользователь может допустить опечатку, переставив местами два соседних символа.

К существующим редакторским операциям добавляется еще одна - X (англ. Exchange), штраф за которую также составляет 1. Тогда функция D вычисляется по формуле 1.5.

$$D(i,j) = \begin{cases} 0 & \text{i = 0, j = 0} \\ i & \text{j = 0, i > 0} \\ j & \text{i = 0, j > 0} \\ \min\{ & \\ \quad D(i,j-1)+1, & \\ \quad D(i-1,j)+1, & \text{i > 0, j > 0} \\ \quad D(i-1,j-1)+m(a[i],b[j]), & \\ \quad \begin{cases} D(i-2,j-2)+1, & \text{если } i,j>1; \\ & a[i]=b[j-1]; \\ & b[j]=a[i-1] \\ \infty, & \text{иначе} \end{cases} & \\ \} & \end{cases} \tag{1.5}$$

1.2.1 Рекурсивный алгоритм для определения расстояния Дамерау-Левенштейна

Рекурсивный вариант напрямую реализует формулу 1.5, вызывая функцию D для величин, равных длине каждой из строк. Сразу можно отметить недостаток данной реализации: одно и то же значение будет подсчитано несколько раз, поэтому будет возникать проблема повторных вычислений.

1.2.2 Рекурсивный алгоритм для определения расстояния Дамерау-Левенштейна с кешированием

В качестве оптимизации предыдущего 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 если да, то будет использовать уже готовый вариант.

1.2.3 Нерекурсивный алгоритм для определения расстояния Дамерау-Левенштейна

Нерекурсивный алгоритм для определения расстояния Дамерау-Левенштейна с использованием матрицы аналогичен нерекурсивному алгоритму для определения расстояния Левенштейна. Добавляется лишь последнее условие из формулы 1.5.

Вывод

Формулы для определения расстояния Левенштейна и Дамерау-Левенштейна между строками задаются рекуррентно, а следовательно, алгоритмы могут быть реализованы рекурсивно или итерационно.

В данном разделе были рассмотрены идеи итеративной реализации алгоритма для нахождения расстояния Левенштейна, а также идеи рекурсивной, рекурсивной с кешированием и итеративной реализации алгоритма для нахождения расстояния Дамерау-Левенштейна.

2 Конструкторский раздел

В данном разделе будут приведены схемы алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна. При разработке этих алгоритмов можно использовать разные подходы к реализации: итеративный, рекурсивный без кеширования, рекурсивный с кешированием.

2.1 Алгоритм поиска расстояния Левенштейна

На рисунке 1 приведена схема нерекурсивного алгоритма поиска расстояния Левенштейна с заполнением матрицы расстояний.

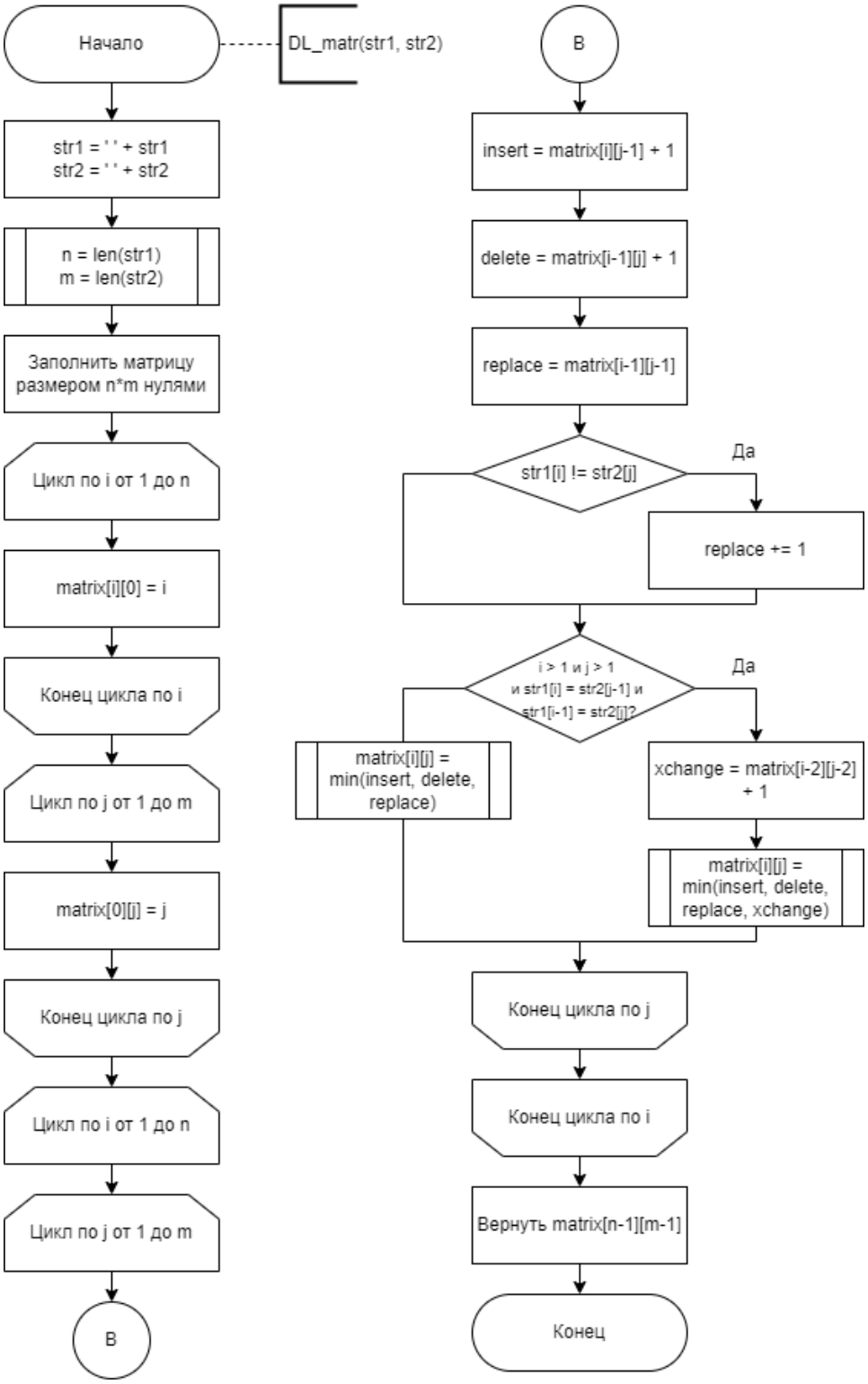

Рисунок 1 – Схема нерекурсивного алгоритма поиска расстояния Левенштейна

2.2 Алгоритмы поиска расстояния Дамерау-Левенштейна

На рисунках 2, 3, 4 приведены схемы следующих алгоритмов:

- итеративного алгоритма поиска расстояния Дамерау-Левенштейна с заполнением матрицы расстояний;
- рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна без кеширования;
- рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кешированием.

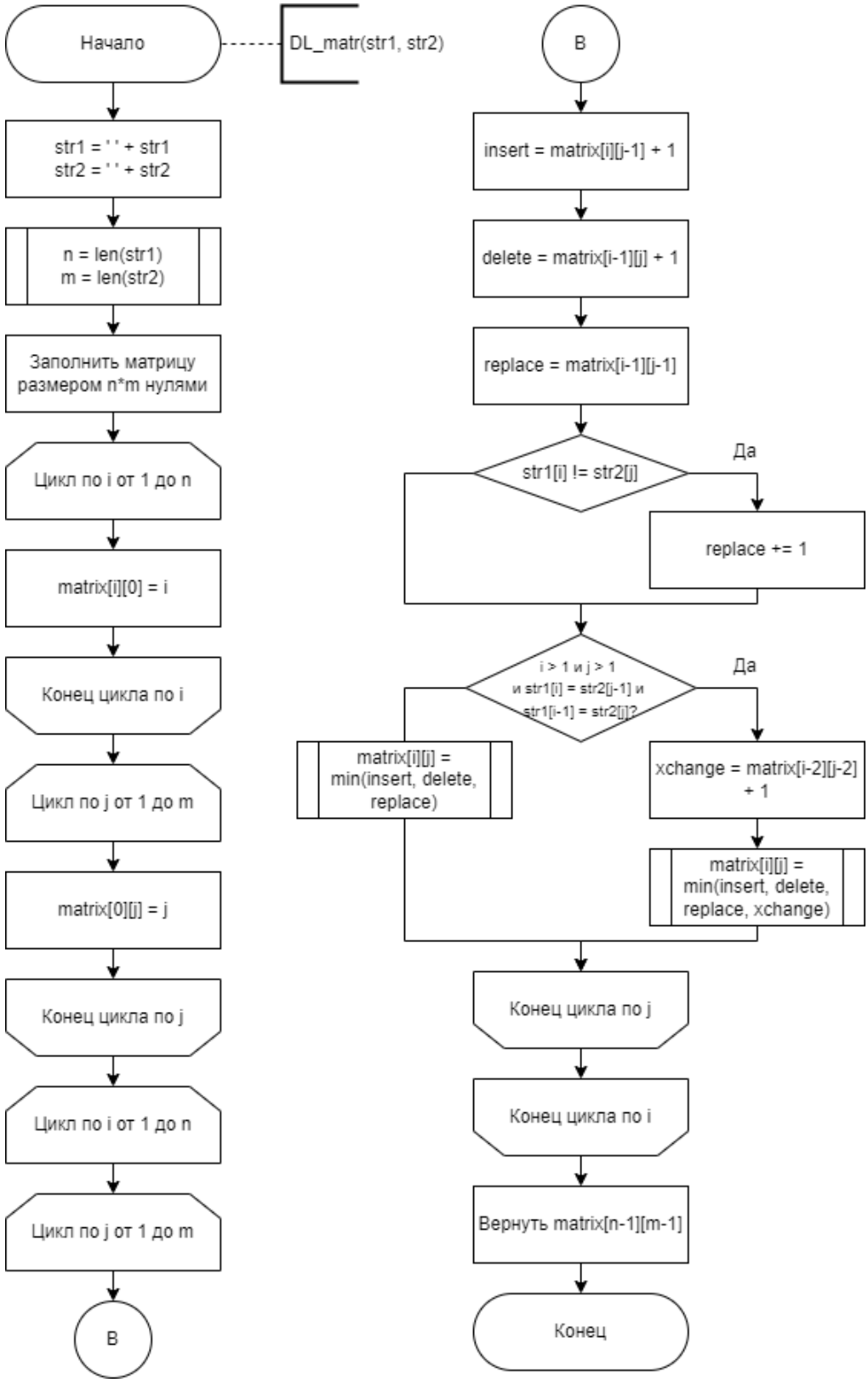

Рисунок 2 – Схема нерекурсивного алгоритма поиска расстояния Дамерау-Левенштейна

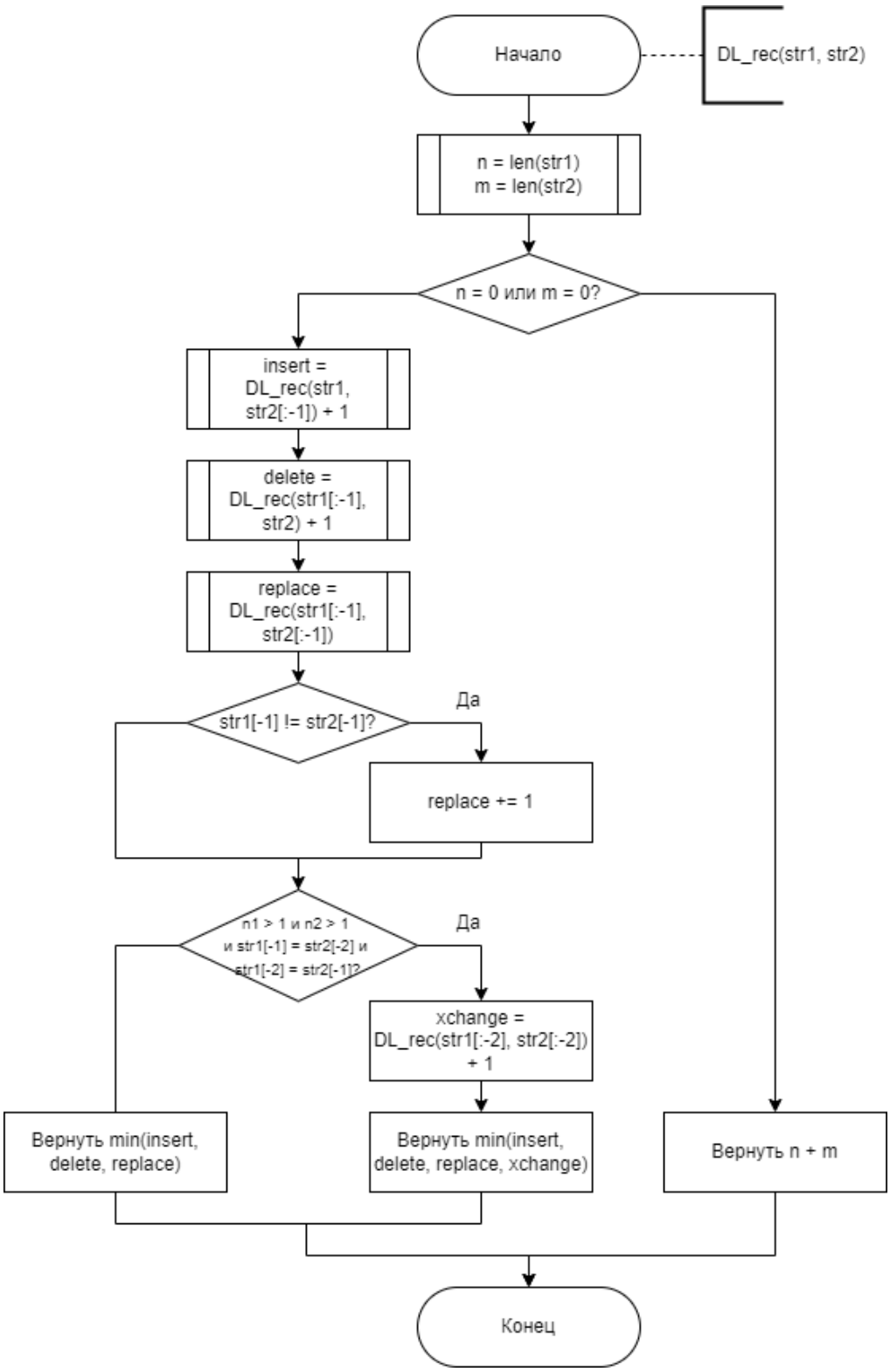

Рисунок 3 – Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна без кеширования

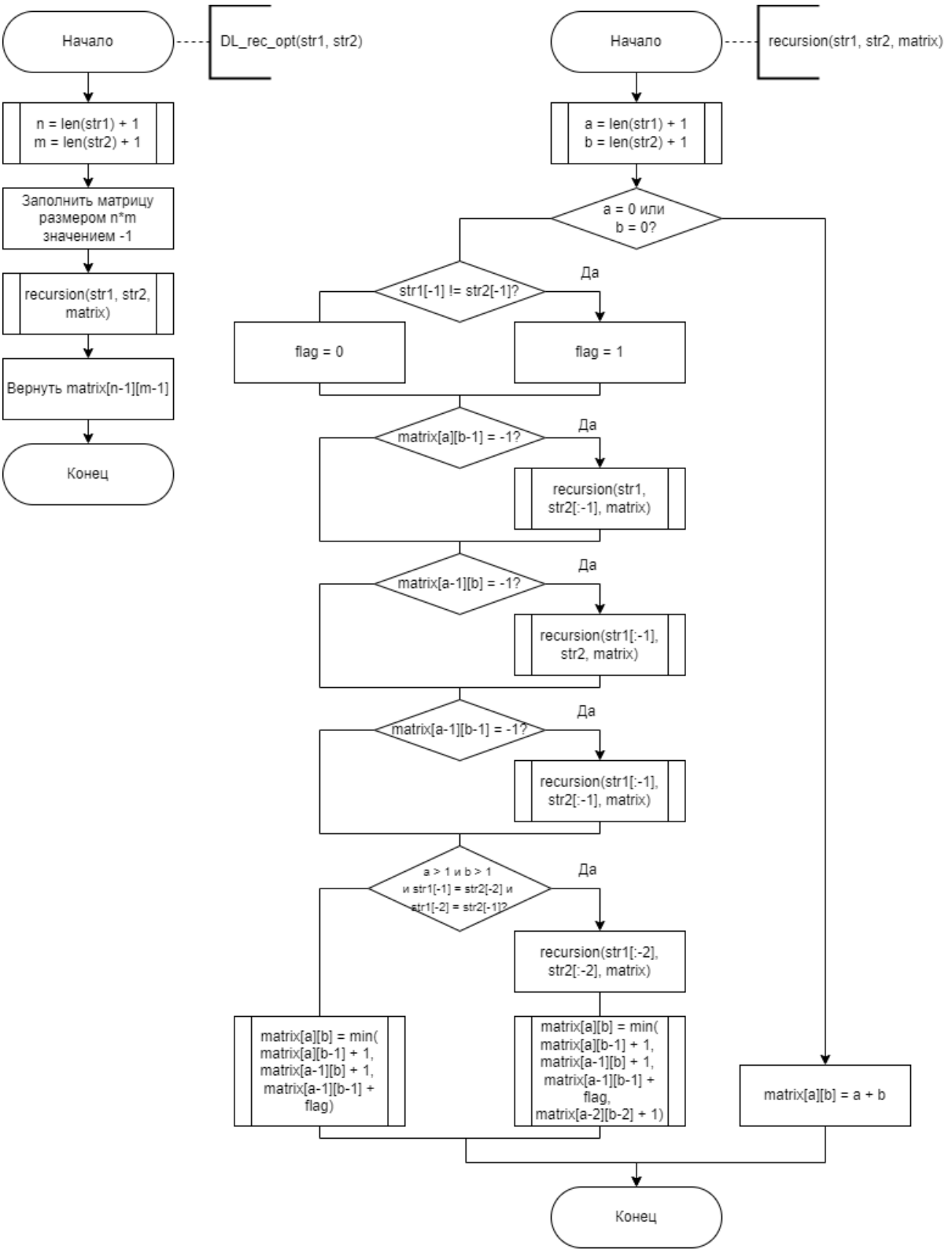

Рисунок 4 – Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кешированием

Вывод

Были разработаны схемы алгоритмов, позволяющих находить расстояния Левенштейна и Дамерау-Левенштейна.

3 Технологический раздел

В данном разделе будут приведены требования к программному обеспечению, средства реализации, листинги реализованных алгоритмов и тесты для программы.

3.1 Требования к программному обеспечению

Требования к входным данным:

- на вход подаются две строки;
- буквы верхнего и нижнего регистров считаются различными;
- строки могут быть пустыми.

Требования к выходным данным:

- искомое расстояние должно быть вычислено каждым из рассматриваемых алгоритмов;
- в режиме отладки должна быть выведена матрица расстояний.

3.2 Средства реализации

В качестве языка программирования для реализации лабораторной работы был выбран язык Python [2]. Данный язык предоставляет возможности работы со строками и матрицами.

Для замера процессорного времени использовалась функция библиотеки time process_time [3].

3.3 Реализации алгоритмов

В листингах 1 – 4 представлены реализации рассматриваемых алгоритмов поиска редакционного расстояния.

Листинг 1 – Функция нерекурсивного поиска расстояния Левенштейна

```
def lowenstein_dist_matrix(str1, str2):
    str1 = ' ' + str1
    str2 = ' ' + str2
    n = len(str1)
    m = len(str2)
    matrix = [[0] * m for _ in range(n)]

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

    for i in range(1, n):
        for j in range(1, m):
            insert = matrix[i][j-1] + 1
            delete = matrix[i-1][j] + 1
            replace = matrix[i-1][j-1] + int(str1[i] != str2[j])

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

    return matrix[n - 1][m - 1]
```

Листинг 2 – Функция нерекурсивного поиска расстояния Дамерау-Левенштейна

```
def damerau_lowenstein_dist_matrix(str1, str2):
    str1 = ' ' + str1
    str2 = ' ' + str2
    n = len(str1)
    m = len(str2)
    matrix = [[0] * m for _ in range(n)]

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

    for i in range(1, n):
        for j in range(1, m):
            insert = matrix[i][j-1] + 1
            delete = matrix[i-1][j] + 1
            replace = matrix[i-1][j-1] + int(str1[i] != str2[j])
            if (i > 1 and j > 1) and (str1[i] == str2[j - 1] and
               str1[i-1] == str2[j]):
                xchange = matrix[i-2][j - 2] + 1
                matrix[i][j] = min(insert,delete,replace,xchange)
            else:
                matrix[i][j] = min(insert,delete,replace)

    return matrix[n - 1][m - 1]
```

Листинг 3 – Функция рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна без кеширования

```
def dam_lowenstein_dist_rec(str1, str2):
    n = len(str1)
    m = len(str2)

    if n == 0 or m == 0:
        return n + m

    insert = dam_lowenstein_dist_rec(str1, str2[:-1]) + 1
    delete = dam_lowenstein_dist_rec(str1[:-1], str2) + 1
    replace = dam_lowenstein_dist_rec(str1[:-1], str2[:-1]) + \
                                        int(str1[-1] != str2[-1])

    if (n > 1 and m > 1) and \
       (str1[-1] == str2[-2] and str1[-2] == str2[-1]):
        xchange = dam_lowenstein_dist_rec(str1[:-2], str2[:-2])+1
        return min(insert,delete,replace,xchange)
    else:
        return min(insert,delete,replace)
```

Листинг 4 – Функция рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кешированием

```
def dam_lowenstein_dist_rec_optimized(str1, str2):
    def rec(str1, str2, a):
        len1 = len(str1)
        len2 = len(str2)
        if len1 == 0 or len2 == 0:
            a[len1][len2] = len1 + len2
        else:
            if a[len1][len2 - 1] == -1:
                rec(str1, str2[:-1], a)
            if a[len1 - 1][len2] == -1:
                rec(str1[:-1], str2, a)
            if a[len1 - 1][len2 - 1] == -1:
                rec(str1[:-1], str2[:-1], a)

            if (len1>1 and len2>1) and (str1[-1] == str2[-2]
                                        and str1[-2] == str2[-1]):
                if a[len1-2][len2-2] == -1:
                    rec(str1[:-2], str2[:-2], a)
                a[len1][len2] = min(a[len1][len2-1]+1,
                a[len1-1][len2]+1, a[len1-2][len2-2]+1,
                a[len1-1][len2-1] + int(str1[-1] != str2[-1]))
            else:
                a[len1][len2] = min(a[len1][len2-1] + 1,
                a[len1-1][len2] + 1,
                a[len1-1][len2 - 1] + int(str1[-1] != str2[-1]))
        return

    n = len(str1) + 1
    m = len(str2) + 1
    matrix = [[-1] * m for _ in range(n)]
    rec(str1, str2, matrix)
    return matrix[n - 1][m - 1]
```

3.4 Тестирование

В таблице 1 приведены функциональные тесты для алгоритмов вычисления расстояний Левенштейна и Дамерау—Левенштейна.

В таблице приняты обозначения: РЛ - алгоритм поиска расстояния Левенштейна, РДЛ - алгоритм поиска расстояния Дамерау-Левенштейна.

Таблица 1 – Функциональные тесты

Строка 1	Строка 2	Ожидаемый результат	
		РЛ	РДЛ
		0	0
кот	скат	2	2
осень	очеьн	3	2
сон	сноп	2	2
мир	мира	1	1
мира	мир	1	1
рим	ром	1	1
дождь	длджь	3	2

Все тесты пройдены успешно.

Вывод

Был произведен выбор средств реализации, реализованы и протестированы алгоритмы поиска расстояний: Левенштейна – итерационный, Дамерау-Левенштейна – итерационный и рекурсивный (с кешем и без).

4 Исследовательский раздел

В данном разделе будет приведена демонстрация работы программы, а также произведен сравнительный анализ алгоритмов на основе времени их работы и затрачиваемой памяти.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система Windows 10.
- память 8 ГБ.
- процессор Intel® Core™ i5-6260U @ 1.80ГГц.

Замеры времени выполнения реализаций алгоритмов проводились на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно разработанным приложением.

4.2 Демонстрация работы программы

На рисунке 5 представлен пример работы программы. Пользователь вводит две строки, в результате работы программы на экран выводятся искомые расстояния и матрицы расстояний для всех случаев, кроме рекурсивного без кеша.

```
Введите строку 1: осень
Введите строку 2: очень

Левенштейн, итерационный
Матрица расстояний:
[0, 1, 2, 3, 4, 5]
[1, 0, 1, 2, 3, 4]
[2, 1, 1, 2, 3, 4]
[3, 2, 2, 1, 2, 3]
[4, 3, 3, 2, 2, 2]
[5, 4, 4, 3, 2, 3]
Ответ: 3

Дамерау-Левенштейн, итерационный
Матрица расстояний:
[0, 1, 2, 3, 4, 5]
[1, 0, 1, 2, 3, 4]
[2, 1, 1, 2, 3, 4]
[3, 2, 2, 1, 2, 3]
[4, 3, 3, 2, 2, 2]
[5, 4, 4, 3, 2, 2]
Ответ: 2

Дамерау-Левенштейн, рекурсивный без кеша
Ответ: 2

Дамерау-Левенштейн, рекурсивный с кешем
Матрица расстояний:
[0, 1, 2, 3, 4, 5]
[1, 0, 1, 2, 3, 4]
[2, 1, 1, 2, 3, 4]
[3, 2, 2, 1, 2, 3]
[4, 3, 3, 2, 2, 2]
[5, 4, 4, 3, 2, 2]
Ответ: 2
```

Рисунок 5 – Пример работы программы

4.3 Сравнение времени выполнения реализаций алгоритмов

Сравнение времени выполнения реализаций алгоритмов производилось на строках длиной от 0 до 10 с шагом 1 для всех реализаций и на строках длиной от 0 до 200 с шагом 10 для реализаций, использующих матрицы расстояний.

Так как замеры времени имеют некоторую погрешность, они производились 20 раз для каждой реализации алгоритма и длины строки, а затем вычислялось среднее время работы реализации с текущей длиной строки.

На рисунке 6 приведены результаты сравнения времени работы всех реализаций в секундах. Как видно на графике, реализации, использующие матрицы расстояний, работают значительно быстрее рекурсивной реализации без кеширования. Это обусловлено отсутвием в первых вызова функций для вычисления значений, которые уже были подсчитаны ранее.

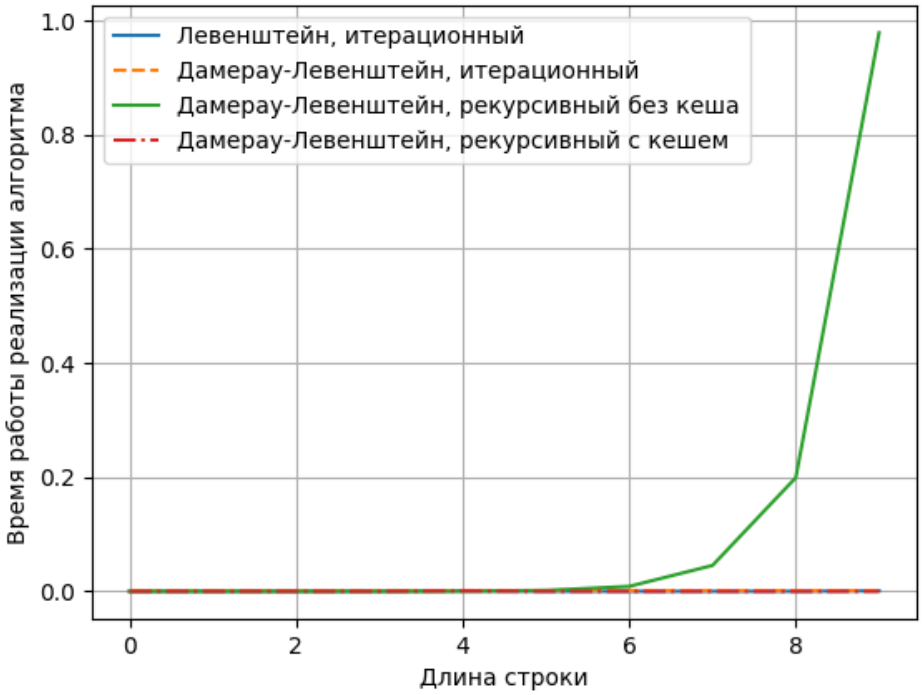

Рисунок 6 – Сравнение времени работы реализаций алгоритмов

Сравним отдельно реализации, использующие матрицы расстояний.

На рисунке 7 приведено сравнение времени выполнения реализаций итерационного алгоритма поиска расстояния Левенштейна, итерационного и рекурсивного с кешированием алгоритмов поика расстояния Дамерау-Левенштейна. Время их работы растет соизмеримо, что обусловлено их схожестью в отсутствии вызова функций для вычисления значений, которые уже были подсчитаны ранее. Однако рекурсивная реализация с кешированием все же работает дольше, так как в ней тратится время на рекурсивный вызов.

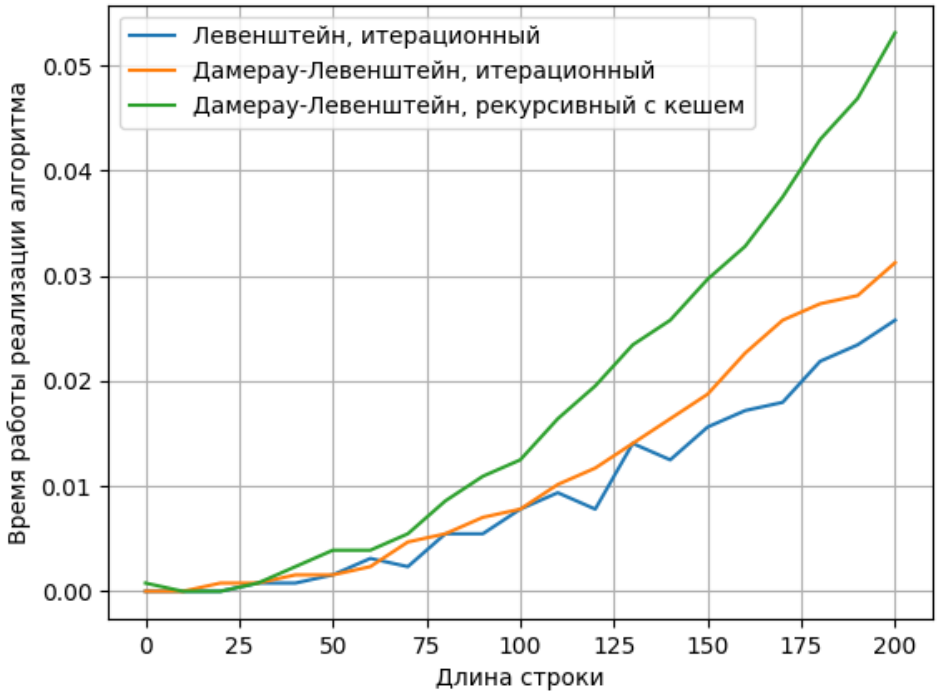

Рисунок 7 – Сравнение времени работы реализациий алгоритмов, использующих матрицы растояний

4.4 Оценка памяти

Пусть m, n - длины строк $S1$ и $S2$ соответственно.

Затраты памяти для итеративного алгоритма поиска расстояния Левенштейна с матрицей расстояний:

- $2 \cdot sizeof(string\_pointer)$ (ссылки на строки $S1, S2$);
- $2 \cdot sizeof(int)$ (длины строк);
- $((m+1) \cdot (n+1)) \cdot sizeof(int)$ (матрица);
- $3 \cdot sizeof(int)$ (вспомогательные переменные).

Результат: $((m+1) \cdot (n+1) + 5) \cdot sizeof(int) + 2 \cdot sizeof(string\_pointer)$

Затраты памяти для итеративного алгоритма поиска расстояния Дамерау-Левенштейна с матрицей расстояний:

- $2 \cdot sizeof(string\_pointer)$ (ссылки на строки $S1, S2$);
- $2 \cdot sizeof(int)$ (длины строк);
- $((m+1) \cdot (n+1)) \cdot sizeof(int)$ (матрица);

- $4 \cdot sizeof(int)$ (вспомогательные переменные).

Результат: $((m+1) \cdot (n+1) + 6) \cdot sizeof(int) + 2 \cdot sizeof(string\_pointer)$

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк. Ниже приведены оценки памяти для каждого вызова рекурсивной функции и итоговая затрачиваемая память с учетом максимальной глубины стека.

Затраты памяти для рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна (для каждого вызова):

- $2 \cdot sizeof(string\_pointer)$ (ссылки на строки $S1, S2$);
- $2 \cdot sizeof(int)$ (длины строк);
- $4 \cdot sizeof(int)$ (вспомогательные переменные).

Результат: $(m+n) \cdot (6 \cdot sizeof(int) + 2 \cdot sizeof(string\_pointer))$

Затраты памяти для рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с использованием кеша (для каждого вызова):

- $2 \cdot sizeof(string\_pointer)$ (ссылки на строки $S1, S2$);
- $2 \cdot sizeof(int)$ (длины строк);
- $((m+1) \cdot (n+1)) \cdot sizeof(int)$ (матрица).

Результат: $(m+n) \cdot (((m+1) \cdot (n+1) + 6) \cdot sizeof(int) + 2 \cdot sizeof(string\_pointer))$

4.5 Вывод

Рекурсивная реализация алгоритма поиска расстояния Дамерау-Левенштейна без кеширования работает значительно дольше реализаций алгоритмов, в которых используется матрица расстояний.

Рекурсивный алгоритм с кешированием сравним с итерационными алгоритмами, однако его реализация все равно несколько дольше нерекурсивных аналогов.

Итерационные реализации алгоритмов Левенштейна и Дамерау-Левенштейна сопоставимы по времени. Реализация алгоритма Дамерау-Левенштейна незначительно дольше реализации Левенштейна, т.к. в первом случае имеет место дополнительная проверка на перестановку соседних сим-

волов.

По расходу памяти нерекурсивные алгоритмы проигрывают рекурсивному алгоритму без кеша на больших длинах строк: в первом случае максимальный размер используемой памяти растет пропорционально произведению длин строк, в то время как во втором – пропорционально сумме длин строк.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были решены следующие задачи:

- изучены расстояния Левенштейна и Дамерау-Левенштейна;
- разработаны и реализованы алгоритм поиска расстояния Левенштейна с заполнением матрицы, алгоритмы поиска расстояния Дамерау-Левенштейна с заполнением матрицы, с использованием рекурсии (с кешем и без);
- проведен сравнительный анализ процессорного времени выполнения реализаций рассматриваемых алгоритмов;
- проведен сравнительный анализ максимальной затрачиваемой алгоритмами памяти.

Поставленная цель была достигнута: были разработаны, реализованы и исследованы алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

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

2. Лутц Марк. Изучаем Python, том 1, 5-е изд. Пер. с англ. — СПб.: ООО “Диалектика”, 2019. с. 832.

3. time — Time access and conversions [Электронный ресурс]. Режим доступа: `https://docs.python.org/3/library/time.html` (дата обращения: 15.09.2022).