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 между строками»

Студент ИУ7-54Б (Группа) _____ (Подпись, дата) Булдаков М. (И. О. Фамилия)

Преподаватель _____ (Подпись, дата) Волкова Л. Л. (И. О. Фамилия)

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Расстояние Левенштейна — минимальное количество редакционных операций, которое необходимо выполнить для преобразования одной строки в другую [1]. Редакционными являются следующие операции:

— I — вставка одного символа (insert);

— D — удаление (delete);

— R — замена (replace).

Также обозначим совпадение как M (match).

Расстояние Дамерау—Левенштейна является модификацией расстояния Левенштейна, отличается от него добавлением операции транспозиции (перестановки).

Редакционные расстояния применяются для решения следующих задач:

— исправление ошибок в словах;

— обучение языковых моделей (расстояние Левенштейна используется в качестве метрики соответствия слов, полученных в результате генерации, и эталонных слов);

— сравнение геномов, хромосом и белков в биоинформатике.

Целью данной лабораторной работы является исследование алгоритмов, вычисляющих расстояния Левенштейна и Дамерау—Левенштейна.

Для достижения поставленной цели необходимо выполнить следующие задачи.

1) Описать алгоритмы, вычисляющие расстояния Левенштейна и Дамерау—Левенштейна.

2) Выбрать инструменты для реализации и замера процессорного времени выполнения реализаций.

3) Разработать программное обеспечение, реализующее следующие алгоритмы:

- нерекурсивный алгоритм поиска расстояния Дамерау—Левенштейна;
- рекурсивный алгоритм поиска расстояния Дамерау—Левенштейна без кеширования;
- рекурсивный алгоритм поиска расстояния Дамерау—Левенштейна с кешированием;
- нерекурсивный алгоритм поиска расстояния Левенштейна.

4) Проанализировать затраты реализаций алгоритмов по времени и по памяти.

1 Аналитический раздел

Каждая редакционная операция имеет свой штраф, который определяет стоимость данной операции. В общем случае:

— $m(a, b)$ — цена замены символа a на b, при $a \neq b$;

— $m(\lambda, a)$ — цена вставки символа a;

— $m(a, \lambda)$ — цена удаления символа a.

1.1 Расстояние Левенштейна

При вычислении расстояния Левенштейна будем считать стоимость каждой редакционной операции равной 1, при этом, если символы совпадают, то штраф равен 0, т. о. штрафы вычисляются следующим способом:

— $m(a, b) = 1$;

— $m(\lambda, a) = 1$;

— $m(a, \lambda) = 1$;

— $m(a, a) = 0$.

Пусть S_1 и S_2 — две строки (длинной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно вычислить по следующей рекуррентной формуле (1.1).

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

Значение $m(a, b)$ можно рассчитать по формуле (1.2).

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

Нерекурсивный алгоритм нахождения расстояния Левенштейна

Используя матрицу $A_{(M+1)\times(N+1)}$ для хранения промежуточных значений, сведем задачу к итерационному заполнению матрицы $A_{(M+1)\times(N+1)}$ значениями $D(i, j)$. Т. о. значение в ячейке $[i, j]$ равно значению $D(S_1[1...i], S_2[1...j])$.

1.2 Расстояние Дамерау—Левенштейна

Расстояние Дамерау—Левенштейна модифицирует расстояние Левенштейна, добавляя ко всем перечисленным операциям, операцию перестановки соседних символов. Штраф новой операции также составляет 1.

Расстояние Дамерау—Левенштейна может быть вычислено по рекуррентной формуле (1.3).

$$D(i,j) = \begin{cases} 0, & \text{i = 0, j = 0,} \\ i, & \text{j = 0, i > 0,} \\ j, & \text{i = 0, j > 0,} \\ min(& \\ D(i, j-1) + 1, & \\ D(i-1, j) + 1, & \\ D(i-1, j-1) + m(S_1[i], S_2[j]), & \\ \begin{cases} & \text{если i > 1, j > 1,} \\ D(i-2, j-2) + 1, & S_1[i] = S_2[j-1], \\ & S_1[i-1] = S_2[j], \\ \infty, & \text{иначе} \end{cases} & \\ ), & \text{иначе.} \end{cases} \tag{1.3}$$

Рекурсивный алгоритм нахождения расстояния Дамерау—Левенштейна

Используя рекурсию можно реализовать формулу (1.3). Рекурсия используется для обработки всех возможных вариантов преобразований (вставка, удаление, замена и транспозиция) и далее выбирается минимальное количество операций.

Рекурсивный алгоритм нахождения расстояния Дамерау—Левенштейна с кешем

Используя кеш, рекурсивный алгоритм вычисления расстояния по формуле (1.3) можно оптимизировать по времени выполнения. В качестве кеша используется матрица. Суть данной оптимизации заключается в сокращении числа лишних операций, производимых над одними и теми же подстроками несколько раз. В случае, если для текущих подстрок, значение расстояния отсутствует в кеше, то оно вычисляется с помощью рекурсивного алгоритма и заносится в матрицу. Если же значение присутствует в кеше, то алгоритм сразу переходит к следующему шагу.

Нерекурсивный алгоритм нахождения расстояния Дамерау—Левенштейна

Можно свести задачу вычисления расстояния Дамерау—Левенштейна к итерационному заполнению матрицы промежуточными значениями $D(i, j)$. При этом матрица будет иметь размер $(M + 1) \times (N + 1)$.

Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояний Левенштейна и Дамерау—Левенштейна, поскольку данные расстояния могут быть вычислены с помощью рекуррентных формул, то алгоритмы могут быть реализованы рекурсивно и итеративно.

2 Конструкторский раздел

В этом разделе будут приведены требования к вводу и программе, а также схемы алгоритмов нахождения расстояний Левенштейна и Дамерау—Левенштейна.

2.1 Требования к программному обеспечению

Программа должна поддерживать два режима работы: массированного замера времени и расчета расстояния. Программа в режиме массового замера должна обладать функциональностью:

— генерации строк различной длины для замеров;

— замера процессорного времени работы реализаций алгоритмов поиска расстояний Левенштейна и Дамерау—Левенштейна;

— вывода результатов в виде таблицы и графика.

К программе в режиме расчета расстояний предъявлен ряд требований:

— на вход подаются две строки, которые могут быть пустыми;

— на выходе — результат работы всех алгоритмов поиска расстояний, 4 целых числа и по 1 заполненной матрице для всех алгоритмов, кроме рекурсивного с расстоянием Дамерау—Левенштейна;

— наличие интерфейса для выбора действий;

— возможность обработки строк, включающих буквы как на латинице, так и на кириллице.

2.2 Разработка алгоритмов

На вход алгоритмов подаются строки $S1$ и $S2$.

На рисунке 2.1 представлена схема алгоритма поиска расстояния Левенштейна. На рисунках 2.2 – 2.6 представлены схемы алгоритмов поиска расстояния Дамерау—Левенштейна.

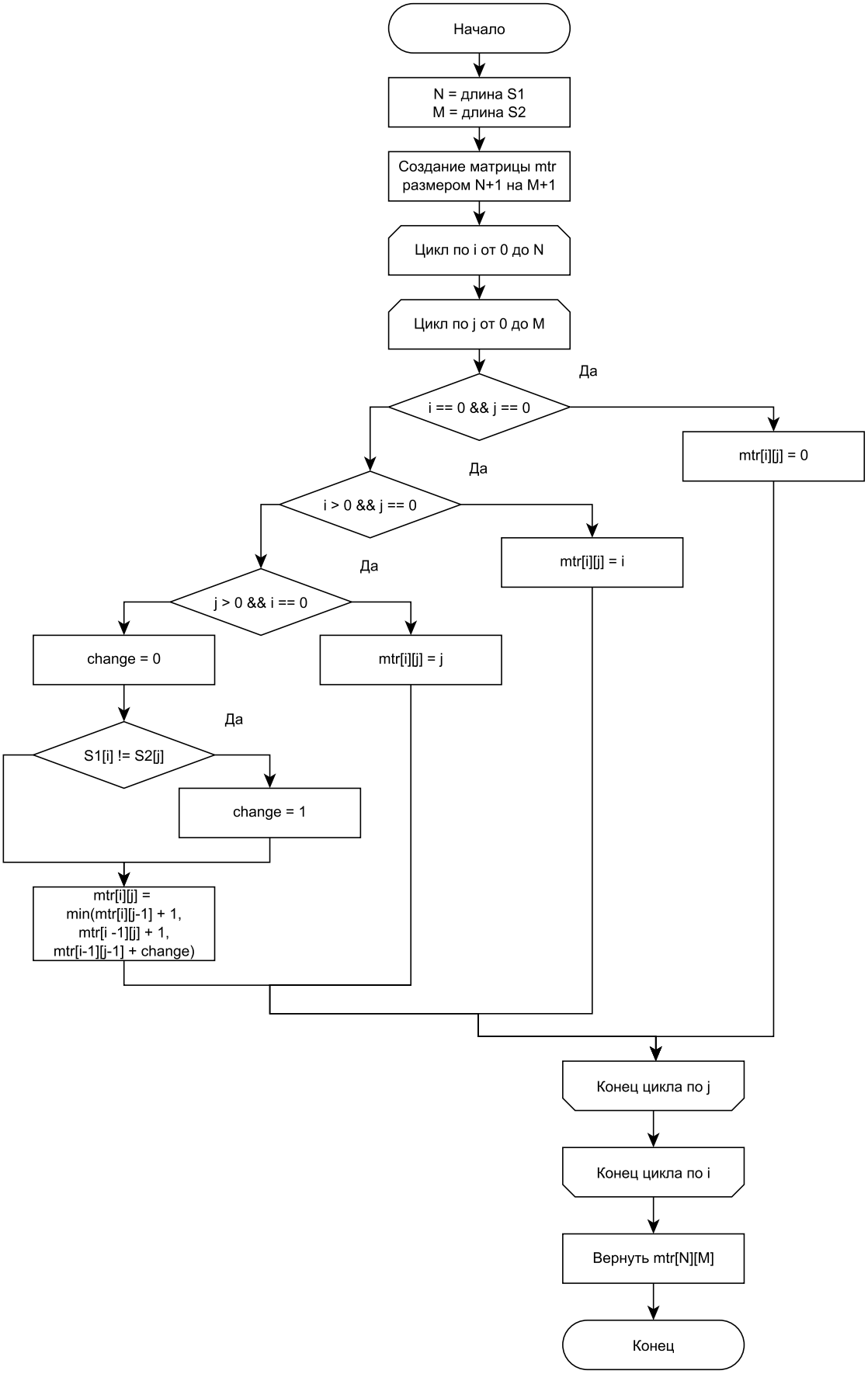

Рисунок 2.1 – Нерекурсивный алгоритм нахождения расстояния Левенштейна

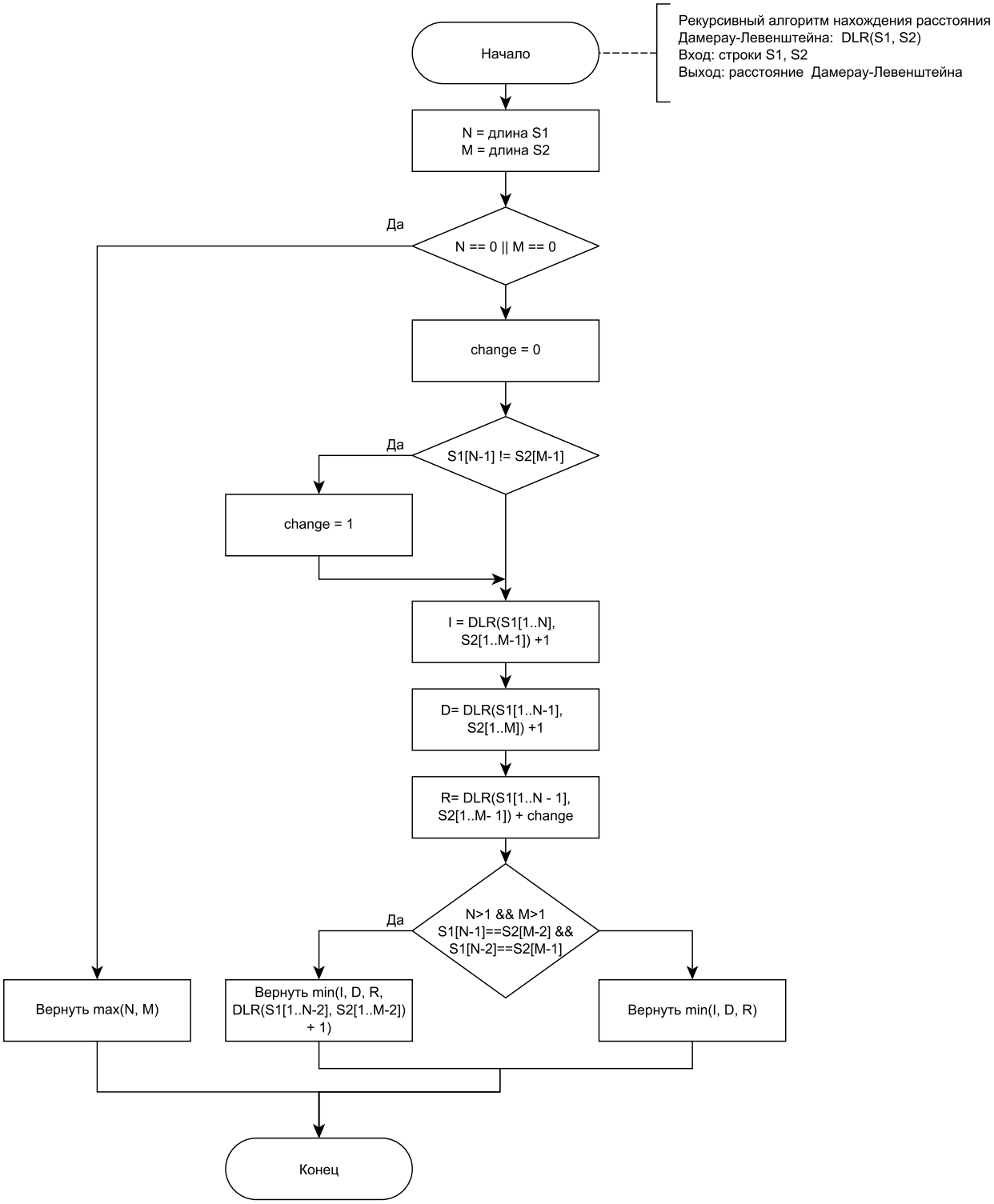

Рисунок 2.2 – Рекурсивный алгоритм нахождения расстояния Дамерау—Левенштейна

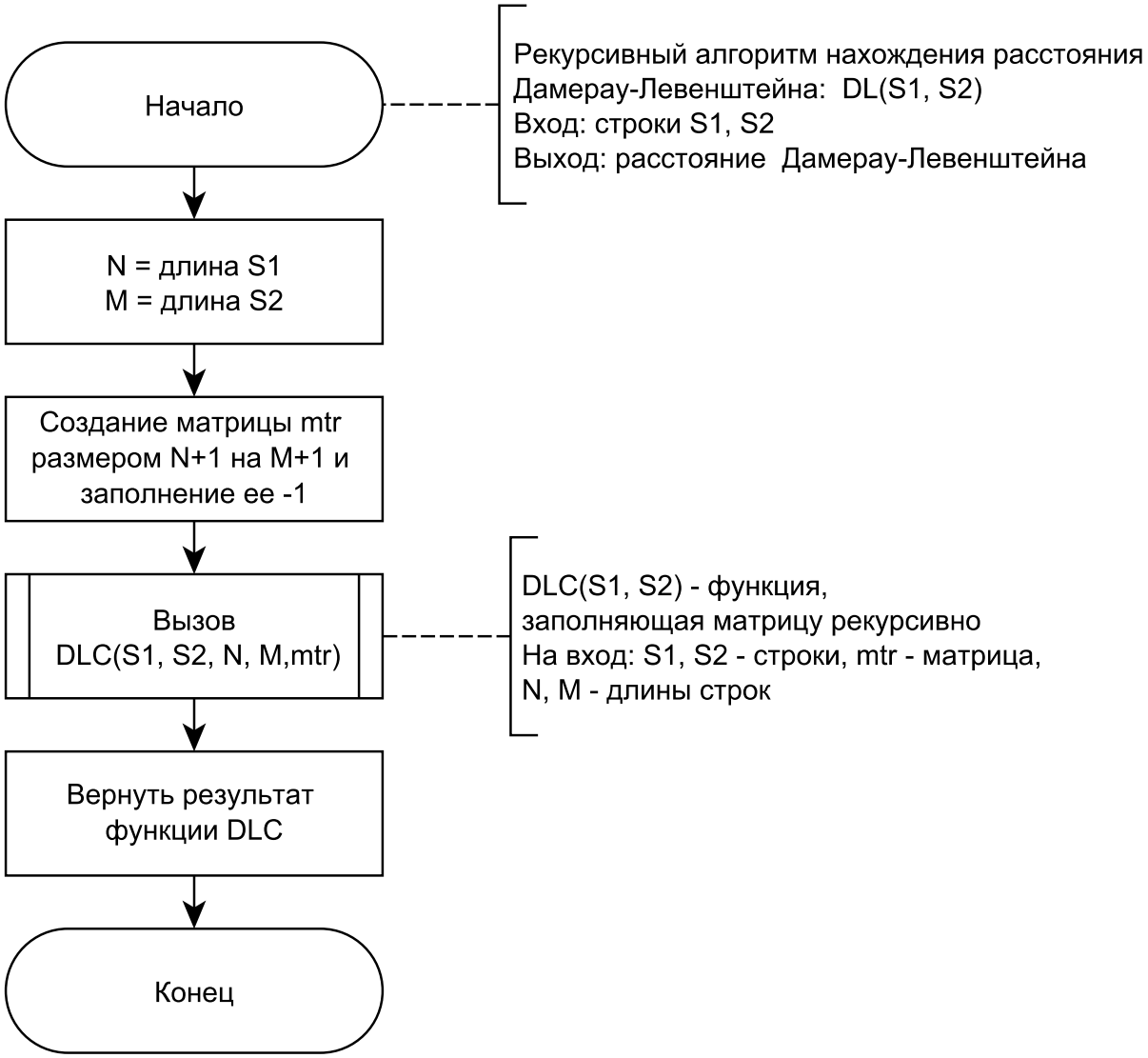

Рисунок 2.3 – Рекурсивный алгоритм нахождения расстояния Дамерау—Левенштейна с кешем

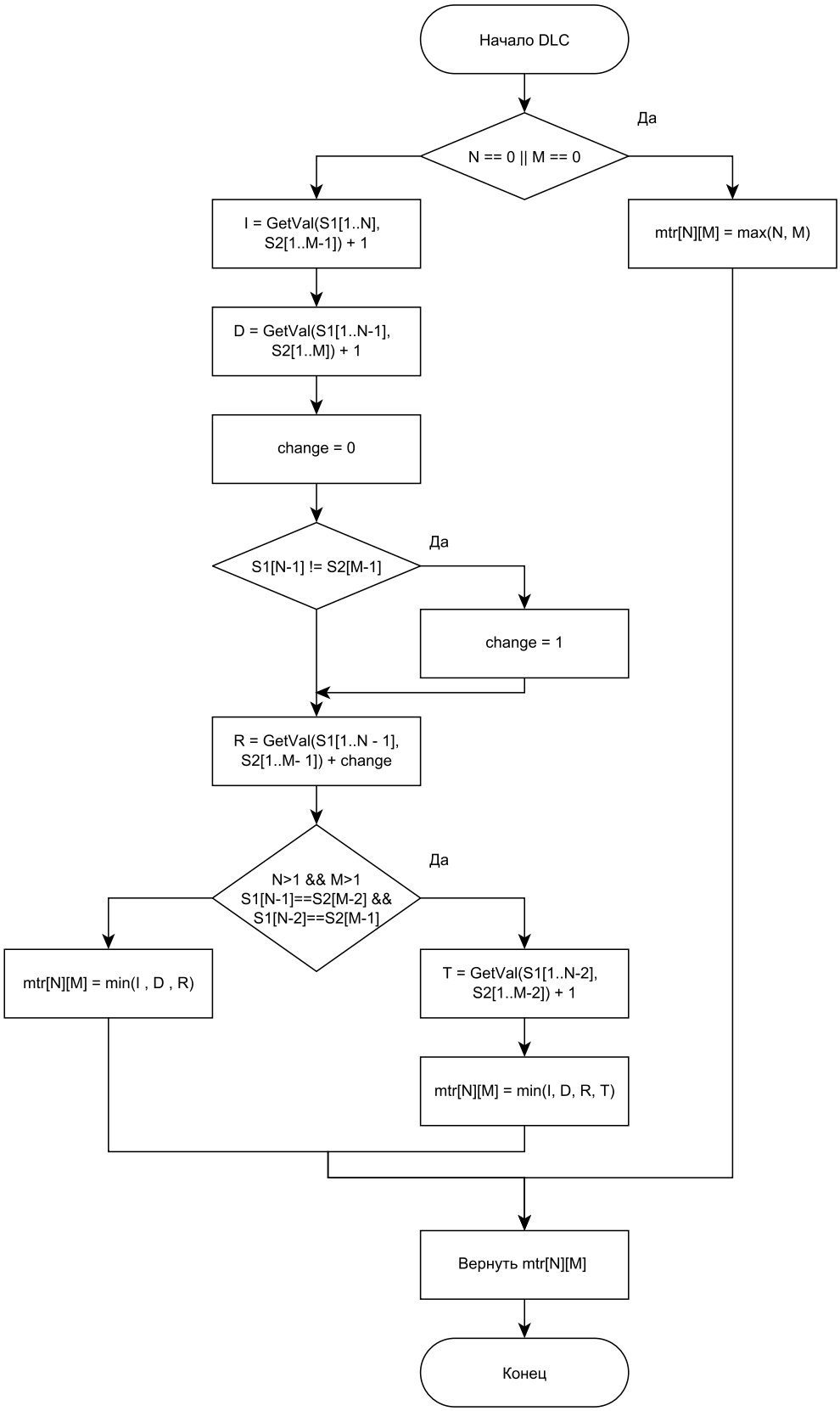

Рисунок 2.4 – Функция, заполняющая матрицу расстояний Дамерау—Левенштейна рекурсивно

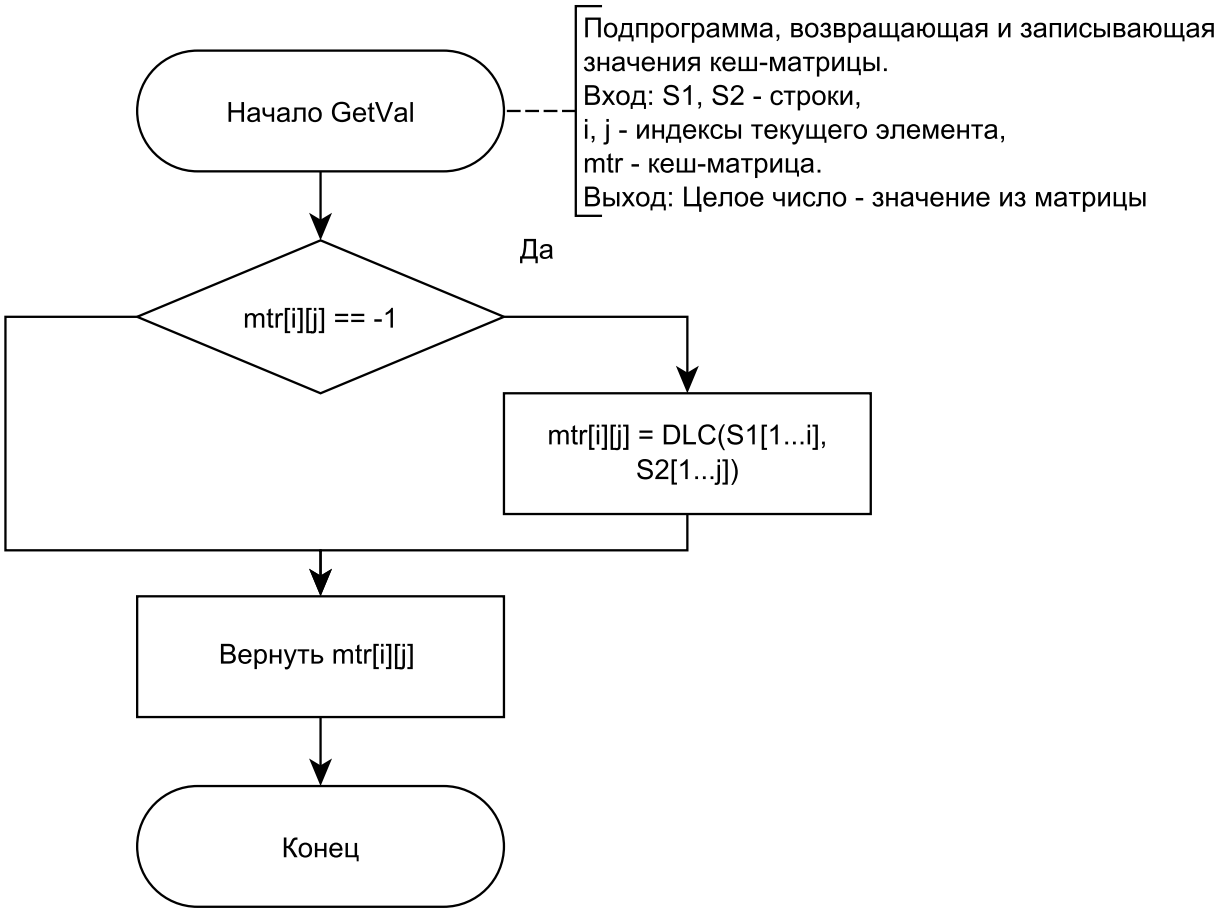

Рисунок 2.5 – Функция, извлекающая и записывающая значения кеш-матрицы

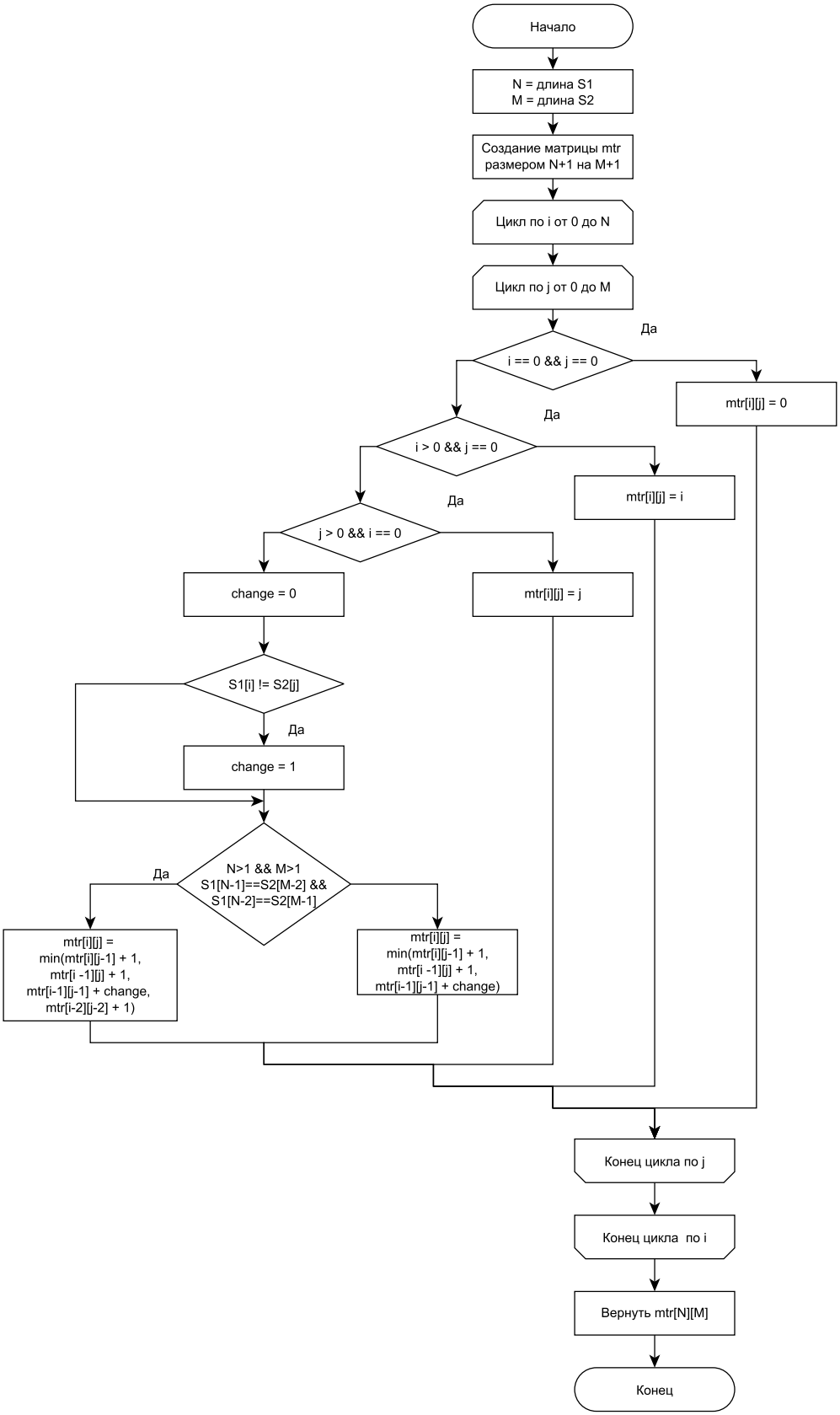

Рисунок 2.6 – Нерекурсивный алгоритм нахождения расстояния Дамерау—Левенштейна

2.3 Описание используемых типов и структур данных

Для реализации алгоритмов, будут использованы следующие типы данных:

— *str* — для двух строк, поданных на вход;

— *int* — для возвращаемого значения искомого расстояния.

При реализации алгоритмов будет использована структура данных — матрица, которая является двумерным списком значений типа *int*.

Вывод

В данном разделе на основе теоретических данных были построены схемы требуемых алгоритмов, выбраны используемые типы данных.

3 Технологический раздел

В данном разделе будут приведены требования к программному обеспечению, средства реализации, листинг кода и функциональные тесты.

3.1 Средства реализации

Для реализации данной работы был выбран язык *Python* [2]. Такой выбор обусловлен опытом работы с этим языком программирования. Также данный язык позволяет замерять процессорное время с помощью модуля *time*.

Процессорное время было замерено с помощью функции *process_time()* из модуля *time* [3].

3.2 Сведения о модулях программы

Данная программа разбита на следующие модули:

— *main.py* — файл, содержащий функцию *main*;

— *algorithms.py* — файл, содержащий код всех алгоритмов нахождения расстояний Левенштейна и Дамерау—Левенштейна;

— *compare_time.py* — файл, в котором содержатся функции для замера и вывода времени выполнения реализаций алгоритмов.

3.3 Реализация алгоритмов

В листингах 3.1 – 3.4 приведены реализации алгоритмов поиска расстояний Левенштейна (только нерекурсивный алгоритм) и Дамерау—Левенштейна (нерекурсивный, рекурсивный, рекурсивный с кешированием).

Листинг 3.1 – Функция нахождения расстояния Левенштейна с использованием матрицы

```
def m(a, b):
    return 0 if a == b else 1


def levenstein(s1, s2):
    matrix = [[0] * (len(s2) + 1) for _ in range(len(s1) + 1)]
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if i == 0 and j == 0:
                matrix[i][j] = 0
            elif i > 0 and j == 0:
                matrix[i][j] = i
            elif j > 0 and i == 0:
                matrix[i][j] = j
            else:
                matrix[i][j] = min(
                    [
                        matrix[i][j - 1] + 1,
                        matrix[i - 1][j] + 1,
                        matrix[i - 1][j - 1] + m(s1[i - 1], s2[j
                            - 1]),
                    ]
                )
    return matrix[-1][-1]
```

Листинг 3.2 – Функция нахождения расстояния Дамерау—Левенштейна с использованием матрицы

```
def damerau_levenstein_iter(s1, s2):
    d = [[0] * (len(s2) + 1) for _ in range(len(s1) + 1)]
    for i in range(len(d)):
        for j in range(len(d[0])):
            if i == 0 and j == 0:
                d[i][j] = 0
            elif i > 0 and j == 0:
                d[i][j] = i
            elif j > 0 and i == 0:
                d[i][j] = j
            elif i > 1 and j > 1 and s1[i - 1] == s2[j - 2] and
               s1[i - 2] == s2[j - 1]:
                d[i][j] = min(
                    [
                        d[i][j - 1] + 1,
                        d[i - 1][j] + 1,
                        d[i - 1][j - 1] + m(s1[i - 1], s2[j -
                           1]),
                        d[i - 2][j - 2] + 1,
                    ]
                )
            else:
                d[i][j] = min(
                    [
                        d[i][j - 1] + 1,
                        d[i - 1][j] + 1,
                        d[i - 1][j - 1] + m(s1[i - 1], s2[j -
                           1]),
                    ]
                )
    return d[-1][-1]
```

Листинг 3.3 – Функция нахождения расстояния Дамерау—Левенштейна рекурсивно

```
def damerau_levenstein_rec(s1, s2):
    if len(s1) == 0 or len(s2) == 0:
        return max(len(s1), len(s2))
    temp = min(
        [
            damerau_levenstein_rec(s1[:-1], s2) + 1,
            damerau_levenstein_rec(s1, s2[:-1]) + 1,
            damerau_levenstein_rec(s1[:-1], s2[:-1]) + m(s1[-1],
               s2[-1]),
        ]
    )

    if len(s1) > 1 and len(s2) > 1 and s1[-1] == s2[-2] and
       s1[-2] == s2[-1]:
        temp = min(temp, damerau_levenstein_rec(s1[:-2],
           s2[:-2]) + 1)

    return temp
```

Листинг 3.4 – Функция нахождения расстояния Дамерау—Левенштейна рекурсивно с кешированием

```
def damerau_levenstein_rec_cash(s1, s2):
    d = [[-1] * (len(s2) + 1) for _ in range(len(s1) + 1)]

    def get_value(s1, s2):
        if d[len(s1)][len(s2)] == -1:
            d[len(s1)][len(s2)] = dlrc(s1, s2)

        return d[len(s1)][len(s2)]

    def dlrc(s1, s2):
        if len(s1) == 0 or len(s2) == 0:
            d[len(s1)][len(s2)] = max(len(s1), len(s2))
            return d[len(s1)][len(s2)]

        dlr1 = get_value(s1[:-1], s2) + 1
        dlr2 = get_value(s1, s2[:-1]) + 1
        dlr3 = get_value(s1[:-1], s2[:-1]) + m(s1[-1], s2[-1])

        temp = min([dlr1, dlr2, dlr3])

        if len(s1) > 1 and len(s2) > 1 and s1[-1] == s2[-2] and
           s1[-2] == s2[-1]:
            dlr4 = get_value(s1[:-2], s2[:-2]) + 1
            temp = min(temp, dlr4)

        d[len(s1)][len(s2)] = temp

        return temp

    return dlrc(s1, s2)
```

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояний Левенштейна и Дамерау—Левенштейна. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Входные данные		**Расстояние и алгоритм**			
		Левенштейна	**Дамерау—Левенштейна**		
Строка 1	**Строка 2**	**Итеративный**	**Итеративный**	**Рекурсивный**	
				Без кеша	**С кешом**
λ	λ	0	0	0	0
a	b	1	1	1	1
a	a	0	0	0	0
кот	скат	2	2	2	2
ab	ba	2	1	1	1
bba	abba	1	1	1	1
aboba	boba	1	1	1	1
abcdef	gh	6	6	6	6

Вывод

Были реализованы алгоритмы поиска расстояния Левенштейна итеративно, поиска расстояния Дамерау–Левенштейна итеративно, рекурсивно и рекурсивно с кешированием. Проведено тестирование реализаций алгоритмов.

4 Исследовательский раздел

В данном разделе будут приведены примеры работы программ, постановка эксперимента и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени, следующие.

— Процессор: AMD Ryzen 5 4600H 3 ГГц [4].

— Оперативная память: 16 ГБайт.

— Операционная система: Windows 10 Pro 64-разрядная система версии 22H2 [5].

При замерах времени ноутбук был включен в сеть электропитания и был нагружен только системными приложениями.

4.2 Демонстрация работы программы

На рисунке 4.1 представлена демонстрация работы разработанного программного обеспечения, а именно показаны результаты вычислений расстояний Левенштейна и Дамерау—Левенштейна для строк «скат», «кот» и «красивый», «карсивый» соответственно.

```
    1 - Расстояние Левенштейна (итеративно)
    2 - Расстояние Дамерау-Левенштейна (итеративно)
    3 - Расстояние Дамерау-Левенштейна (рекурсивно)
    4 - Расстояние Дамерау-Левенштейна (рекурсивно с кешированием)
    5 - Вывести результаты тестов
    6 - Замер времени
    0 - Выход


Выберите опцию: 1
Введите строку 1: скат
Введите строку 2: кот
Матрица:
[0, 1, 2, 3]
[1, 1, 2, 3]
[2, 1, 2, 3]
[3, 2, 2, 3]
[4, 3, 3, 2]
Расстояние = 2


    1 - Расстояние Левенштейна (итеративно)
    2 - Расстояние Дамерау-Левенштейна (итеративно)
    3 - Расстояние Дамерау-Левенштейна (рекурсивно)
    4 - Расстояние Дамерау-Левенштейна (рекурсивно с кешированием)
    5 - Вывести результаты тестов
    6 - Замер времени
    0 - Выход


Выберите опцию: 4
Введите строку 1: красивый
Введите строку 2: карсивый
Матрица:
[0, 1, 2, 3, 4, 5, 6, 7, 8]
[1, 0, 1, 2, 3, 4, 5, 6, 7]
[2, 1, 1, 1, 2, 3, 4, 5, 6]
[3, 2, 1, 1, 2, 3, 4, 5, 6]
[4, 3, 2, 2, 1, 2, 3, 4, 5]
[5, 4, 3, 3, 2, 1, 2, 3, 4]
[6, 5, 4, 4, 3, 2, 1, 2, 3]
[7, 6, 5, 5, 4, 3, 2, 1, 2]
[8, 7, 6, 6, 5, 4, 3, 2, 1]
Расстояние = 1
```

Рисунок 4.1 – Демонстрация работы программы при поиске расстояний Левенштейна и Дамерау—Левенштейна

4.3 Время выполнения реализаций алгоритмов

Результаты замеров времени выполнения реализаций алгоритмов нахождения расстояний Левенштейна и Дамерау–Левенштейна приведены в таблице 4.1. Замеры времени проводились на строках одинаковой длины и усреднялись для каждого набора одинаковых экспериментов. В таблице 4.1 используются следующие обозначения:

— Л (и) — итеративная реализация алгоритма поиска расстояния Левенштейна;

— Д-Л (и) — итеративная реализация алгоритма поиска расстояния Дамерау—Левенштейна;

— Д-Л (р) — рекурсивная реализация алгоритма поиска расстояния Дамерау—Левенштейна;

— Д-Л (рк) — рекурсивная с кешированием реализация алгоритма поиска расстояния Дамерау—Левенштейна.

Таблица 4.1 – Время работы реализации алгоритмов (в мс)

Длина строк	Л (и)	Д-Л (и)	Д-Л (р)	Д-Л (рк)
1	$3.13 \cdot 10^{-6}$	$3.13 \cdot 10^{-6}$	$1.56 \cdot 10^{-6}$	$6.25 \cdot 10^{-6}$
2	$6.25 \cdot 10^{-6}$	$4.69 \cdot 10^{-6}$	$6.25 \cdot 10^{-6}$	$1.25 \cdot 10^{-5}$
3	$9.37 \cdot 10^{-6}$	$6.25 \cdot 10^{-6}$	$3.59 \cdot 10^{-5}$	$2.19 \cdot 10^{-5}$
4	$1.25 \cdot 10^{-5}$	$1.25 \cdot 10^{-5}$	$1.91 \cdot 10^{-4}$	$3.13 \cdot 10^{-5}$
5	$1.87 \cdot 10^{-5}$	$1.87 \cdot 10^{-5}$	$9.78 \cdot 10^{-4}$	$4.69 \cdot 10^{-5}$
6	$2.50 \cdot 10^{-5}$	$2.50 \cdot 10^{-5}$	$5.26 \cdot 10^{-3}$	$6.56 \cdot 10^{-5}$
7	$3.13 \cdot 10^{-5}$	$3.13 \cdot 10^{-5}$	$2.86 \cdot 10^{-2}$	$8.75 \cdot 10^{-5}$
8	$4.06 \cdot 10^{-5}$	$4.37 \cdot 10^{-5}$	$1.58 \cdot 10^{-1}$	$1.19 \cdot 10^{-4}$
9	$5.31 \cdot 10^{-5}$	$5.62 \cdot 10^{-5}$	$8.70 \cdot 10^{-1}$	$1.50 \cdot 10^{-4}$
10	$5.94 \cdot 10^{-5}$	$6.56 \cdot 10^{-5}$	$4.75 \cdot 10^{0}$	$1.75 \cdot 10^{-4}$

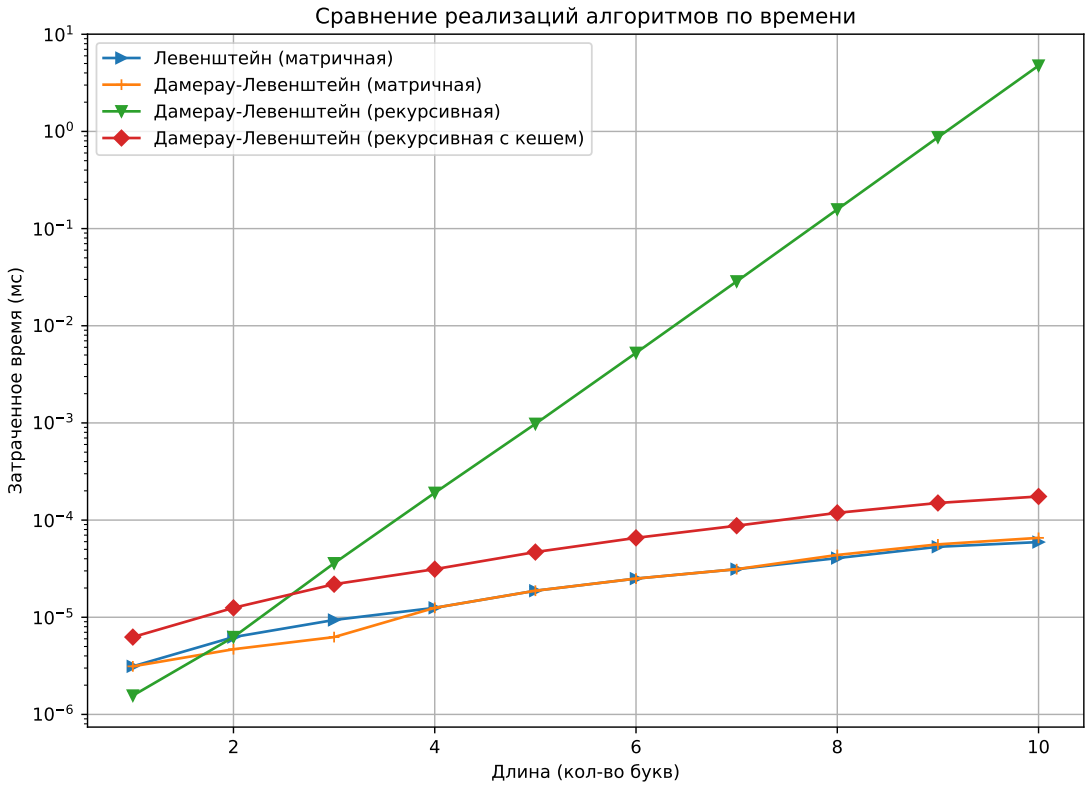

Рисунок 4.2 – Сравнение реализаций алгоритмов по времени выполнения

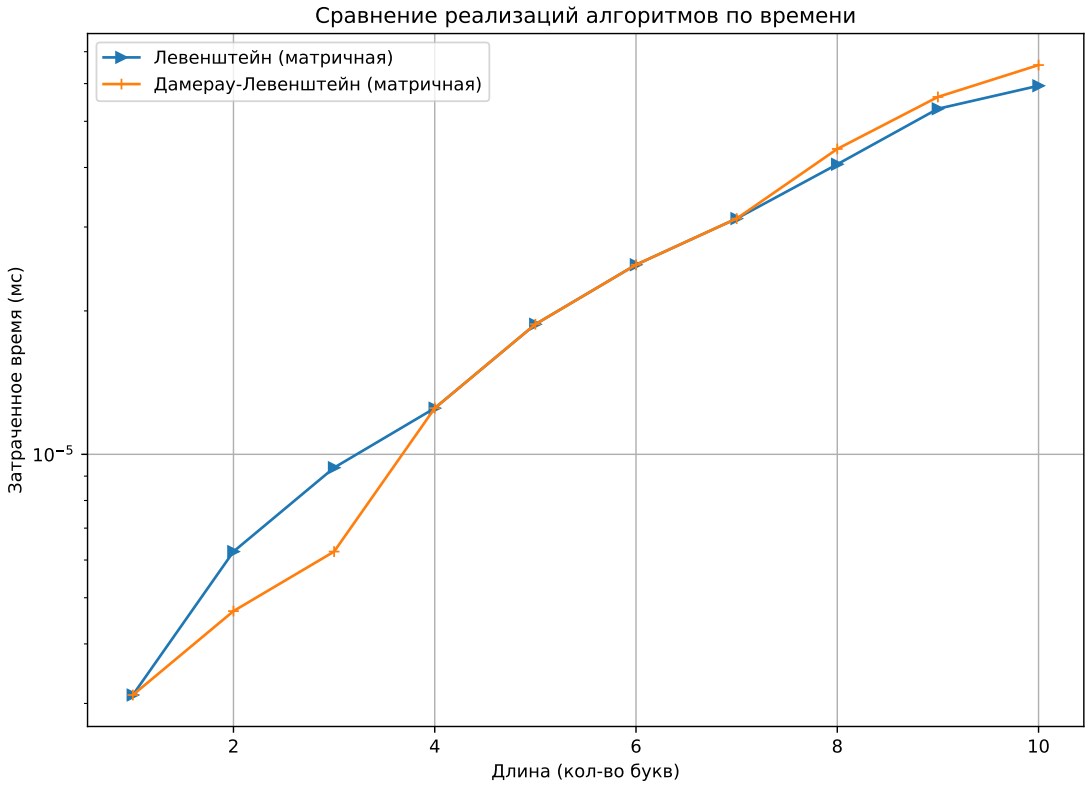

Рисунок 4.3 – Сравнение реализаций итеративных алгоритмов по времени выполнения

Наиболее эффективными по времени являются реализации алгоритмов, использующие матрицы, после них по времени идет реализация алгоритма, использующая кеш, это обусловлено тем, что по сравнению с обычной рекурсией, не вычисляются повторно одни и те же значения, но все равно тратим время на рекурсивные вызовы.

4.4 Характеристики по памяти

Введем следующие обозначения:

— n — длина строки S_1;

— m — длина строки S_2;

— $size()$ — функция, вычисляющая размер в байтах;

— int — целочисленный тип данных;

— $string$ — строковый тип данных.

Рассмотрим итеративную, рекурсивную и рекурсивную с кешированием реализации алгоритмов вычисления расстояния Дамерау—Левенштейна.

Использование памяти при итеративной реализации теоритически рассчитывается по формуле (4.1).

$$(n+1) \cdot (m+1) \cdot size(int) + 2 \cdot size(string) + 2 \cdot size(int), \qquad (4.1)$$

где

— $(n+1) \cdot (m+1) \cdot size(int)$ — хранение матрицы;

— $2 \cdot size(string)$ — хранение двух строк;

— $2 \cdot size(int)$ — адрес возврата и возвращаемое значение.

Максимальная глубина стека вызовов при рекурсивной реализации нахождения расстояния Дамерау—Левенштейна равна сумме длин входящих строк, соответственно, максимальный расход памяти рассчитывается по (4.2).

$$(n+m) \cdot (2 \cdot size(string) + 3 \cdot size(int)), \qquad (4.2)$$

где

— $(n + m)$ — максимальная глубина стека вызовов;

— $2 \cdot size(string)$ — хранение двух строк;

— $2 \cdot size(int)$ — адрес возврата и возвращаемое значение;

— $size(int)$ — временная переменная.

Для алгоритма, использующего кеширование требуется дополнительно память под кеш и 4 временных переменных (4.3).

$$(n + m) \cdot (2 \cdot size(string) + 6 \cdot size(int)) + (n + 1) \cdot (m + 1) \cdot size(int), \quad (4.3)$$

где

— $(n + m)$ — максимальная глубина стека вызовов;

— $2 \cdot size(string)$ — хранение двух строк;

— $2 \cdot size(int)$ — адрес возврата и возвращаемое значение;

— $4 \cdot size(int)$ — временные переменные;

— $(n + 1) \cdot (m + 1) \cdot size(int)$ — хранение кеша.

По расходу памяти итеративные реализации алгоритмов проигрывают рекурсивной без кеширования: максимальный размер используемой памяти в итеративной реализации растет как произведение длин строк, в то время как у рекурсивной без кеширования — как сумма длин строк.

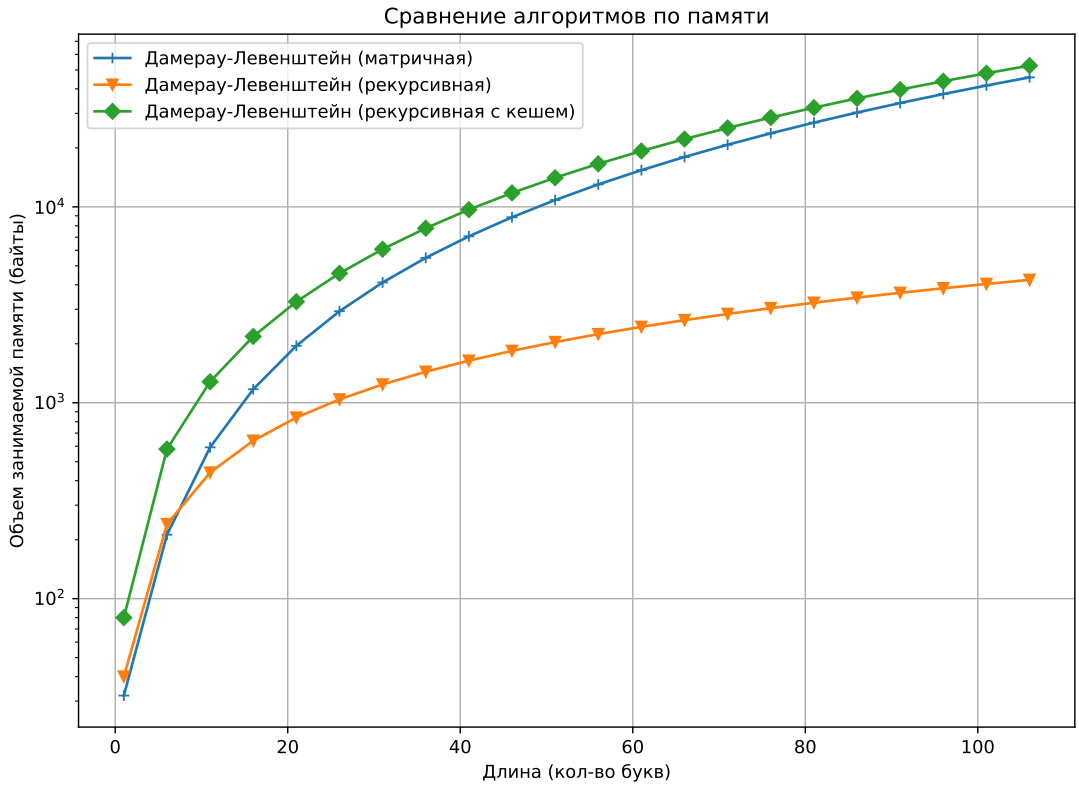

Рисунок 4.4 – Сравнение алгоритмов по памяти

Вывод

В данном разделе было произведено сравнение количества затраченного времени и памяти реализаций алгоритмов поиска расстояний Левенштейна и Дамерау—Левенштейна. Наименее затратной по времени оказалась итеративная реализация алгоритма нахождения расстояния Левенштейна.

По таблице 4.1 видно, что рекурсивная реализация в 72447 раз проигрывает итеративной по времени выполнения при длине строк 10. Такая огромная разница может быть объяснена тем, что рекурсивная реализация алгоритма вычисляет расстояния для одних и тех же подстрок множество раз, т. к. не требует дополнительную память для хранения ранее вычисленных значений. Поэтому рекурсивные реализации следует использовать лишь при малых длинах строк.

При этом как было замечено в пункте 4.4, рекурсивные реализации алгоритмов требуют меньше памяти, чем итеративные.

Рекурсивная реализация алгоритма поиска расстояния Дамерау—Левенштейна будет более затратной по времени, но менее затратной по памяти в сравнении с итеративной реализацией алгоритма поиска расстояния Дамерау—Левенштейна. При этом рекурсивная с кешированием реализация проигрывает по памяти и по времени итеративной.

ЗАКЛЮЧЕНИЕ

Цель данной лабораторной работы была достигнута, а именно были исследованы алгоритмы, вычисляющие расстояния Левенштейна и Дамерау—Левенштейна.

Для достижения поставленной цели были выполнены следующие задачи.

1) Описаны алгоритмы поиска расстояний Левенштейна и Дамерау—Левенштейна;

2) Выбраны инструменты для реализации алгоритмов и замера процессорного времени их выполнения.

3) Разработано программное обеспечение, реализующее следующие алгоритмы:

   — нерекурсивный метод поиска расстояния Левенштейна;
   — нерекурсивный метод поиска расстояния Дамерау—Левенштейна;
   — рекурсивный метод поиска расстояния Дамерау—Левенштейна;
   — рекурсивный с кешированием метод поиска расстояния Дамерау—Левенштейна.

4) Проведен анализ затрат реализаций алгоритмов по времени и по памяти.

В результате исследования реализаций алгоритмов было выявлено, что рекурсивная реализация в 72477 раз проигрывает итеративной по времени выполнения при длине строк 10. Это обусловлено тем, что рекурсивная реализация не хранит вычисленные значения расстояний для подстрок и поэтому вычисляет их множество раз. При этом рекурсивная реализация требует меньше памяти, чем итеративные, поэтому лучше подходит для коротких строк.

Рекурсивная реализация алгоритма поиска расстояния Дамерау—Левенштейна с кешированием проигрывает в 3 раза итеративной реализации, но выигрывает в 27142 раза рекурсивную реализацию по времени выполнения при длине строк 10. Такой результат обуславливается тем, что в отличии от рекурсивной реализации ранее вычисленные значения сохраняются в памяти,

но все равно затрачивается время на рекурсивные вызовы. В то же время реализация с кешированием проигрывает всем реализациям по памяти.

Итеративные реализации алгоритмов поиска расстояний Дамерау—Левенштейна и Левенштейна практически не различаются по времени выполнения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Левенштейн В. И.* Двоичные коды с исправлением выпадений, вставок и замещений символов // Докл. АН СССР. — М.: Наука, 1965. — Т. 163, № 4. — С. 845—848.
2. The official home of the Python Programming Language [Электронный ресурс]. — Режим доступа: `https://www.python.org/` (дата обращения: 19.09.2023).
3. time — Time access and conversions [Электронный ресурс]. — Режим доступа: `https://docs.python.org/3/library/time.html` (дата обращения: 19.09.2023).
4. Amd [Электронный ресурс]. — Режим доступа: `https://www.amd.com/en.html` (дата обращения: 28.09.2023).
5. Windows 10 Pro 22h2 64-bit [Электронный ресурс]. — Режим доступа: `https://www.microsoft.com/ru-ru/software-download/windows10` (дата обращения: 28.09.2023).