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: «Расстояния Левенштейна и Дамерау-Левенштейна»

Студент ИУ7-54Б (Группа) __________ (Подпись, дата) Писаренко Д. П. (И. О. Фамилия)

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

*2023 г.*

# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

Зачастую при использовании поисковых платформ в спешке пользователи допускают ошибки в написании слов. После обработки поискового запроса пользователь получает предложение об исправлении. Метрика, реализующая поиск подобных исправлений, называется редакционным расстоянием. Очевидно, что ее применение не заканчивается на этом. Проблема посимвольного сравнения при работе со словами встречается повсеместно.

Впервые задачу определения редакционного расстояния поставил советский математик Владимир Левенштейн в 1965 году во время изучения последовательностей 0–1. Стоит заметить, что более общую задачу назвали его именем. Алгоритм расстояния Левенштейна допускает следующие операции: вставка, удаление и замена символа.

На данный момент расстояние Левенштейна активно применяется:

— для сравнения текстовых файлов утилитой $diff$;

— для исправления ошибок в слове в поисковых системах, базах данных, системах автоматического распознавания сканированного текста или речи;

— в биоинформатике для сравнения генов, хромосом и белков.

Позднее Фредерик Дамерау обнаружил, что зачастую ошибки связаны с неправильным порядком записи соседних букв, и добавил операцию транспозиции (перестановки) символов.

Целью данной лабораторной работы является описание, изучение и сравнение нескольких алгоритмов поиска редакционного расстояния.

Для выполнения поставленной цели необходимо выполнить следующие задачи:

— описать расстояние Левенштейна и Дамерау—Левенштейна;

— создать программный продукт с реализованными алгоритмами поиска расстояний Левенштейна и Дамерау—Левенштейна;

— исследовать затраты времени и памяти при различных реализациях алгоритмов;

— выполнить сравнение алгоритмов по затратам процессорного времени и памяти.

# 1 Аналитический раздел

Будут рассмотрены две группы реализаций алгоритмов:

- матричный (нерекурсивный) — алгоритм работает не рекурсивно, обрабатывая матрицу на каждой итерации;
- рекурсивный — алгоритм реализуется посредством вложенных вызовов себя же с измененными аргументами.

В лабораторной работе будут рассмотрены следующие реализации алгоритмов:

- матричная (нерекурсивная) реализация алгоритма поиска расстояния Левенштейна;
- матричная (нерекурсивная) реализация алгоритма поиска расстояния Дамерау-Левенштейна;
- рекурсивная реализация алгоритма поиска расстояния Дамерау—Левенштейна;
- рекурсивная реализация алгоритма поиска расстояния Дамерау—Левенштейна с кэшированием.

## 1.1 Расстояние Левенштейна

Расстояние Левенштейна - это число, которое показывает, насколько различны две строки [**definition-lev**].

Любая операция имеет свою цену, но в общем виде:

- $w(a, b)$ — замена символа $a$ на $b$, R (англ. replace);
- $w(\lambda, a)$ — вставка символа $a$, I (англ. insert);
- $w(a, \lambda)$ — удаление символа $a$, D (англ. delete);

Пусть стоимость каждой такой операции равна 1, тогда:

- $w(a, b) = 1, \ a \neq b$;
- $w(\lambda, a) = 1$;

— $w(a, \lambda) = 1$;

— $w(a, b) = 0,\ a = b$ — операция M — совпадение (англ. match).

Однако существует проблема выравнивания строк различной длины, при котором есть более, чем один вариант сопоставления символов. В таком случае эта проблема решается введением рекуррентной формулы, где

— $l_1$ — длина $str_1$;

— $l_2$ — длина $str_2$;

— $str_1[1 \ldots i]$ — подстрока $str_1$ длиной $i$, начиная с 1-го символа;

— $str_2[1 \ldots j]$ — подстрока $str_2$ длиной $j$, начиная с 1-го символа.

В таком случае расстояние Левенштейна между строками $str_1$ (длиной $l_1$) и $str_2$ (длиной $l_2$) можно рассчитать следующим образом:

$$D(i, j) = \begin{cases} 0, & \text{если } i = 0, j = 0 \\ j, & \text{если } i = 0, j > 0 \\ i, & \text{если } j = 0, i > 0 \\ min(D(i, j-1) + 1, & \\ \qquad D(i-1, j) + 1, & \text{если } i > 0, j > 0 \\ \qquad D(i-1, j-1) + change(str_1[i], str_2[j]), & \end{cases} \tag{1.1}$$

При этом $str_1[i]$ и $str_2[j]$ сравниваются так:

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

## 1.2 Расстояние Дамерау—Левенштейна

Фредерик Дамерау заметил, что иногда несовпадение пар соседних символов между строками является проблемой неправильного порядка записи и ввел операцию замены S (англ. swap). Иными словами, операция применяется только в тех случаях, когда $str_1[i] = str_2[j-1]$ и $str_1[i-1] = str_2[j]$. Тогда используется следующая формула:

$$D(i,j) = \begin{cases} 0, & \text{если } i = 0, j = 0 \\ j, & \text{если } i = 0, j > 0 \\ i, & \text{если } j = 0, i > 0 \\ min(min(D(i,j-1)+1, & \\ \qquad D(i-1,j)+1), & \\ \quad D(i-1,j-1)+change(str1[i], str2[j]), & \\ \quad \left[\begin{array}{ll} D(i-2,j-2)+1, & \text{если } i > 1, j > 1, \\ & str_1[i-1] == str_2[j-2], \\ & str_1[i-2] == str_2[j-1] \end{array}\right], & \text{иначе} \end{cases} \tag{1.3}$$

**Вывод:** Таким образом, в этом разделе была поставлена цель, определены задачи, введены понятия расстояния Левенштейна и Дамерау—Левенштейна и выведены формулы поиска значений.

# 2 Конструкторский раздел

Различные реализации алгоритмов подразумевают использование матрицы, рекурсии или применение кэширования.

## 2.1 Структуры данных

Для реализациия алгоритмов будут использованы следующие структуры данных:

- список;
- матрица — список из вложенных списков;
- строка;
- целое число — необходимо для хранения размера строки.

## 2.2 Матричная реализация алгоритма поиска расстояния Левенштейна

На рисунке 2.1 приведена схема матричной реализации алгоритма поиска расстояния Левенштейна.

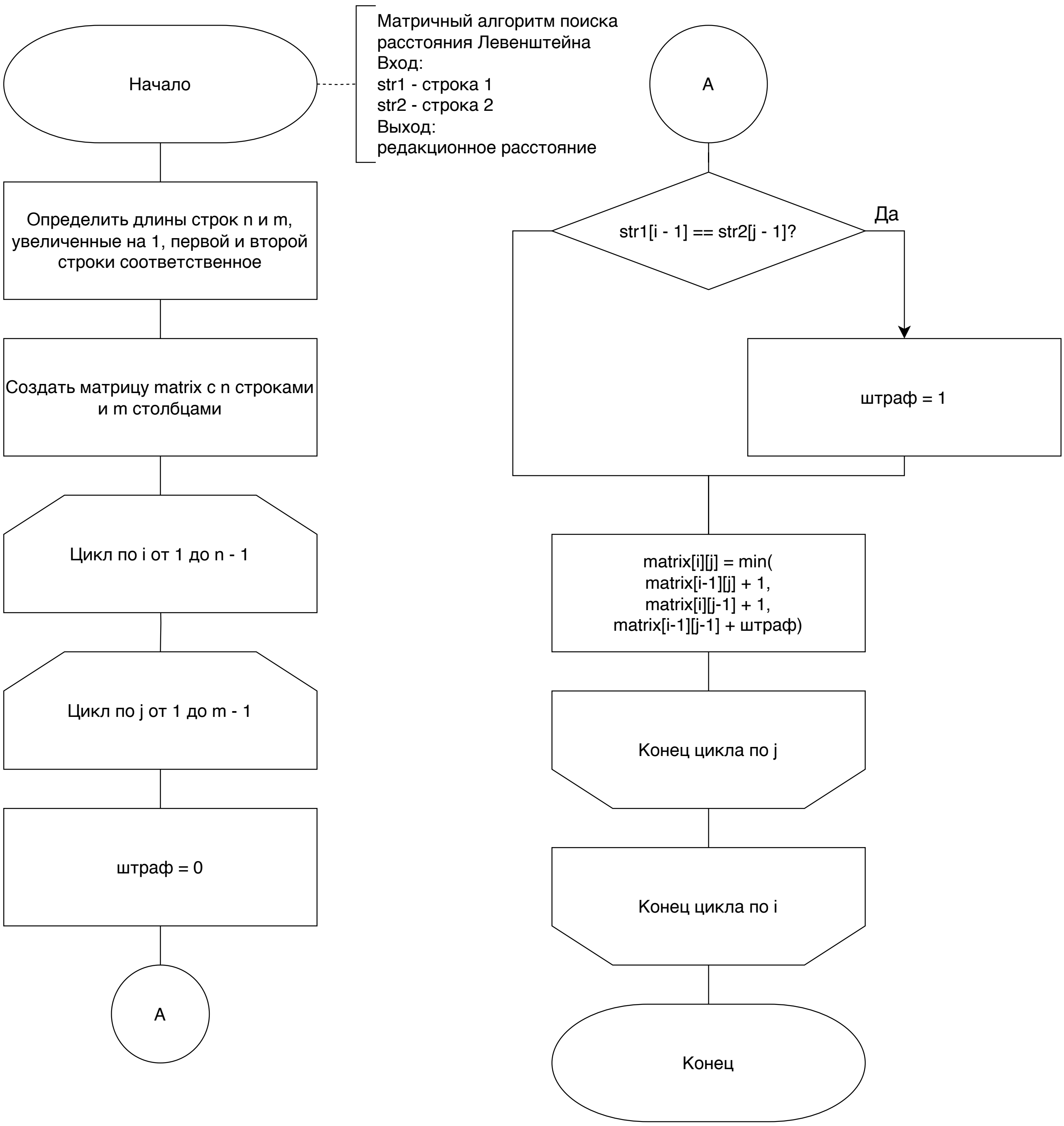


Рисунок 2.1 – Схема матричной реализации алгоритма поиска расстояния Левенштейна

## 2.3 Матричная реализация алгоритма поиска расстояния Дамерау—Левенштейна

На рисунке 2.2 приведена схема матричной реализации алгоритма поиска расстояния Дамерау—Левенштейна.

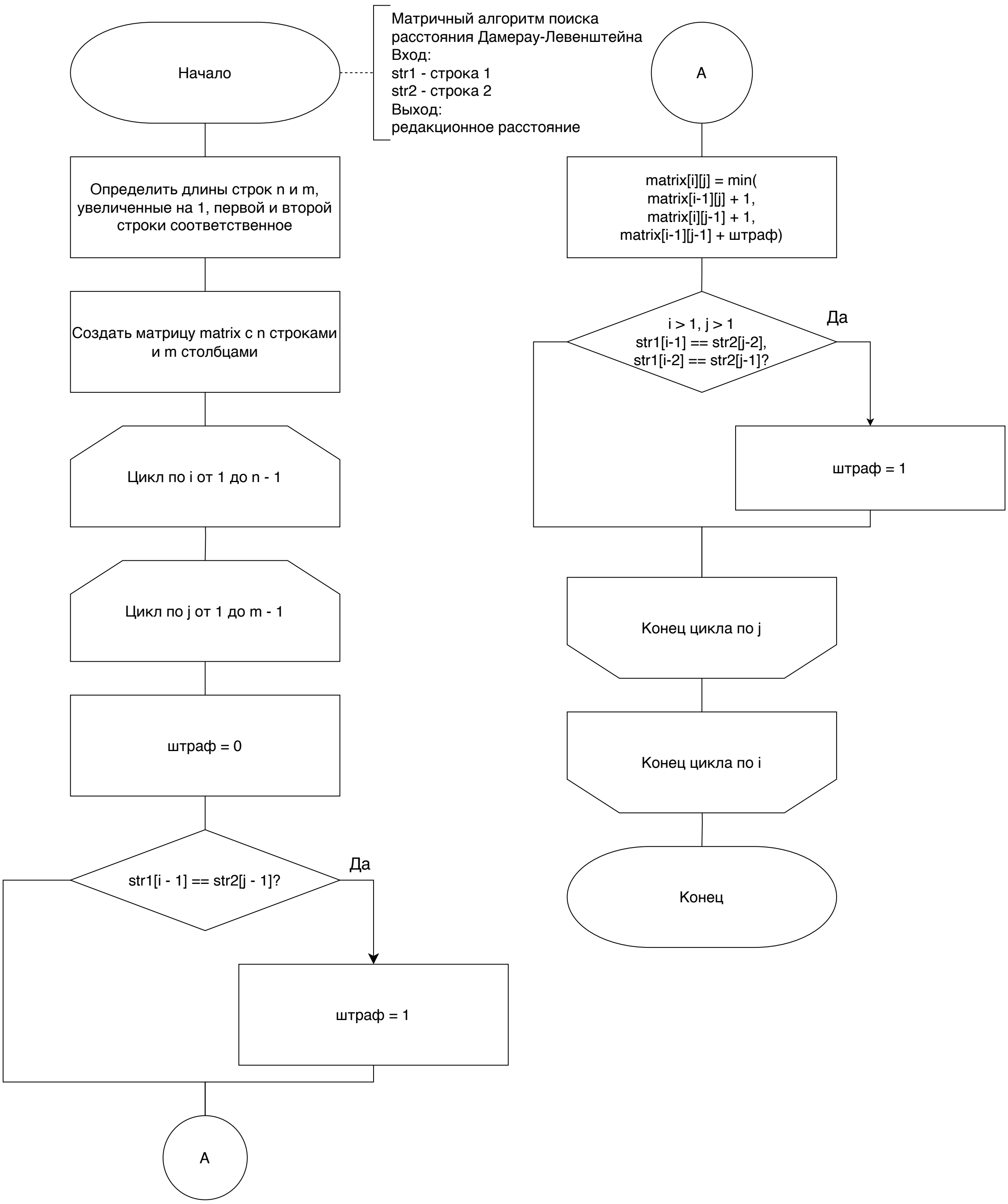


Рисунок 2.2 – Схема матричной реализации алгоритма поиска расстояния Дамерау—Левенштейна

## 2.4 Рекурсивная реализация алгоритма поиска расстояния Дамерау—Левенштейна

На рисунке 2.3 приведена схема рекурсивной реализации алгоритма поиска расстояния Дамерау—Левенштейна.

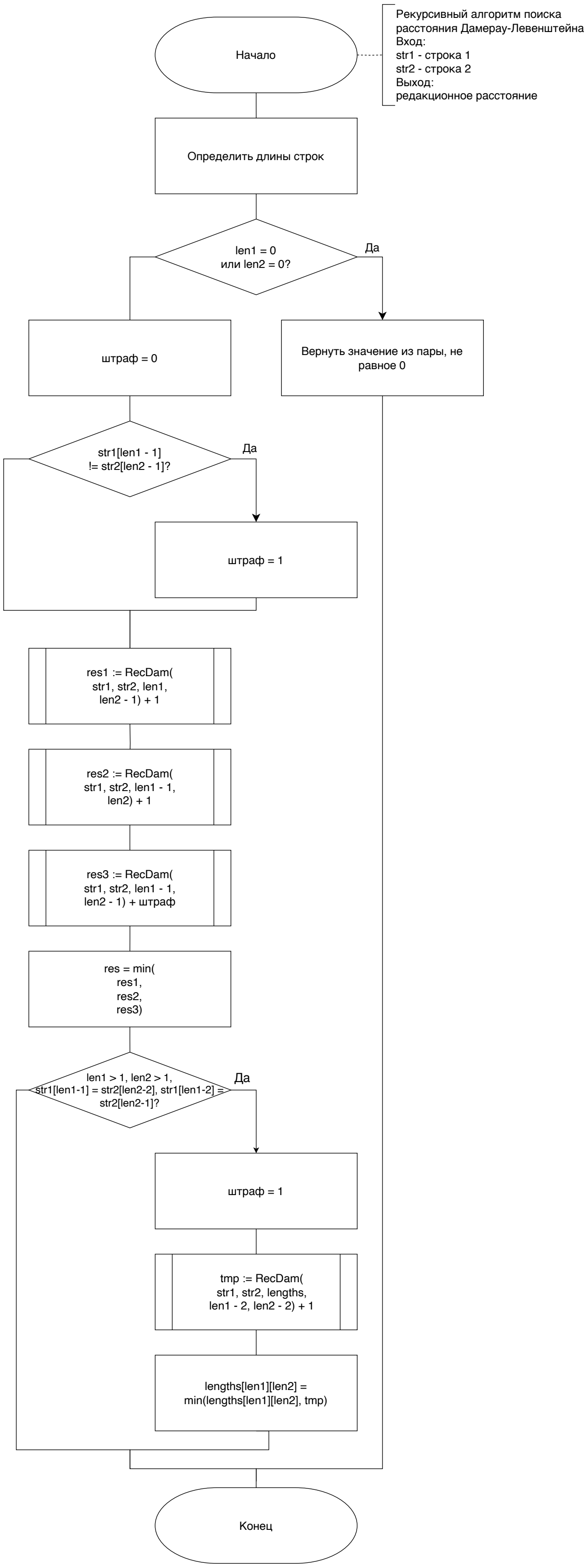


Рисунок 2.3 – Схема рекурсивной реализации алгоритма поиска расстояния Дамерау—Левенштейна

## 2.5 Рекурсивная реализация алгоритма поиска расстояния Дамерау—Левенштейна с кэшированием

В качестве кэша используется матрица, инициализированная значениями «-1».

На рисунке 2.4 приведена схема рекурсивной реализации алгоритма поиска расстояния Дамерау—Левенштейна с кэшированием.

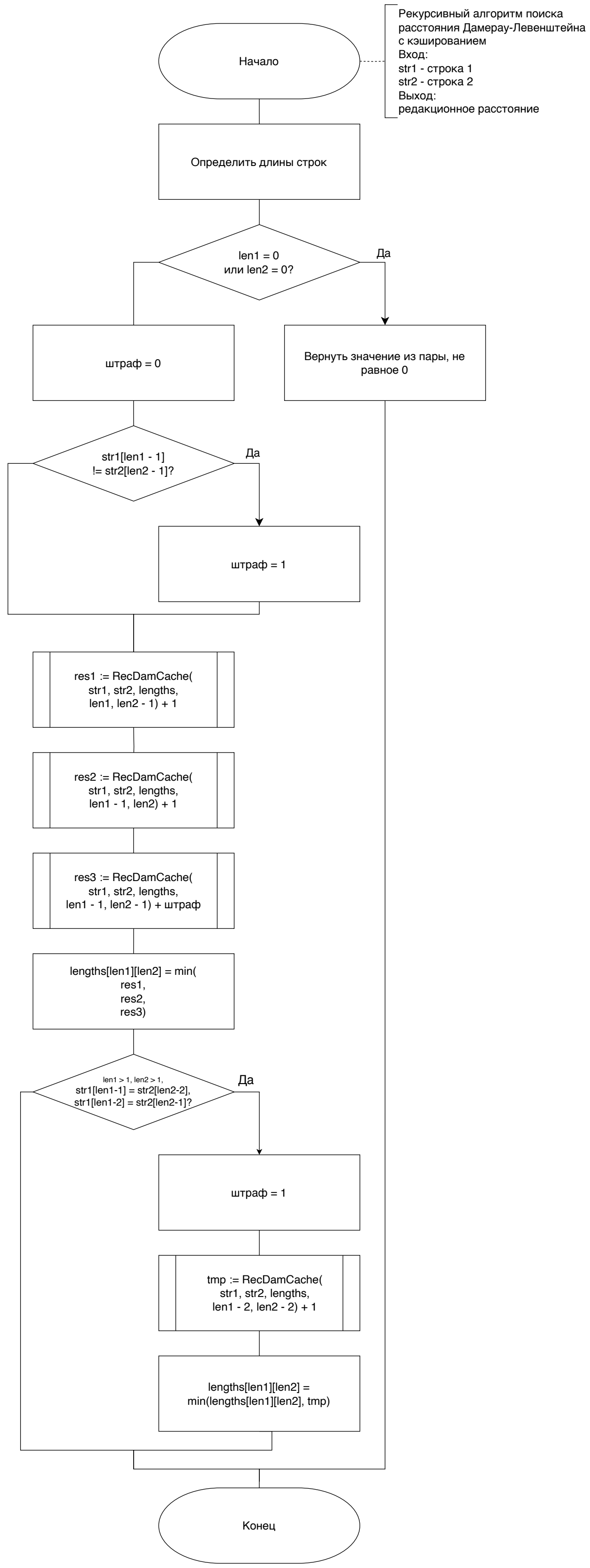


Рисунок 2.4 – Схема рекурсивной реализации алгоритма поиска расстояния Дамерау—Левенштейна с кэшированием

# 3 Технологический раздел

Для создания программного продукта был выбран язык Python [**python**].

Для разработки была выбрана среда разработки Visual Studio Code в связи с тем, что она предоставляет возможности управления и ведения проекта для выполнения поставленной задачи [**vscode**].

## 3.1 Реализация алгоритмов

На листингах 3.1 представлены реализации алгоритмов поиска расстояний Левенштейна и Дамерау—Левенштейна.

Листинг 3.1 – Реализации алгоритмов поиска расстояний Левенштейна и Дамерау—Левенштейна

```
from functions import *

def levenshtein_distance(str1, str2, flag_output=False):
    n, m = len(str1), len(str2)
    matrix = create_levenshtein_matrix(n + 1, m + 1)

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            action_add = matrix[i - 1][j] + 1
            action_delete = matrix[i][j - 1] + 1
            action_change = matrix[i - 1][j - 1]

            if str1[i - 1] != str2[j - 1]:
                action_change += 1

            matrix[i][j] = min(action_add, action_delete,
               action_change)

    if flag_output:
        print("\nМАТРИЦА:")
        print_matrix(matrix, str1, str2)

    result = matrix[n][m]

    return result


def damerau_levenshtein_distance(str1, str2, flag_output=False):
    n, m = len(str1), len(str2)
    matrix = create_levenshtein_matrix(n + 1, m + 1)

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            action_add = matrix[i - 1][j] + 1
            action_delete = matrix[i][j - 1] + 1
            action_change = matrix[i - 1][j - 1]

            if str1[i - 1] != str2[j - 1]:
                action_change += 1
```

```

            action_swap = n
            if i > 1 and j > 1 and str1[i - 1] == str2[j - 2] and str1[i - 2] == str2[j - 1]:
                action_swap = matrix[i - 2][j - 2] + 1

            matrix[i][j] = min(action_add, action_delete, action_change, action_swap)

    if flag_output:
        print("\nМАТРИЦА:")
        print_matrix(matrix, str1, str2)

    result = matrix[n][m]

    return result


def damerau_levenshtein_distance_recursive(str1, str2):
    n, m = len(str1), len(str2)

    flag = 0
    if n == 0 or m == 0:
        return abs(n - m)

    if str1[-1] != str2[-1]:
        flag = 1

    result = min(
        damerau_levenshtein_distance_recursive(str1[:-1], str2) + 1,
        damerau_levenshtein_distance_recursive(str1, str2[:-1]) + 1,
        damerau_levenshtein_distance_recursive(str1[:-1], str2[:-1]) + flag)

    if n > 1 and m > 1 and str1[-1] == str2[-2] and str1[-2] == str2[-1]:
        result = min(result, damerau_levenshtein_distance_recursive(str1[:-2], str2[:-2]) + 1)
```

```

    return result


def damerau_levenshtein_distance_recursive_cache(str1, str2,
   flag_output=False):
    n, m = len(str1), len(str2)
    matrix = create_levenshtein_matrix(n + 1, m + 1)

    for i in range(n + 1):
        for j in range(m + 1):
            matrix[i][j] = -1

    recursive_cache(str1, str2, n, m, matrix)

    if flag_output:
        print("\nМАТРИЦА:")
        print_matrix(matrix, str1, str2)

    result = matrix[n][m]

    return result


def recursive_cache(str1, str2, n, m, matrix):
    if matrix[n][m] != -1:
        return matrix[n][m]

    if n == 0:
        matrix[n][m] = m
        return matrix[n][m]

    if n > 0 and m == 0:
        matrix[n][m] = n
        return matrix[n][m]

    action_add = recursive_cache(str1, str2, n - 1, m, matrix) +
       1
    action_delete = recursive_cache(str1, str2, n, m - 1,
       matrix) + 1
```

```
    action_change = recursive_cache(str1, str2, n - 1, m - 1, matrix)

    if str1[n - 1] != str2[m - 1]:
        action_change += 1

    matrix[n][m] = min(action_add, action_delete, action_change)

    if n > 1 and m > 1 and str1[n - 1] == str2[m - 2] and str1[n - 2] == str2[m - 1]:
        matrix[n][m] = min(matrix[n][m], recursive_cache(str1, str2, n - 2, m - 2, matrix) + 1)

    return matrix[n][m]
```

## 3.2 Тестовые случаи

Обозначения:

— Lev — матричная реализация алгоритма поиска расстояния Левенштейна;

— Dam-Lev — матричная реализация алгоритма поиска расстояния Дамерау-Левенштейна;

— Rec — рекурсивная реализация алгоритма поиска расстояния Дамерау-Левенштейна;

— Rec-Cache — рекурсивная реализация алгоритма поиска расстояния Дамерау-Левенштейна с кэшированием.

В таблице 3.1 приведены тестовые случаи.

Таблица 3.1 – Тестовые случаи

| № | $S_1$ | $S_2$ | Lev | Dam-Lev | Rec | Rec-Cache |
|---|---|---|---|---|---|---|
| 1 | привет | пока | 5 | 5 | 5 | 5 |
| 2 | poka | privet | 5 | 5 | 5 | 5 |
| 3 | привет | пирвет | 2 | 1 | 1 | 1 |
| 4 | коммуникация | комуна | 6 | 6 | 6 | 6 |
| 5 | wife | husband | 7 | 7 | 7 | 7 |
| 6 | sunday | saturday | 3 | 3 | 3 | 3 |
| 7 | John | Jhno | 2 | 2 | 2 | 2 |

# 4 Исследовательский раздел

## 4.1 Технические характеристики

Технические характеристики компьютера, на котором проводился замерный эксперимент:

— процессор Intel Core i5-10400F (6 ядер) [**intel**];

— 16 Гб оперативная память DDR4;

— операционная система Windows 10 Pro [**windows**].

Во время проведения исследования ноутбук был нагружен только системными приложениями и целевой программой.

В библиотеке *time* языка *python* есть встроенный функционал для замеров времени. Для замеров процессорного времени используется функция $process_time$, которая возвращает время в секундах. Происходит запуск фиксированного числа тестов для каждого случая и вычисляется среднее затраченное время. [**python-time**].

## 4.2 Временные показатели

Обозначения:

— матричный Левенштейн (Lev) — матричная реализация алгоритма поиска расстояния Левенштейна;

— матричный ДЛ (Dam-Lev) — матричная реализация алгоритма поиска расстояния Дамерау-Левенштейна;

— рекурсивный ДЛ (Rec) — рекурсивная реализация алгоритма поиска расстояния Дамерау-Левенштейна;

— кэширование ДЛ (Rec-Cache) — рекурсивная реализация алгоритма поиска расстояния Дамерау-Левенштейна с кэшированием.

В таблице 4.1 приведены результаты замеров времени для строк различной длины (в каждом эксперименте длины строк одинаковы) для каждой реализации алгоритмов.

Таблица 4.1 – Временные замеры различных реализаций поиска редакционных расстояний

| Длина слова | Время, мс | | | |
|---|---|---|---|---|
| | Матричный Левенштейн | Матричный ДЛ | Рекурсивный ДЛ | Кэширование ДЛ |
| 1 | 0.02325 | 0.02375 | 0.0135 | 0.025 |
| 2 | 0.025 | 0.02625 | 0.0265 | 0.02675 |
| 3 | 0.0275 | 0.0275 | 0.0645 | 0.0325 |
| 4 | 0.02875 | 0.03 | 0.15625 | 0.035 |
| 5 | 0.03125 | 0.03125 | 0.9375 | 0.0425 |
| 6 | 0.04 | 0.0425 | 3.90625 | 0.0525 |
| 7 | 0.05025 | 0.055 | 22.03125 | 0.075 |
| 8 | 0.06775 | 0.0675 | 122.96875 | 0.08225 |
| 9 | 0.07825 | 0.0925 | 699.21875 | 0.1425 |
| 10 | 0.16125 | 0.18125 | 3971.5625 | 0.17125 |
| 20 | 0.3125 | 0.35625 | – | 0.625 |
| 40 | 0.78125 | 0.78125 | – | 1.09375 |
| 60 | 1.71875 | 2.34375 | – | 3.125 |
| 80 | 2.5 | 3.4375 | – | 5.625 |
| 100 | 3.59375 | 4.84375 | – | 9.53125 |
| 120 | 7.03125 | 7.1875 | – | 13.4375 |
| 140 | 7.65625 | 10 | – | 17.8125 |
| 160 | 10.78125 | 11.71875 | – | 22.34375 |
| 180 | 11.71875 | 15.9375 | – | 27.96875 |

На рисунке 4.1 представлен график зависимости процессорного времени от длины слова для матричных реализаций.

На рисунках 4.2–4.5 представлены графики зависимости процессорного времени от длины слова.

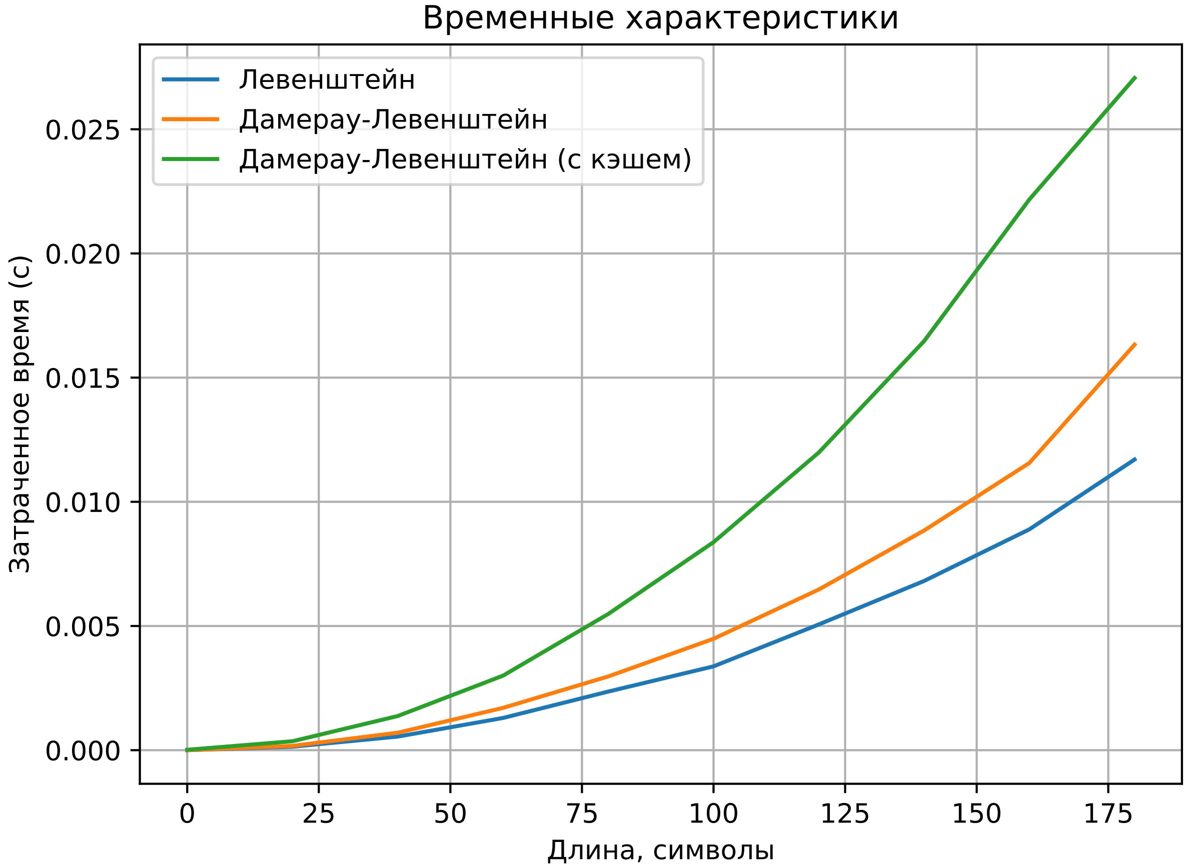


Рисунок 4.1 – График зависимости процессорного времени от длины слова

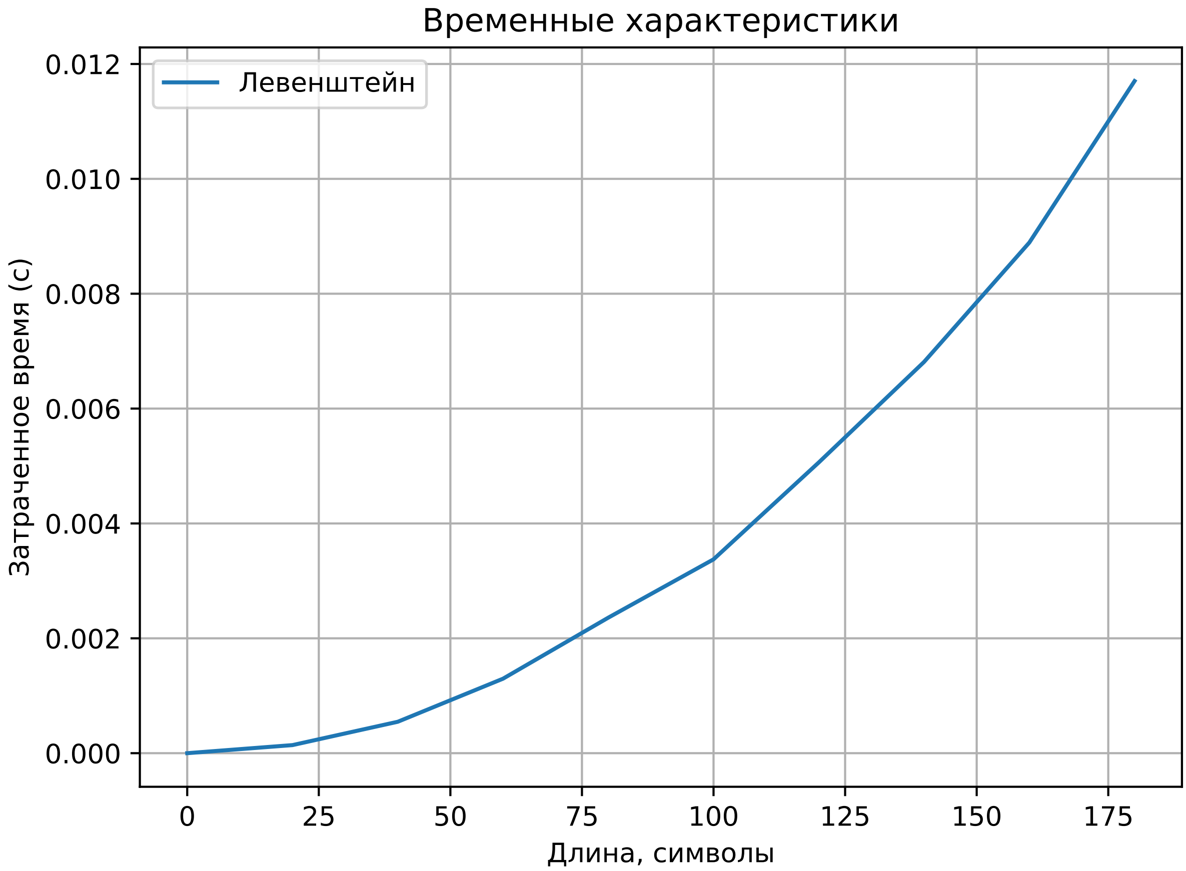


Рисунок 4.2 – График зависимости процессорного времени от длины слова


Рисунок 4.3 – График зависимости процессорного времени от длины слова

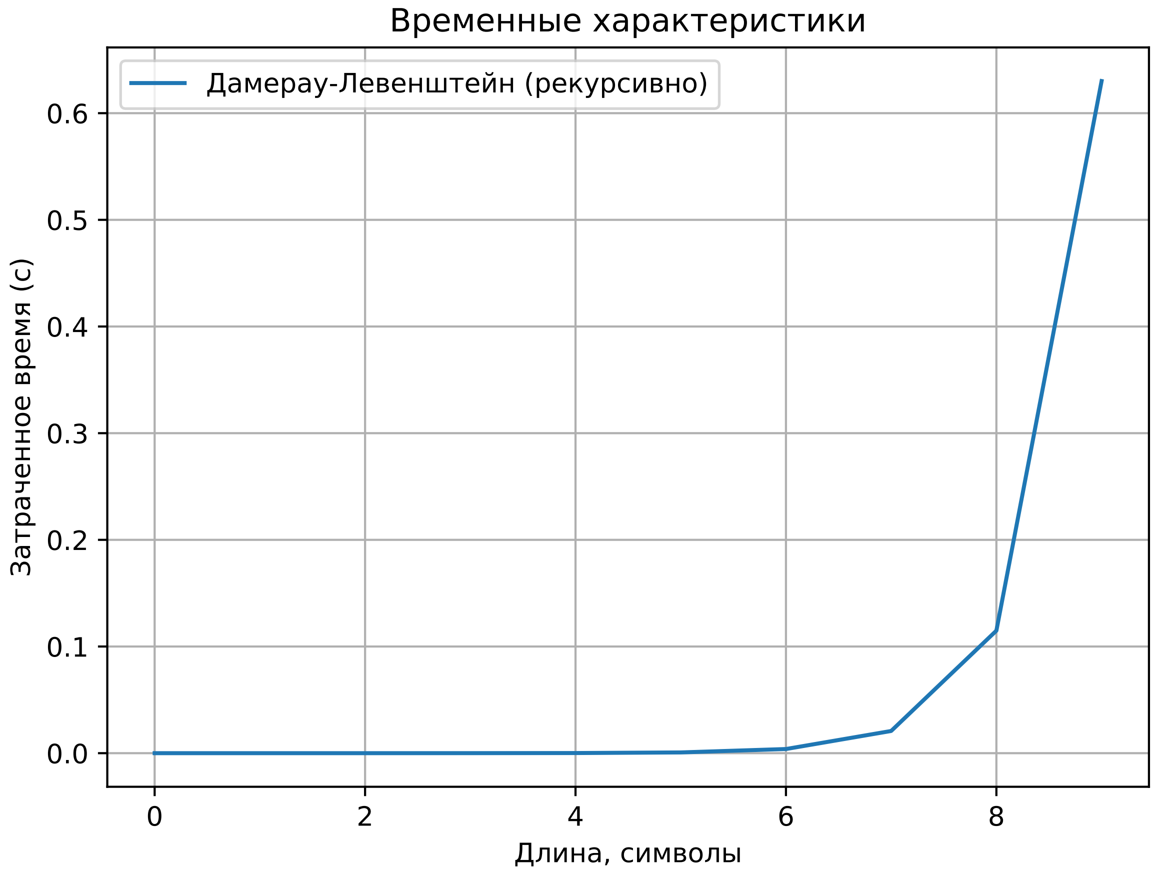


Рисунок 4.4 – График зависимости процессорного времени от длины слова

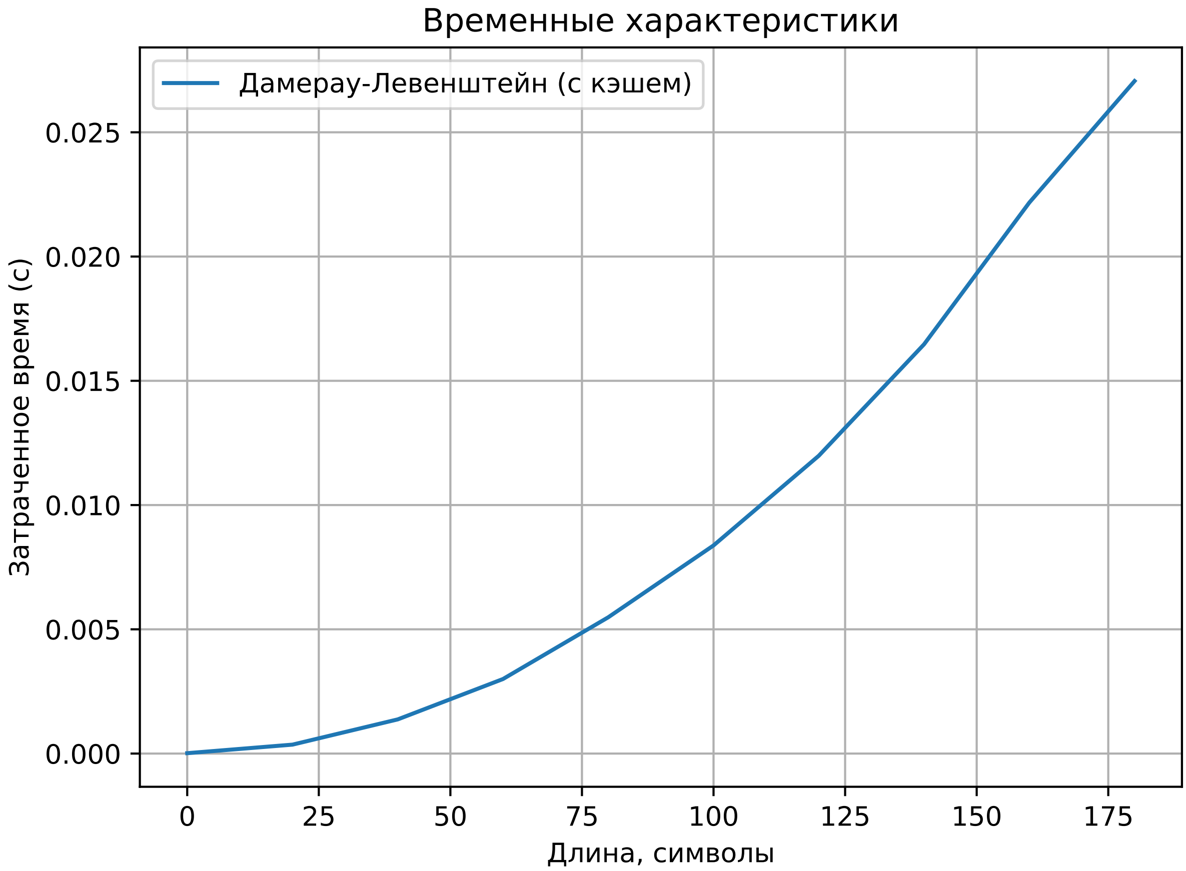


Рисунок 4.5 – График зависимости процессорного времени от длины слова

При длине слова, большей 10, замеры для рекурсивной реализации алгоритма поиска расстояния Дамерау-Левенштейна не производились в связи с заметным отставанием от остальных реализаций.

## 4.3 Теоретические затраты памяти

Пусть

— $len_1$ — длина строки $str_1$;

— $len_2$ — длина строки $str_2$;

— $string$ — массив символьного типа;

— $int$ — целочисленный тип;

— $[]int32$ — массив целочисленного типа;

— $[][]int32$ — матрица целочисленного типа;

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

Использование памяти при итеративной реализации алгоритма поиска расстояния Левенштейна теоретически равно:

$$(len_1 + 1) \cdot (len_2 + 1) \cdot size(int32) + 2 \cdot size(string) + 5 \cdot size(int32), \quad (4.1)$$

где

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

— $5 \cdot size(int32)$ — хранение размеров матрицы, адреса возврата и дополнительная переменная для хранения результата.

Использование памяти при итеративной реализации алгоритма поиска расстояния Дамерау—Левенштейна теоретически равно:

$$(len_1 + 1) \cdot (len_2 + 1) \cdot size(int) + 2 \cdot size(string) + 5 \cdot size(int32), \quad (4.2)$$

где

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

— $5 \cdot size(int32)$ — хранение размеров матрицы, адреса возврата и дополнительная переменная для хранения результата.

Так как при каждом вызове в рекурсивном алгоритме нахождения расстояния Дамерау—Левенштейна требуется 2 дополнительные переменные и, при этом, максимальная глубина стека вызовов равна сумме длин входящих строк, то в худшем случае расход памяти равен:

$$(len_1 + len_2) \cdot (2 \cdot size(string) + 5 \cdot size(int32)), \quad (4.3)$$

где:

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

— $5 \cdot size(int32)$ — дополнительные переменные, адрес возврата и длины строк.

В случае использования рекурсивного алгоритма поиска расстояния Дамерау—Левенштейна с кэшированием необходимо так же учитывать размеры матрицы, поэтому в худшем случае расход памяти равен:

$$\begin{aligned}(len_1 + len_2) \cdot (2 \cdot size(string) + 5 \cdot size(int32)+ \\ +(len_1 + 1) \cdot (len_2 + 1) \cdot size(int32))\end{aligned} \tag{4.4}$$

## Вывод

В этом разделе было осуществлено сравнение различных алгоритмов, решающих проблему поиска расстояний Левенштейна и Дамерау—Левенштейна по затрачиваемой памяти и процессорному времени. Самыми быстрыми алгоритмами с равными затратами памяти можно считать матричные реализации алгоритма нахождения расстояний Левенштейна и Дамерау—Левенштейна. Рекурсивная реализация алгоритма нахождения расстояния Дамерау—Левенштейна с кэшированием проигрывает матричным реализациям в 1.5-2 раза при одинаковых затратах памяти. Рекурсивная реализация алгоритма нахождения расстояния Дамерау-Левенштейна многократно выигрывает все остальные реализации по затратам памяти, однако многократно проигрывает по затратам процессорного времени при любой длине слова.

# ЗАКЛЮЧЕНИЕ

Из результатов проведения лабораторной работы и исследования можно сделать вывод, что временные затраты всех реализаций алгоритмов нахождения расстояний Левенштейна и Дамерау—Левенштейна растут в зависимости от длины строки. Рекурсивная реализация алгоритма поиска расстояния Дамерау-Левенштейна выигрывает остальные реализации в затрачиваемой памяти, но проигрывает по процессорному времени. Матричные реализации затрачивают наименьшее количество процессорного времени.

Была достигнута цель лабораторной работы и выполнены все задачи, а именно были описаны, изучены и сравнены несколько алгоритмов поиска редакционного расстояния.

Для достижения поставленной цели были выполнены следующие задачи:

— описаны расстояния Левенштейна и Дамерау—Левенштейна;

— создан программный продукт с реализованными алгоритмами поиска расстояний Левенштейна и Дамерау—Левенштейна;

— исследованы затраты времени и памяти при различных реализациях алгоритмов;

— выполнено сравнение алгоритмов по процессорному времени и затратам оперативной памяти.