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 Динь Вьет Ань

Группа ИУ7И-54Б

Оценка (баллы) __________

Преподаватели Волкова Л. Л..

Москва — 2022 г.

Содержание

Введение

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

Расстояние Левенштейн — метрика, измеряющая по модулю разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (вставки, удаления, замены), необходимых для превращения одной последовательности символов в другую.

Расстояния Левенштейна и Дамерау-Левенштейна широко применяются для решения задач:

- компьютерной лингвистики (исправление ошибок в слове, автоматическое распознавание отсканированного текста или речи);
- теории информации;
- биоинформатики (для сравнения генов, хромосом и белков)

Расстояние Дамерау-Левенштейна — модификация расстояния Левештейна. Это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую.

В рамках выполнения лабораторной работы необходимо решить следующие задачи:

- Изучить расстояния Левенштена и Дамерау-Левенштейна;
- Построить схемы алгоритмов следующих методов: нерекурсивный метод поиска расстояния Левенштейна, нерекурсивный метод поиска Дамерау-Левенштейна, рекурсивный метод поиска Дамерау-Левенштейна, рекурсивный с кешированием метод поиска Дамерау-Левенштейна;
- Создать ПО, реализующее перечисленные выше алгоритмы;

- Сравнить алгоритмы определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
- Описать и обосновать полученные результаты в отчете о выполненной лабораторной работе;

1 Аналитическая часть

В данном разделе будут разобраны алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна.

1.1 Расстояние Левенштейна

Расстояние Левенштейна между двумя строками, позволяющая определить «схожесть» двух строк — минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую (каждая операция имеет свою цену – штраф).

Редакционное предписание – последовательность действий, необходимых для получения из первой строки вторую, и минимизирующую суммарную цену (и является расстоянием Левенштейна).
Пусть S_1 и S_2 – две строки, длиной N и M соответственно. Введены следующие обозначения:

- I (англ. Insert) – вставка символа в произвольной позиции ($w(\lambda, b) = 1$);
- D (англ. Delete) – удаление символа в произвольной позиции ($w(\lambda, b) = 1$);
- R (англ. Replace) – замена символа на другой ($w(a, b) = 1,\ a \neq b$);
- M (англ. Match) – совпадение двух символов ($w(a, a) = 0$).

С учетом введенных обозначений, расстояние Левенштейна может быть подсчитано по формуле 1.1:

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

Функция 1.2 $m(a, b)$ определена как:

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

1.1.1 Матричный алгоритм нахождения расстояния

Рекурсивный алгоритм вычисления расстояния Левенштейна может быть не эффективен при больших i и j, так как множество промежуточных значений $D(i, j)$ вычисляются не один раз, что сильно замедляет время выполнения программы.

В качестве структуры данных для хранения промежуточных значений можно использовать *матрицу*, имеющую размеры:

$$(length(S1) + 1) \times ((length(S2) + 1), \tag{1.3}$$

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 заполнены нулями.

Всю таблицу (за исключением первого столбца и первой строки) заполняем в соответствии с формулой 1.4:

$$A[i][j] = min \begin{cases} A[i-1][j]+1 \\ A[i][j-1]+1 \\ A[i-1][j-1]+m(S1[i],S2[j])) \end{cases} . \tag{1.4}$$

Функция $m(S1[i], S2[j])$ определена как:

$$m(S1[i], S2[j]) = \begin{cases} 0, & \text{если } S1[i-1] = S2[j-1], \\ 1, & \text{иначе} \end{cases} . \tag{1.5}$$

Результат вычисления расстояния Левенштейна будет ячейка матрицы с индексами $i = length(S1)$ и $j = length(S2)$.

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна между двумя строками, состоящими из конечного числа символов — это минимальное число операций вставки, удаления, замены одного символа и транспозиции двух соседних символов, необходимых для перевода одной строки в другую.

Является модификацией расстояния Левенштейна – добавлена операции *транспозиции*, то есть перестановки, двух символов.

Расстояние Дамерау – Левенштейна может быть найдено по формуле 1.6, которая задана как

$$
d_{a,b}(i,j) = \begin{cases}
\max(i,j), & \text{если } \min(i,j)=0,\\
\min\{ & \\
\quad d_{a,b}(i,j-1)+1, & \\
\quad d_{a,b}(i-1,j)+1, & \\
\quad d_{a,b}(i-1,j-1)+m(a[i],b[j]), & \text{иначе}\\
\quad \left[\begin{array}{ll} d_{a,b}(i-2,j-2)+1, & \text{если } i,j>1;\\ & a[i]=b[j-1];\\ & b[j]=a[i-1]\\ \infty, & \text{иначе}\end{array}\right. & \\
\} &
\end{cases},
\tag{1.6}
$$

Формула выводится по тем же соображениям, что и формула (1.1).

1.2.1 Рекурсивный алгоритм нахождения расстояния

Рекурсивный алгоритм вычисления расстояния Дамерау-Левенштейна реализует формулу 1.6

Минимальная цена преобразования – минимальное значение приведенных вариантов.

Если полагать, что *a', b'* – строки a и b без последнего символа соответственно, а *a", b"* – строки a и b без двух последних символов, то цена преобразования из строки a в b выражается из элементов, представленных ниже:

1) сумма цены преобразования строки a' в b и цены проведения операции удаления, которая необходима для преобразования a' в a;

2) сумма цены преобразования строки a в b' и цены проведения операции вставки, которая необходима для преобразования b' в b;

3) сумма цены преобразования из a' в b' и операции замены, предполагая, что a и b оканчиваются на разные символы;

4) сумма цены преобразования из a'' в b'' и операции перестановки, предполагая, что длины a'' и b'' больше 1 и последние два символа a'', поменянные местами, совпадут с двумя последними символами b'';

5) цена преобразования из a' в b', предполагая, что a и b оканчиваются на один и тот же символ.

1.2.2 Матричный алгоритм нахождения расстояния

Рекурсивный алгоритм вычисления расстояния Дамерау-Левенштейна может быть не эффективен при больших i и j, так как множество промежуточных значений $D(i, j)$ вычисляются не один раз, что сильно замедляет время выполнения программы.

В качестве структуры данных для хранения промежуточных значений можно использовать *матрицу*, имеющую размеры:

$$(length(S1) + 1) \times ((length(S2) + 1), \tag{1.7}$$

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 1.8.

$$A[i][j] = min \begin{cases} A[i-1][j] + 1 \\ A[i][j-1] + 1 \\ A[i-1][j-1] + m(S1[i], S2[j])) \\ A[i-2][j-2] + 1, \text{если } i > 1,\ j > 1 \text{ и} \\ \qquad S1[i-2] = S2[j-1],\ S2[i-1] = S2[j-2] \end{cases} \tag{1.8}$$

Функция $m(S1[i], S2[j])$ определена как:

$$m(S1[i], S2[j]) = \begin{cases} 0, & \text{если } S1[i-1] = S2[j-1], \\ 1, & \text{иначе} \end{cases} \tag{1.9}$$

Результат вычисления расстояния Дамерау-Левенштейна будет ячейка матрицы с индексами $i = length(S1)$ и $j = length(S2)$.

1.2.3 Рекурсивный алгоритм нахождения расстояния с использованием кеша

Чтобы уменьшить время работы рекурсивного алгоритма заполнения можно использовать *кеш*, который будет представлять собой матрицу.

Ускорение достигается за счет использования матрицы для предотвращения повторной обработки уже обработанных данных.

Если данные ещё не были обработаны, то результат работы рекурсивного алгоритма заносится в матрицу. В случае, если обработанные данные встречаются снова, то для них расстояние не находится и выполняется следующий шаг.

1.3 Вывод

В данном разделе были рассмотрены алгоритмы поиска расстояния Левенштейна и расстояния Дамерау-Левенштейна. В частности были приведены рекурентные формулы работы алгоритмов, объяснена разница между расстоянием Левенштейна и расстоянием Дамерау-Левенштейна.

2 Конструкторская часть

В этом разделе представлены описания используемых типов данных, а также схемы алгоритмов вычисления расстояния Левенштейна и Дамерау-Левенштейна.

2.1 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

- две переменных строкового типа;
- длина строки – целое число;
- в матричной реализации алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна, а также рекурсивной реализации с кешем – матрица, которая является двумерным списком целочисленного типа.

2.2 Сведения о модулях программы

Программа состоит из шести модулей:

- *main.py* – файл, содержащий точку входа и меню программы;
- *compareTime.py* – файл, содержаший код для измерения времени выпонения алгоритмов;
- *algorythms.py* – файл, содержащий код всех алгоритмов.

2.3 Разработка алгоритмов

На рисунках 2.1-2.2 представлены схемы алгоритмов вычисления расстояния Левенштейна и Дамерау-Левенштейна.

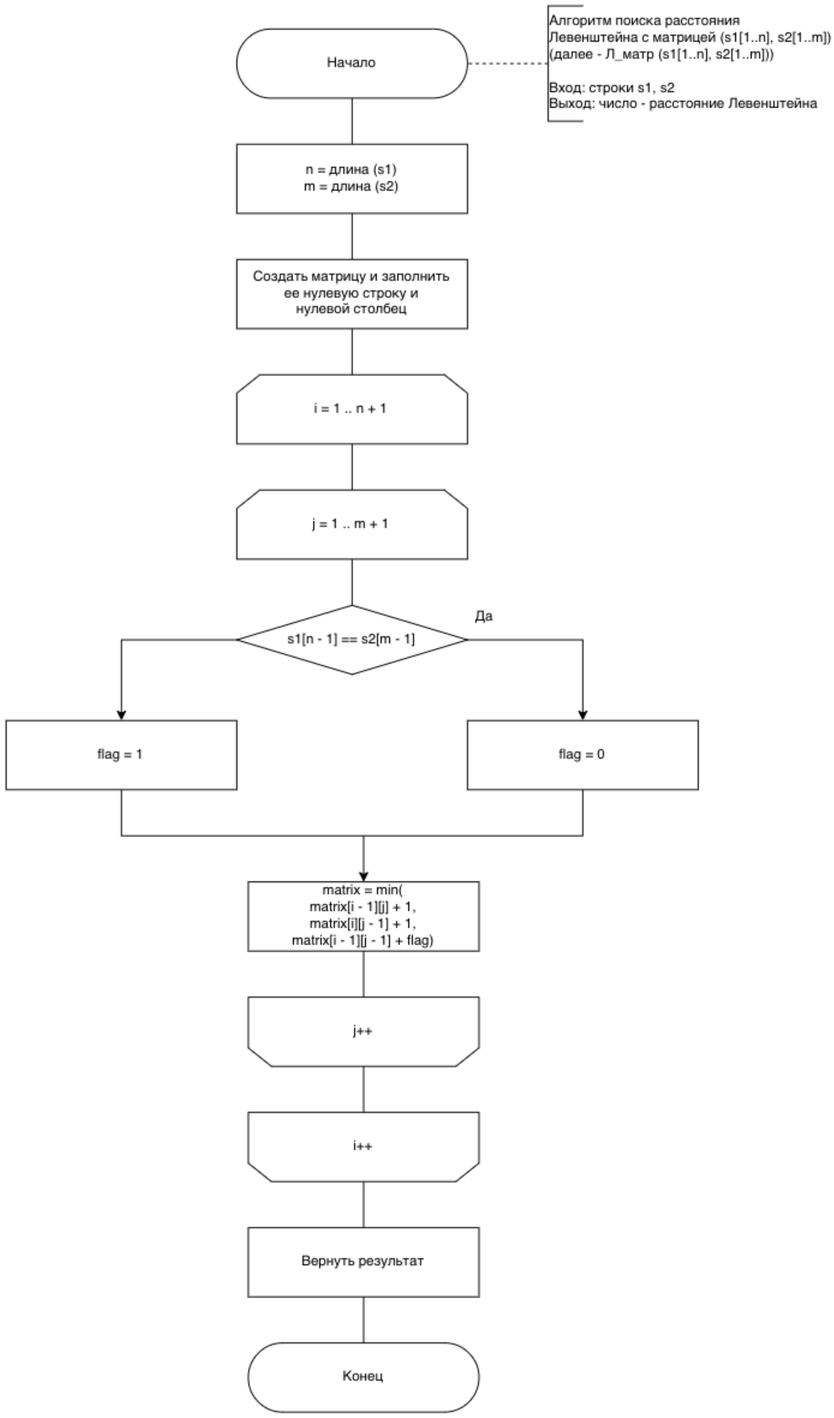

Рисунок 2.1 – Схема матричного алгоритма нахождения расстояния Левенштейна

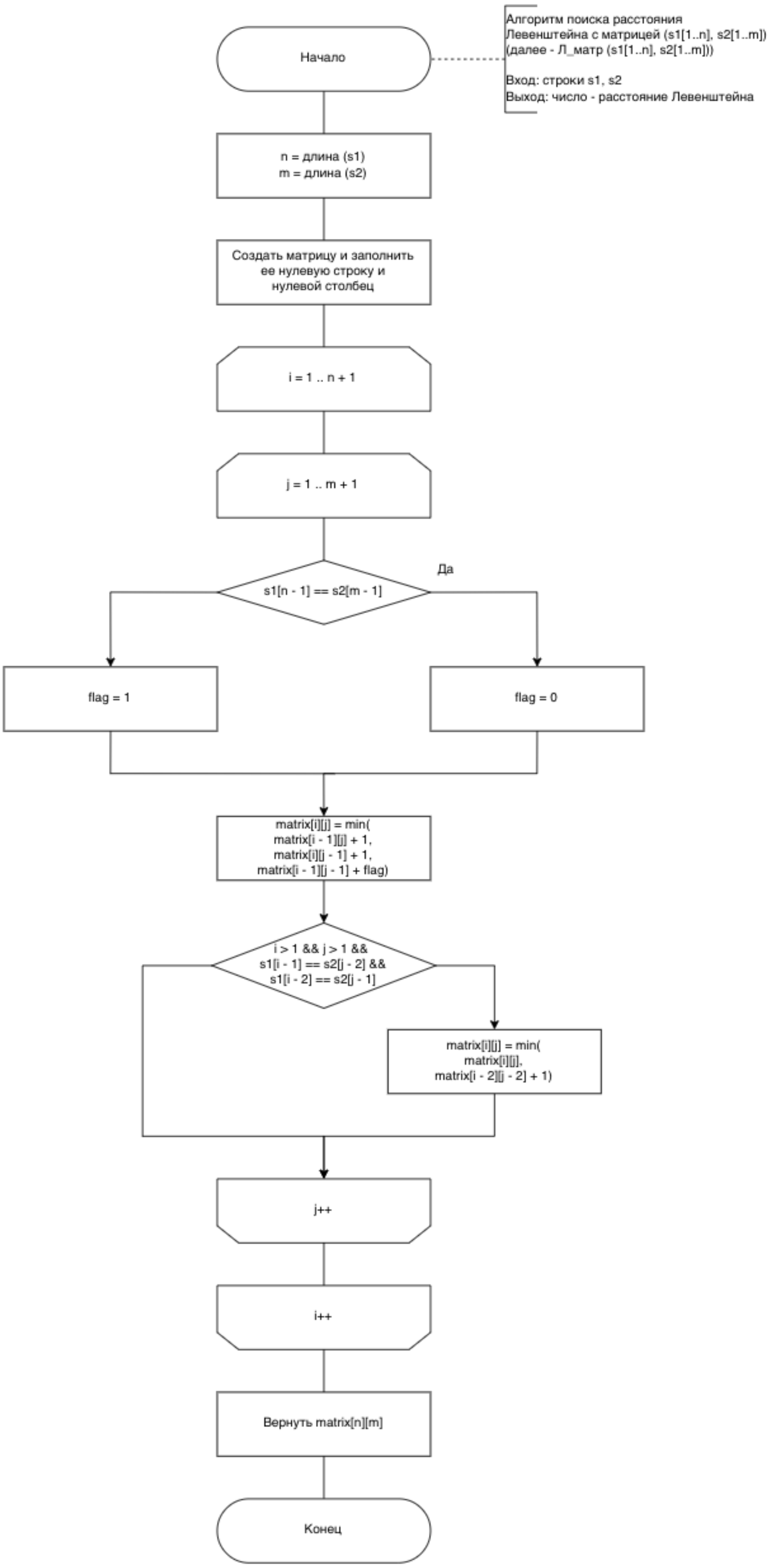

Рисунок 2.2 – Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

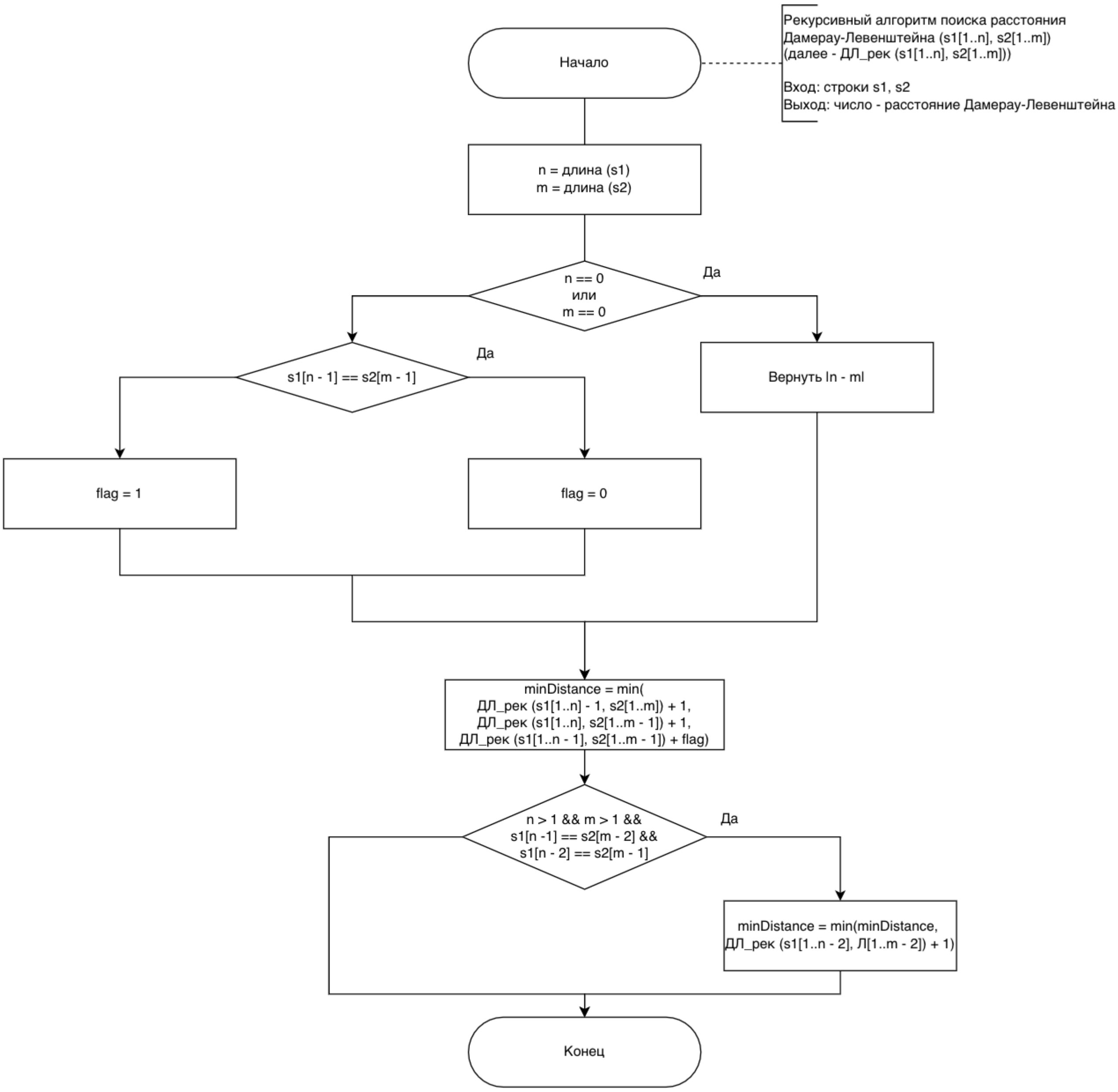

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

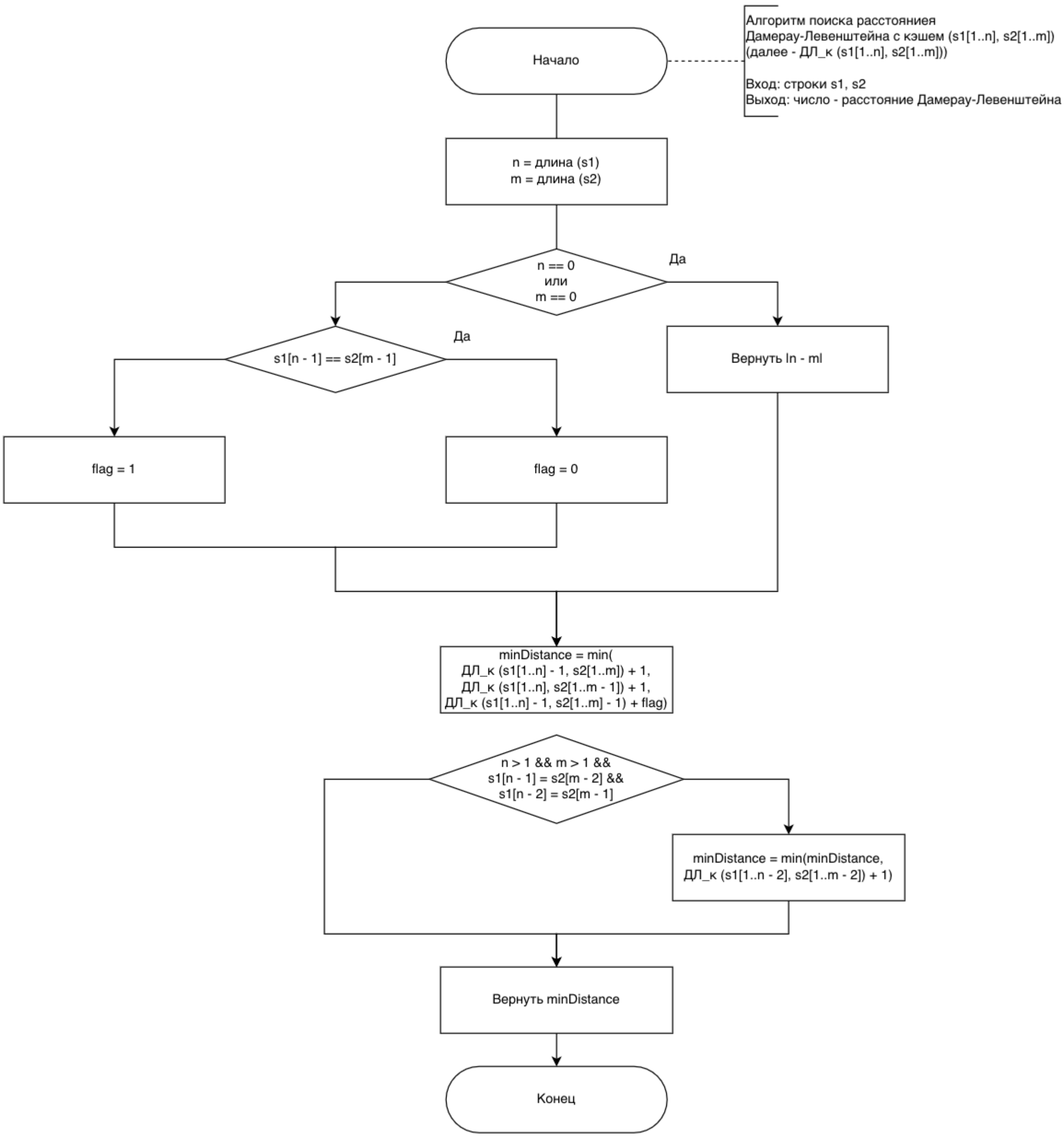

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с использованием кеша (матрицы)

2.4 Использование памяти

Замеры времени работы и используемой памяти алгоритмов Левенштейна и Дамерау-Левенштейна могут быть произведены одним и тем же способом. Тогда рассмотрим только рекурсивную и матричную реализации данных алгоритмов.

Пусть n – длина строки S1, m – длина строки S2.

Тогда затраты по памяти будут такими:

- алгоритм нахождения расстояния Левенштейна (матричный):
  - для матрицы – $((n + 1) \cdot (m + 1)) \cdot \text{sizeof(int)})$;
  - для S1, S2 – $(n + m) \cdot \text{sizeof(char)}$;
  - для n, m – $2 \cdot \text{sizeof(int)}$;
  - доп. переменные – $3 \cdot \text{sizeof(int)}$;
  - адрес возврата.
- алгоритм нахождения расстояния Дамерау-Левенштейна (матричный):
  - для матрицы – $((n + 1) \cdot (m + 1)) \cdot \text{sizeof(int)})$;
  - для S1, S2 – $(n + m) \cdot \text{sizeof(char)}$;
  - для n, m – $2 \cdot \text{sizeof(int)}$;
  - доп. переменные – $4 \cdot \text{sizeof(int)}$;
  - адрес возврата.
- алгоритм нахождения расстояния Дамерау-Левенштейна (рекурсивный), где для каждого вызова:
  - для S1, S2 – $(n + m) \cdot \text{sizeof(char)}$;
  - для n, m – $2 \cdot \text{sizeof(int)}$;
  - доп. переменные - $2 \cdot \text{sizeof(int)}$;
  - адрес возврата.

- алгоритм нахождения расстояния Дамерау-Левенштейна с использованием кеша в виде матрицы (память на саму матрицу: $((n + 1) \cdot (m + 1)) \cdot \text{sizeof(int)}$) (рекурсивный), где для каждого вызова:
  - для S1, S2 – $(n + m) \cdot \text{sizeof(char)}$;
  - для n, m – $2 \cdot \text{sizeof(int)}$;
  - доп. переменные – $2 \cdot \text{sizeof(int)}$;
  - ссылка на матрицу - 8 байт;
  - адрес возврата.

2.5 Вывод

В данном разделе были представлены описания используемых типов данных, а также схемы алгоритмов, рассматриваемых в лабораторной работе. Можно сделать вывод, что алгоритмы нахождения расстояния Левенштейна, использующие матрицу (матричный подход), а также рекурсивные алгоритмы с кешем, используют значительно больше памяти, чем рекурсивная реализация (примерно на $(n + m) \cdot \text{sizeof(char)}$ байт - размер используемой матрицы/кеша).

3 Технологическая часть

В данном разделе будут рассмотрены средства реализации, а также представлены листинги алгоритмов определения расстояния Левенштейна и Дамерау-Левенштейна.

3.1 Средства реализации

В данной работе для реализации был выбран язык программирования *Python*. В текущей лабораторной работе требуется замерить процессорное время работы выполняемой программы и визуализировать результаты при помощи графиков. Инструменты для этого присутствуют в выбранном языке программирования.

Время работы было замерено с помощью функции *process_time(...)* из библиотеки *time*.

3.2 Реализация алгоритмов

В листингах 3.1-3.4 представлена реализация алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Алгоритм нахождения расстояния Левенштейна (матричный)

```
def levenshteinDistance(str1, str2, output = True):
    n = len(str1)
    m = len(str2)
    matrix = createLevenshteinMatrix(n + 1, m + 1)

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

        if (str1[i - 1] != str2[j - 1]):
            change += 1

        matrix[i][j] = min(add, delete, change)

    if (output):
        printMatrix(matrix, str1, str2)

    return matrix[n][m]
```

Листинг 3.2 – Алгоритм нахождения расстояния Дамерау-Левенштейна (матричный)

```
def dameraulevenshteinDistance(str1, str2, output = True):
    n = len(str1)
    m = len(str2)
    matrix = createLevenshteinMatrix(n + 1, m + 1)

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            add = matrix[i - 1][j] + 1
            delete = matrix[i][j - 1] + 1
            change = matrix[i - 1][j - 1]
            if (str1[i - 1] != str2[j - 1]):
                change += 1
            swap = n
            if (i > 1 and j > 1 and
                str1[i - 1] == str2[i - 2] and
                str1[i - 2] == str2[i - 1]):
```

```
                    swap = matrix[i - 2][j - 2] + 1

            matrix[i][j] = min(add, delete, change, swap)

    if (output):
        printMatrix(matrix, str1, str2)

    return matrix[n][m]
```

Листинг 3.3 – Алгоритм нахождения расстояния Дамерау-Левенштейна (рекурсивный)

```
def damerauLevenshteinDistanceRecursive(str1, str2, output = True):
    n = len(str1)
    m = len(str2)
    flag = 0
    if ((n == 0) or (m == 0)):
    return abs(n - m)

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

    minDistance = min(
        damerauLevenshteinDistanceRecursive(str1[:-1], str2) + 1,
        damerauLevenshteinDistanceRecursive(str1, str2[:-1]) + 1,
        damerauLevenshteinDistanceRecursive(str1[:-1], str2[:-1]) +
           flag
    )
    if (n > 1 and m > 1 and str1[-1] == str2[-2] and str1[-2] ==
        str2[-1]):
        minDistance = min(
            minDistance,
            damerauLevenshteinDistanceRecursive(str1[:-2],
               str2[:-2]) + 1
    )

    return minDistance
```

Листинг 3.4 – Алгоритм нахождения расстояния Дамерау-Левенштейна (рекурсивный с использованием кеша)

```python
def recursiveWithCache(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]

    add = recursiveWithCache(str1, str2, n - 1, m, matrix) + 1
    delete = recursiveWithCache(str1, str2, n, m - 1, matrix) + 1
    change = recursiveWithCache(str1, str2, n - 1, m - 1, matrix)
    if (str1[n - 1] != str2[m - 1]):
        change += 1 # flag

    matrix[n][m] = min(add, delete, 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],
            recursiveWithCache(str1, str2, n - 2, m - 2, matrix) + 1
            )

    return matrix[n][m]

def damerauLevenshteinDistanceRecurciveCache(str1, str2, output =
    True):
     n = len(str1)
    m = len(str2)
    matrix = createLevenshteinMatrix(n + 1, m + 1)

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

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

    if (output):
```

```
        printMatrix(matrix, str1, str2)

    return matrix[n][m]
```

3.3 Функциональные тесты

В таблице 3.1 приведены тесты для функций, реализующих алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна. Тесты *для всех алгоритмов* пройдены успешно.

Таблица 3.1 – Функциональные тесты

			Ожидаемый результат	
№	Строка 1	Строка 2	Левенштейн	Дамерау-Л.
1	"пустая строка"	"пустая строка"	0	0
2	"пустая строка"	слово	5	5
3	проверка	"пустая строка"	8	8
4	ремонт	емонт	1	1
5	гигиена	иена	3	3
6	нисан	автоваз	6	6
7	спасибо	пожалуйста	9	9
8	что	кто	1	1
9	ты	тыква	3	3
10	есть	кушать	4	4
11	abba	baab	3	2
12	abcba	bacab	4	2

3.4 Вывод

В данном разделе были выбраны средства разработки алгоритмов. Также была представлена реализация всех алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна, которые были описаны в предыдущем разделе.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программы, а также проведен сравнительный анализ алгоритмов при различных ситуациях на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование представлены далее:

- операционная система: Window 10 Home Single Language;
- память: 8 Гб;
- процессор: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz .

Во время тестирования устройство было подключено к сети электропитания, нагружено приложениями окружения и самой системой тестирования

4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы.

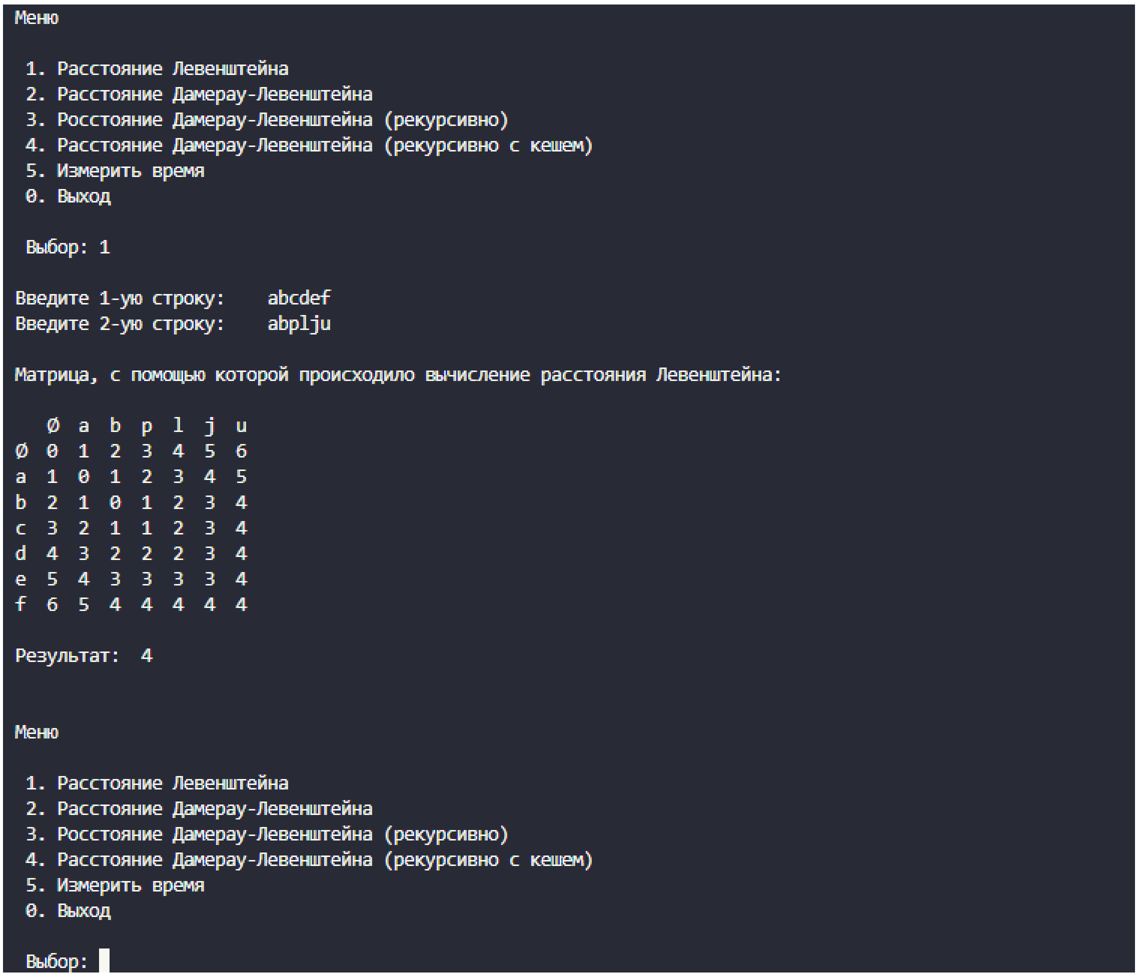

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения алгоритмов

Для замера времени используется функция замера процессорного времени process_time(...) из библиотеки time на Python. Она возвращает пользовательское процессорное время типа float.

Использовать функцию приходится дважды, затем из конечного времени нужно вычесть начальное, чтобы получить результат.

Замеры проводились для длины слова от 0 до 7 по 300 раз на различных входных данных.

На рисунках 4.2, 4.3, 4.4 приведены графические результаты замеров.

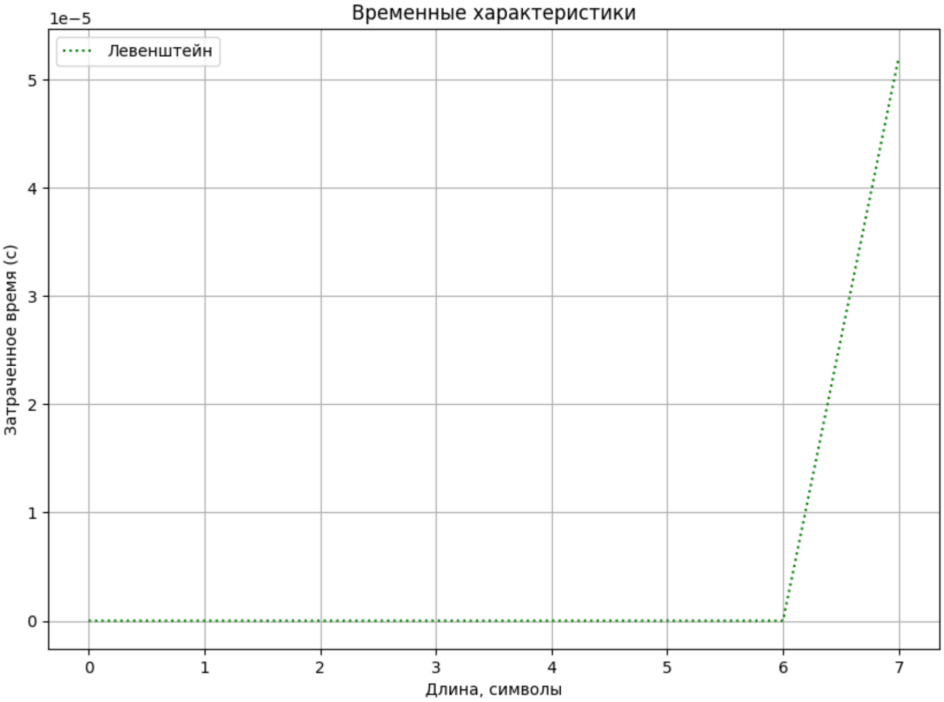

Рисунок 4.2 – Результат работы алгоритма нахождения расстояния Левештейна (матричного)

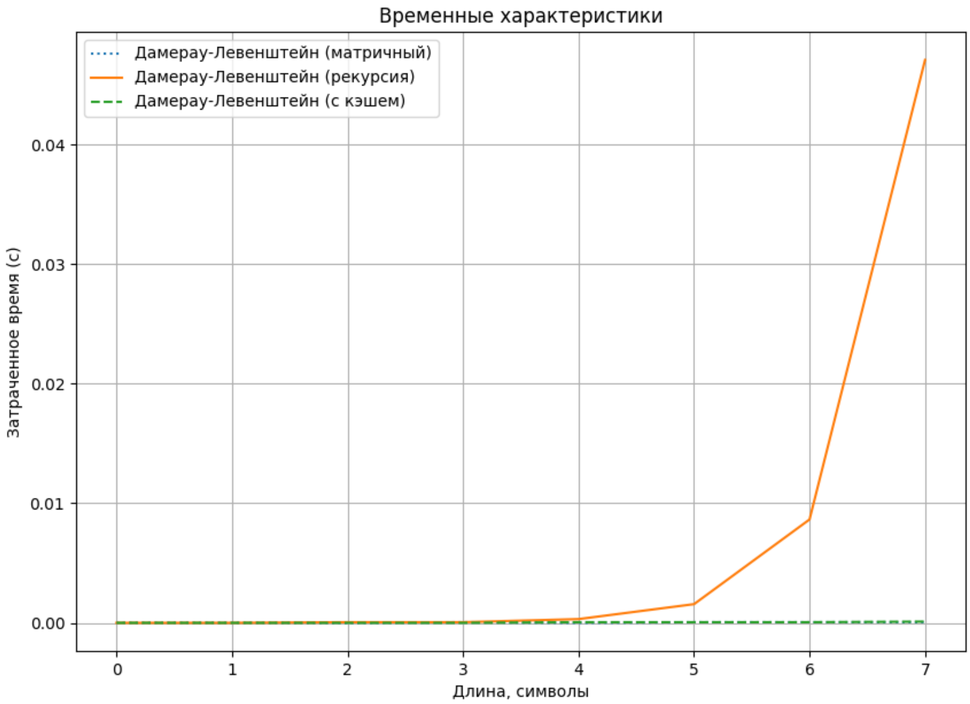

Рисунок 4.3 – Сравнение алгоритмов нахождения расстояния Дамерау-Левенштейна (матричного, рекурсивного и рекурсивного с использованием кеша)

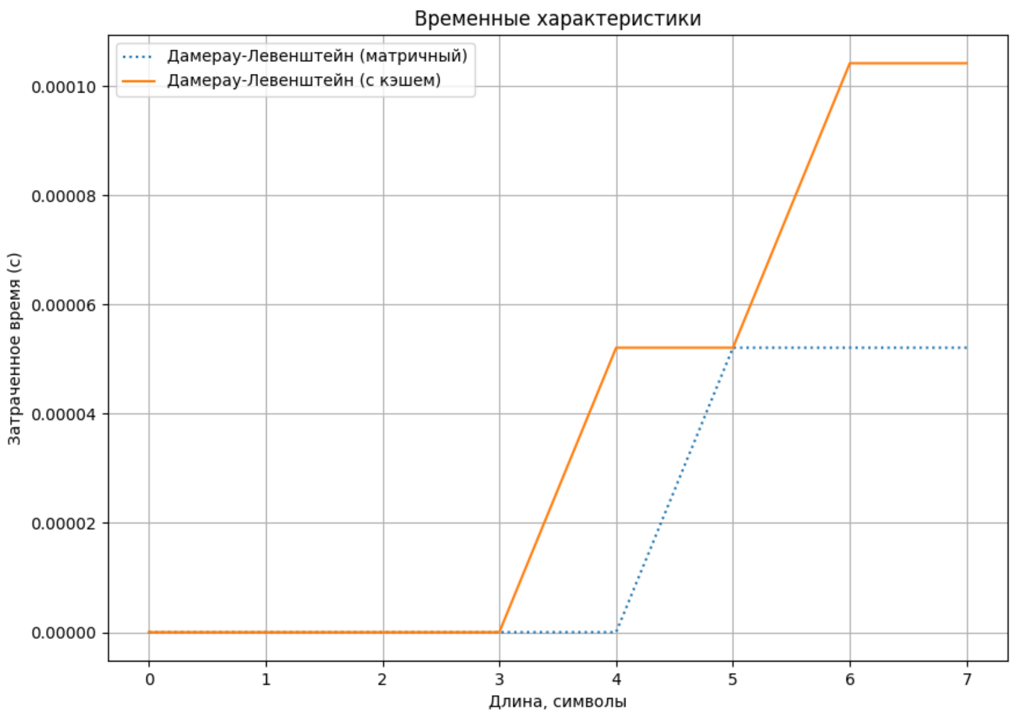

Рисунок 4.4 – Сравнение алгоритмов нахождения расстояния Дамерау-Левенштейна (матричного и рекурсивного с использованием кеша)

Сложность матричного алгоритма нахождения расстояния Левенштейна составляет $O(n^2)$ (рисунок 4.2).

В общем случае рекурсивный алгоритм алгоритм нахождения расстояния Дамерау-Левенштейна медленнее, чем его реализация с кешем или матричная реализация (рисунок 4.3), а также что матричная реализация несколько быстрее рекурсивного алгоритма с использованием кеша (рисунок 4.4).

4.4 Вывод

В результате замеров можно прийти к выводу, что матричная реализация алгоритмов нахождения расстояний заметно выигрывает по времени при росте строк, но проигрывает по количеству затрачиваемой памяти.

Заключение

В результате выполнения лабораторной работы при исследовании алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна были применены и отработаны навыки динамического программирования.

В ходе выполнения лабораторной работы были решены следующие задачи:

- изучены алгоритмы нахождения расстояний Левенштейна и Дамерау–Левенштейна;
- реализованы алгоритмы поиска расстояния Левенштейна и расстояния Дамерау-Левенштейна без рекурсии;
- реализованы рекурсивные алгоритмы поиска расстояния Дамерау - Левенштейна с и без матрицы-кеша;
- проведен сравнительный анализ линейной и рекурсивной реализаций алгоритмов определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
- подготовлен отчет о лабораторной работе.