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 Калашков П. А.

Группа ИУ7-56Б

Оценка (баллы) ______

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

Москва — 2022 г.

Содержание

Введение

Операции работы со строками являются важной частью всего программирования. Часто возникает потребность в использовании строк для различных задач, для решения которых нужны алгоритмы сравнения строк, о которых и пойдет речь в данной работе. Они используются при решении таких задач, как исправление ошибок в тексте, предлагая заменить введенное слово с ошибкой на наиболее подходящее.

Целью данной работы является изучение, реализация и исследование алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна. Для достижения поставленной цели необходимо выполнить следующие задачи:

- изучить и реализовать алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна;
- протестроивать по времени и по памяти для алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна;
- сравнить по времени алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна;
- сравнить анализ по времени матричной, рекурсивной, а также рекурсивной с использованием кеша реализаций алгоритма нахождения расстояния Дамерау-Левенштейна;
- описать и обосновать полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

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

1.1 Расстояние Левенштейна

Расстояние Левенштейна [1] между двумя строками – метрика, 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 [2] 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

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

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 приведены формулы вычисления расстояния Левенштейна и Дамерау-Левенштейна, которые являются рекуррентными, что позволяет реализовать их как рекурсивно, так и итеративно.

В качестве входных данных в программу будут подаваться две строки на русском или английском языке. Требуется реализовать меню для вызова алгоритмов и замеров времени и учесть, что программа не должна аварийно завершаться в случае ввода пустых строк.

Реализуемое программное обеспечение будет работать в двух режимах – пользовательский, в котором можно выбрать алгоритм и вывести результат его работы, а также экспериментальный режим, в котором можно произвести сравнение алгоритмов по времени работы на различных входных данных.

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;
- *menu.py* – файл, содержащий код меню программы;
- *test.py* – файл, содержаший код тестирования алгоритмов;
- *utils.py* – файл, содержащий служебные алгоритмы;
- *constants.py* – файл, содержаший константы программы;
- *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.

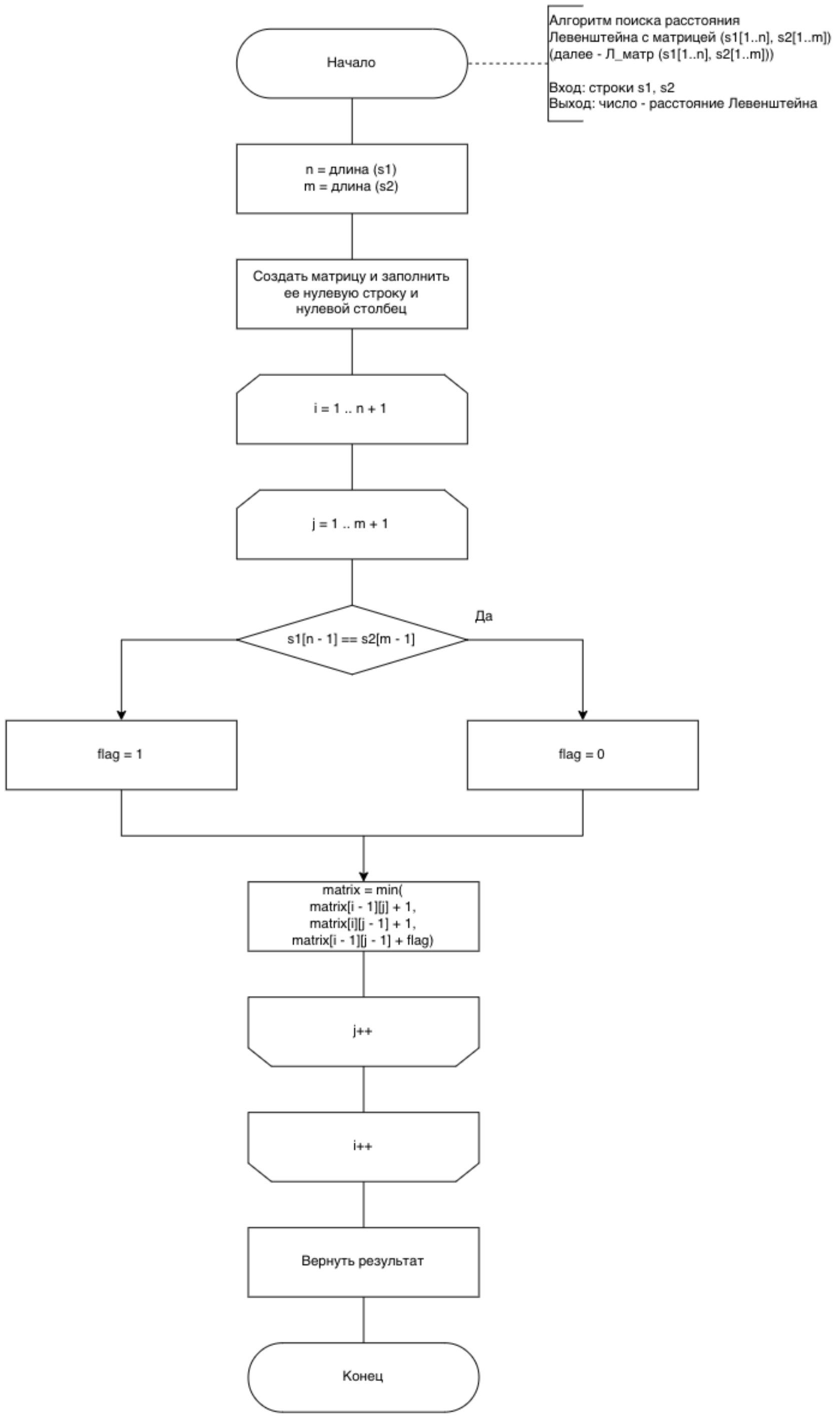

Рисунок 2.1 – Схема матричного алгоритма нахождения расстояния Левенштейна

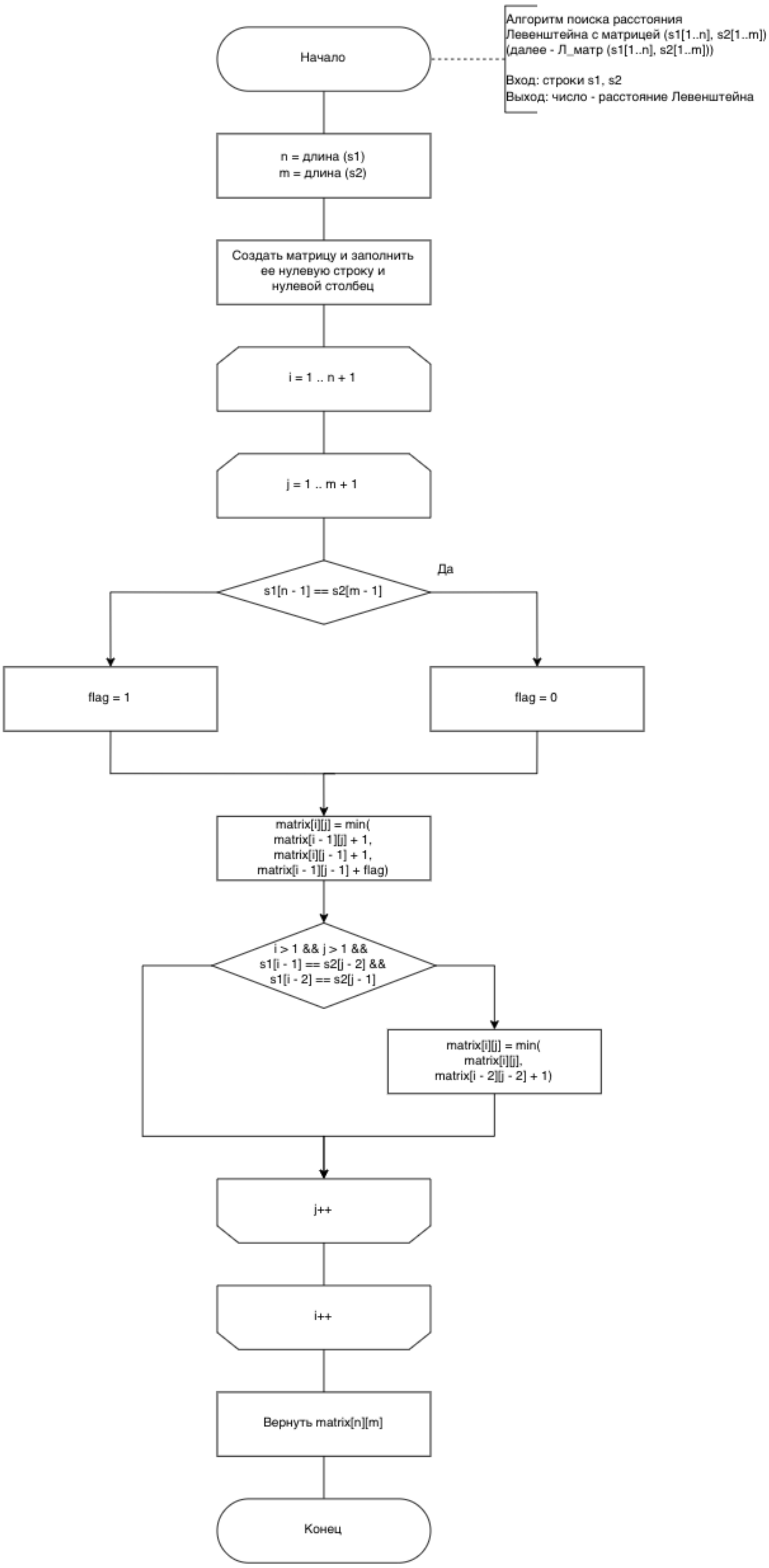

Рисунок 2.2 – Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

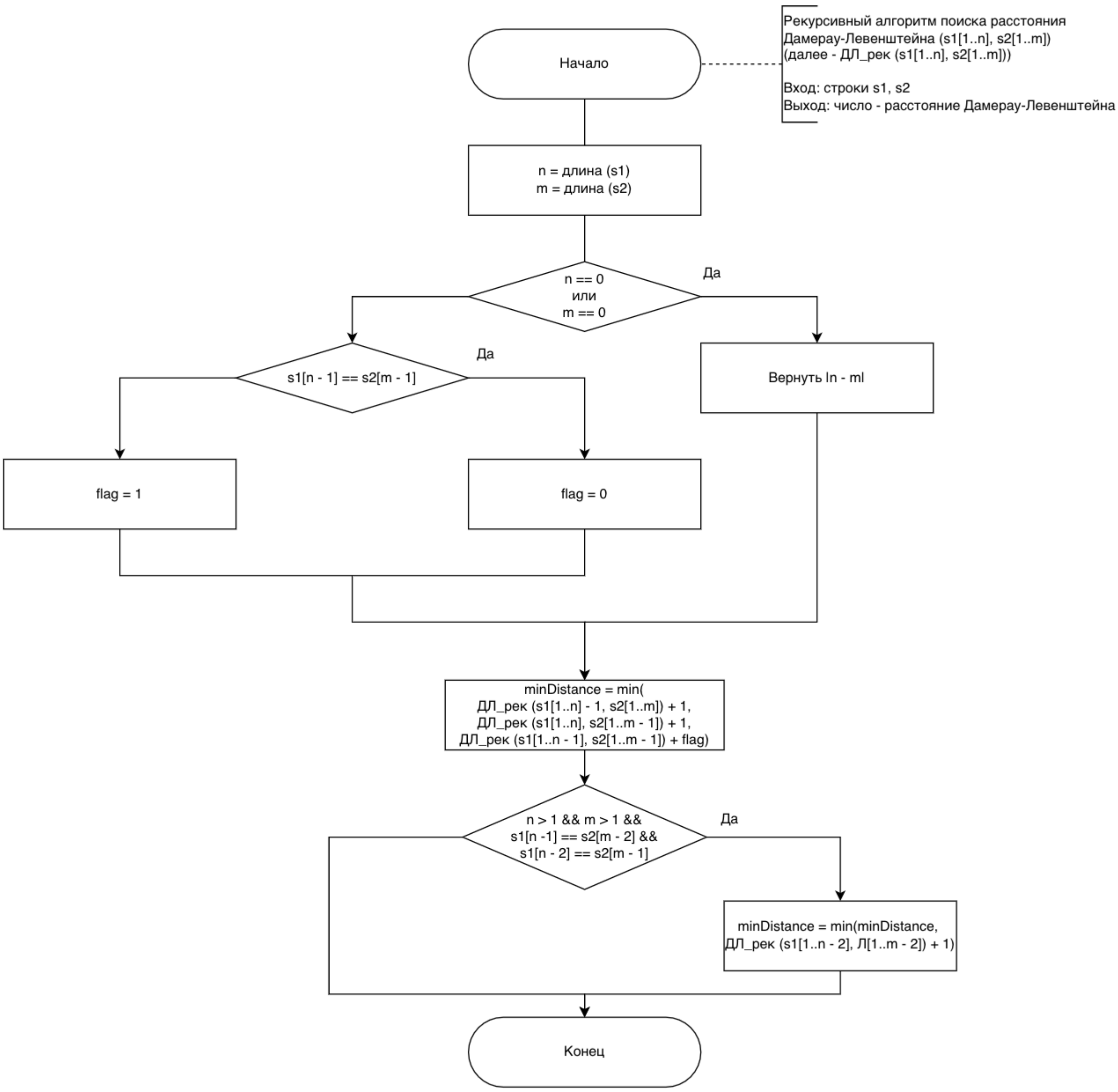

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

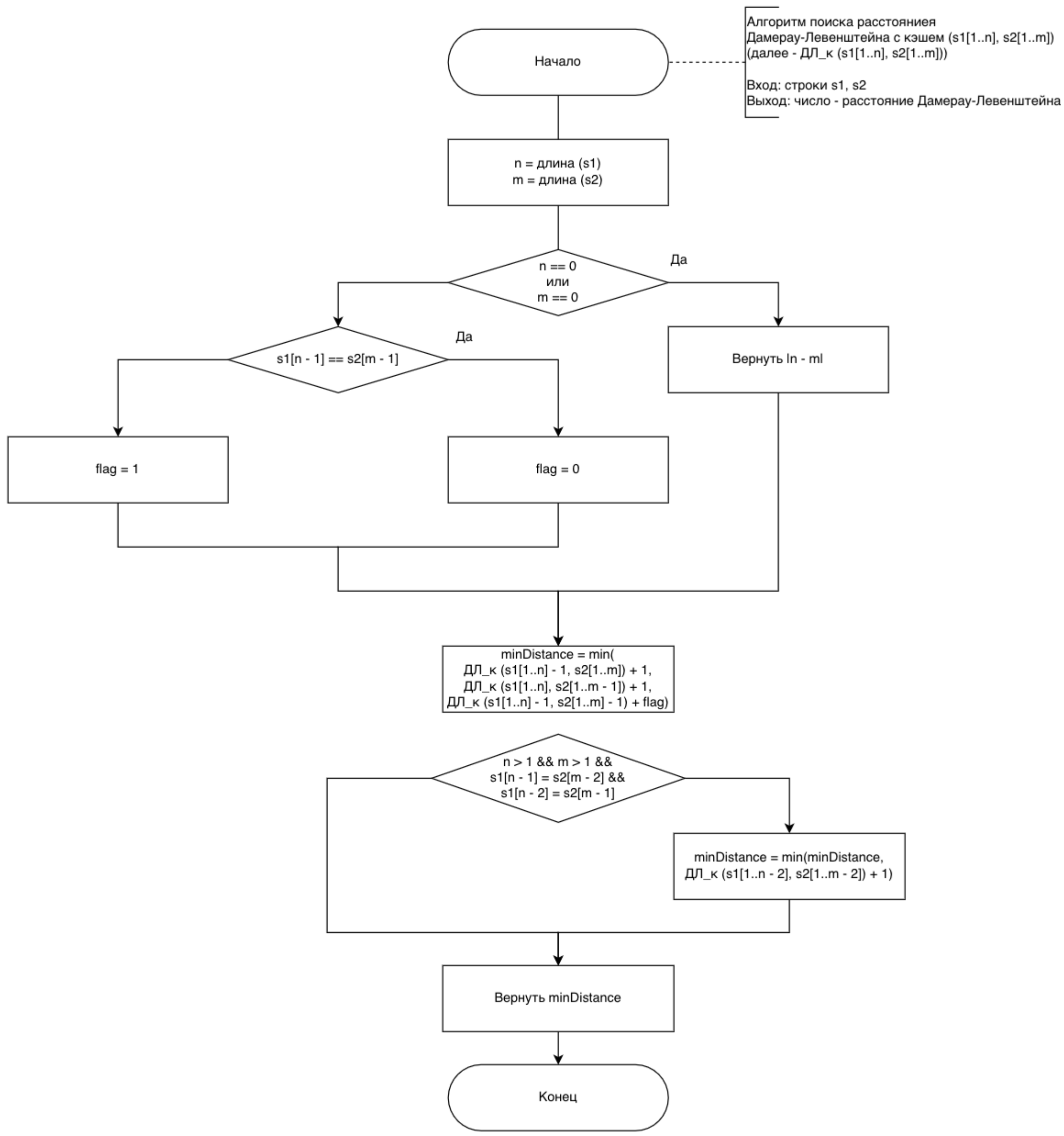

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с использованием кеша (матрицы)

2.4 Классы эквивалентности функционального тестирования

Для функционального тестирования выделены классы эквивалентности, представленные ниже.

1. Ввод двух пустых строк.
2. Одна из строк – пустая.
3. Расстояния, которые вычислены алгоритмами Левенштейна и Дамерау-Левенштейна, равны.
4. Расстояния, которые вычислены алгоритмами Левенштейна и Дамерау-Левенштейна, отличаются

2.5 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 – $((n + 1)\cdot (m + 1)) \cdot$ sizeof(int));
  - для S1, S2 – $(n + m) \cdot$ sizeof(char);
  - для n, m – $2 \cdot$ sizeof(int);
  - доп. переменные – $3 \cdot$ sizeof(int);
  - адрес возврата.
- алгоритм нахождения расстояния Дамерау-Левенштейна (матричный):

– для матрицы – $((n + 1) \cdot (m + 1)) \cdot sizeof(int))$;
– для S1, S2 – $(n + m) \cdot sizeof(char)$;
– для n, m – $2 \cdot sizeof(int)$;
– доп. переменные – $4 \cdot sizeof(int)$;
– адрес возврата.

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

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

2.6 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 $(n + m) \cdot sizeof(char)$ 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 $Python$[3]. 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 $time$[4].

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: str, str2: str, output: bool = True)
    -> int:
    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: str, str2: str, output: bool =
    True) -> int:
    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: str, str2: str, output: bool = True) -> int:
    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)

```
def recursiveWithCache(
str1: str, str2: str, n: int, m: int, matrix: Matrix) -> None:
    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: str, str2: str, output: bool = True) -> int:
    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: Mac OS Monterey Версия 12.5.1 (21G83) [5] x86_64;
- память: 16 Гб;
- процессор: 2,7 ГГцz 4-ядерный процессор Intel Core i7 [6].

При тестировании ноутбук был включен в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также системой тестирования.

4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы.

```
Меню

 1. Расстояние Левенштейна
 2. Расстояние Дамерау-Левенштейна
 3. Росстояние Дамерау-Левенштейна (рекурсивно)
 4. Расстояние Дамерау-Левенштейна (рекурсивно с кешем)
 5. Измерить время
 0. Выход

 Выбор: 1

Введите 1-ую строку:    нирыба
Введите 2-ую строку:    нимясо

Матрица, с помощью которой происходило вычисление расстояния Левенштейна:

   ø  н  и  м  я  с  о
ø  0  1  2  3  4  5  6
н  1  0  1  2  3  4  5
и  2  1  0  1  2  3  4
р  3  2  1  1  2  3  4
ы  4  3  2  2  2  3  4
б  5  4  3  3  3  3  4
а  6  5  4  4  4  4  4

Результат:  4


Меню

 1. Расстояние Левенштейна
 2. Расстояние Дамерау-Левенштейна
 3. Росстояние Дамерау-Левенштейна (рекурсивно)
 4. Расстояние Дамерау-Левенштейна (рекурсивно с кешем)
 5. Измерить время
 0. Выход

 Выбор: █
```

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 9 по 100 раз на различных входных данных.

Результаты замеров приведены в таблице 4.1 (время в мс).

Таблица 4.1 – Результаты замеров времени

Длина	Л.(матр)	Д-Л.(матр.)	Д-Л.(рек.)	Д.-Л.(рек. с кешем)
0	0.0033	0.0074	0.0089	0.0032
1	0.0138	0.0130	0.0091	0.0216
2	0.0154	0.0169	0.0326	0.0363
3	0.0225	0.0227	0.1430	0.0521
4	0.0284	0.0331	0.6516	0.0751
5	0.0410	0.0472	3.1557	0.1328
6	0.0509	0.0633	16.7735	0.1755
7	0.0653	0.0854	89.8081	0.2375
8	0.0866	0.1064	496.2408	0.3080
9	0.1049	0.1354	2724.1102	0.3792

Также на рисунках 4.2, 4.3, 4.4 приведены графические результаты замеров.

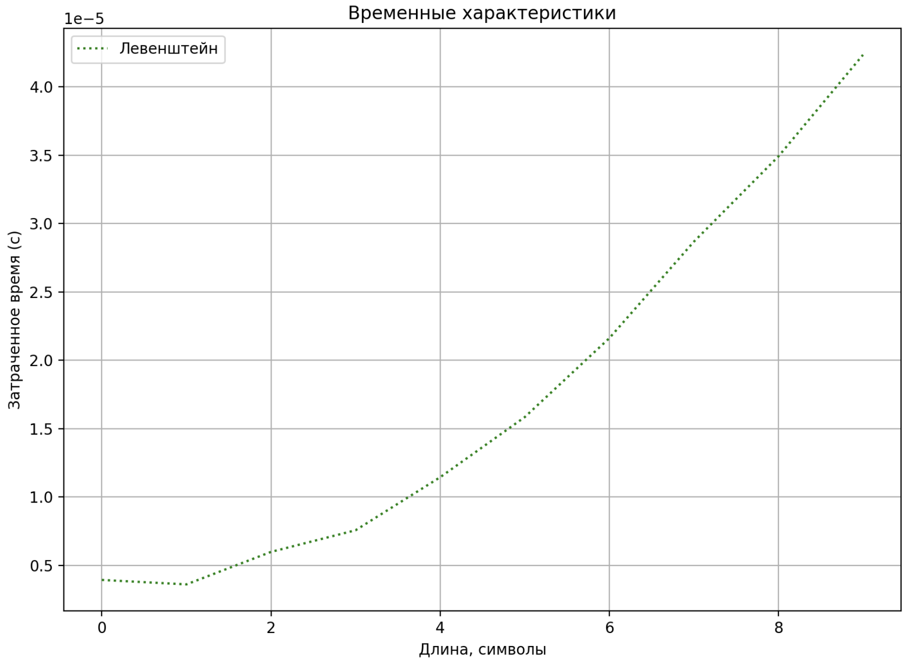

Рисунок 4.2 – Результат работы алгоритма нахождения расстояния Левештейна (матричного)

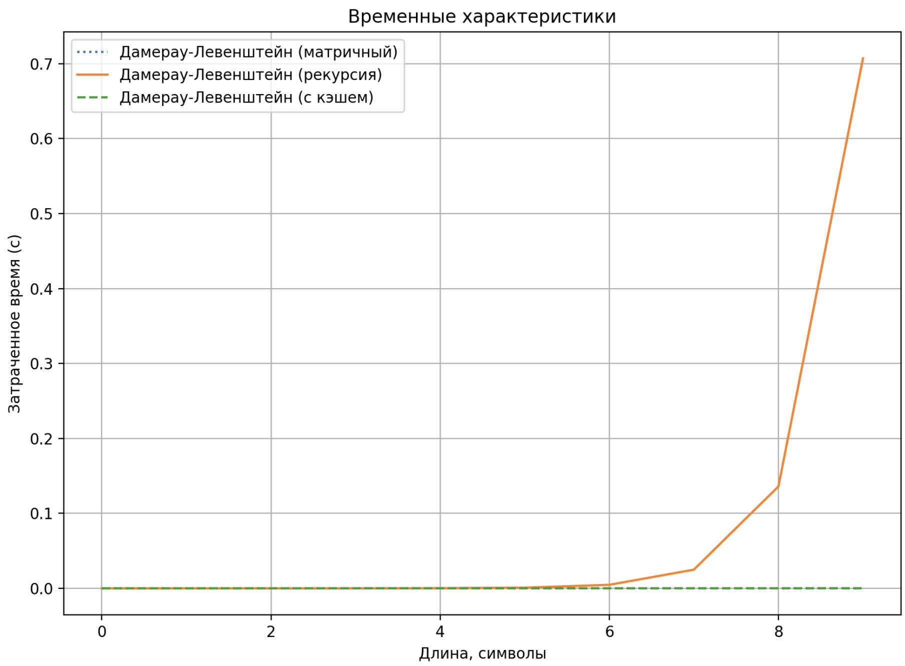

Рисунок 4.3 – Сравнение алгоритмов нахождения расстояния Дамерау-Левенштейна (матричного, рекурсивного и рекурсивного с использованием кеша)

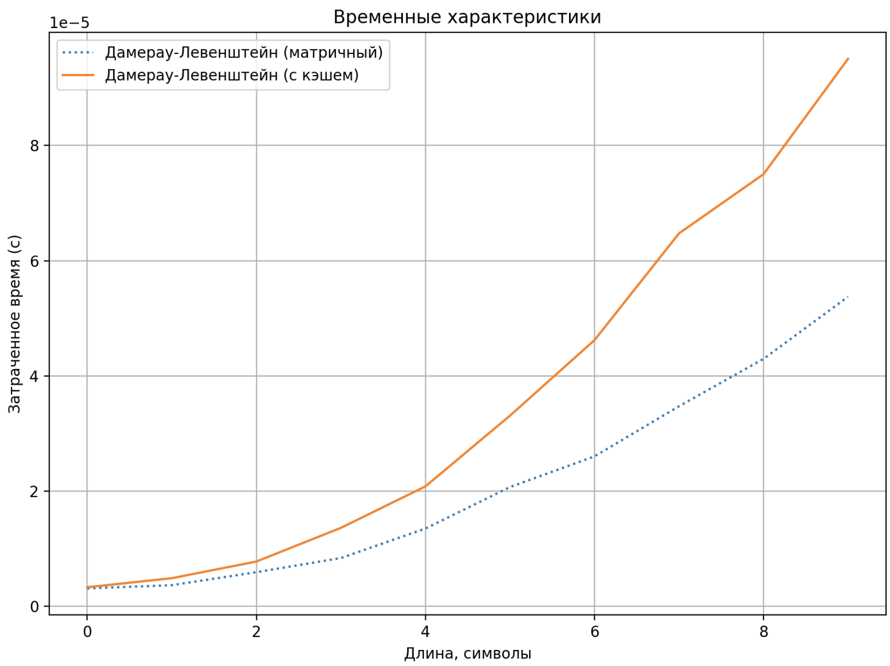

Рисунок 4.4 – Сравнение алгоритмов нахождения расстояния Дамерау-Левенштейна (матричного и рекурсивного с использованием кеша)

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 нескольлко быстрее рекурсивного алгоритма с использованием кеша (рисунок 4.4).

4.4 Вывод

Исходя из замеров по памяти, итеративные алгоритмы проигрывают рекурсивным, потому что максимальный размер памяти в них растет, как произведение длин строк, а в рекурсивных – как сумма длин строк.

В результате проведенных измерений было получено, что обычно матричный алгоритм нахождения расстояния Дамерау-Левенштейна быстрее рекурсивного алгоритма с использованием кеша, однако занимает он намного больше памяти. Так, для длины слова в 9 символов матричная реализация быстрее рекурсивной в 2 раза, однако занимает в 5 раз больше памяти

Также было выявлено, что на длине строк в 4 символа рекурсивная реализация алгоритма Дамерау-Левенштейна в уже в 21 раз медленнее матричной реализации алгоритма. При увеличении длины строк в геометрической прогрессии растет и время работы рекурсивной реализации. Следовательно, для строк длиной более 4 символов стоит использовать матричную реализацию.

Заключение

В результате исследования было определено, что время алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна растет в геометрической прогрессии при увеличении длин строк. Лучшие показатели по времени дает матричная реализация алгоритма нахождения расстояния Дамерау-Левенштейна и его рекурсивная реализация с кешем, использование которых приводит к 21-кратному превосходству по времени работы уже на длине строки в 4 символа за счет сохранения необходимых промежуточных вычислений. При этом матричные реализации занимают довольно много памяти при большой длине строк.

Цель, которая была поставлена в начале лабораторной работы, была достигнута, а также в ходе выполнения лабораторной работы были решены следующие задачи:

- были изучены и реализованы алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна;
- были также изучены матричная реализация, а также реализация с использованием кеша в виде матрицы для алгоритма нахождения расстояния Дамерау-Левенштейна;
- проведен сравнительный анализ алгоритмов нахождения расстояний Дамерау-Левенштейна в матричной, рекурсивной и рекурсивной и использованием кеша реализациях;
- подготовлен отчет о лабораторной работе.

Список источников

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

[2] Черненький В. М. Гапанюк Ю. Е. Методика идентификации пассажира по установочным данным. – М.: Вестник МГТУ им. Н.Э. Баумана. Сер. “Приборостроение”, 2012. Т. 163. С. 30–34.

[3] Welcome to Python [Электронный ресурс]. Режим доступа: `https://www.python.org` (дата обращения: 17.09.2022).

[4] time — Time access and conversions [Электронный ресурс]. Режим доступа: `https://docs.python.org/3/library/time.html#functions` (дата обращения: 17.09.2022).

[5] macOS Monterey [Электронный ресурс]. Режим доступа: `https://www.apple.com/macos/monterey/` (дата обращения: 17.09.2022).

[6] Процессор Intel® Core™ i7 [Электронный ресурс]. Режим доступа: `https://www.intel.com/processors/core/i7/docs` (дата обращения: 17.09.2022).