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 Маслюков П.В.

Группа ИУ7-52Б

Оценка (баллы) ______

Преподаватель Волкова Л. Л.

Москва — 2023 г.

Содержание

Введение

Важная часть программирования состоит в операциях работы со строками, которые часто используются для различных задач, таких как создание обычных статей или записей в базу данных. В связи с этим возникает несколько важных задач, для которых требуются алгоритмы сравнения строк. В данной работе будет рассмотрено обсуждение этих алгоритмов.. Подобные алгоритмы используются при:

- исправлении ошибок в тексте, предлагая заменить введенное слово с ошибкой на наиболее подходящее;
- поиске слова в тексте по подстроке;
- сравнении целых текстовых файлов.

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-Левенштейнаа;
- изучить и реализовать матричую реализацию, а также реализацию с использованием кеша в виде матрицы для алгоритма Левенштейна;
- провести сравнительный анализ алгоритмов Левенштейна и Дамерау-Левенштейна, а также сравнение рекурсивной и матричной реализаций, матричной реализации и реализаций с кешом алгоритма Левенштейнаа;
- подготовить отчет о лабораторной работе.

1 Аналитическая часть

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

1.1 Расстояние Левенштейна

Расстояние Левенштейна между двумя строками - определяющая степень «схожести» двух строк, это наименьшее количество операций, которое необходимо выполнить для преобразования одной строки в другую. Каждая операция может быть вставкой символа, удалением символа или заменой одного символа на другой, и каждая операция имеет свою стоимость в виде штрафа.[1]

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. Введем 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 следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0 & \text{i = 0, j = 0} \\ i & \text{j = 0, i > 0} \\ j & \text{i = 0, j > 0} \\ \min\{ & \\ \qquad D(i, j-1) + 1 & \\ \qquad D(i-1, j) + 1 & \text{i > 0, j > 0} \\ \qquad D(i-1, j-1) + m(a[i], b[j]) & (1.2) \\ \} & \end{cases} \tag{1.1}$$

Функция 1.2 определена как:

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

1.2 Рекурсивный алгоритм нахождения расстояния Левенштейна

Рекурсивный алгоритм вычисления расстояния Левенштейна реализует формулу 1.1

Минимальная цена преобразования - минимальное значение приведенных вариантов.

Если полагать, что *a', b'* - строки a и b без последнего символа соответственно, то цена преобразования из строки a в b может быть выражена так:

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

1.3 Матричный алгоритм нахождения расстояния Левенштейна

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 оптимизации можно использовать *матрицу* для хранения промежуточных значений. Матрица имеет размеры:

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

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 тривиальны.

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

$$A[i][j] = min \begin{cases} A[i-1][j] + 1 \\ A[i][j-1] + 1 \\ A[i-1][j-1] + m(S1[i], S2[j]) \quad (1.5) \end{cases} . \tag{1.4}$$

Функция 1.5 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 $j = length(S2)$.

1.4 Рекурсивный алгоритм нахождения расстояния Левенштейна с использованием кеша

В качестве оптимизации рекурсивного алгоритма заполнения можно использовать *кеш*, который будет представлять собой матрицу.

Суть оптимизации - при выполнении рекурсии происходит параллельное заполнение матрицы.

Если рекурсивный алгоритм выполняет прогон для данных, которые еще не были обработаны, то результат нахождения заносится в матрицу. Иначе, если обработанные данные встречаются снова, то для них расстояние не находится и алгоритм переходит к следующему шагу.

1.5 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

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

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

1.6 Вывод

В данном разделе были рассмотрены теоретические аспекты формул Левенштейна и Дамерау-Левенштейна. Обе формулы являются рекуррентными, что значит, что их можно реализовать как рекурсивно, так и с помощью итерационно.

Программа будет принимать две строки в качестве входных данных. Также мы реализовали меню, которое позволяет вызывать алгоритмы и замерять время их работы. Ограничениями программы является то, что она должна корректно работать со строками на английском и русском языках, а также учитывать случай ввода пустых строк.

Разрабатываемое программное обеспечение будет функционировать в двух режимах: пользовательском и экспериментальном. В пользовательском режиме можно выбрать алгоритм и получить вычисленное значение. В экспериментальном режиме можно сравнить время работы алгоритмов на различных входных данных.

2 Конструкторская часть

В этом разделе будут представлено описание 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 строки типа *str*;
- длина строки - целое число типа *int*;
- в матричной реализации алгоритма Левенштейна и рекурсивной реализации с кешем - матрица, которая является двумерным списком типа *int*.

2.2 Сведения о модулях программы

Программа состоит из двух модулей:

- *main.py* - файл, содержащий весь служебный код;
- *algorythms.py* - файл, содержащий код всех алгоритмов.

2.3 Схемы алгоритмов

На рисунках 2.1-2.4 представлены схемы алгоритмов вычисления расстояния Левенштейна и Дамерау-Левенштейна.

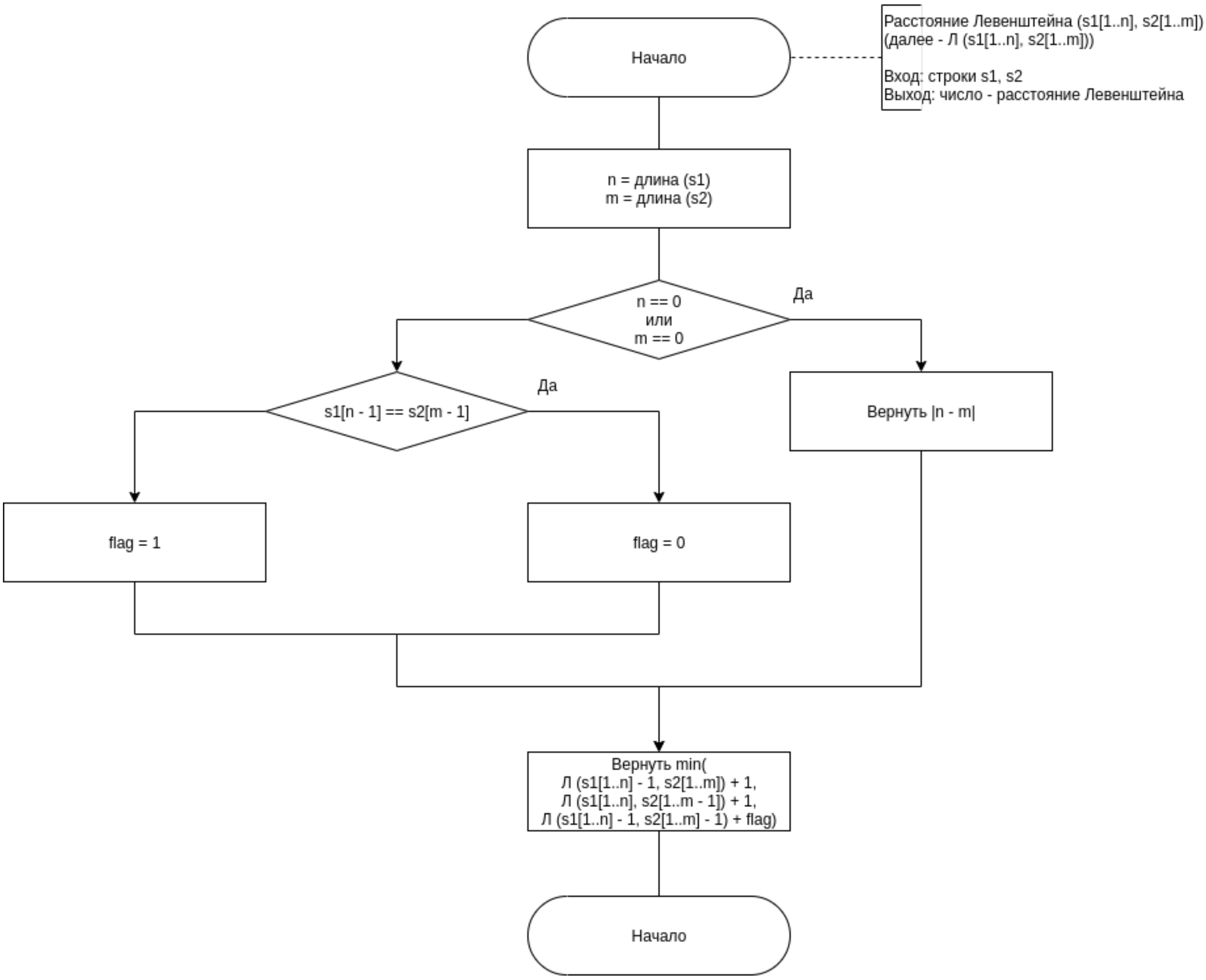

Рисунок 2.1 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна

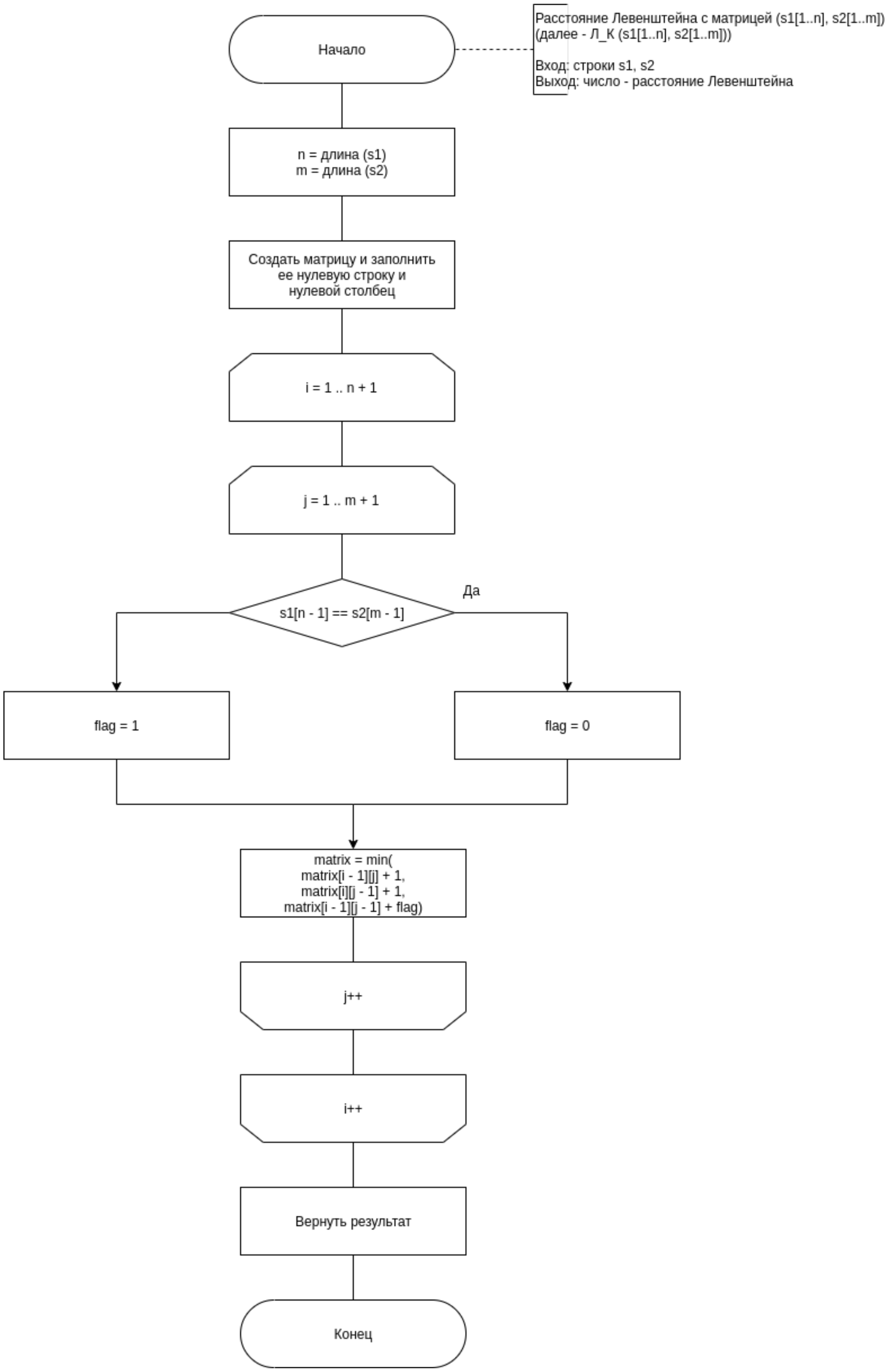

Рисунок 2.2 – Схема матричного алгоритма нахождения расстояния Левенштейна

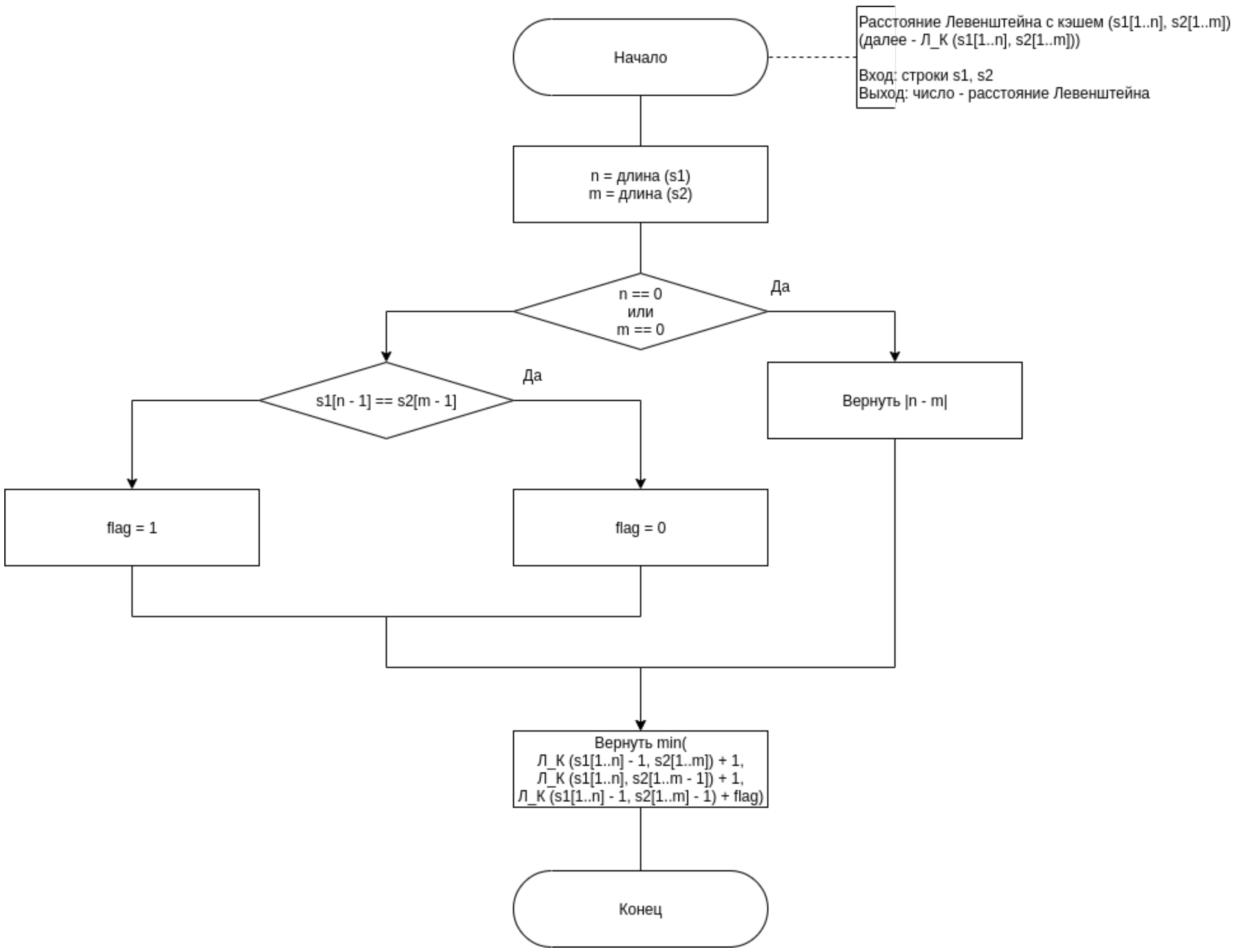

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна с использованием кеша (матрицы)

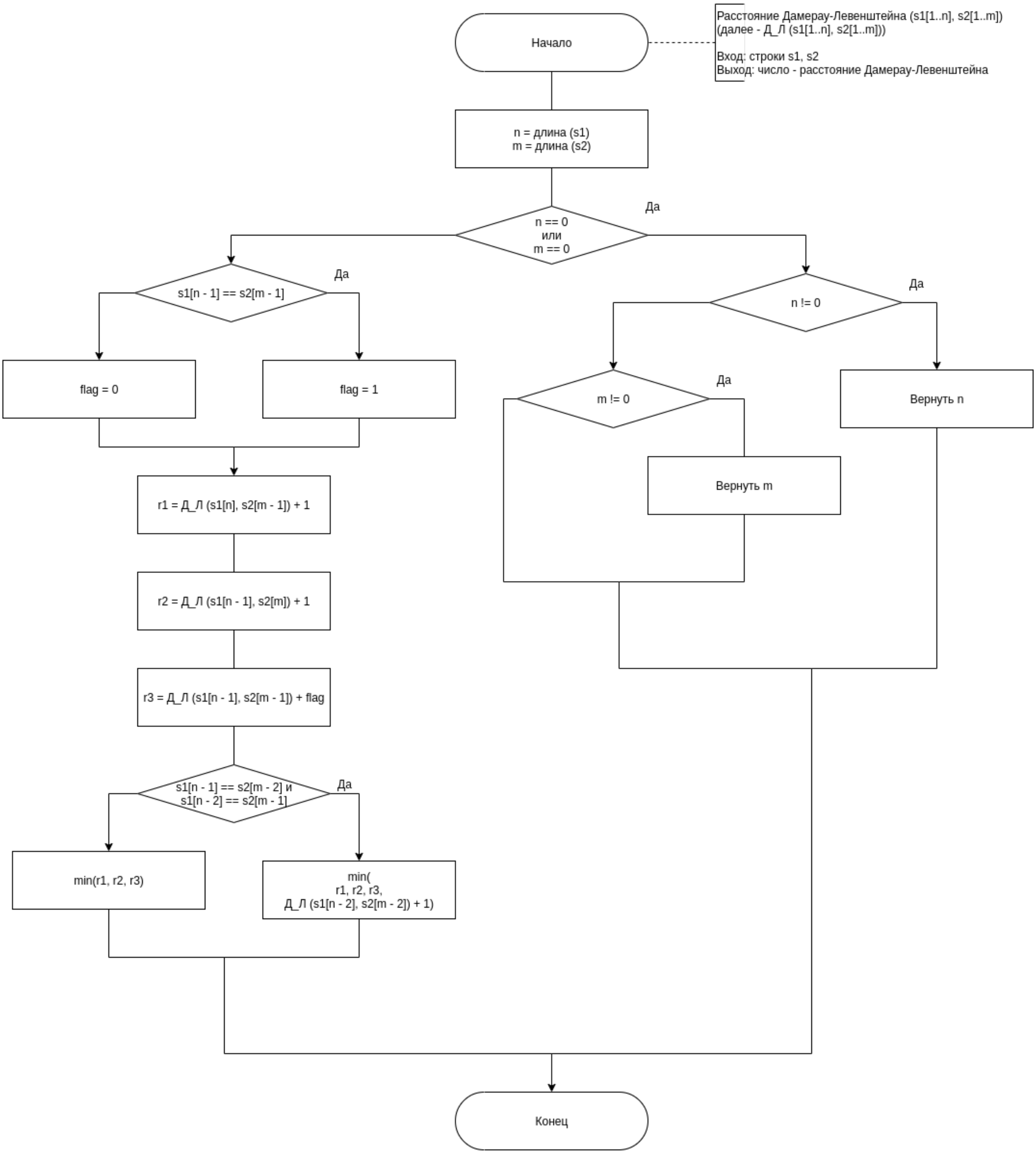

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

2.4 Классы эквивалентности тестирования

Для 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, дают разные результаты.

2.5 Использование памяти

С точки зрения замеров и сравнения используемой памяти, алгоритмы Левенштейна и Дамерау-Левенштейна не отличаются друг от друга. Тогда рассмотрим только рекурсивную и матричную реализации данных алгоритмов.

Пусть:

- l1 - длина строки S1;
- l2 - длина строки S2;

Тогда затраты по памяти будут такими:

- алгоритм нахождения расстояния Левенштейна (рекурсивный), где для каждого вызова:
  - для S1, S2 - (l1 + l2) * sizeof(char);
  - для l1, l2 - 2 * sizeof(int);
  - доп. переменные - 2 * sizeof(int);
  - адрес возврата.

- алгоритм нахождения расстояния Левенштейна с использованием кеша в виде матрицы (память на саму матрицу: ((l1 + 1) * (l2 + 1)) * sizeof(int)) (рекурсивный), где для каждого вызова:
  - для S1, S2 - (l1 + l2) * sizeof(char);
  - для l1, l2 - 2 * sizeof(int);
  - доп. переменные - 2 * sizeof(int);
  - ссылка на матрицу - 8 байт;
  - адрес возврата.
- алгоритм нахождения расстояния Дамерау-Левенштейна (рекурсивный), где для каждого вызова:
  - для S1, S2 - (l1 + l2) * sizeof(char);
  - для l1, l2 - 2 * sizeof(int);
  - доп. переменные - 2 * sizeof(int);
  - адрес возврата.
- алгоритм нахождения расстояния Левенштейна (матричный):
  - для матрицы - ((l1 + 1) * (l2 + 1)) * sizeof(int)) ;
  - текущая строка матрицы - (l1 + 1) * sizeof(int);
  - для S1, S2 - (l1 + l2) * sizeof(char);
  - для l1, l2 - 2 * sizeof(int);
  - доп. переменные - 3 * sizeof(int);
  - адрес возврата.

2.6 Вывод

В данном разделе были представлено описание используемых типов данных, а также схемы алгоритмов, рассматриваемых в лабораторной работе.

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*[2]. В текущем лабораторном задании необходимо определить процессорное время, затрачиваемое на выполнение программы, а также построить графики. Все необходимые инструменты для этого уже имеются в выбранном языке программирования.

Время работы было замерено с помощью функции *process_time(...)* из библиотеки *time*[3].

3.2 Реализация алгоритмов

В листингах 3.1-3.4 представлены реализации алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Алгоритм нахождения расстояния Левенштейна (рекурсивный)

```
def levenstein_recursive(str1, str2):
l1 = len(str1)
l2 = len(str2)

if ((l1 == 0) or (l2 == 0)):
return abs(l1 - l2)

flag = 0

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

min_distance = min(levenstein_recursive(str1[:-1], str2) +
    1,
levenstein_recursive(str1, str2[:-1]) + 1,
levenstein_recursive(str1[:-1], str2[:-1]) + flag)

return min_distance
```

Листинг 3.2 – Алгоритм нахождения расстояния Левенштейна (матричный)

```
def levenstein_matrix(str1, str2):
l1 = len(str1)
l2 = len(str2)
matrix = create_lev_matrix(l1 + 1, l2 + 1)

for i in range(1, l1 + 1):
for j in range(1, l2 + 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)

return matrix[l1][l2]
```

Листинг 3.3 – Алгоритм нахождения расстояния Левенштейна с использованием кеша в виде матрицы

```
def recursive_for_levenstein_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]

add = recursive_for_levenstein_cache(str1, str2, n - 1, m, matrix) + 1
delete = recursive_for_levenstein_cache(str1, str2, n, m - 1, matrix) + 1
change = recursive_for_levenstein_cache(str1, str2, n - 1, m - 1, matrix)

if (str1[n - 1] != str2[m - 1]):
change += 1 # flag

matrix[n][m] = min(add, delete, change)

return matrix[n][m]

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

matrix = create_lev_matrix(n + 1, m + 1)

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

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

```

```
return matrix[n][m]
```

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

```
def damerau_levenstein_recursive(str1, str2):
l1 = len(str1)
l2 = len(str2)

if ((l1 == 0) or (l2 == 0)):
if (l1 != 0):
return l1

if (l2 != 0):
return l2

return 0

flag = 0 if (str1[-1] == str2[-1]) else 1

add = damerau_levenstein_recursive(str1[:l1 - 1], str2) + 1
delete = damerau_levenstein_recursive(str1, str2[:l2 - 1])
    + 1
change = damerau_levenstein_recursive(str1[:l1 - 1],
    str2[:l2 - 1]) + flag
extra_change = damerau_levenstein_recursive(str1[:l1 - 2],
    str2[:l2 - 2]) + 1

if ((l1 > 1) and (l2 > 1) and (str1[-1] == str2[-2]) and
    (str1[-2] == str2[-1])):
minimum  = min(add, delete, change, extra_change)

else:
minimum = min(add, delete, change)

return minimum
```

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"	текст	4	4
3	слово	"пустая строка"	5	5
4	слово	лово	1	1
5	символ	вол	3	3
6	буква	цифра	4	4
7	нос	лицо	4	4
8	кот	кит	1	1
9	танк	танкист	3	3
10	я	рябина	5	5
11	крот	корт	2	1
12	глина	лгиан	4	2

3.4 Вывод

Были представлены всех алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна, которые были описаны в предыдущем разделе. Также в данном разделе была приведена информации о выбранных средствах для разработки алгоритмов.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программа, 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: Windows 10 Pro;
- память: 32 ГБ;
- процессор: 12th Gen Intel(R) Core(TM) i5-12400 2.50 ГГц [4].

Во время тестирования компьютер 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) Рекурсивный алгоритм Левенштейна
 2) Алгоритм Левенштейна с использованием матрицы
 3) Рекурсивный алгоритм Левенштейна с кешем
 4) Рекурсивный алгоритм Дамерау-Левенштейна
 5) Сравнение времени
 0) Выход

 Введите номер пункта: 2

Введите 1-ую строку:     скат
Введите 2-ую строку:     скот

Матрица:

0  0  с  к  а  т
0  0  1  2  3  4
с  1  0  1  2  3
к  2  1  0  1  2
о  3  2  1  1  2
т  4  3  2  2  1

Результат:  1
```

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения алгоритмов

Как было сказано выше, используется функция замера процессорного времени process_time(...) из библиотеки time на Python. Функция возвращает пользовательское процессорное время типа float.

Использовать функцию приходится дважды, из времени завершения необходимо вычесть время начала работы функции, 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.(рек)	Л.(матр.)	Л.(рек с матр.)	Д.-Л.(рек.)
0	0.0023	0.0038	0.0075	0.0028
1	0.0074	0.0110	0.0122	0.0088
2	0.0362	0.0122	0.0125	0.0295
3	0.0535	0.0104	0.0132	0.0575
4	0.5422	0.0136	0.0210	0.9562
5	0.9125	0.0175	0.0292	1.5250
6	1.3875	0.0228	0.0388	2.7580
7	12.3438	0.0354	0.0544	21.4985
8	69.3750	0.0548	0.0618	126.4228
9	378.2812	0.0604	0.0834	723.4395

Где Л – реализация алгоритма Левенштейна, а Д.-Л. – реализация алгоритма Дамерау-Левенштейна.

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

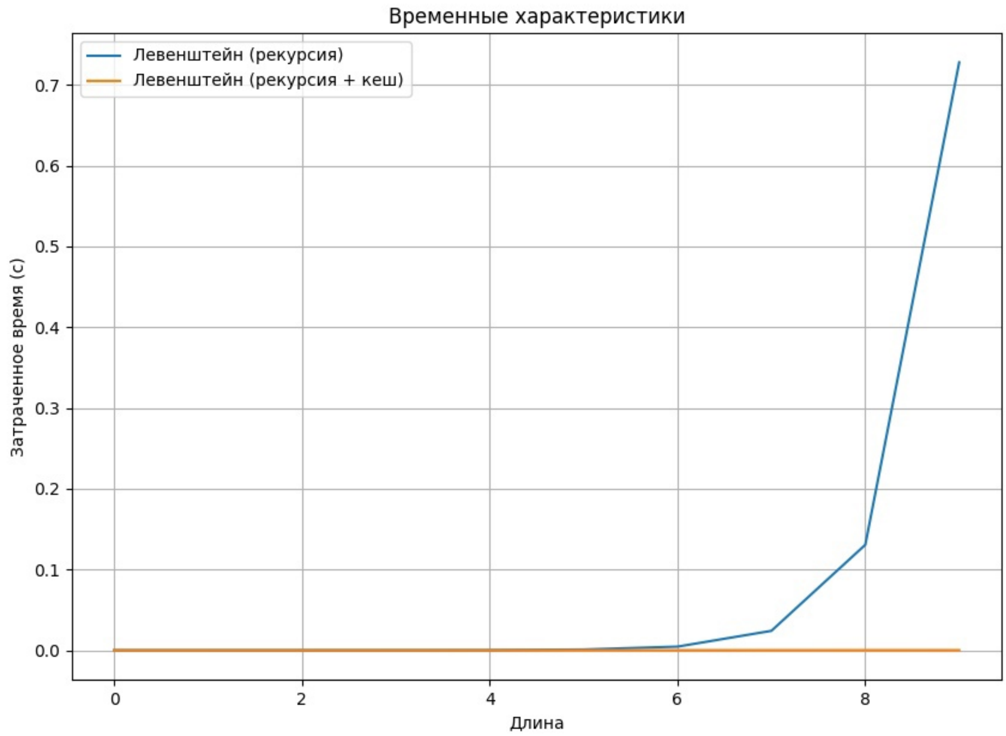

Рисунок 4.2 – Сравнение по времени алгоритмов Левенштейна с использованием рекурсии и с использованием кеша (матрица + рекурсия)

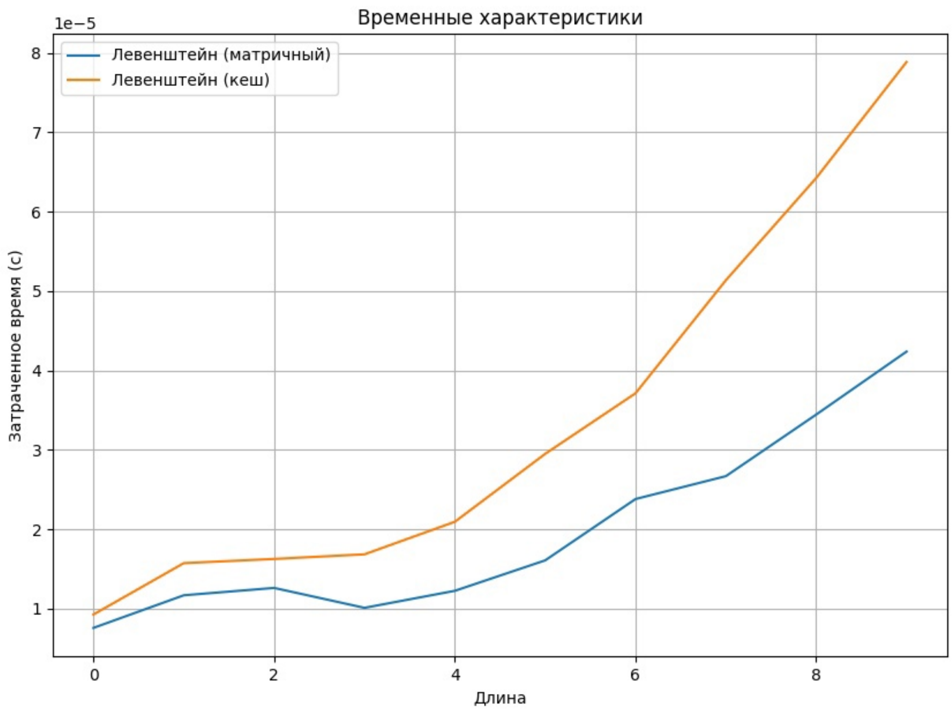

Рисунок 4.3 – Сравнение алгоритмов нахождения расстояния Левенштейна матричного и с кешем в виде матрицы

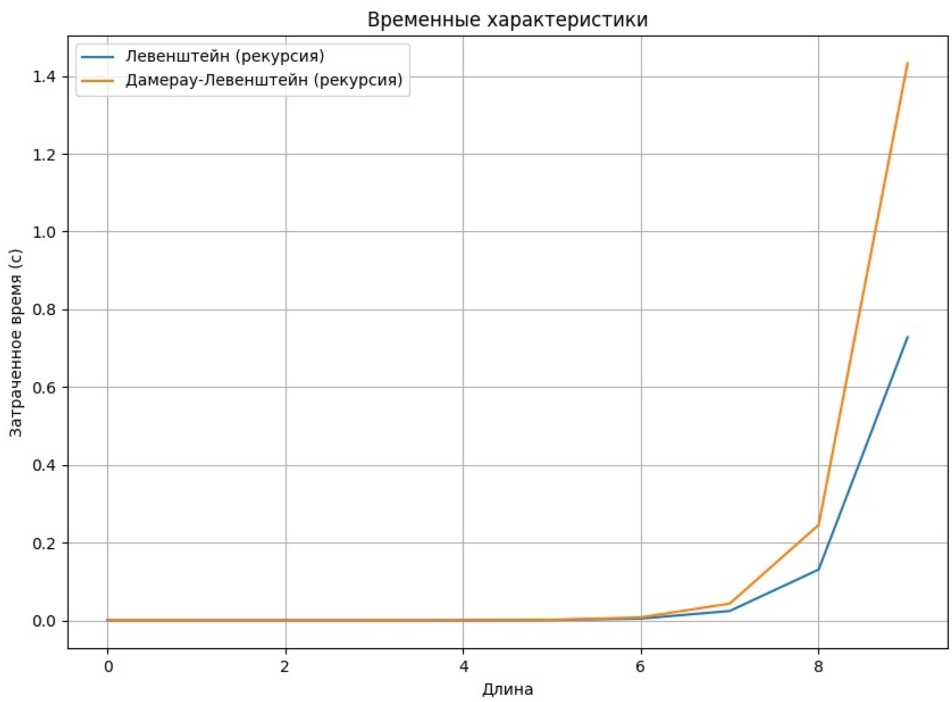

Рисунок 4.4 – Сравнение по времени рекурсивных алгоритмов Левенштейна и Дамерау-Левенштейна

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 эксперимента было получено, что при длине строк в более 5 символов, алгоритм Левенштейна быстрее Дамерау-Левенштейна в 2 раза. В итоге, можно сказать, что при таких данных следует использовать алгоритм Левенштейна.

Также при проведении эксперимента было выявлено, что на длине строк в 4 символа рекурсивная реализация алгоритма Левенштейна в уже в 14 раз медленнее матричной реализации алгоритма. При увеличении длины строк в геометрической прогрессии растет и время работы рекурсивной реализации. Следовательно, стоит использовать матричную реализацию для строк длиной более 4 символов.

Заключение

По результатам исследования было установлено, что при увеличении длины строк для алгоритмов Левенштейна и Дамерау-Левенштейна время выполнения растет в геометрической прогрессии. Рекомендуется отдавать предпочтение алгоритму Левенштейна, так как он в два раза быстрее, при длине строк более шести символов. Однако, наилучшие показатели по времени достигаются при использовании матричной реализации алгоритма Левенштейна и рекурсивной реализации с кешем, которые обеспечивают 12-кратное превосходство по времени работы уже при длине строки в 4 символа, за счет сохранения промежуточных вычислений. В то же время следует учитывать, что матричные реализации требуют значительного объема 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 Левенштейна;
- проведен сравнительный анализ алгоритмов Левенштейна и Дамерау-Левенштейна, а также сравнение рекурсивной и матричной реализаций, матричной реализации и реализаций с кешом алгоритма Левенштейна;
- подготовлен отчет о лабораторной работе.

Список источников

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

[2] Welcome to Python [Электронный ресурс]. https://www.python.org.

[3] time — Time access and conversions [Электронный ресурс]. https://docs.python.org/3/library/time.html#functions.

[4] Intel(R) Core(TM) i5-12400 [Электронный ресурс]. https://www.intel.com/content/www/us/en/support/ru-banner-inside.html?wapkw=12400f.