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 Расстояния Левенштейна и Дамерау-Левенштейна

Студент Косарев А.А.

Группа ИУ7-51Б

Оценка (баллы)

Преподаватель Волкова Л. Л.

Москва — 2022 г.

Содержание

Введение

В программировании работа со строками занимает отдельное важное место. Обработка строк необходима для решения огромного количества различных задач, например:

- исправление ошибок в тексте при рукописном вводе;
- поиск слова или строки в тексте;
- сравнение текстовых файлов.

Но одной из основных задач является сравнение строк. Для осуществления этого процесса были разработаны специальные алгоритмы, которые и будут рассмотрены в данной работе.

1 Аналитическая часть

1.1 Цель и задачи

Цель – изучение метода динамического программирования на основании определения редакционных расстояний по алгоритмам Левенштейна и Дамерау-Левенштейна.

Для достижения поставленной цели следует решить следующие задачи:

- изучить расстояния Левенштейна и Дамерау-Левенштейна;
- разработать алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна;
- реализовать разработанные алгоритмы;
- выполнить оценку затрат алгоритмов по памяти;
- выполнить замеры процессорного времени работы реализаций алгоритмов;
- провести сравнительный анализ нерекурсивных алгоритмов для поиска расстояний Левенштейна и Дамерау-Левенштейна;
- провести сравнительный анализ трех алгоритмов поиска расстояний Дамерау-Левенштейна.

1.2 Итерационный алгоритм нахождения расстояния Левенштейна

Расстояние Левенштейна - минимальное количество редакционных мероприятий, необходимых для преобразования одной строки в другую.

Пусть S_1 и S_2 – две строки, длиной N и M соответственно, а редакционные операции:

- вставка символа в произвольной позиции (I - Insert);
- удаление символа в произвольной позиции (D - Delete);
- замена символа на другой (R - Replace).

Принято, что для этих операций "штраф" равен 1.

Для поиска расстояния Левенштейна используют рекуррентную формулу, то есть такую, которая использует предыдущие члены ряда или последовательности для вычисления последующих:

$$D(i,j) = \begin{cases} 0, & \text{i} = 0, \text{j} = 0 \\ i, & \text{j} = 0, \text{i} > 0 \\ j, & \text{i} = 0, \text{j} > 0 \\ \min\{ & \\ \quad D(i,j-1)+1, & \\ \quad D(i-1,j)+1, & \text{i} > 0, \text{j} > 0 \\ \quad D(i-1,j-1) + \begin{cases} 0, & \text{если } S_i = S_j, \\ 1, & \text{иначе} \end{cases} & \\ \}, & \end{cases} \tag{1.1}$$

Итерационный алгоритм поиска расстояния Левенштейна будет выполнять расчёт по формуле (1.1).

1.3 Итерационный алгоритм нахождения расстояния Дамерау-Левенштейна

Поскольку одна из самых частых ошибок при наборе текстов на естественном языке – это перестановка двух соседних символов местами, Дамерау предложил включить в число редакционных операций операцию перестановки двух соседних символов со "штрафом" 1.

Тогда для данного алгоритма определены следующие редакционные операции:

- вставка символа в произвольной позиции (I - Insert);
- удаление символа в произвольной позиции (D - Delete);
- замена символа на другой (R - Replace);
- транспозиция двух символов (M - Match).

Таким образом, чтобы получить формулу нахождения расстояния Дамерау-Левенштейна, необходимо в формулу для поиска расстояния Левенштейна добавить еще одно определение минимума, но уже из четырех вариантов, если возможна перестановка двух соседних символов:

$$
D(i,j) = \begin{cases}
0, & \text{i} = 0,\ \text{j} = 0, \\
i, & \text{j} = 0,\ \text{i} > 0, \\
j, & \text{i} = 0,\ \text{j} > 0, \\
\min\{ & \\
\quad D(i,j-1)+1, & \text{i} > 0,\ \text{j} > 0, \\
\quad D(i-1,j)+1, & S_i = S_{j-1}, \\
\quad D(i-2,j-2)+1, & S_{i-1} = S_j, \\
\quad D(i-1,j-1) + \begin{cases} 0, & \text{если } S_i = S_j, \\ 1, & \text{иначе} \end{cases} & \\
\}, & \\
\min\{ & \\
\quad D(i,j-1)+1, & \\
\quad D(i-1,j)+1, & \text{иначе} \\
\quad D(i-1,j-1) + \begin{cases} 0, & \text{если } S_i = S_j, \\ 1, & \text{иначе} \end{cases} & \\
\} &
\end{cases}
\tag{1.2}
$$

Итерационный алгоритм поиска расстояния Дамерау-Левенштейна будет выполнять расчёт по формуле (1.2).

1.4 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна решает ту же задачу, что и его итерационная реализация. Отличие лишь в том, что в данном случае не используется матрица, так как все значения необходимые для расчета последующих, вычисляются рекурсивно. Но в связи с этим такая реализация не отличается быстродействием.

1.5 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с использованием кеша

Данный алгоритм является оптимизацией рекурсивной реализации. Суть оптимизации заключается в использовнии кеша, который представляет собой матрицу.

При выполнении рекурсии выполняется заполнение матрицы значениями редакционных расстояний для подстрок заданной длины. Если данные, для которых выполняется расчет на очередном шаге рекурсии, еще не обрабатывались (значение ячейки матрицы равно -1), то результат будет занесен в матрицу. Если же это расстояние уже было найдено, то вычисление не выполняется, а алгоритм переходит к следующему шагу рекурсии.

Вывод

В данном разделе были теоретически разобраны алгоритмы Левенштейна и Дамерау-Левенштейна. Эти алгоритмы позволяют найти редакционное расстояние для двух строк.

2 Конструкторская часть

В данном разделе будут представлены схемы алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна, выбранные типы данных и теоретическая оценка затрат по памяти.

2.1 Алгоритмы поиска редакционных расстояний

Схемы алгоритмов поиска редакционных расстояний представлены на рисунках 2.1-2.5.

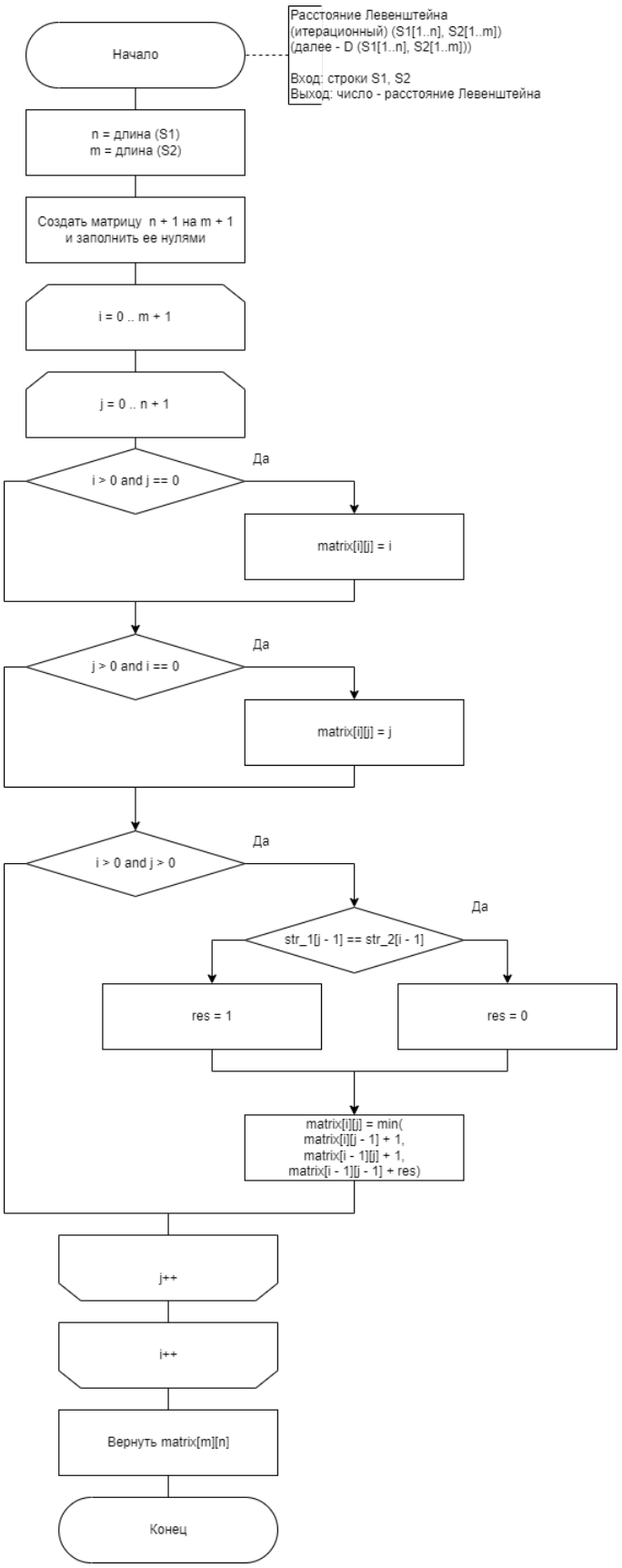

Рисунок 2.1 – Схема итерационного алгоритма нахождения расстояния Левенштейна

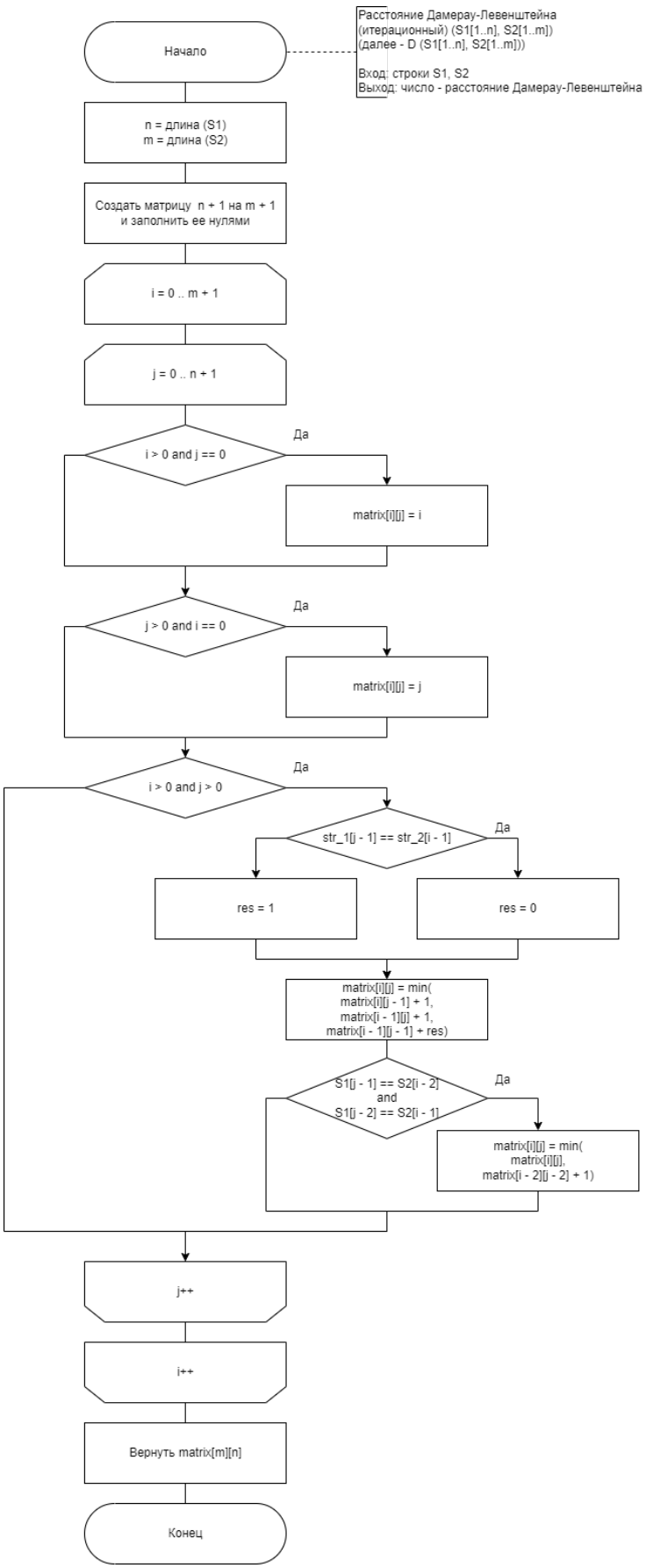

Рисунок 2.2 – Схема итерационного алгоритма нахождения расстояния Дамерау-Левенштейна

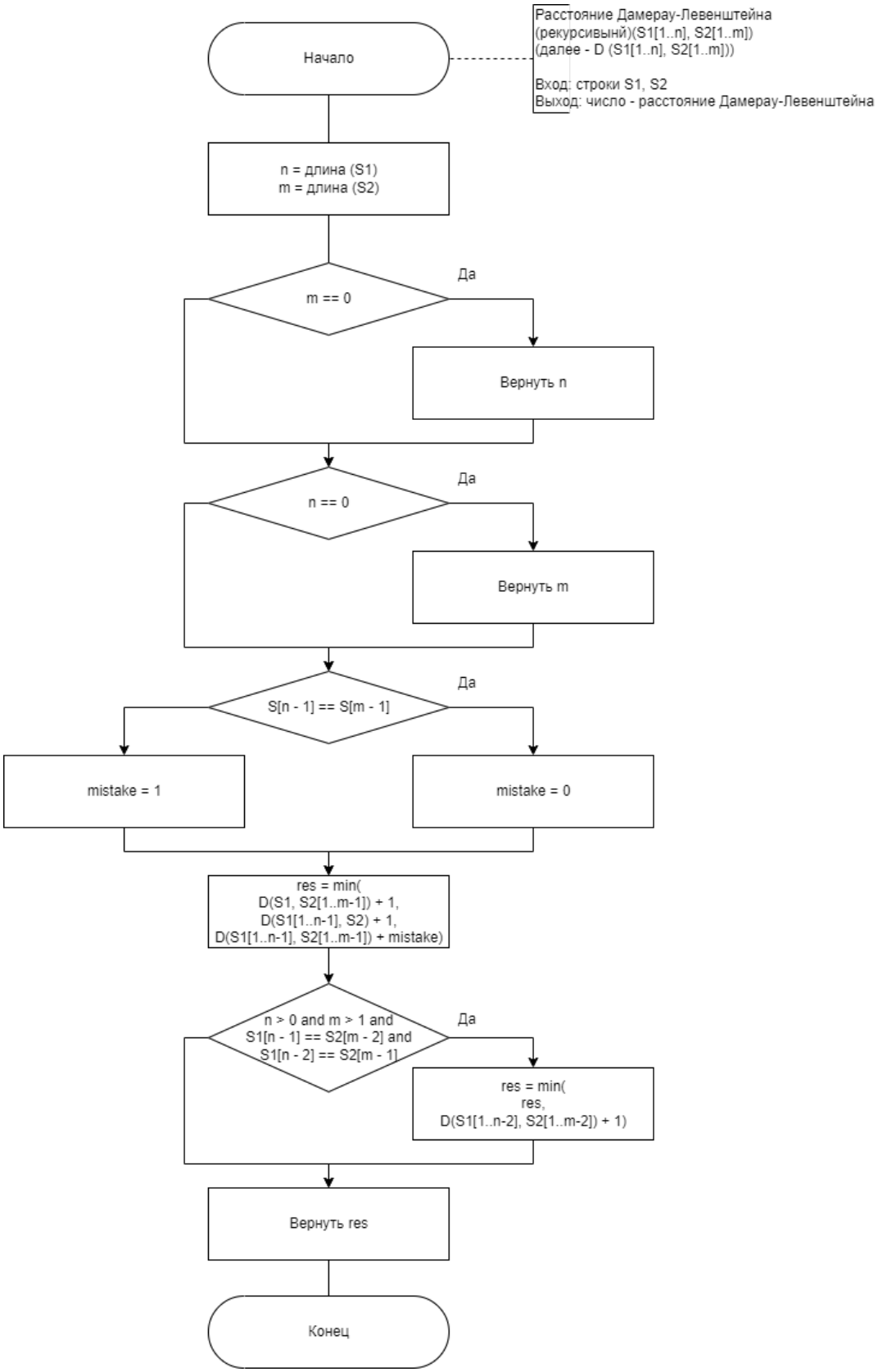

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

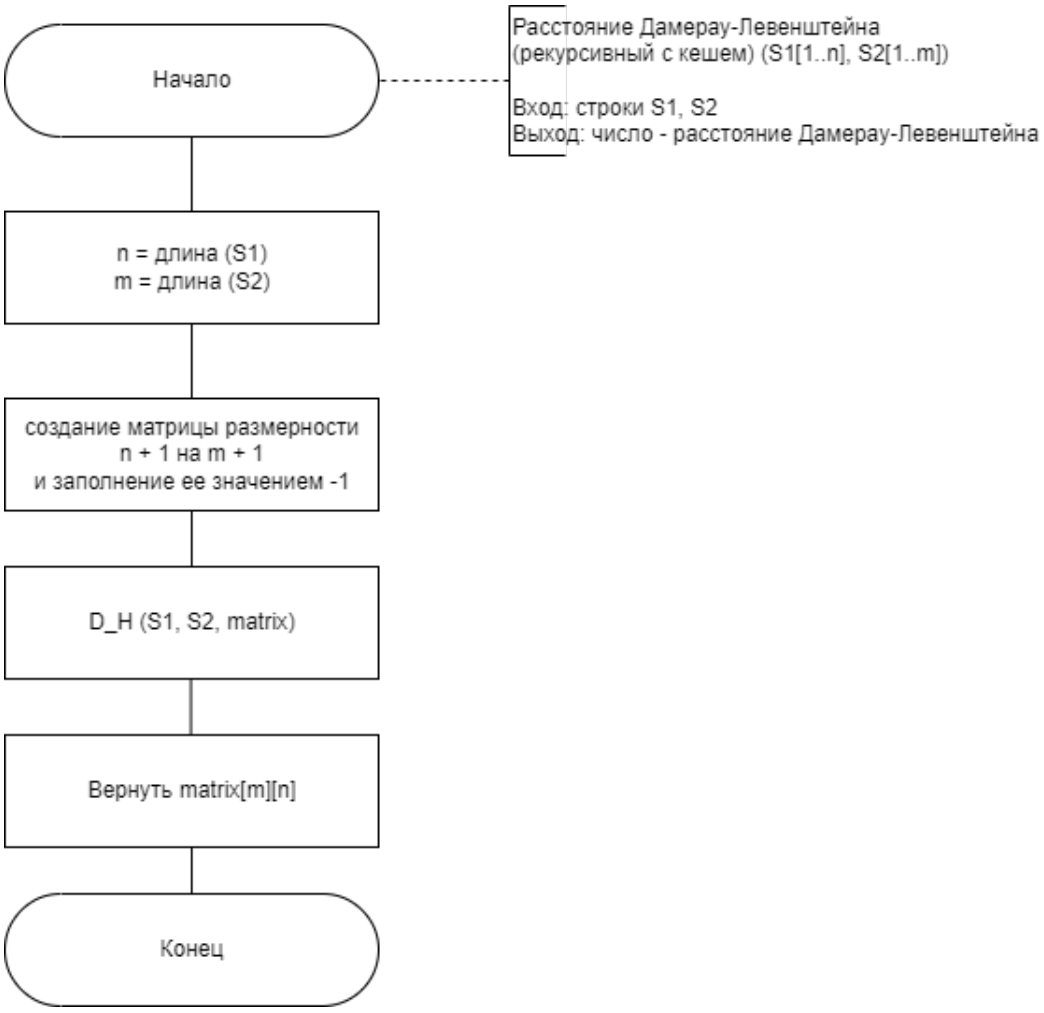

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кешем (создание матрицы)

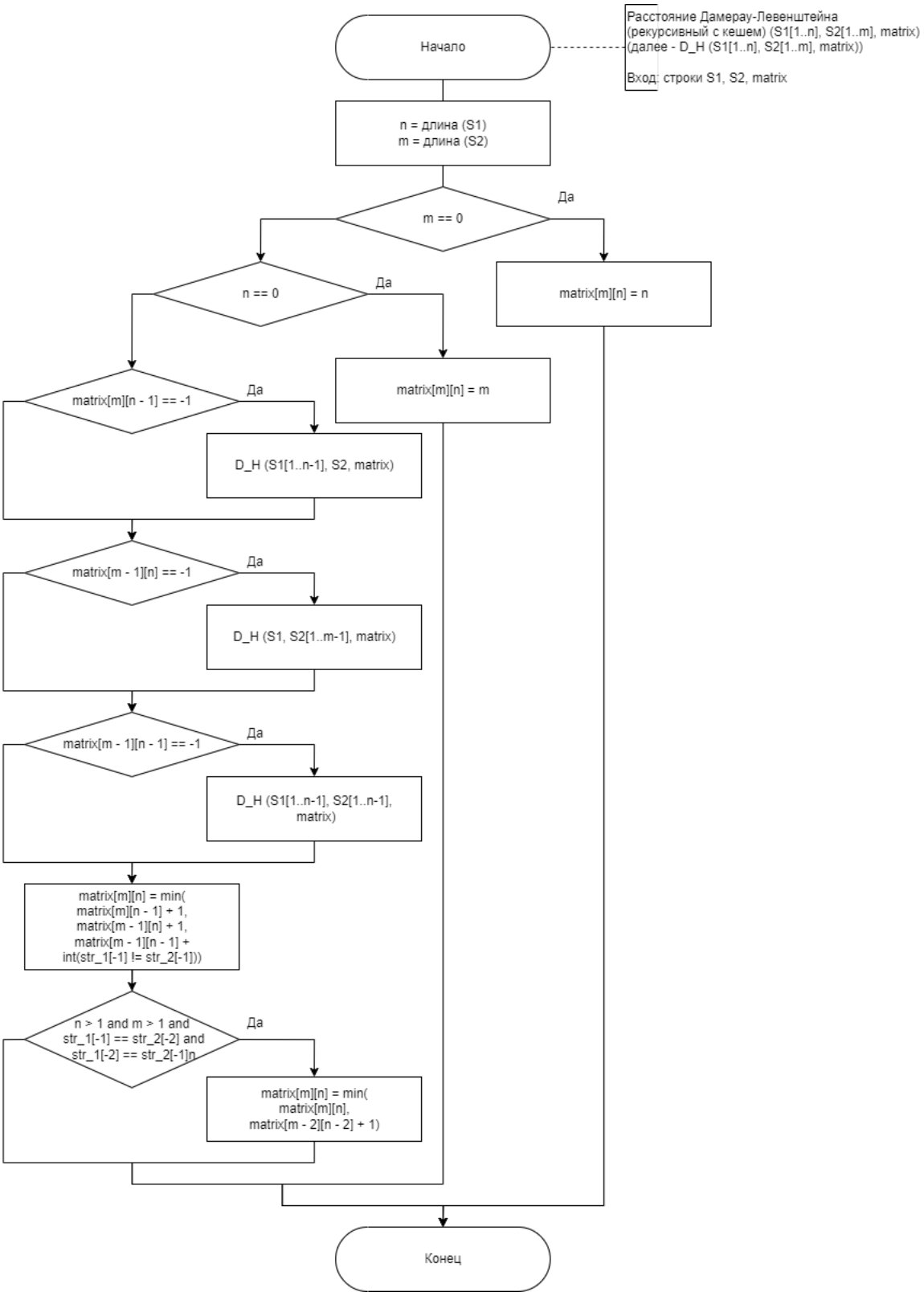

Рисунок 2.5 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кешем

2.2 Описание типов данных

Выбранные типы данных:

- две строки - тип *str*;
- длина строки - тип *int*;
- матрица (для итерационного представления алгоритма Левенштейна и Дамерау-Левенштейна и для рекурсивной реализации с кешем) - двумерный массив типа *int*.

2.3 Оценка затрат алгоритмов по памяти

Затраты по памяти для алгоритма поиска расстояния Левенштейна (итерационный):

- матрица - (n + 1) * (m + 1) * sizeof(int);
- строки str_1, str_2 - (n + m + 2) * sizeof(char);
- длины строк n, m - 2 * sizeof(int);
- дополнительные переменные (i, j, res) - 3 * sizeof(int);
- адрес возврата.

Итого:
(n + 1) * (m + 1) * sizeof(int) + (n + m + 2) * sizeof(char) + 5 * sizeof(int)

Затраты по памяти для алгоритма поиска расстояния Дамерау-Левенштейна (итерационный):

- матрица - (n + 1) * (m + 1) * sizeof(int);
- строки str_1, str_2 - (n + m + 2) * sizeof(char);
- длины строк n, m - 2 * sizeof(int);

- дополнительные переменные (i, j, res) - 3 * sizeof(int);
- адрес возврата.

Итого:
(n + 1) * (m + 1) * sizeof(int) + (n + m + 2) * sizeof(char) + 5 * sizeof(int)

Затраты по памяти для алгоритма поиска расстояния Дамерау-Левенштейна (рекурсивный), для одного вызова:

- строки str_1, str_2 - (n + m + 2) * sizeof(char);
- длины строк n, m - 2 * sizeof(int);
- дополнительные переменные (mistake) - sizeof(int);
- адрес возврата.

Итого (K - количество вызовов рекурсии):
((n + m + 2) * sizeof(char) + 3 * sizeof(int)) * K

Затраты по памяти для алгоритма поиска расстояния Дамерау-Левенштейна (рекурсивный с кешем), для одного вызова:

- строки str_1, str_2 - (n + m + 2) * sizeof(char);
- длины строк n, m - 2 * sizeof(int);
- адрес возврата;

и матрица - (n + 1) * (m + 1) * sizeof(int).

Итого (K - количество вызовов рекурсии):
((n + m + 2) * sizeof(char) + 2 * sizeof(int)) * K + (n + 1) * (m + 1) * sizeof(int)

2.4 Исходные файлы программы

Программа состоит из следующих модулей:

- *main.py* – файл с главной функцией, вызывающей меню программы;
- *algorithms.py* – файл, содержащий код алгоритмов поиска редакционных расстояний и функцию вывода полученной матрицы;
- *proc_time.py* – файл с функциями подсчета процессорного времени алгоритмов;
- *tests.py* – файл с тестирующей функцией.

Вывод

В этом разделе были представлены схемы алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна, выбранные типы данных и теоретическая оценка затрат по памяти.

Проведя анализ оценки алгоритмов по памяти, можно сказать, что рекурсивные алгоритмы менее затратны, так как для них максимальный размер памяти растет прямо пропорционально сумме длин строк, а в итерационных – пропорционально их произведению.

3 Технологическая часть

В данном разделе будут рассмотрены средства реализации задания, представлены листинги алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна, а также тестовые данные, которые использовались для проверки корректности работы алгоритмов.

3.1 Выбор языка программирования и среды разработки

Для реализации алгоритмов был выбран язык программирования *Python*, а в качестве среды разработки - *PyCharm*. Для замеров процессорного времени использовалась функция *process_time()* [3] из библиотеки *time*, а для построения графиков - библиотека *matplotlib* [4].

3.2 Реализация алгоритмов

Ниже представлены реализации алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна (листинги 3.1–3.4).

Листинг 3.1 – Итерационный алгоритм поиска расстояния Левенштейна

```
1  def levenstein(str_1, str_2):
2      n, m = len(str_1), len(str_2)
3      matrix = [[0] * (n + 1) for _ in range(m + 1)]
4
5      for i in range(m + 1):
6          for j in range(n + 1):
7              if j == 0 and i > 0:
8                  matrix[i][j] = i
9              if i == 0 and j > 0:
10                 matrix[i][j] = j
11             if i > 0 and j > 0:
12                 if str_1[j - 1] == str_2[i - 1]:
13                     res = 0
14                 else:
```

```
15                      res = 1
16                  matrix[i][j] = min(
17                      matrix[i][j - 1] + 1,
18                      matrix[i - 1][j] + 1,
19                      matrix[i - 1][j - 1] + res)
20
21      return matrix[m][n]
```

Листинг 3.2 – Итерационный алгоритм поиска расстояния Дамерау-Левенштейна

```
1  def damerau_levenstein(str_1, str_2):
2      n, m = len(str_1), len(str_2)
3      matrix = [[0] * (n + 1) for _ in range(m + 1)]
4
5      for i in range(m + 1):
6          for j in range(n + 1):
7              if i == 0 and j == 0:
8                  matrix[i][j] = 0
9              if j == 0 and i > 0:
10                 matrix[i][j] = i
11             if i == 0 and j > 0:
12                 matrix[i][j] = j
13             if i > 0 and j > 0:
14                 if str_1[j - 1] == str_2[i - 1]:
15                     res = 0
16                 else:
17                     res = 1
18                 matrix[i][j] = min(
19                     matrix[i][j - 1] + 1,
20                     matrix[i - 1][j] + 1,
21                     matrix[i - 1][j - 1] + res)
22
23                 if str_1[j - 1] == str_2[i - 2] and str_1[j - 2] ==
                       str_2[i - 1]:
24                     matrix[i][j] = min(
25                         matrix[i][j],
26                         matrix[i - 2][j - 2] + 1)
27
28     return matrix[m][n]
```

Листинг 3.3 – Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна

```
def damerau_levenstein_recursive(str_1, str_2):
    n, m = len(str_1), len(str_2)

    if m == 0:
        return n
    if n == 0:
        return m

    if str_1[-1] == str_2[-1]:
        mistake = 0
    else:
        mistake = 1

    res = min(damerau_levenstein_recursive(str_1, str_2[:-1]) + 1,
              damerau_levenstein_recursive(str_1[:-1], str_2) + 1,
              damerau_levenstein_recursive(str_1[:-1], str_2[:-1])
                + mistake)
    if n > 1 and m > 1 and str_1[-1] == str_2[-2] and str_1[-2] ==
       str_2[-1]:
        res = min(res, damerau_levenstein_recursive(str_1[:-2],
           str_2[:-2]) + 1)
    return res
```

Листинг 3.4 – Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кешем

```
def damerau_levenstein_recursive_cache(str_1, str_2, matrix):
    n, m = len(str_1), len(str_2)

    if m == 0:
        matrix[m][n] = n
    elif n == 0:
        matrix[m][n] = m
    else:
        if matrix[m][n - 1] == -1:
            damerau_levenstein_recursive_cache(str_1[:-1], str_2,
               matrix)
        if matrix[m - 1][n] == -1:
```

```
12              damerau_levenstein_recursive_cache(str_1, str_2[:-1],
                    matrix)
13          if matrix[m - 1][n - 1] == -1:
14              damerau_levenstein_recursive_cache(str_1[:-1],
                    str_2[:-1], matrix)
15
16          matrix[m][n] = min(matrix[m][n - 1] + 1,
17                             matrix[m - 1][n] + 1,
18                             matrix[m - 1][n - 1] + int(str_1[-1] !=
                                   str_2[-1]))
19
20          if n > 1 and m > 1 and str_1[-1] == str_2[-2] and str_1[-2]
              == str_2[-1]:
21              matrix[m][n] = min(matrix[m][n],
22                                 matrix[m - 2][n - 2] + 1)
23
24          return
25
26
27  def damerau_levenstein_recursive_cache_matrix(str_1, str_2):
28      n, m = len(str_1), len(str_2)
29
30      matrix = [[-1] * (n + 1) for _ in range(m + 1)]
31      damerau_levenstein_recursive_cache(str_1, str_2, matrix)
32
33      if info:
34          print_matrix(str_1, str_2, matrix)
35      return matrix[m][n]
```

3.3 Тестирование

Классы эквивалентности:

- обе строки пустые;
- одна строка пустая, другая - нет;
- одинаковые строки;

- строки, дающие один и тот же результат и для алгоритма Левенштейна, и для алгоритмов Дамерау-Левенштейна;
- строки, дающие разные результаты для алгоритма Левенштейна и для алгоритмов Дамерау-Левенштейна.

В таблице 3.1 представлены модульные тесты. Все тесты пройдены успешно.

Таблица 3.1 – Модульные тесты

	Входные данные		Ожидаемый результат	
№	str_1	str_2	Левенштейн	Дамерау-Левенштейн
1	"Пустая строка"	"Пустая строка"	0	0
2	"Пустая строка"	hugo	4	4
3	applejack	"Пустая строка"	9	9
4	house	house	0	0
5	car	cars	1	1
6	decode	decoding	3	3
7	class	clsas	2	1
8	kingtoys	ikdfotpq	7	6
9	unique	nueqeu	5	3
10	qwertyuiop	wqreytiupo	6	5
11	sweetbloody	croatlbiawe	10	9

Вывод

В этом разделе были рассмотрены средства реализации задания, представлены листинги алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна, а также тесты.

4 Исследовательская часть

В данном разделе будут представлены примеры работы программы, проведены замеры процессорного времени и предоставлена информация о технических характеристиках устройства.

4.1 Примеры работы программы

На рисунках 4.1-4.2 представлен результат работы программы. В каждом примере пользователем введены две строки и получены результаты вычислений редакционных расстояний.

```
Введите первое слово: cat
Введите второе слово: coal

 ================================================================================
Алгоритм Левенштейна
  |    C  A  T
--------------
  | 0  1  2  3
C | 1  0  1  2
O | 2  1  1  2
A | 3  2  1  2
L | 4  3  2  2
Ответ: 2

 ================================================================================
Алгоритм Дамерау-Левенштейна (нерекурсивный)
  |    C  A  T
--------------
  | 0  1  2  3
C | 1  0  1  2
O | 2  1  1  2
A | 3  2  1  2
L | 4  3  2  2
Ответ: 2

 ================================================================================
Алгоритм Дамерау-Левенштейна (рекурсивный)
Ответ:  2

 ================================================================================
Алгоритм Дамерау-Левенштейна (рекурсивный с кешем)
  |    C  A  T
--------------
  | 0  1  2  3
C | 1  0  1  2
O | 2  1  1  2
A | 3  2  1  2
L | 4  3  2  2
Ответ: 2
```

Рисунок 4.1 – Пример работы программы (1)

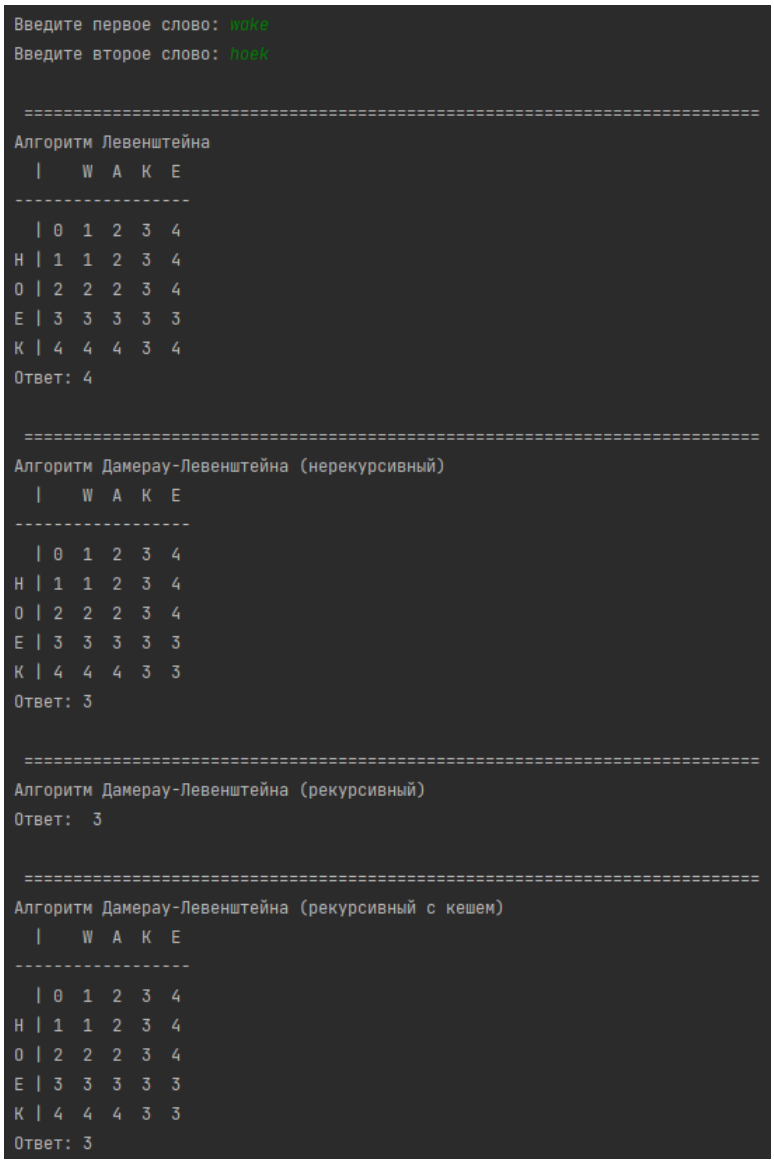

Рисунок 4.2 – Пример работы программы (2)

4.2 Время выполнения алгоритмов

Для замера процессорного времени использовалась функция *process_ time()* библиотеки *time*. Возвращаемый результат - время в миллисекундах, число типа *float*. Чтобы получить достаточно точное значение, производилось усреднение времени. Количество запусков замера процессорного времени:

- для итерационного алгоритма поиска Левенштейна - 10000;
- для итерационного алгоритма поиска Дамерау-Левенштейна - 10000;
- для рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна - 50;
- для рекурсивного алгоритма (с кешем) поиска расстояния Дамерау-Левенштейна - 10000.

В замерах использовались строки длиной от 0 до 10 символов.

В таблице 4.1 представлено процессорное время работы реализаций алгоритмов поиска редакционного расстояния.

Таблица 4.1 – Результаты замеров времени

Длина	Лев.	Д.-Л.(итер.)	Д.-Л.(рек.)	Д.-Л.(рек. с кешем)
0	0.000002	0.000002	0.000000	0.000000
1	0.000003	0.000002	0.000000	0.000003
2	0.000002	0.000003	0.000000	0.000005
3	0.000005	0.000008	0.000000	0.000008
4	0.000009	0.000009	0.000313	0.000014
5	0.000013	0.000016	0.000625	0.000022
6	0.000017	0.000019	0.003125	0.000028
7	0.000022	0.000025	0.017500	0.000037
8	0.000028	0.000033	0.095000	0.000050
9	0.000034	0.000041	0.523438	0.000061
10	0.000041	0.000048	2.912500	0.000078

На рисунках 4.3-4.5 также приведены результаты замеров процессорного времени.

При данном масштабе на рисунке 4.4 графики времени работы итерационной и рекурсивной реализации с кешем алгоритма Дамерау-Левенштейна совпадают.

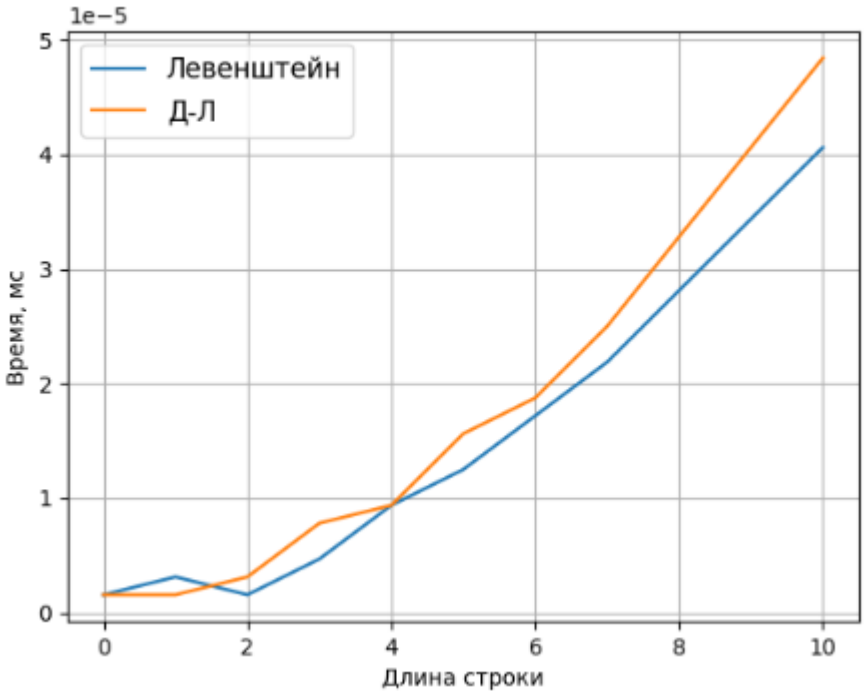

Рисунок 4.3 – Сравнение итерационных алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна

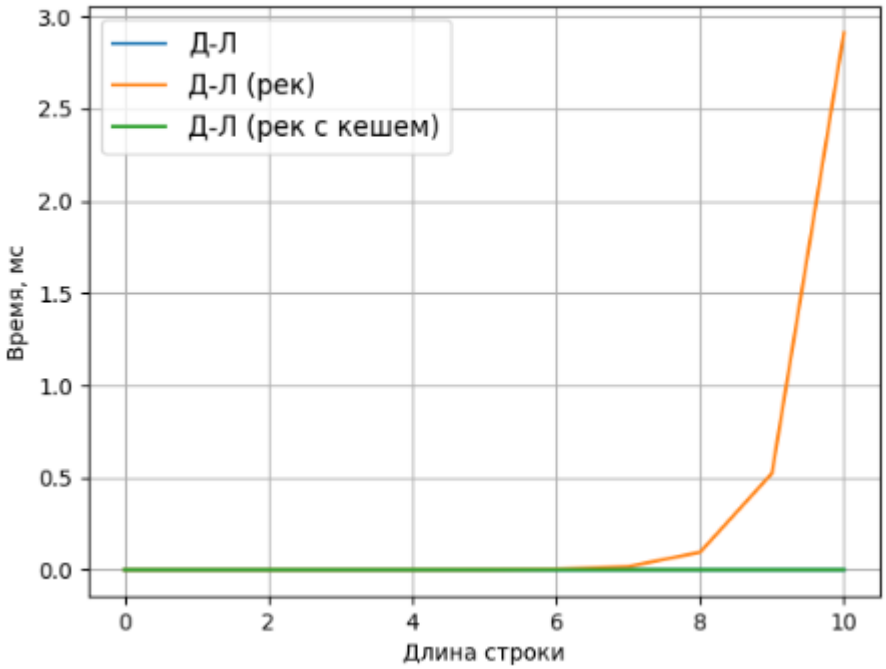

Рисунок 4.4 – Сравнение 3 алгоритмов поиска расстояний Дамерау-Левенштейна

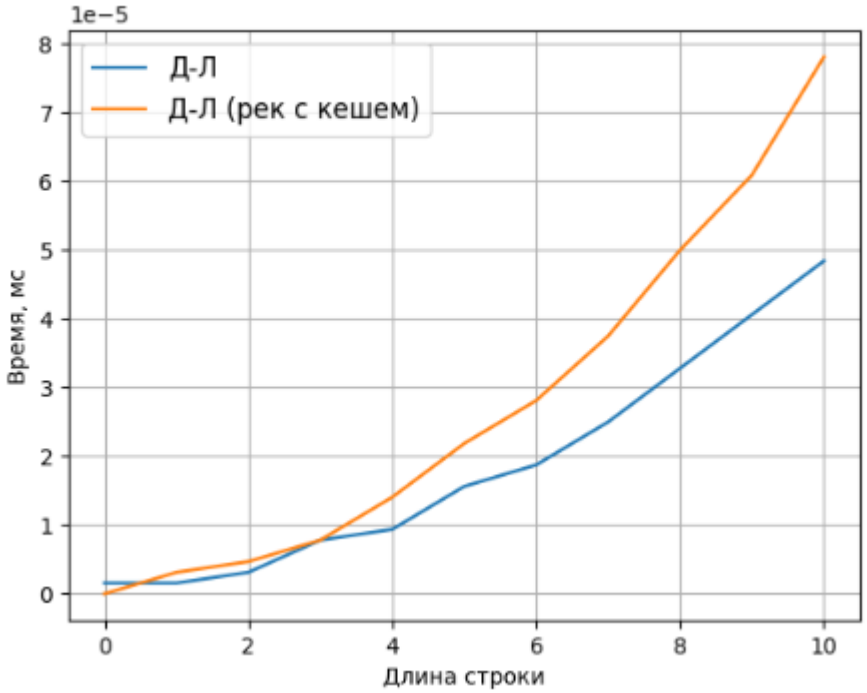

Рисунок 4.5 – Сравнение итерационного алгоритма поиска расстояний Дамерау-Левенштейна и рекурсивного с кешем

4.3 Технические характеристики устройства

Ниже представлены характеристики компьютера, на котором проводилось тестирование программы:

- операционная система Windows 10 Домашняя 21H2;
- оперативная память 16 Гб;
- процессор Intel(R) Core(TM) i7-10870H CPU @ 2.20 ГГц.

Во время тестирования ноутбук был подключен к сети электропитания. Из программного обеспечения были запущены только среда разработки *PyCharm* и браузер *Chrome*.

Загруженность компонентов:

- процессор - 20%;
- оперативная память - 49%.

4.4 Вывод

В результате замеров процессорного времени выделены следущие аспекты:

- итерационные алгоритмы Левенштейна и Дамерау-Левенштейна работают за сходное время;
- рекурсивный алгоритм Дамерау-Левенштейна уже при длине строк равной 4 символа проигрывает в 26 раз по времени итерационной и рекурсивной с кешем реализациям;
- итерационный алгоритм поиска расстояний Дамерау-Левенштейна в среднем на 42% - 54% быстрее рекурсивного с кешем для длин строк от 0 до 10 символов;

Заключение

Цель, которая была поставлена в начале лабораторной работы, была достигнута: изучен метод динамического программирования на основании определения редакционных расстояний по алгоритмам Левенштейна и Дамерау-Левенштейна.

Решены все поставленные задачи:

- изучены, разработаны и реализованы алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна;
- выполнена оценка затрат алгоритмов по памяти;
- выполнены замеры алгоритмов процессорного времени работы реализаций алгоритмов;
- проведен сравнительный анализ нерекурсивных алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна;
- проведен сравнительный анализ трех алгоритмов поиска расстояний Дамерау-Левенштейна.

В результате проведенных экспериментов было определено, что при увеличении длины строк время работы изученных алгоритмов увеличивается в геометрической прогрессии. Самой медленной реализацией оказалась рекурсивная реализация алгоритма поиска расстояния Дамерау-Левенштейна. Из оставшихся трех вариантов наиболее быстрым является итерационный алгоритм Левенштейна. Но стоит отметить, что матричная реализация алгоритма Дамерау-Левенштейна отрабатывает примерно за такое же время.

Несмотря на то, что итерационные алгоритмы обладают высоким быстродействием, при больших длинах строк они занимают довольно много памяти под матрицу.

Список использованной литературы

[1] Ингерсолл, Г. С. Обработка неструктурированных текстов / Г. С. Ингерсолл, Т. С. Мортон, Э. Л. Фэррис. - ЛитРес, 2022

[2] Погорелов, Д. А. Сравнительный анализ алгоритмов редакционного расстояния Левенштейна и Дамерау-Левенштейна / Д. А. Погорелов, А. М. Таразанов. - Синергия Наук. – 2019. - URL: https://elibrary.ru/item.asp?id=36907767 (дата обращения: 28.09.2022)

[3] time — Time access and conversions [Электронный ресурс]. - URL: https://docs.python.org/3/library/time.html#time.process_time (дата обращения: 28.09.2022)

[4] Matplotlib documentation [Электронный ресурс]. - URL: https://matplotlib.org/stable/index.html (дата обращения: 28.09.2022)

[5] Windows 10 Home [Электронный ресурс]. - URL: https://www.microsoft.com/en-us/d/windows-10-home/d76qx4bznwk4?activetab=pivot:overviewtab (дата обращения: 01.10.2022)

[6] Intel(R) Core(TM) i7-10870H CPU @ 2.20GHz [Электронный ресурс]. - URL: https://ark.intel.com/content/www/ru/ru/ark/products/208018/intel-core-i710870h-processor-16m-cache-up-to-5-00-ghz.html (дата обращения: 01.10.2022)