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» (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.01 Информатика и вычислительная техника**

О Т Ч Е Т

по лабораторной работе № 1

Название: Расстояние Левенштейна и Дамерау-Левенштейна

Дисциплина: Анализ алгоритмов

Студент ИУ7-52Б _____ Сучков А.Д.

(Группа) (Подпись, дата) (И.О. Фамилия)

Преподаватель _____ Волкова Л.Л.

(Подпись, дата) (И.О. Фамилия)

Москва, 2020

Оглавление

Введение

Расстояние Левенштейна - это минимальное количество редакторских операций, которые необходимы для превращения одной строки в другую [1]. Впервые задачу поставил в 1925 году советский математик Владимир Левенштейн при изучении последовательностей 0 - 1, впоследствии более общую задачу для произвольного алфавита связали с его именем.

Существуют следующие редакторские операции:

- вставка символа;
- удаление символа;
- замена символа.

Расстояние Дамерау-Левенштейна включает также операцию перестановки 2 символов (или транспозицию).

Расстояние Левенштейна и его обобщения активно применяются для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста ил и речи), а также, к примеру, в биоинформатике для сравнения генов, хромосом и белков.

1. Аналитическая часть

Цель данной лабораторной работы заключается в реализации и последующем сравнении алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

Для начала можно выделить следующие задачи лабораторной работы:

- математически описать расстояние Левенштейна и Домерау-Левенштейна;
- описать и реализовать алгоритмы поиска расстояний;
- замерить процессорное время работы алгоритмов при различных размерах строк;
- оценить наибольшую затрачиваемую память для каждого из алгоритмов;
- провести сравнительный анализ алгоритмов на основании проведённых экспериментов;

Поиск расстояний несёт в себе задачу нахождения такой последовательности операций, применение которых даст в результате минимальный суммарный штраф.

Существуют следующие штрафы:

- вставка (I, от англ. insert) - 1;
- замена (R, от англ. replace) - 1;
- удаление (D, от англ. delete) - 1;
- совпадение (M, от англ. match) - 0;
- транспозиция (T, от англ. transposition) - 1.

Данная проблема решается использованием рекуррентной формулы вычисления расстояний. Пусть D(S1[1..i], S2[1..j]) - расстояние Левенштейна для подстроки S1 и S2 с длинами i и j соответственно.

Тогда, формула для вычисления D имеет следующий вид:

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

Аналогично и для Дамерау-Левенштейна:

$$D(i,j) = \begin{cases} j, \text{если i} = 0 \\ i, \text{если j} = 0 \\ min(D(S1[1..i], S2[1..j-1]) + 1, \\ D(S1[1..i-1], S2[1..j]) + 1, \\ D(S1[1..i-1], S2[1..j-1] + \begin{cases} 0, \text{если S1[i]} = \text{S2[j]} \\ 1, \text{иначе} \end{cases}, \\ \begin{cases} D(S1[1..i-2], S2[1..j-2]) + 1, \text{ если} \begin{cases} i > 1, j > 1 \\ S1[i] = S2[j-1] \\ S1[i-1] = S2[j] \end{cases} \\ +\infty, \text{иначе} \end{cases} \end{cases})$$

2. Конструкторская часть

Рассмотрим алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна для строк S1 и S2 с длинами l1 и l2 соответственно.

2.1. Расстояние Левенштейна, рекурсивный метод

В методе используется рекурсивная формула нахождения D(S1[1..i], S2[1..j]), реализовать это можно с помощью рекурсивной функции. Функция будет принимать в качестве входных данных строки S1 и S2, а также их длины i и j соответственно. Метод основан на последующем вызове той же функции для тех же строк, но для длин (i-1, j-1), (i-1, j), (i, j-1), где возвращается минимальное из этих значений. Схема на рис. 2.1.

2.2. Расстояние Левенштейна, матричный метод

Данный матричный метод основан на использовании рекуррентной формулы. В начале работы создаётся целочисленная матрица с размерами (l1 + 1) на (l2 + 1), затем заполняются первый столбец и первая строка, которые являются базой для рекуррентной формулы. Матрица заполняется построчно, в каждой ячейке [i][j] матрицы записывается значение D(S1[1..i-1], S2[1..j-1]). В том случае, когда i=1 и j=1 будет значить, что строки пусты. Итоговым результатом является значение в нижней правой ячейке матрицы, т.е. в ячейке с индексом [l1 + 1][l2 + 1]. Схема на рис. 2.2.

2.3. Расстояние Левенштейна, рекурсивный метод с заполнением матрицы

В данном методе создаётся матрица размерами (l1 + 1)x(l2 + 1), все ячеки которой изначально заполнены значением бесконечности. В каждой клетке [i][j] этой матрицы будет записано значение D(s1[1..i-1], s2[1..j-1]).

Рекурсивная функция получает матрицу, индексы i, j положения в ней и две строки. Алгоритм начинает свою работу с ячейки [1][1], которая заполняется значением 0. Из положения [i][j] рассматривается переход в соседние ячейки [i + 1][j + 1], [i + 1][j], [i][j + 1]. В случае, если соседняя ячейка расположена в пределах матрицы и расстояние R при переходе из данной ячейки меньше ныне хранимого в ней значения, то значение соседней ячейки меня-

ется на R, после чего функция запускается уже для соседней ячейки. После завершения работы всех функций, расстояние Левенштейна расположено в ячейке [l1 + 1][l2 + 1]. Схема на рис. 2.3.

2.4. Расстояние Дамерау-Левенштейна, матричный метод

Метод является модифицированным методом подсчёта расстояния Левенштейна матричным методом. В матрице для ячейки [i][j], где i $>$ 2 и j $>$ 2, учитывается также вариант перехода из клетки [i-2][j-2], в случае когда S1[i] = S2[j-1] и S1[i-1] = S[j]. Результатом всё также будет правое нижнее значение ячейки [l1 + 1][l2 + 1]. Схема на рис 2.4.

2.5. Требования к программе

Для дальнейшего тестирования программы необходимо обеспечить консольный ввод двух строк, а также обеспечить выбор алгоритма поиска. На выходе должны получить редакционное расстояние и, если использовался один из матричных методов, программа должна выводить саму матрицу подсчёта.

Также необходимо реализовать функцию подсчёта процессорного времени, которое могут затрачивать функции поиска.

2.6. Тесты

Для детального тестирования программы, можно выделить несколько случаев поступаемых данных:

- равные (одинаковые) строки;
- одна из строк пуста;
- две строки пусты;
- проверка транспозиции (для Дамерау-Левенштейна).

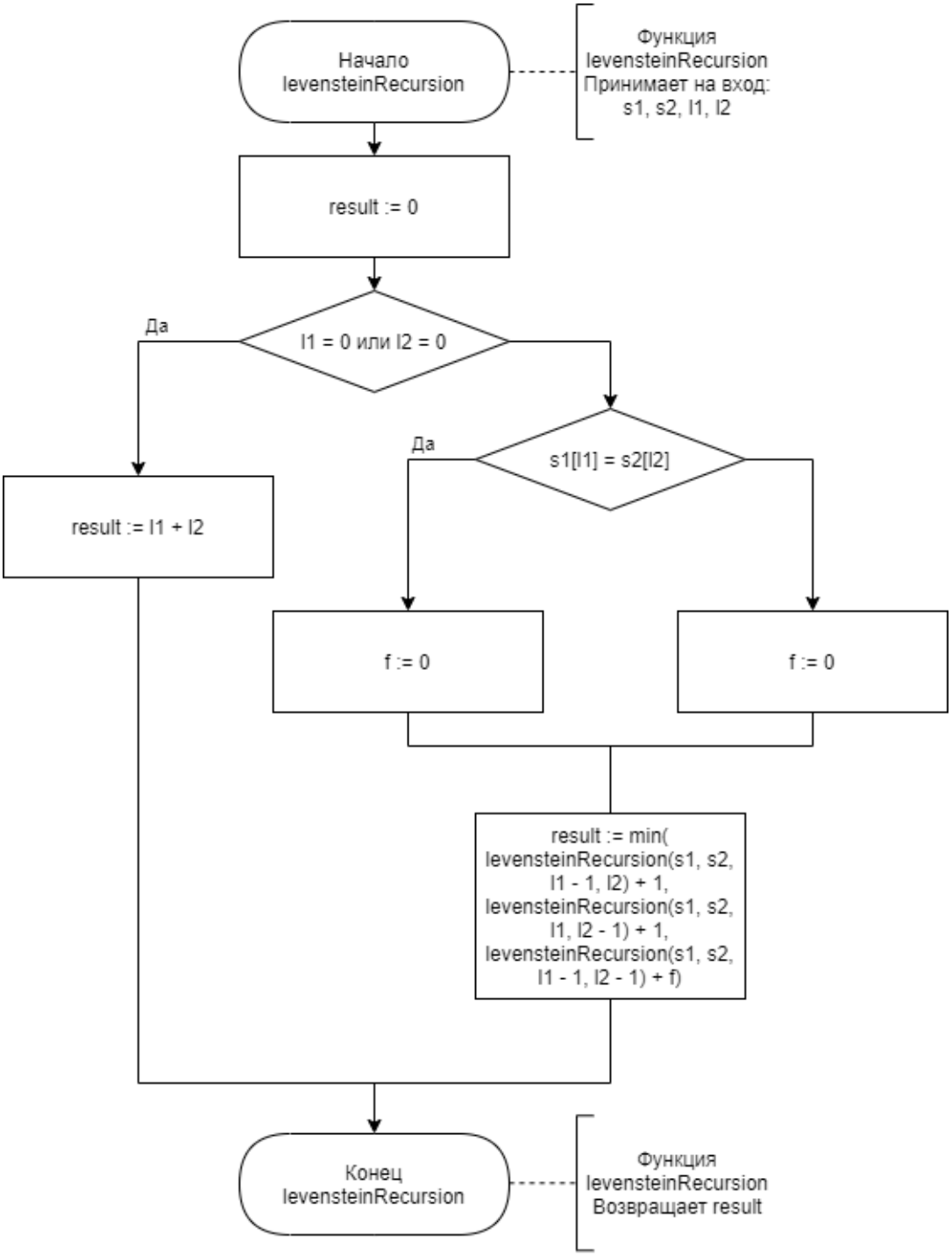

Рис. 2.1: Расстояние Левенштейна, рекурсивный метод

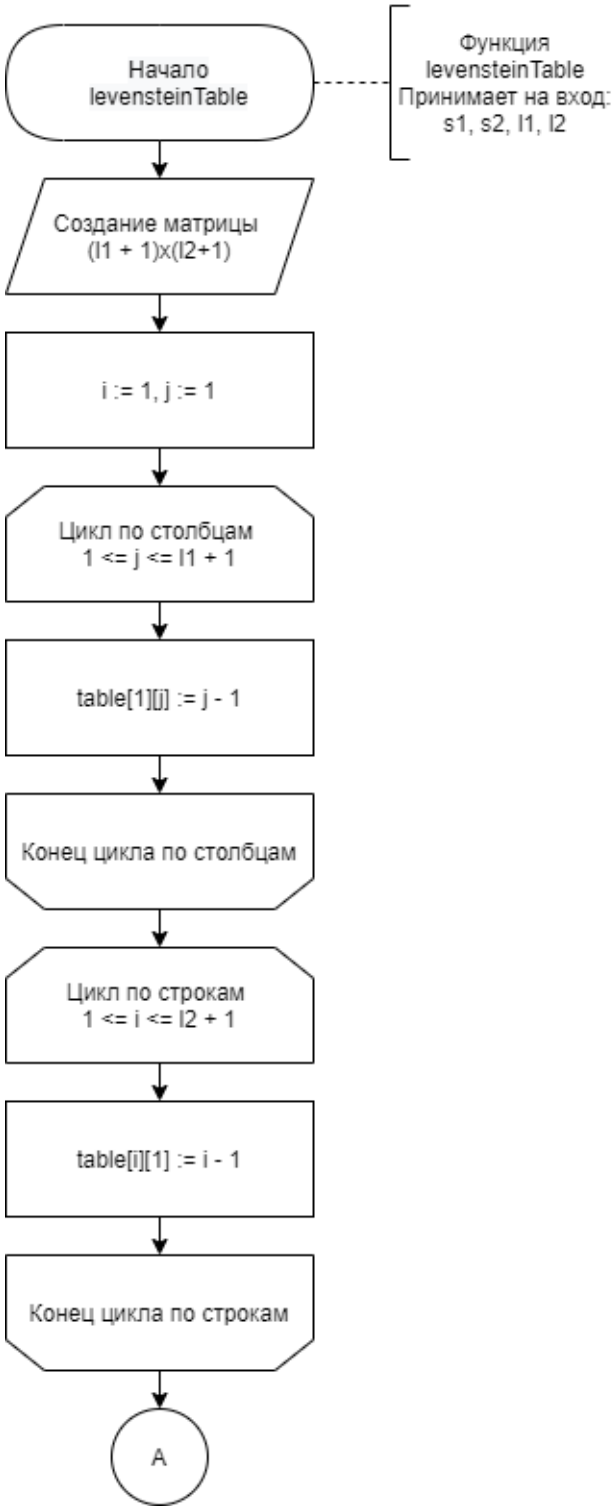

Рис. 2.2: Расстояние Левенштейна, матричный метод, часть 1

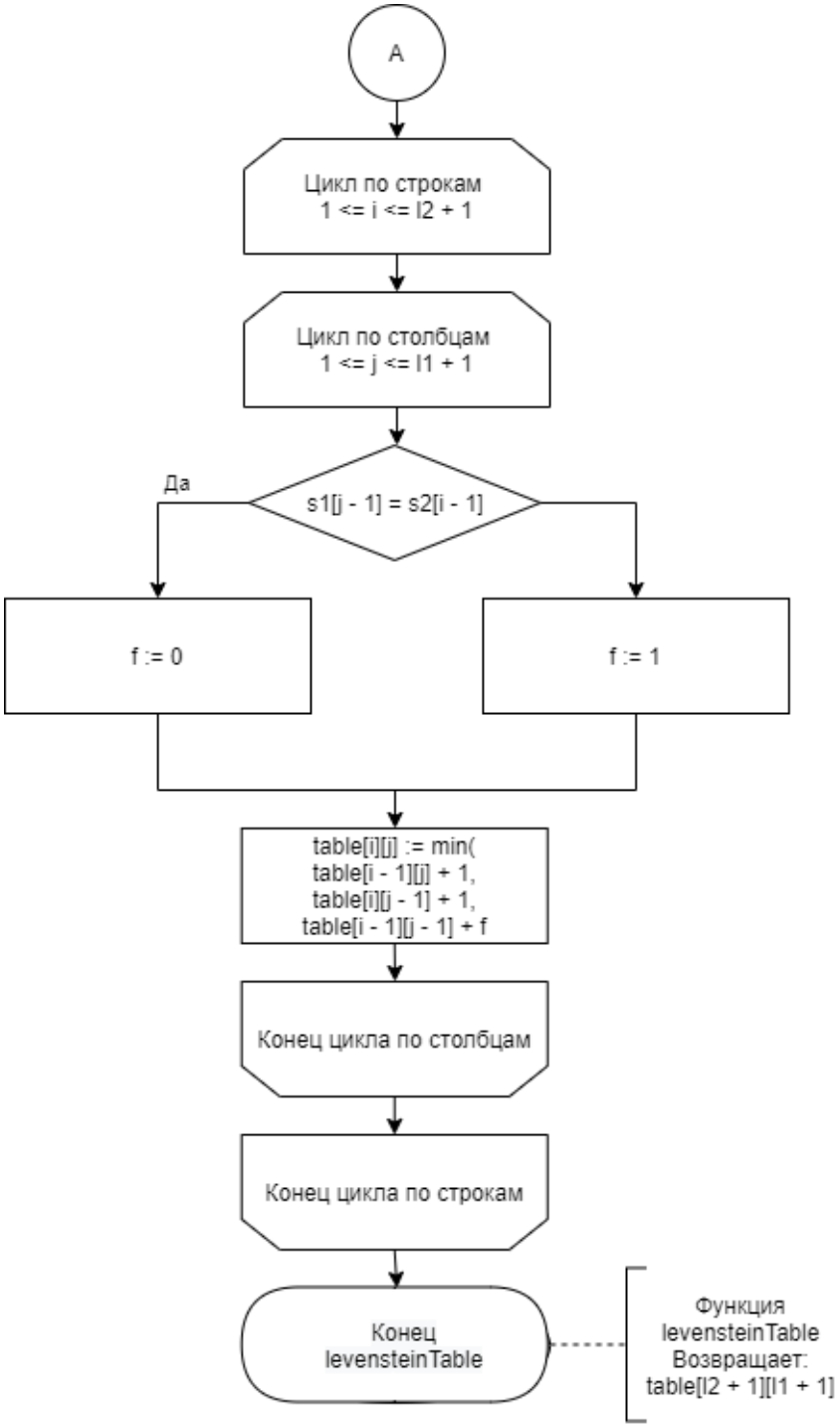

Рис. 2.3: Расстояние Левенштейна, матричный метод, часть 2

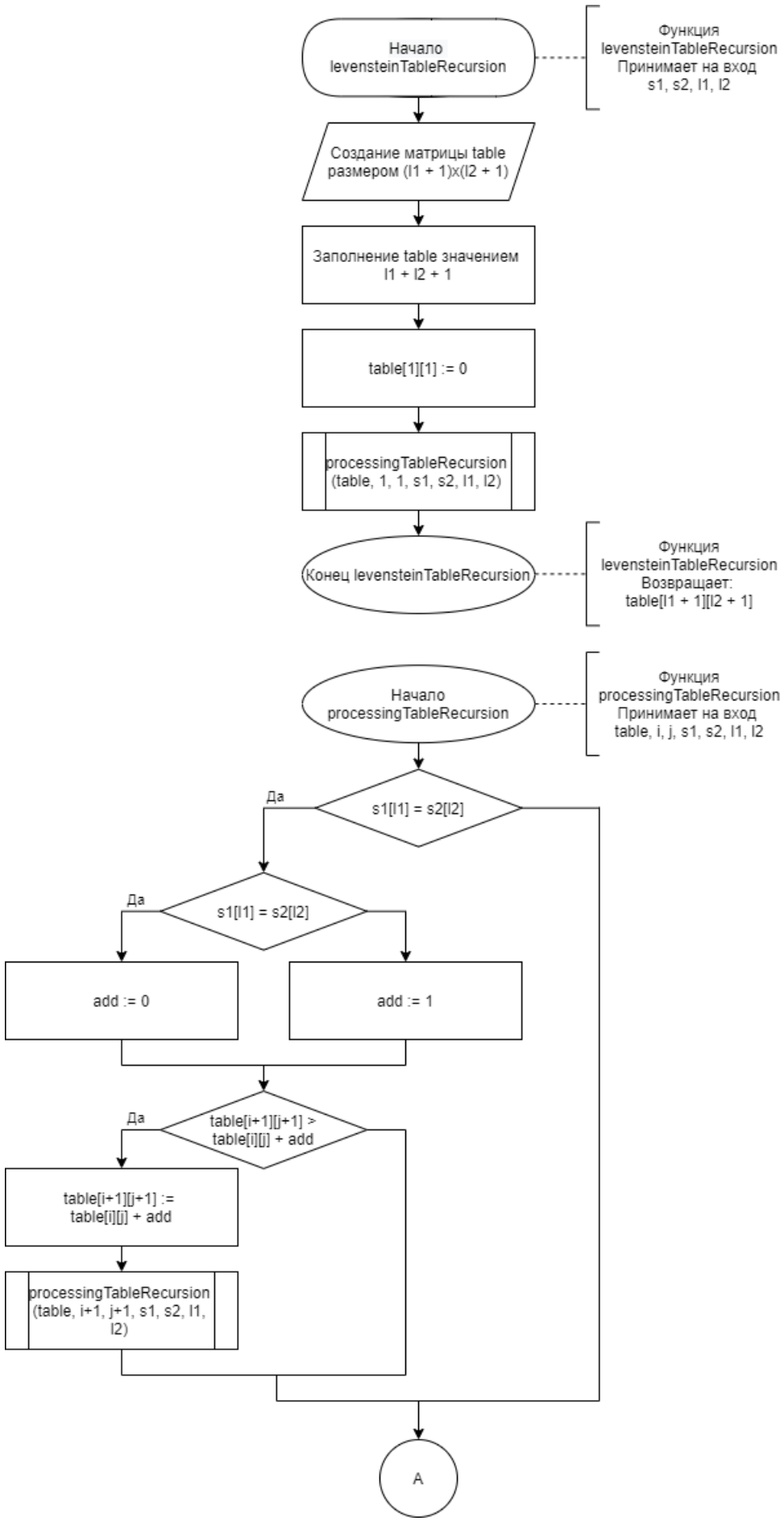

Рис. 2.4: Расстояние Левенштейна, рекурсивный метод с заполнением матрицы, часть 1

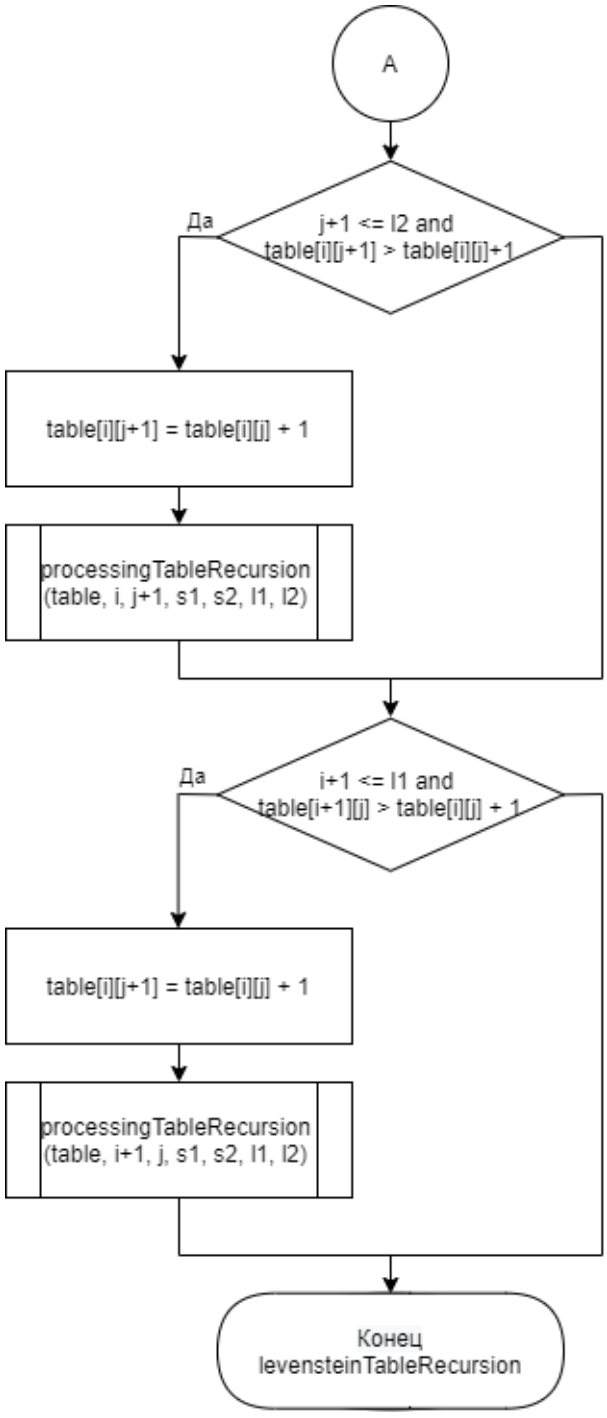

Рис. 2.5: Расстояние Левенштейна, рекурсивный метод с заполнением матрицы, часть 2

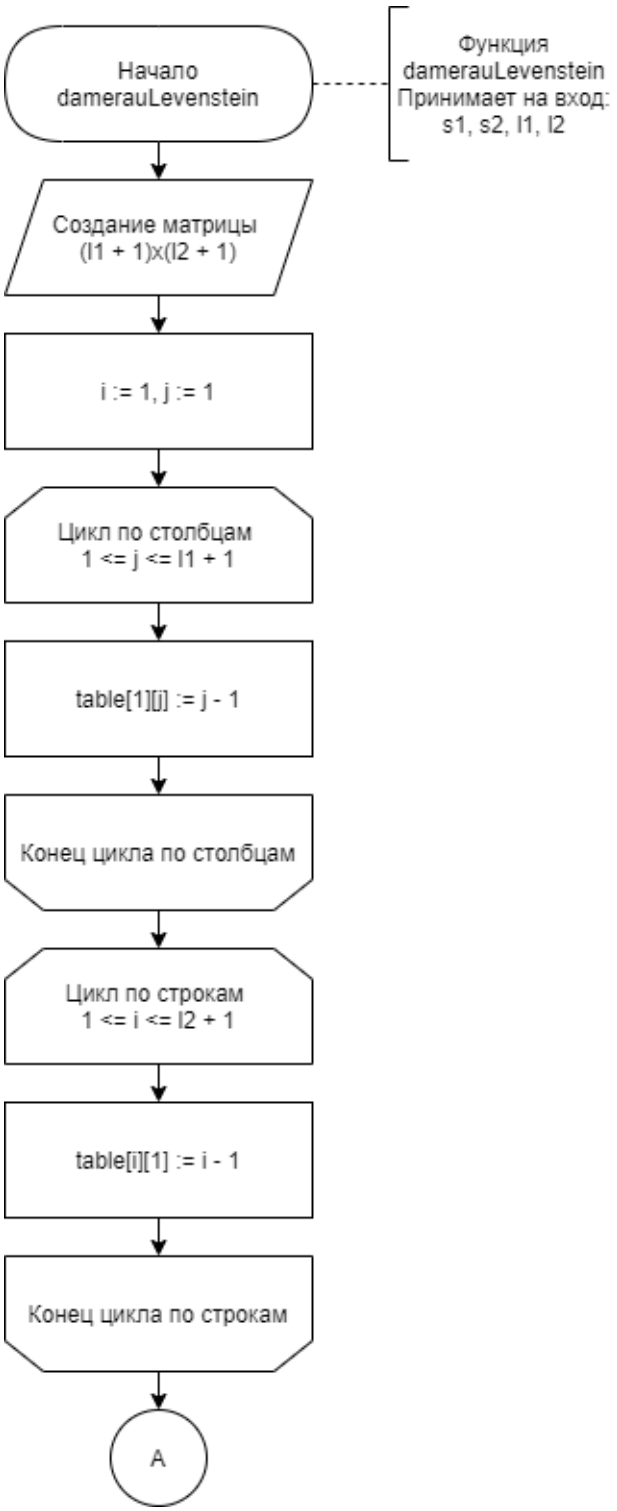

Рис. 2.6: Расстояние Дамерау-Левенштейна, матричный метод, часть 1

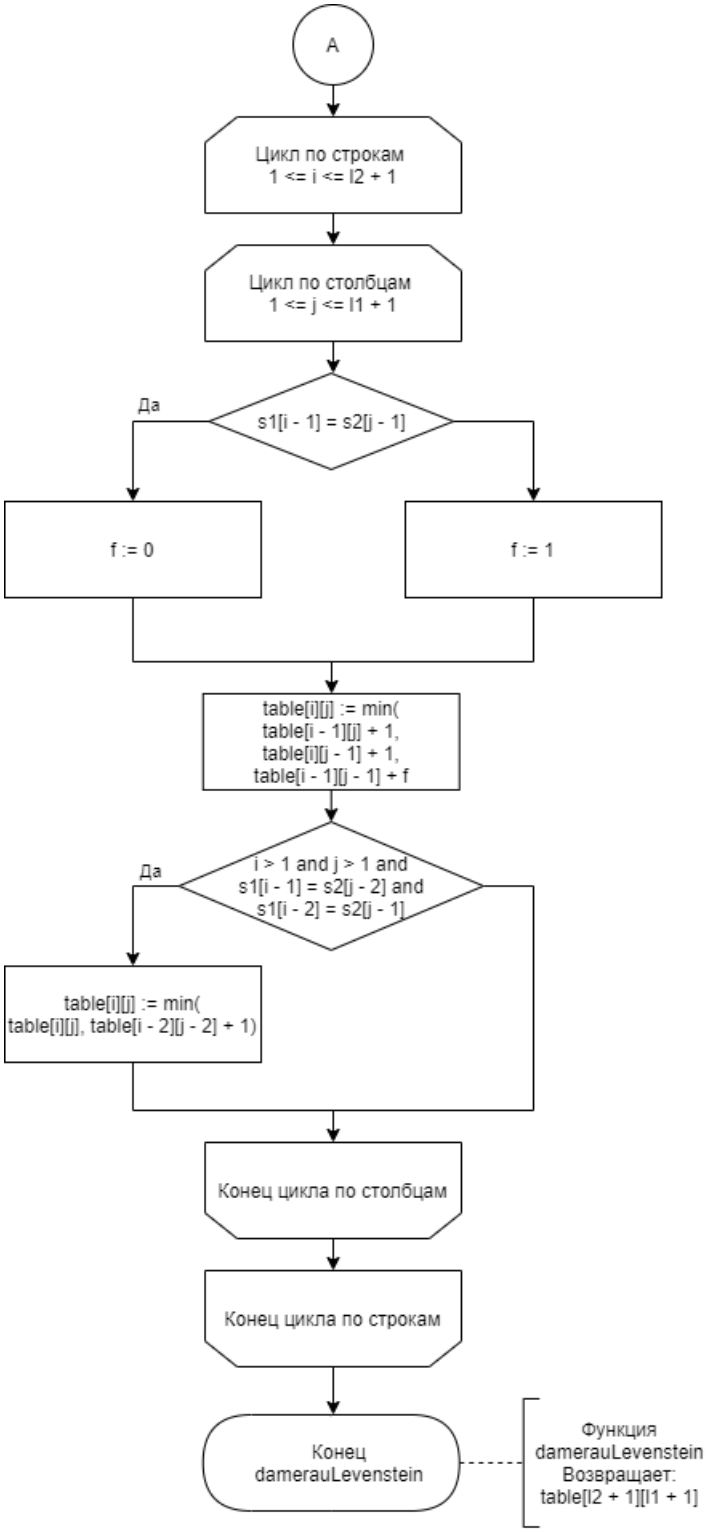

Рис. 2.7: Расстояние Дамерау-Левенштейна, матричный метод, часть 2

3. Технологическая часть

3.1. Выбор языка программирования

В качестве языка программирования было решено выбрать Python 3, так как уже имеется опыт работы с библиотеками и инструментами языка, которые позволяют реализовать и провести исследования алгоритмов подсчёта расстояния.

3.2. Листинг кода реализованных алгоритмов

Далее с листинга 3.1 по 3.4 приведены реализации алгоритмов подсчёта расстояния Левенштейна и Дамерау-Левенштейна.

Листинг 3.1: расстояние Левенштейна, рекурсивный метод

```
1  def levensteinRecursion(s1, s2):
2      if (s1 == "" or s2 == ""):
3          return len(s1) + len(s2)
4
5      if (s1[-1] == s2[-1]):
6          f = 0
7      else:
8          f = 1
9
10     return min(levensteinRecursion(s1, s2[:-1]) + 1,
11                levensteinRecursion(s1[:-1], s2) + 1,
12                levensteinRecursion(s1[:-1], s2[:-1]) + f)
```

Листинг 3.2: расстояние Левенштейна, матричный метод

```
1  def levensteinTable(s1, s2, isPrint):
2      lenI = len(s1) + 1
3      lenJ = len(s2) + 1
4
5      table = [[i + j for j in range(lenJ)]
6                      for i in range(lenI)]
7
8      for i in range(1, lenI):
```

```
9           for j in range(1, lenJ):
10              if (s1[i - 1] == s2[j - 1]):
11                  f = 0
12              else:
13                  f = 1
14              table[i][j] = min(table[i - 1][j] + 1,
15                                table[i][j - 1] + 1,
16                                table[i - 1][j - 1] + f)
17      if (isPrint):
18          tablePrint(table)
19
20      return table[-1][-1]
```

Листинг 3.3: расстояние Левенштейна, рекурсивный метод с заполнением матрицы

```
1  def processingTableRecursion(table, i, j, s1, s2):
2      if (i + 1 < len(table)) and (j + 1 < len(table[0])):
3          if s1[j] == s2[i]:
4              add = 0
5          else:
6              add = 1
7
8          if table[i + 1][j + 1] > table[i][j] + add:
9              table[i + 1][j + 1] = table[i][j] + add
10             processingTableRecursion(table, i + 1,
11                                      j + 1, s1, s2)
12
13     if (j + 1 < len(table[0])) and
14        (table[i][j + 1] > table[i][j] + 1):
15         table[i][j + 1] = table[i][j] + 1
16         processingTableRecursion(table, i, j + 1, s1, s2)
17
18     if (i + 1 < len(table)) and
19        (table[i + 1][j] > table[i][j] + 1):
20         table[i + 1][j] = table[i][j] + 1
```

```
21          processingTableRecursion(table, i + 1, j, s1, s2)
22
23  def levensteinTableRecursion(s1, s2, isPrint):
24      lenI = len(s1) + 1
25      lenJ = len(s2) + 1
26
27      maxLen = max(len(s1), len(s2)) + 1
28
29      table = [[maxLen] * lenI for i in range(lenJ)]
30      table[0][0] = 0
31
32      processingTableRecursion(table, 0, 0, s1, s2)
33
34      if isPrint:
35          tablePrint(table)
36
37      return table[-1][-1]
```

Листинг 3.4: расстояние Дамерау-Левенштейна, матричный метод

```
1   def damerauLevenstein(s1, s2, isPrint):
2       lenI = len(s1) + 1
3       lenJ = len(s2) + 1
4
5       table = [[i + j for j in range(lenJ)]
6                       for i in range(lenI)]
7
8       for i in range(1, lenI):
9           for j in range(1, lenJ):
10              if (s1[i - 1] == s2[j - 1]):
11                  f = 0
12              else:
13                  f = 1
14
15              table[i][j] = min(table[i - 1][j] + 1,
16                                table[i][j - 1] + 1,
```

```
17                                  table[i - 1][j - 1] + f)
18
19
20                 if (i > 1 and j > 1) and
21                     (s1[i - 1] == s2[j - 2]) and
22                     (s1[i - 2] == s2[j - 1]):
23                      table[i][j] = min(table[i][j],
24                      table[i - 2][j - 2] + 1)
25
26     if isPrint:
27         tablePrint(table)
28
29     return table[-1][-1]
```

3.3. Результаты тестирования

Для модульного тестирования реализованных алгоритмов, были написаны специальные функции с наборами тестов:

Листинг 3.5: Модульные тесты

```
1  def doUnitTest(testArray, testName, levFunction, isTable):
2      for i in range(len(testArray)):
3          if (isTable):
4              result = levFunction(testArray[i][0],
5                                   testArray[i][1], False)
6          else:
7              result = levFunction(testArray[i][0],
8                                   testArray[i][1])
9
10         if (result == testArray[i][2]):
11             print(testName, "test", i, "succesfully")
12         else:
13             print(testName, "test", i, "failure")
14             return False
15
16     return True
```

```
17
18
19 def unitTests(levFunction, isTable):
20     # Tests with empty string
21     test_empty = [["f", "f", 0],
22                   ["f", "", 1],
23                   ["", "f", 1]]
24     # Test with match
25     test_match = [["asd", "asd", 0],
26                   ["f", "f", 0],
27                   ["F", "f", 1]]
28     # Others tests
29     test_others = [["a", "s", 1],
30                    ["asd", "bsf", 2],
31                    ["asd", "as", 1],
32                    ["a", "adws", 3]]
33
34     if doUnitTest(test_empty, "Empty",
35                   levFunction, isTable):
36         print()
37         if doUnitTest(test_match, "Match",
38                       levFunction, isTable):
39             print()
40             if doUnitTest(test_others, "Others",
41                           levFunction, isTable):
42                 print("\n>>>All tests done!\n")
```

Все тесты прошли успешно

3.4. Оценка затрачиваемой памяти

Для подсчёта наибольшей занимаемой памяти M_{max} каждого алгоритма, пусть для удобства строки s1 и s2 имеют одиноковую длину l.

Расстояние Левенштейна, рекурсивный метод

Функция будет запрашивать память при каждом вызове. Функция принимает на вход две строки и каждый раз подсчитывает и хранит два

размера этих строк. Максимальная глубина рекурсии = l + l.

M_{max} = (l + l) * (2l * sizeof(char) + 2 * sizeof(int)) = 2l * (2l + 32) = $4l^2$ + 64l байт.

Расстояние Левенштейна, матричный метод

В данном алгоритме память тратится на матрицу и две строки.

M_{max} = (l + 1) * (l + 1) * sizeof(int) + (l + l) * sizeof(char) = (l + 1) * (l + 1) * 16 + 2l = $16l^2$ + 2 * 17l + 16 байт.

Расстояние Левенштейна, рекурсивный метод с заполнением матрицы

Память тратится на матрицу и при каждом вызове функции. Максимальная глубина рекурсии = l + l.

M_{max} = (l + 1) * (l + 1) * sizeof(int) + (l + l) * (2l * sizeof(char) + 2 * sizeof(int)) = (l^2 + 2l + 1) * 16 + 2l * (2l + 32) = $20l^2$ + 96l + 16 байт.

Расстояние Дамерау-Левенштейна, матричный метод

Память, также как и в матричном алгоритме Левенштейна тратится на матрицу и две строки.

M_{max} = 16 * l^2 + 2 * 17l + 16 байт.

3.5. Оценка затрачиваемого времени

Для замера процессорного времени выполнения алгоритмов используется библиотека time [2]. В листинге 3.6 приведена функция генерации случайной строки заданной длины. В листинге 3.7 приведены функции, с помощью которых производились замеры.

Листинг 3.6: Функция генерации строки заданной длинны с произвольными символами

```
def takeRandomString(size):
    return ''.join(random.choice(string.ascii_letters)
                   for _ in range(size))
```

Листинг 3.7: Функции для замера времени

```
def doTimeTestsRecursion(levFunc, iterations, strLength):
    t1 = process_time()

    for _ in range(iterations):
        s1 = takeRandomString(strLength)
        s2 = takeRandomString(strLength)
        levFunc(s1, s2)

    t2 = process_time()

    return (t2 - t1) / iterations


def doTimeTestsTable(levFunc, iterations, strLength):
    t1 = process_time()

    for _ in range(iterations):
        s1 = takeRandomString(strLength)
        s2 = takeRandomString(strLength)
        levFunc(s1, s2, False)

    t2 = process_time()

    return (t2 - t1) / iterations

def timeTests(levFunction, isTable, isRecursion):
    lengthsArray = [1, 3, 10, 20, 100, 1000]
    iterations   = [100000, 10000, 100, 200, 100, 10]
```

```
29
30      if (isRecursion):
31          lastIndex = 3
32      else:
33          lastIndex = len(lengthsArray)
34
35      for i in range(lastIndex):
36          if (isTable):
37              timeResult = doTimeTestsTable(levFunction,
38                                            iterations[i],
39                                            lengthsArray[i])
40          else:
41              timeResult = doTimeTestsRecursion(levFunction,
42                  iterations[i], lengthsArray[i])
43
44          print("For length =", lengthsArray[i],
45                "\ttime =", timeResult)
```

4. Исследовательская часть

Измерения процессорного времени проводятся при одинаковых длинах строк s1 и s2, при этом всём содержание наполняется случайно генерируемыми символами. Эксперименты по замеру времени проводились при длинах 1, 3, 7, 20, 100, 1000.

4.1. Результаты экспериментов

Проведя измерения процессорного времени выполнения реализованных алгоритмов, можно составить таблицу 4.1

4.2. Сравнительный анализ

Анализируя полученные результаты можно сказать, можно выделить несколько утверждений:

- наименее время затратным является алгоритм, использующий матрицу;
- рекурсивный алгоритм с заполнением матрицы демонстрирует значительно низкую скорость роста времени по сравнению с рекурсивным алгоритмом;
- алгоритм поиска расстояний Левенштейна и Дамерау-Левенштейна с помощью матриц показывают схожую скорость роста времени, однако первый алгоритм несколько быстрее.

Таблица 4.1: Результаты замеров процессорного времени в секундах

Название метода	1	3	7	20	100	1000
Лев. рекурсия	$3.8 * 10^{-6}$	$2 * 10^{-4}$	$6 * 10^{-1}$	–	–	–
Лев. матрица	$7 * 10^{-6}$	$2.2 * 10^{-5}$	$1.3 * 10^{-4}$	$4.7 * 10^{-4}$	$9 * 10^{-3}$	1.321
Лев. рекурсия-матрица	$1 * 10^{-5}$	$4 * 10^{-5}$	$2 * 10^{-4}$	$2.5 10^{-3}$	–	–
Дамерау-Левенштейна	$7.5 * 10^{-6}$	$2.7 * 10^{-5}$	$1.5 * 10^{-4}$	$6 * 10^{-4}$	0.012	1.725

Заключение

В ходе выполнения лабораторной работы были решены все задачи и достигнута цель, а именно реализация и сравнение алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна. Были изучены и описаны понятия расстояний Левенштейна и Дамерау-Левенштейна, а также были описаны и реализованы алгоритмы поиска расстояния, проведены замеры процессорного времени работы каждого алгоритма при различных строках. Была произведена оценка наибольшей занимаемой памяти. Проведён сравнительный анализ алгоритмов.

Список литературы

1. Мосалев П.М. Обзор методов нечеткого поиска текстовой информации // Вестник МГУП. 2016. №2. (дата обращения 23.09.2020)

2. Документация на официальном сайте Python про библиотеку time [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html (дата обращения 23.09.2020)