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

о лабораторной работе №1

по дисциплине Анализ Алгоритмов

на тему «Расстояния Левенштейна и Дамерау-Левенштейна»

Студент **Брянская Е.В.**

Группа **ИУ7-52Б**

Преподаватель **Волкова Л.Л.**

Москва
2020

Оглавление

Введение

Расстояние Левенштейна (редакционное расстояние) – это минимальное количество редакторских операций, которые необходимы для превращения одной строки в другую [1].

Под редакторскими операциями подразумеваются следующие операции:

1) вставка (обозначается, как I - insert);
2) замена (R - replace);
3) удаление (D - delete).

Также введём совпадение (M - match) со штрафом 0.

Расстояние Левенштейна имеет широкий спектр применения, например, используется в поисковых строках, в программах, отвечающих за автоисправление, автозамену. Помимо этого, оно также применяется в биоинформатике (строение белков представляется строками, состоящими из букв ограниченного алфавита, таким образом, упрощается их анализ).

Существует много алгоритмов, рассчитывающих расстояние Левенштейна, а также их модификаций, которые и будут рассмотрены далее.

1. Аналитическая часть

Цель данной работы – реализовать и сравнить алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна.

Для достижения поставленной цели необходимо решить ряд следующих **задач**:

1) дать математическое описание расстояний;

2) описать алгоритмы поиска расстояний;

3) оценить затрачиваемую алгоритмами память;

4) реализовать эти алгоритмы ;

5) провести замеры процессорного времени работы алгоритмов на материале серии экспериментов;

6) провести сравнительный анализ алгоритмов.

Поиск расстояния Левенштейна можно описать разными алгоритмами:

– матричный расчёт;

– рекурсивный расчёт по формуле;

– рекурсивный алгоритм, заполняющий незаполненные клетки матрицы.

Пусть S1 и S2 – строки длиной N и M соответственно. Тогда расстояние Левенштейна можно рассчитать по следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} j, & \text{если } i = 0 \\ i, & \text{если } j = 0, i > 0 \\ min(D(S1[1..i], S2[1..j-1]) + 1, & \\ \qquad D(S1[1..i-1], S2[1..j]) + 1, & \text{если } i > 0, j > 0 \\ \qquad D(S1[1..i-1], S2[1..j-1]) + & \\ \qquad + \left[\begin{array}{ll} 0, & \text{если } S1[i] == S2[j],) \\ 1, & \qquad \text{иначе} \end{array} \right. & \end{cases} \tag{1.1}$$

При таком способе расчёта расстояния нужно использовать матрицу размера Len(S1) + 1 x Len(S2) + 1, элементы которого рассчитываются по формуле выше.

Расстояние Дамерау-Левенштейна дополнительно включает операцию перестановки двух соседних символов (транспозицию) и формула выглядит следующим образом:

$$D(i,j) = \begin{cases} j, & \text{если } i = 0 \\ i, & \text{если } j = 0, i > 1 \\ min(D(S1[1..i], S2[1..j-1]) + 1, & \\ \quad D(S1[1..i-1], S2[1..j]) + 1, & \\ \quad D(S1[1..i-1], S2[1..j-1]) + & \\ \quad + \left[\begin{array}{ll} 0, & \text{если } S1[i] == S2[j], \\ 1, & \text{иначе} \end{array} \right. & \\ \quad \left[\begin{array}{ll} D(S1[1..i-2], S2[1..j-2]) + 1), & \text{если } i > 1, j > 1, \\ & S1[i] == S2[j-1], \\ & S1[i-1] == S2[j] \\ +\infty, & \text{иначе} \end{array} \right. & \\ min(D(S1[1..i], S2[1..j-1]) + 1, & \\ \quad D(S1[1..i-1], S2[1..j]) + 1, & \\ \quad D(S1[1..i-1], S2[1..j-1]) + & \\ \quad + \left[\begin{array}{ll} 0, & \text{если } S1[i] == S2[j], \\ 1, & \text{иначе} \end{array} \right.), & \text{если } i > 0, j > 0 \end{cases} \tag{1.2}$$

2. Конструкторская часть

Рассмотрим алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна для строк S1, S2, каждая из которых имеет длину N и M соответственно.

2.1. Расстояние Левенштейна, матричный алгоритм

В основе этого алгоритма лежит формула (1.2).

Задаётся матрица размером (N + 1)x(M + 1). Отдельно обрабатывается тривиальный случай: первая строка и первый столбец. Далее компоненты матрицы заполняются по формуле так, что выбирается ход с наименьшей стоимостью. Попасть в очередную клетку матрицы можно из левой, верхней и диагональной клеток.

Результат вычисления будет находится в ячейке [N][M] (то есть в самом углу справа снизу).

Схема алгоритма представлена на Рис. 2.1.

2.2. Расстояние Левенштейна, рекурсивный алгоритм

Этот алгоритм использует рекурсивную формулу для вычисления наименьшего расстояния.

На вход подаётся две строки и длины обрабатываемых подстрок i, j, которые в последующем будут рекурсивно изменяться, то есть, (i, j - 1), (i - 1, j - 1), (i - 1, j), до тех пор, пока хотя бы одна из строк не обработается полностью (длина подстроки станет равна нулю).

И по завершению работы алгоритмы выбирается наименьшее из трёх полученных значений.

Схема алгоритма представлена на Рис. 2.2.

2.3. Расстояние Левенштейна, рекурсивный метод с заполнением матрицы

Принцип работы этого алгоритма схож с алгоритмом Дейкстры поиска расстояний в графе.

Сначала задаётся матрица размером (N + 1)x(M + 1), все её ячейки заполняются значением $+\infty$. Элемент [0][0] заполняется 0, с него и будет начинаться работа алгоритма.

На вход рекурсивной функции подаётся матрица, индексы i, j, задающие текущее положение, и обрабатываемые строки. По ходу выполнения функции делается выбор, в какую следующую клетку стоит перейти из рассматриваемого ([i][j]). Выбор осуществляется так же, как это было в предыдущих алгоритмов: рассматривается три ячейки с индексами [i + 1][j + 1], [i][j + 1], [i + 1][j] и выбирается та, при переходе из которой расстояние будет наименьшим. И уже из неё осуществляется последующий запуск рекурсивной функции. Важно делать дополнительную проверку на то, чтобы соседняя клетка находилась в пределах матрицы.

Результат вычисления будет находится в ячейке [N][M] (то есть в самом углу справа снизу).

Схема алгоритма представлена на Рис. 2.3.

2.4. Расстояние Дамерау-Левенштейна, нерекурсивный алгоритм

В основе алгоритма лежит формула (1.2). В отличие от предыдущих, этот метод нахождения минимального расстояния дополнительно учитывает операцию перестановки двух соседних символов. Такая операция называется *транспозицией*

Так как этот алгоритм является модификацией описанного выше метода поиска расстояния Левенштейна, то принцип его работы аналогичен. Также создаётся матрица, отдельно отрабатываются тривиальные случаи, выбирается ход с наименьшей стоимостью, только дополнительно проверяется возможность транспозиции.

Результат также будет находится в ячейке [N][M].

Схема алгоритма представлена на Рис. 2.4.

2.5. Требования к ПО

Для корректной работы алгоритмов и проведения тестов необходимо сделать следующее.

1) Обеспечить возможность ввода двух строк через консоль и выбора алгоритма для расчёта минимального расстояния.
2) Программа должна рассчитать искомое значение и вывести его на экран, также, если в выбранном методе используется матрица, нужно вывести и её.
3) Реализовать функцию замера процессорного времени, которое выбранный метод затрачивает на вычисление результата. Дать возможность пользователю ввести длины рассматриваемых строк через консоль. Вывести результаты замеров на экран.

2.6. Заготовки тестов

При проверке на корректность работы реализованных функций необходимо провести следующие тесты:

1) обе строки пустые;
2) только одна из строк пустая;
3) полностью совпадающие строки;
4) элементарные тесты на 1 расстояние;
5) с расстоянием больше, чем 1;
6) с возможной транспозицией.

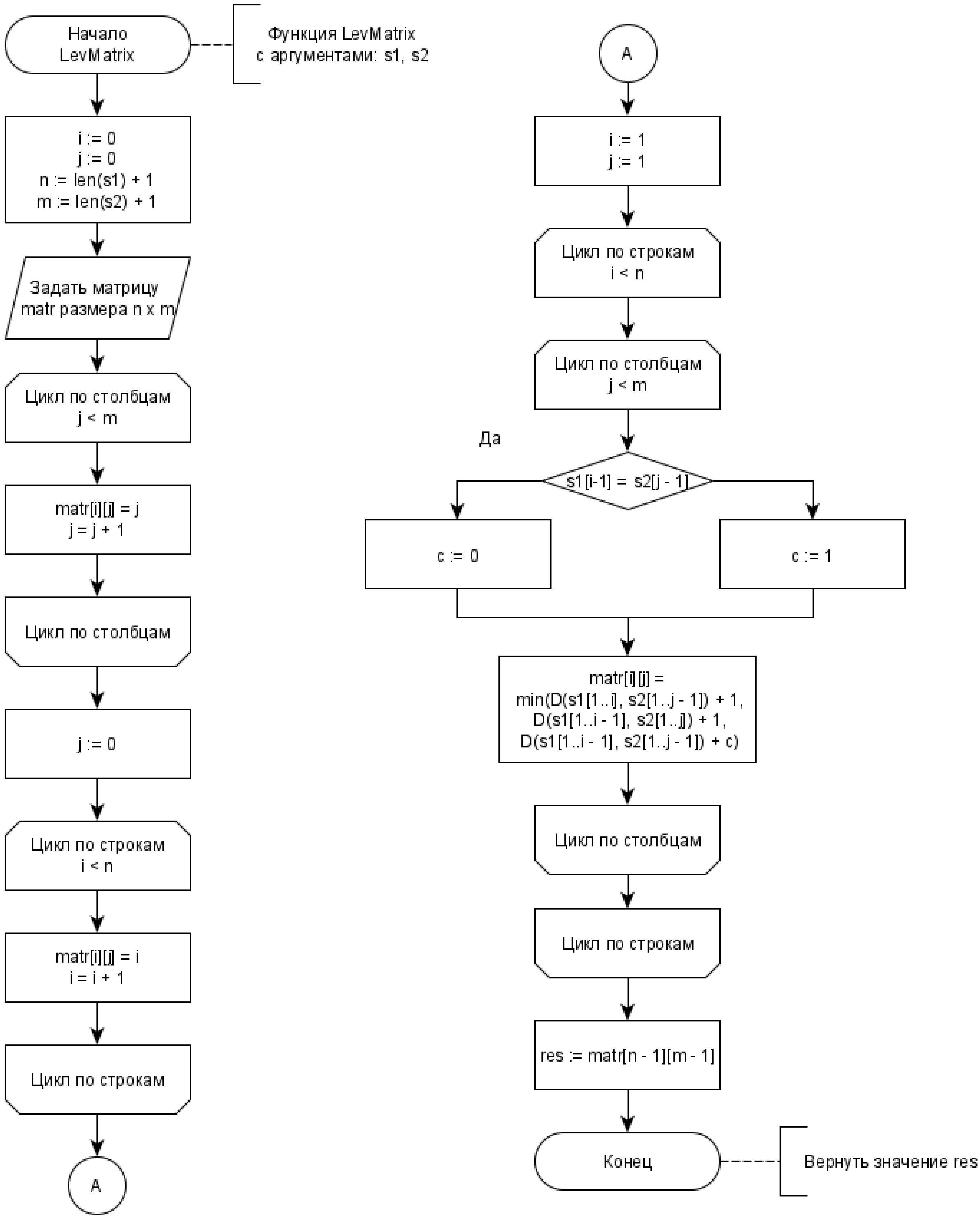

Рис. 2.1: Матричный алгоритм нахождения расстояния Левенштейна

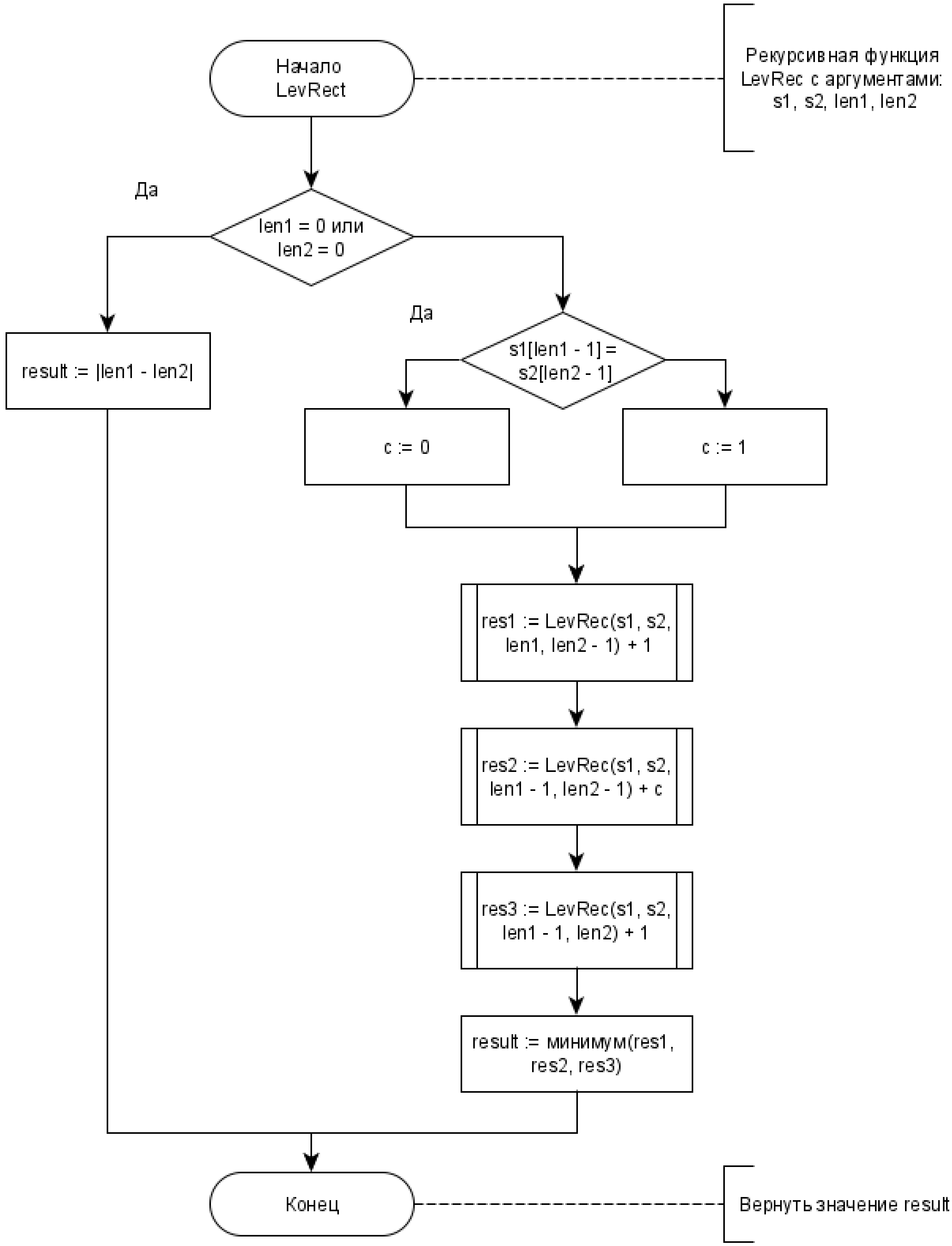

Рис. 2.2: Рекурсивный расчёт

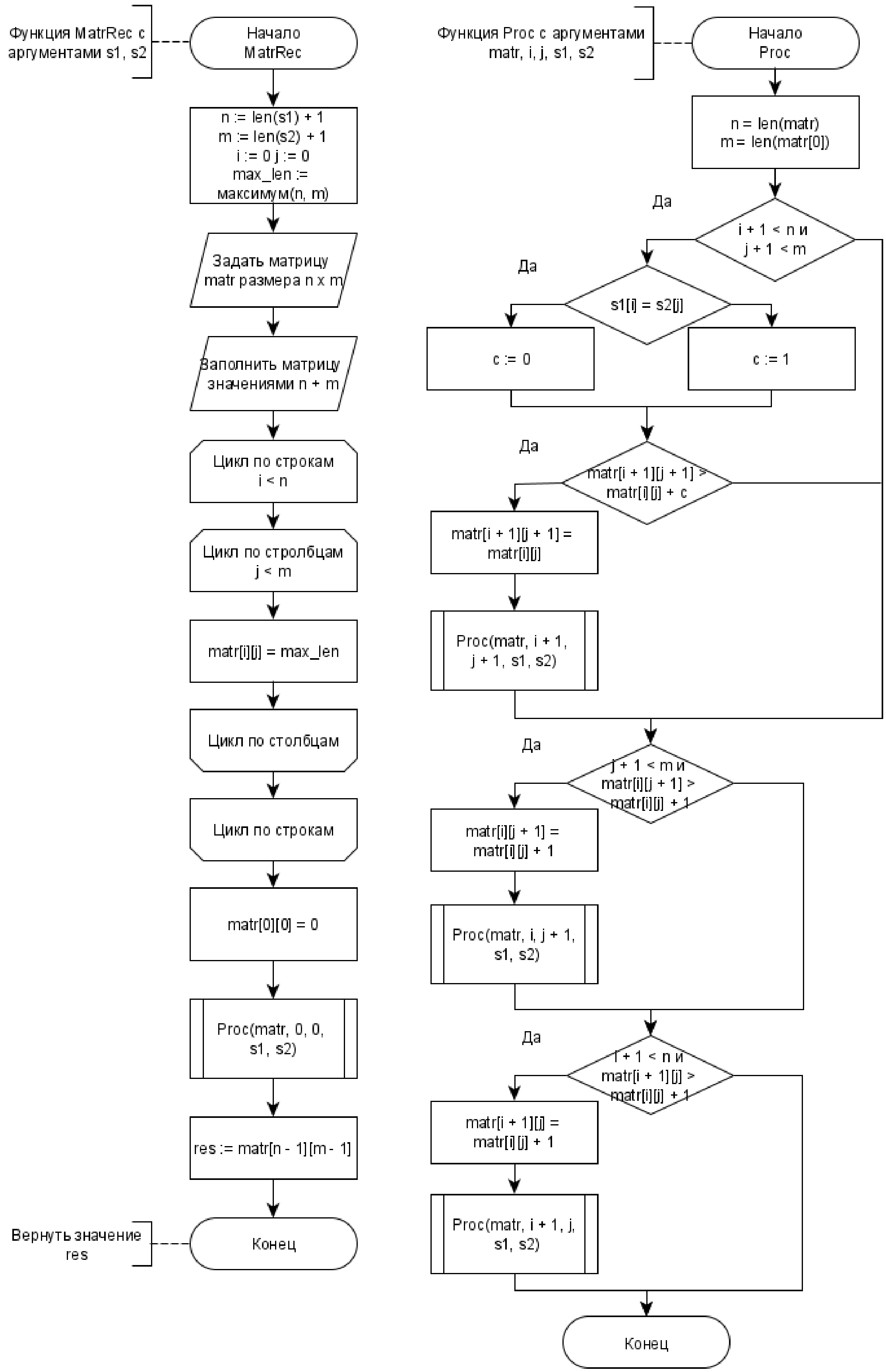

Рис. 2.3: Алгоритм, использующий рекурсию и матрицу

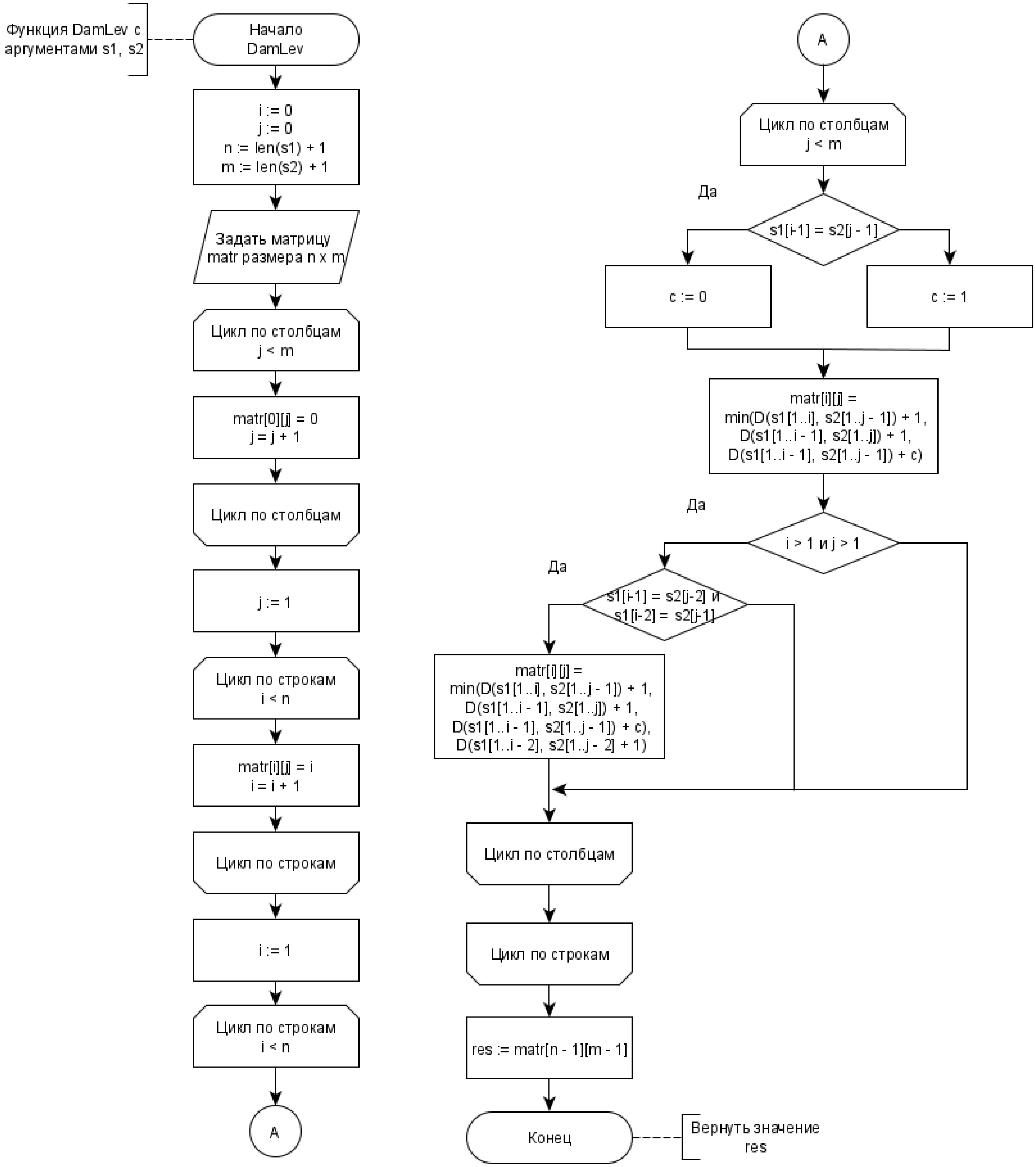

Рис. 2.4: Алгоритм нахождения расстояния Дамерау-Левенштейна

3. Технологическая часть

3.1. Выбранный язык программирования

Для выполнения этой лабораторной работы был выбран язык программирования Python, так как есть большой навык работы с ним и с подключаемыми библиотеками, которые также использовались для проведения замеров.

Для замера процессорного времени будет использован метод time из библиотеки time.h [2].

3.2. Листинг кода

Ниже представлены Листиги 3.1 - 3.4 функций, реализующих алгоритмы поиска расстояний.

Листинг 3.1: Матричный алгоритм нахождения расстояния Левенштейна

```
def LevMatrix(s1, s2):
  n = len(s1) + 1
  m = len(s2) + 1

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

  for i in range(1, n):
    for j in range(1, m):
      const = 0 if (s1[i - 1] == s2[j - 1]) else 1

      matrix[i][j] = min(matrix[i][j - 1] + 1,
                         matrix[i - 1][j] + 1,
                         matrix[i - 1][j - 1] + const)

  return matrix[n - 1][m - 1]


```

Листинг 3.2: Расстояние Левенштейна - рекурсивный расчёт по формуле

```
def LevRecursion(s1, s2, len1, len2):
  if len1 == 0 or len2 == 0:
    return abs(len1 - len2)

```

```
  const = 0 if (s1[len1 - 1] == s2[len2 - 1]) else 1
  return min(LevRecursion(s1, s2, len1, len2 - 1) + 1,
             LevRecursion(s1, s2, len1 - 1, len2 - 1) + const,
             LevRecursion(s1, s2, len1 - 1, len2) + 1)

def LevRecursion(s1, s2):
  return LevRecursion_process(s1, s2, len(s1), len(s2))
```

Листинг 3.3: Расстояние Левенштейна - алгоритм с рекурсией и матрицей

```
def LevMatrixRecursion_process(matrix, i, j, s1, s2):
  if i + 1 < len(matrix) and j + 1 < len(matrix[0]):
    const = 0 if s1[i] == s2[j] else 1
    if matrix[i + 1][j + 1] > matrix[i][j] + const:
      matrix[i + 1][j + 1] = matrix[i][j] + const
      LevMatrixRecursion_process(matrix, i + 1, j + 1, s1, s2)

  if j + 1 < len(matrix[0]) and matrix[i][j + 1] > matrix[i][j] + 1:
    matrix[i][j + 1] = matrix[i][j] + 1
    LevMatrixRecursion_process(matrix, i, j + 1, s1, s2)

  if i + 1 < len(matrix) and matrix[i + 1][j] > matrix[i][j] + 1:
    matrix[i + 1][j] = matrix[i][j] + 1
    LevMatrixRecursion_process(matrix, i + 1, j, s1, s2)

def LevMatrixRecursion(s1, s2):
  n = len(s1) + 1
  m = len(s2) + 1
  max_len = max(n, m)

  matrix = [[max_len for j in range(m)] for i in range(n)]
  matrix[0][0] = 0

  LevMatrixRecursion_process(matrix, 0, 0, s1, s2)

  return matrix[-1][-1]
```

Листинг 3.4: Расстояние Дамерау-Левенштейна

```
def DamLev(s1, s2):
  n = len(s1) + 1
  m = len(s2) + 1
```

```

    matrix = [[0] * m for i in range(n)]

    for j in range(m):
      matrix[0][j] = j

    for i in range(n):
      matrix[i][0] = i

    for i in range(1, n):
      for j in range(1, m):
        const = 0 if (s1[i - 1] == s2[j - 1]) else 1

        matrix[i][j] = min(matrix[i][j - 1] + 1,
                           matrix[i - 1][j] + 1,
                           matrix[i - 1][j - 1] + const)

        if i > 1 and j > 1:
          if s1[i - 1] == s2[j - 2] and s2[j - 1] == s1[i - 2]:
            matrix[i][j] = min(matrix[i][j - 1] + 1,
                               matrix[i - 1][j] + 1,
                               matrix[i - 1][j - 1] + const,
                               matrix[i - 2][j - 2] + 1)

    return matrix[n - 1][m - 1]
```

3.3. Результаты тестов

При тестировании использовалась специальная библиотека **unittest**. Заранее были написаны необходимые тесты в соответствии с заготовками, приведёнными выше. Тестирование происходит следующим образом: функция возвращает расчитанное значение, и оно сравнивается с тем, которое заранее было внесено в систему, и выводится соответствующий результат.

Все тесты пройдены успешно. Сами тесты представлены ниже (Листинг 3.5).

Листинг 3.5: Тесты

```
import unittest
import main

```

```python
# General tests for all algorithms
class GeneralTest(unittest.TestCase):
  @unittest.skip("General Tests were skipped")

  def setUp(self):
    self.function = None

  # Обработка пустых строк
  def test_empty_str(self):
    self.assertEqual(self.function("", ""), 0)
    self.assertEqual(self.function("", "12345"), 5)
    self.assertEqual(self.function("98765", ""), 5)

    # Совпадающие строки
  def test_match(self):
    self.assertEqual(self.function("1", "1"), 0)
    self.assertEqual(self.function("12qw", "12qw"), 0)
    self.assertEqual(self.function("AbC", "AbC"), 0)
    self.assertEqual(self.function("Abc", "abc"), 1)

  # Простые тесты
  def test_easy(self):
    self.assertEqual(self.function("1", "2"), 1)
    self.assertEqual(self.function("123", "1"), 2)
    self.assertEqual(self.function("1", "123"), 2)
    self.assertEqual(self.function("a", "ab"), 1)
    self.assertEqual(self.function("ab", "a"), 1)
    self.assertEqual(self.function("a5c", "abc"), 1)

# Тесты для алгоритма поиска расстояния Левенштейна
class LevTest(GeneralTest):
  def test_lev(self):
    self.assertEqual(self.function("тело", "столб"), 3)
    self.assertEqual(self.function("увлечение", "развлечения"), 4)
    self.assertEqual(self.function("касаться", "опасаться"), 2)
    self.assertEqual(self.function("abccde", "cdeabc"), 6)

# Тесты для алгоритма поиска расстояния ДамерауЛевенштейна—
class DemLevTest(GeneralTest):
  def setUp(self):
```

```
      self.function = main.DamLev

  # Тесты на поиск транспозиций
  def test_demlev(self):
    self.assertEqual(self.function("qw", "wq"), 1)
    self.assertEqual(self.function("132", "123"), 1)
    self.assertEqual(self.function("1001", "0110"), 2)
    self.assertEqual(self.function("2143", "1234"), 2)

# Все алгоритмы поиска расстояния Левенштейна проходят не только общие тесты, но
    и специально написанные LevTest

# Алгоритм поиска расстояния Левенштейна матричный()
class LevMatrixTest(LevTest):
  def setUp(self):
    self.function = main.LevMatrix

# Алгоритм поиска расстояния Левенштейна рекурсия()
class LevRecursionTest(LevTest):
  def setUp(self):
    self.function = main.LevRecursion

# Алгоритм поиска расстояния Левенштейна матрица( + рекурсия)
class LevMatrixRecursionTest(LevTest):
  def setUp(self):
    self.function = main.LevMatrixRecursion


# Запуск тестов
if __name__ == "__main__":
  unittest.main()
```

3.4. Оценка памяти

Рассчитаем память, максимально затрачиваемую каждым алгоритмом при обработке строк s1 и s2. Для упрощения вычислений примем длины строк равными n.

Расстояние Левенштейна (матрица)

Память в этом алгоритме затрачивается на хранение самой матрицы и двух строк.

$M_{matrix} = (n+1) * (n+1) * sizeof(int) = (n+1)^2 * 16$

$M_{strings} = 2 * n * sizeof(char) = 2 * n$

$M = (n+1)^2 * 16 + 2 * n = 16 * n^2 + 34 * n + 16$

Расстояние Левенштейна (рекурсивный расчёт)

Так как это рекурсивный расчёт, то память используется при каждом вызове функции. Функция принимает на вход 2 строки (по значению) и 2 значения, которые являются размерами строк. Максимальная глубина рекурсии n + n.

$M = (n+n) * (2 * n * sizeof(char) + 2 * sizeof(int)) = 2 * n * (2 * n + 2 * 16) = 2 * n * (2 * n + 32) = 4 * n^2 + 64 * n$

Расстояние Левенштейна (матрица + рекурсия)

Память в этом алгоритме затрачивается для хранения матрицы и при каждом вызове функции. И максимальная глубина рекурсии n + n.

$M_{matrix} = (n+1) * (n+1) * sizeof(int) = (n+1)^2 * 16$

$M_{recursion} = (n+n) * (2 * n * sizeof(char) + 2 * sizeof(int)) = 2 * n * (2 * n + 2 * 16) = 2 * n * (2 * n + 32) = 4 * n^2 + 64 * n$

$M = 16 * n^2 + 32 * n + 16 + 4 * n^2 + 64 * n = 20 * n^2 + 96 * n + 16$

Расстояние Дамерау-Левенштейна (матрица)

Затраты на память такие же, как в матричном методе поиска расстояния Левенштейна.

$M = (n+1)^2 * 16 + 2 * n = 16 * n^2 + 34 * n + 16$

3.5. Среда и инструменты для замера времени

Замер процессорного времени осуществлялся с помощью специальной библиотеки **time**. Осуществление замеров указано ниже (Листинг 3.6).

При замерах пользователь указывает длину строк (для того, чтобы проще было составить далее сравнительную таблицу, длины строк примем одинаковыми). Далее программно генерируются строки, которые далее будет анализироваться.

Листинг 3.6: Замеры процессорного времени

```
# Generating a random line
def RandomString(number):
    letters = string.ascii_lowercase
    return ''.join(random.choice(letters) for i in range(number))

def TestTime(function, s1, s2):
    time_start = time.process_time()
    num_oper = 0
    while time.process_time() - time_start < 1:
        function(s1, s2)
        num_oper += 1

    result_time = time.process_time() - time_start

    print("Выполнено {:} операций, затрачено {:} секунд".format(num_oper, result_time))
    print("Время: {:.4} секунд".format(result_time / num_oper))
    print()

def MeasureTime(length):
    s1 = RandomString(length)
    s2 = RandomString(length)

    print(">>> Generated string 1: ", s1)
    print(">>> Generated string 2: ", s2)

    print("\n---Levenshtein distance (matrix)---")
    TestTime(LevMatrix, s1, s2)

```

```
print("-----Levenshtein distance (recurtion)-----")
TestTime(LevRecursion, s1, s2)

print("-----Levenshtein distance (matrix + recurtion)-----")
TestTime(LevMatrixRecursion, s1, s2)

print("-----Damerau-Levenshtein distance (matrix)-----")
TestTime(DamLev, s1, s2)

```

4. Исследовательская часть

Как было упомянуто выше, для облегчения проведения анализа принимается, что длины обрабатываемых строк равны. $len1 = len2 \in \{3, 6, 10, 15, 20, 40, 70, 100, 300\}$. Содержимое строк генерируется случайным образом.

Каждый замер проводится 5 раз для получения более точного среднего результата.

В таблице 4.1 представлены результаты замеров процессорного времени работы реализаций алгоритмов (в сек).

Таблица 4.1: Результаты измерений

Длина n / Алгоритм	3	6	10	15	20	40	70	100	300
Левенштейн (матрица)	$7.998 * 10^{-6}$	$2.383 * 10^{-5}$	$6.208 * 10^{-5}$	0.0002	0.0002	0.001	0.003	0.005	0.049
Левенштейн (рекурсия)	$2.69 * 10^{-5}$	0.004	3.587	—	—	—	—	—	—
Левенштейн (матрица+ рекурсия	$1.854 * 10^{-5}$	$6.123 * 10^{-5}$	0.0002	0.0004	0.0008	0.005	0.029	0.085	2.687
Дамерау-Левенштейн	$9.732 * 10^{-6}$	$3.05 * 10^{-5}$	$7.658 * 10^{-5}$	0.0002	0.0003	0.006	0.004	0.008	0.068

Важно отметить, что замеры для рекурсивного алгоритма поиска расстояния Левенштейна не были произведены в ряде случае, так как время расчёта превышает 7 минут.

Согласно полученным данным, можно сделать несколько выводов.

1) Достаточно ожидаемым результатом стали показатели алгоритма поиска расстояния Левенштейна с использованием рекурсии. При обработке строк, длина которых больше 10, время выполнения резко возрастает и достигает значения в несколько минут.

2) Самым быстродейственным оказался матричный метод поиска расстояния Левенштейна.

3) Алгоритм Дамерау-Левенштейна также достаточно эффективный, лишь немного уступает матричному алгоритму.

4) Алгоритм поиска расстояния Левенштейна, построенный на рекурсии и матрице, имеет показатели гораздо лучшие, чем у алгоритма, использующего только рекурсию; но уступает другим по быстродействию.

Заключение

В ходе лабораторной работы была достигнута поставленная цель, а именно, были изучены, реализованы алгоритмы поиска наименьшего расстояния, также произведён сравнительный анализ.

В процессе выполнения были решены все задачи. Описаны все рассматриваемые алгоритмы, оценена затрачиваемая память при их реализации, кроме того, были сделаны замеры процессорного времени работы каждого на материале серии экспериментов и сделаны соответствующие выводы.

Список литературы

1. В. И. Левенштейн. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965. 163.4:845-848.

2. Официальный сайт Python, документация [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html, свободный (дата обращения: 13.09.20).