

лаб01

# Оглавление

|   |           |
|---|-----------|
| <b>Введение</b>   | <b>3</b>  |
| <b>1 Аналитическая часть</b>  | <b>4</b>  |
| <b>2 Конструкторская часть</b>  | <b>6</b>  |
| 2.1 Расстояние Левенштейна, матричный алгоритм . . . . .                | 6         |
| 2.2 Расстояние Левенштейна, рекурсивный алгоритм . . . . .              | 6         |
| 2.3 Расстояние Левенштейна, рекурсивный метод с заполнением матрицы . . | 7         |
| 2.4 Расстояние Дамерау-Левенштейна . . . . .                            | 7         |
| 2.5 Требования к ПО . . . . .   | 8         |
| 2.6 Заготовки тестов . . . . .  | 8         |
| <b>3 Технологическая часть</b>  | <b>13</b> |
| 3.1 Выбранный язык программирования . . . . .                           | 13        |
| 3.2 Листинг кода . . . . .  | 13        |
| 3.3 Результаты тестов . . . . .   | 15        |
| 3.4 Оценка памяти . . . . .   | 17        |
| 3.5 Среда и инструменты для замера времени . . . . .                    | 19        |
| <b>4 Исследовательская часть</b>  | <b>20</b> |
| <b>Заключение</b>   | <b>21</b> |

# Введение

**Расстояние Левенштейна** (рациональное расстояние) – это минимальное количество редакторских операций, которые необходимы для превращения одной строки в другую.

Под редакторскими операциями подразумеваются:

- вставка (обозначается, как I - insert);
- замена (R - replace);
- удаление (D - delete);
- также сюда относится совпадение (M - match).

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:

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

При таком способе расчёта расстояния нужно использовать матрицу размера  $\text{Len}(S1) + 1 \times \text{Len}(S2) + 1$ , элементы которого рассчитываются по формуле выше.

Что касается **рекурсивного расчёта**, то возникает проблема большого количества повторных вычислений. Это очень сильно влияет как на время выполнения, так и на

занимаемую память.

**Рекурсивный алгоритм, заполняющий незаполненные клетки матрицы,** работает по аналогии с бесконечностями в алгоритме Дейкстры поиска расстояний в графе.

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

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

## 2. Конструкторская часть

Рассмотрим алгоритмы поиска расстояния Левенштейна и Дameraу-Левенштейна для строк  $S_1$ ,  $S_2$ , 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  $(N + 1) \times (M + 1)$ . 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  $[N - 1][M - 1]$  (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  $(N + 1) \times (M + 1)$ , все её ячейки заполняются значением  $+\infty$ . Элемент  $[0][0]$  заполняется 0, с него и будет начинаться работы 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 алгоритмах: 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  $[N - 1][M - 1]$  (то есть в самом углу справа снизу).

Схема алгоритма представлена на Рис. 2.3.

## 2.4. Расстояние Дамерау-Левенштейна

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 обрабатываются тривиальные случаи, выбирается ход с наименьшей стоимостью, только дополнительно проверяется возможность транспозиции.

Результат также будет находится в ячейке  $[N - 1][M - 1]$ .

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.



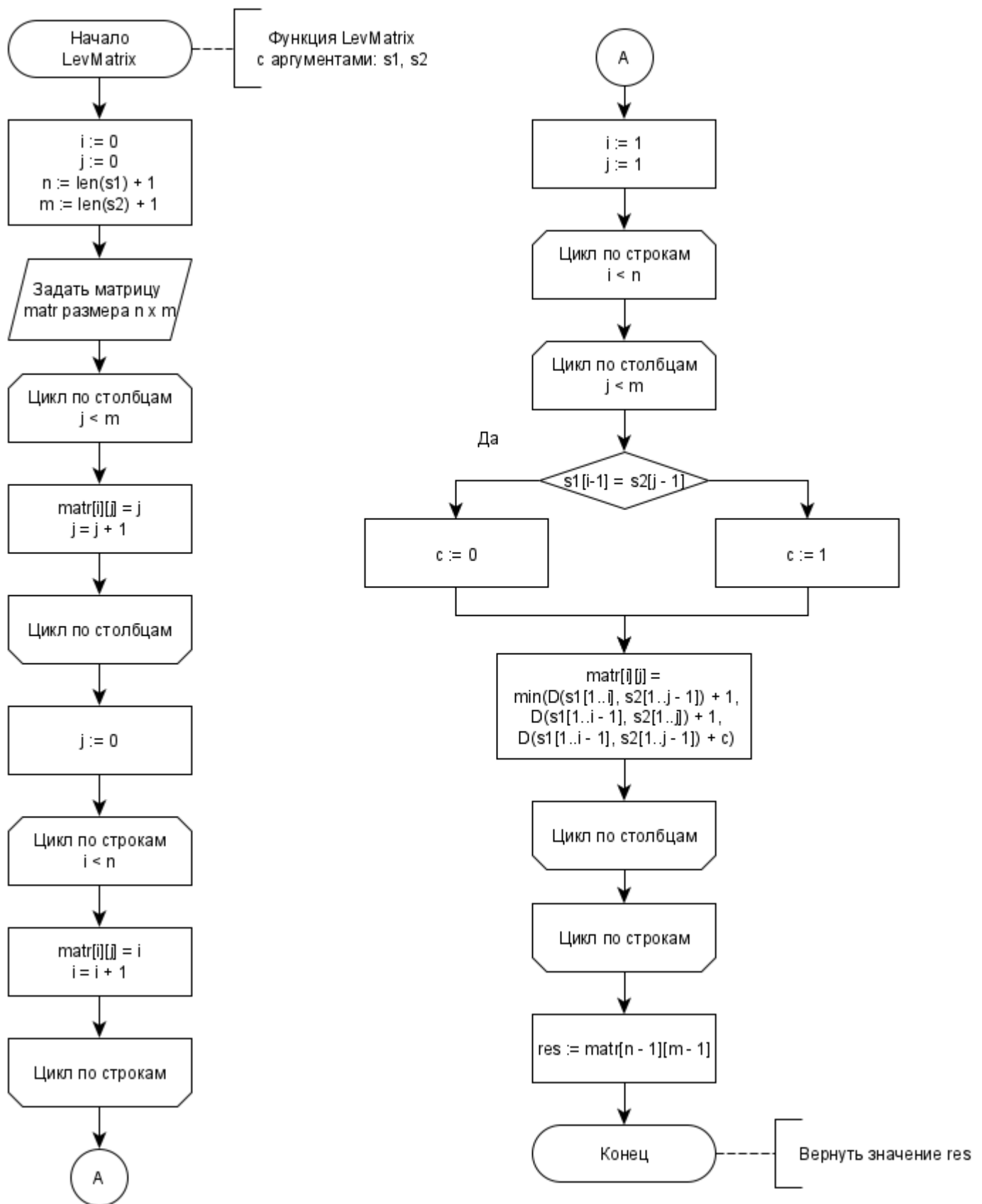


Рис. 2.1: Матричный алгоритм нахождения расстояния Левенштейна

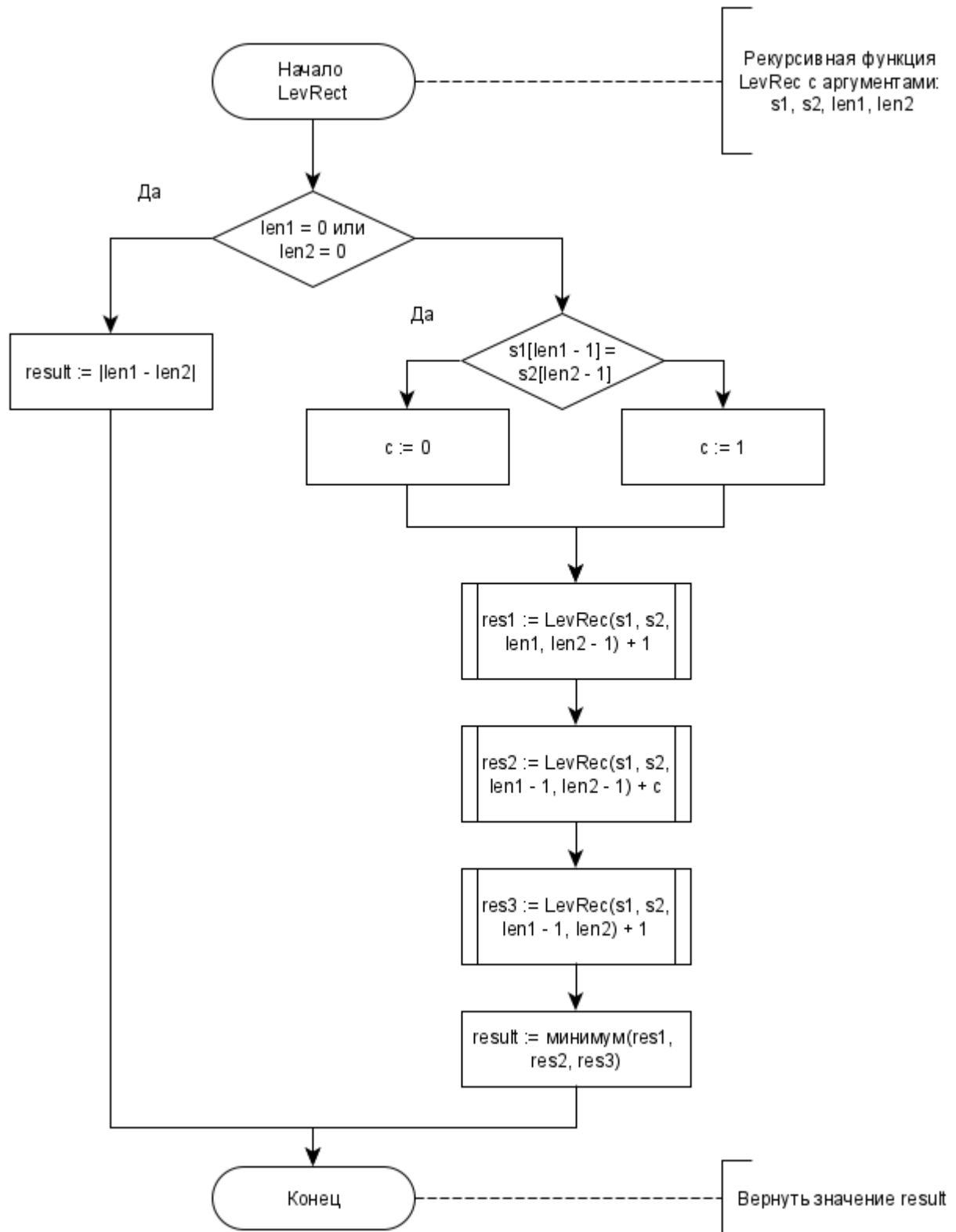


Рис. 2.2: Рекурсивный расчёт

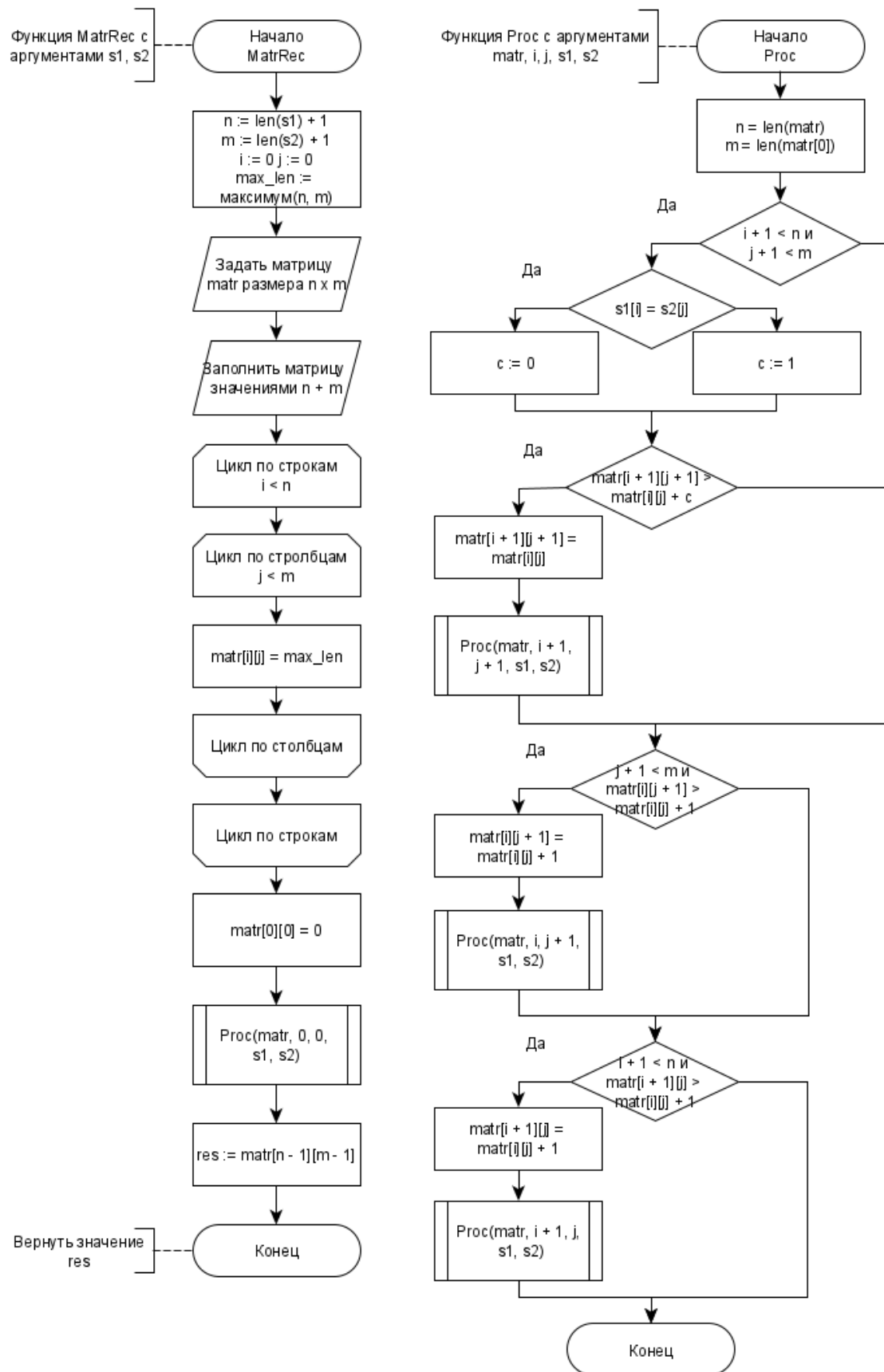


Рис. 2.3: Алгоритм, использующий рекурсию и матрицу

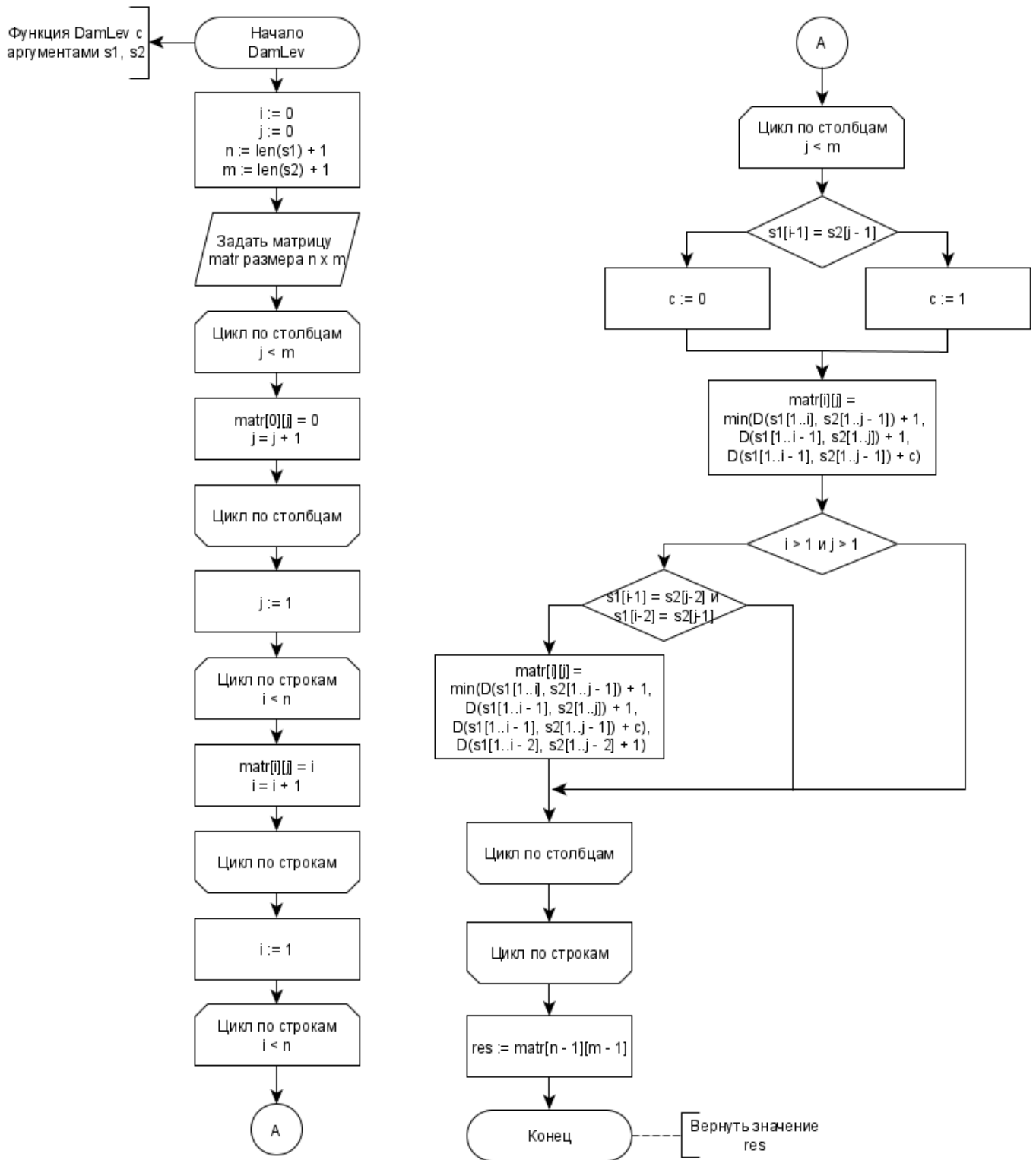


Рис. 2.4: Алгоритм нахождения расстояния Дамерау-Левенштейна

## 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.

### 3.2. Листинг кода

Листинг 3.1: Матричный алгоритм нахождения расстояния Левенштейна

```
1 def LevMatrix(s1, s2):
2     n = len(s1) + 1
3     m = len(s2) + 1
4
5     matrix = [[i + j for j in range(m)] for i in range(n)]
6
7     for i in range(1, n):
8         for j in range(1, m):
9             const = 0 if (s1[i - 1] == s2[j - 1]) else 1
10
11             matrix[i][j] = min(matrix[i][j - 1] + 1,
12                                matrix[i - 1][j] + 1,
13                                matrix[i - 1][j - 1] + const)
14
15     return matrix[n - 1][m - 1]
```

Листинг 3.2: Расстояние Левенштейна - рекурсивный расчёт по формуле

```
1 def LevRecursion(s1, s2, len1, len2):
2     if len1 == 0 or len2 == 0:
3         return abs(len1 - len2)
4
5     const = 0 if (s1[len1 - 1] == s2[len2 - 1]) else 1
6     return min(LevRecursion(s1, s2, len1, len2 - 1) + 1,
7                LevRecursion(s1, s2, len1 - 1, len2 - 1) + const,
8                LevRecursion(s1, s2, len1 - 1, len2) + 1)
9
10 def LevRecursion(s1, s2):
11     return LevRecursion_process(s1, s2, len(s1), len(s2))
```

Листинг 3.3: Расстояние Левенштейна - алгоритм с рекурсией и матрицей

```

1 def LevMatrixRecursion_process(matrix, i, j, s1, s2):
2     if i + 1 < len(matrix) and j + 1 < len(matrix[0]):
3         const = 0 if s1[i] == s2[j] else 1
4         if matrix[i + 1][j + 1] > matrix[i][j] + const:
5             matrix[i + 1][j + 1] = matrix[i][j] + const
6             LevMatrixRecursion_process(matrix, i + 1, j + 1, s1, s2)
7
8     if j + 1 < len(matrix[0]) and matrix[i][j + 1] > matrix[i][j] + 1:
9         matrix[i][j + 1] = matrix[i][j] + 1
10        LevMatrixRecursion_process(matrix, i, j + 1, s1, s2)
11
12    if i + 1 < len(matrix) and matrix[i + 1][j] > matrix[i][j] + 1:
13        matrix[i + 1][j] = matrix[i][j] + 1
14        LevMatrixRecursion_process(matrix, i + 1, j, s1, s2)
15
16 def LevMatrixRecursion(s1, s2):
17     n = len(s1) + 1
18     m = len(s2) + 1
19     max_len = max(n, m)
20
21     matrix = [[max_len for j in range(m)] for i in range(n)]
22     matrix[0][0] = 0
23
24     LevMatrixRecursion_process(matrix, 0, 0, s1, s2)
25
26     return matrix[-1][-1]

```

Листинг 3.4: Расстояние Дамерау-Левенштейна

```

1 def DamLev(s1, s2):
2     n = len(s1) + 1
3     m = len(s2) + 1
4
5     matrix = [[0] * m for i in range(n)]
6
7     for j in range(m):
8         matrix[0][j] = j
9
10    for i in range(n):
11        matrix[i][0] = i

```

```

12
13 for i in range(1, n):
14     for j in range(1, m):
15         const = 0 if (s1[i - 1] == s2[j - 1]) else 1
16
17         matrix[i][j] = min(matrix[i][j - 1] + 1,
18                             matrix[i - 1][j] + 1,
19                             matrix[i - 1][j - 1] + const)
20
21     if i > 1 and j > 1:
22         if s1[i - 1] == s2[j - 2] and s2[j - 1] == s1[i - 2]:
23             matrix[i][j] = min(matrix[i][j - 1] + 1,
24                                 matrix[i - 1][j] + 1,
25                                 matrix[i - 1][j - 1] + const,
26                                 matrix[i - 2][j - 2] + 1)
27
28 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 рассчитанное значение, и оно сравнивается с тем, которое заранее было внесено в систему, и выводится соответствующий результат.

**Все тесты выполнены.**

Сами тесты представлены ниже (Листинг 3.5).

Листинг 3.5: Тесты

```

1 import unittest
2 import main
3
4 # General tests for all algorithms
5 class GeneralTest(unittest.TestCase):
6     @unittest.skip("General Tests were skipped")
7
8     def setUp(self):

```

```

9     self.function = None
10
11     # Processing of empty strings
12     def test_empty_str(self):
13         self.assertEqual(self.function("", ""), 0)
14         self.assertEqual(self.function("", "12345"), 5)
15         self.assertEqual(self.function("98765", ""), 5)
16
17     # Same strings
18     def test_match(self):
19         self.assertEqual(self.function("1", "1"), 0)
20         self.assertEqual(self.function("12qw", "12qw"), 0)
21         self.assertEqual(self.function("AbC", "AbC"), 0)
22         self.assertEqual(self.function("Abc", "abc"), 1)
23
24     # Simple tests
25     def test_easy(self):
26         self.assertEqual(self.function("1", "2"), 1)
27         self.assertEqual(self.function("123", "1"), 2)
28         self.assertEqual(self.function("1", "123"), 2)
29         self.assertEqual(self.function("a", "ab"), 1)
30         self.assertEqual(self.function("ab", "a"), 1)
31         self.assertEqual(self.function("a5c", "abc"), 1)
32
33     # Tests for Levenshtein distance search algorithm
34     class LevTest(GeneralTest):
35         def test_lev(self):
36             self.assertEqual(self.function("1234", "51437"), 3)
37             self.assertEqual(self.function("012343563", "7891234356a"), 4)
38             self.assertEqual(self.function("01213425", "671213425"), 2)
39             self.assertEqual(self.function("abccde", "cdeabc"), 6)
40
41     # Tests for Damerau–Levenshtein distance search algorithm
42     class DemLevTest(GeneralTest):
43         def setUp(self):
44             self.function = main.DamLev
45
46     # Transposition tests
47     def test_demlev(self):
48         self.assertEqual(self.function("qw", "wq"), 1)

```



```

49     self.assertEqual(self.function("132", "123"), 1)
50     self.assertEqual(self.function("1001", "0110"), 2)
51     self.assertEqual(self.function("2143", "1234"), 2)
52
53 # All Levenshtein distance search algorithms are tested by not only
    general tests, but also by special ones for them — LevTest
54
55 # Levenshtein distance search algorithm (matrix)
56 class LevMatrixTest(LevTest):
57     def setUp(self):
58         self.function = main.LevMatrix
59
60 # Levenshtein distance search algorithm (recursion)
61 class LevRecursionTest(LevTest):
62     def setUp(self):
63         self.function = main.LevRecursion
64
65 # Levenshtein distance search algorithm (matrix + recursion)
66 class LevMatrixRecursionTest(LevTest):
67     def setUp(self):
68         self.function = main.LevMatrixRecursion
69
70
71 # Start testing
72 if __name__ == "__main__":
73     unittest.main()

```

### 3.4. Оценка памяти

Рассчитаем память, максимально затрачиваемую каждым алгоритмом при обработке строк  $s_1$  и  $s_2$ . 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  $n + n$ .

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

### Расстояние Левенштейна (матрица + рекурсия)

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

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

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

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

### 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

```
1 # Generating a random line
2 def RandomString(number):
3     letters = string.ascii_lowercase
4     return ''.join(random.choice(letters) for i in range(number))
5
6 def MeasureTime(length):
7     s1 = RandomString(length)
8     s2 = RandomString(length)
9
10    print(">>> Generated string 1: ", s1)
11    print(">>> Generated string 2: ", s2)
12
13    print("\n——Levenshtein distance (matrix)——")
14    TestTime(LevMatrix, s1, s2)
15
16    print("——Levenshtein distance (recurtion)——")
17    TestTime(LevRecursion, s1, s2)
18
19    print("——Levenshtein distance (matrix + recurtion)——")
20    TestTime(LevMatrixRecursion, s1, s2)
21
22    print("——Damerau-Levenshtein distance (matrix)——")
23    TestTime(DamLev, s1, s2)
```

## 4. Исследовательская часть

Как было упомянуто выше, для облегчения проведения анализа принимается, что длины обрабатываемых строк равны.  $len1 = len2 \in \{1, 3, 6, 10, 15, 20, 40, 70, 100, 1000\}$ . Содержимое строк генерируется случайным образом.

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

## Заключение