лаб01

Оглавление

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:

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

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

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:

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

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 - 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, использующий рекурсию и матрицу

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 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 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 - 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 отрабатываются 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.
5.
6.
7.

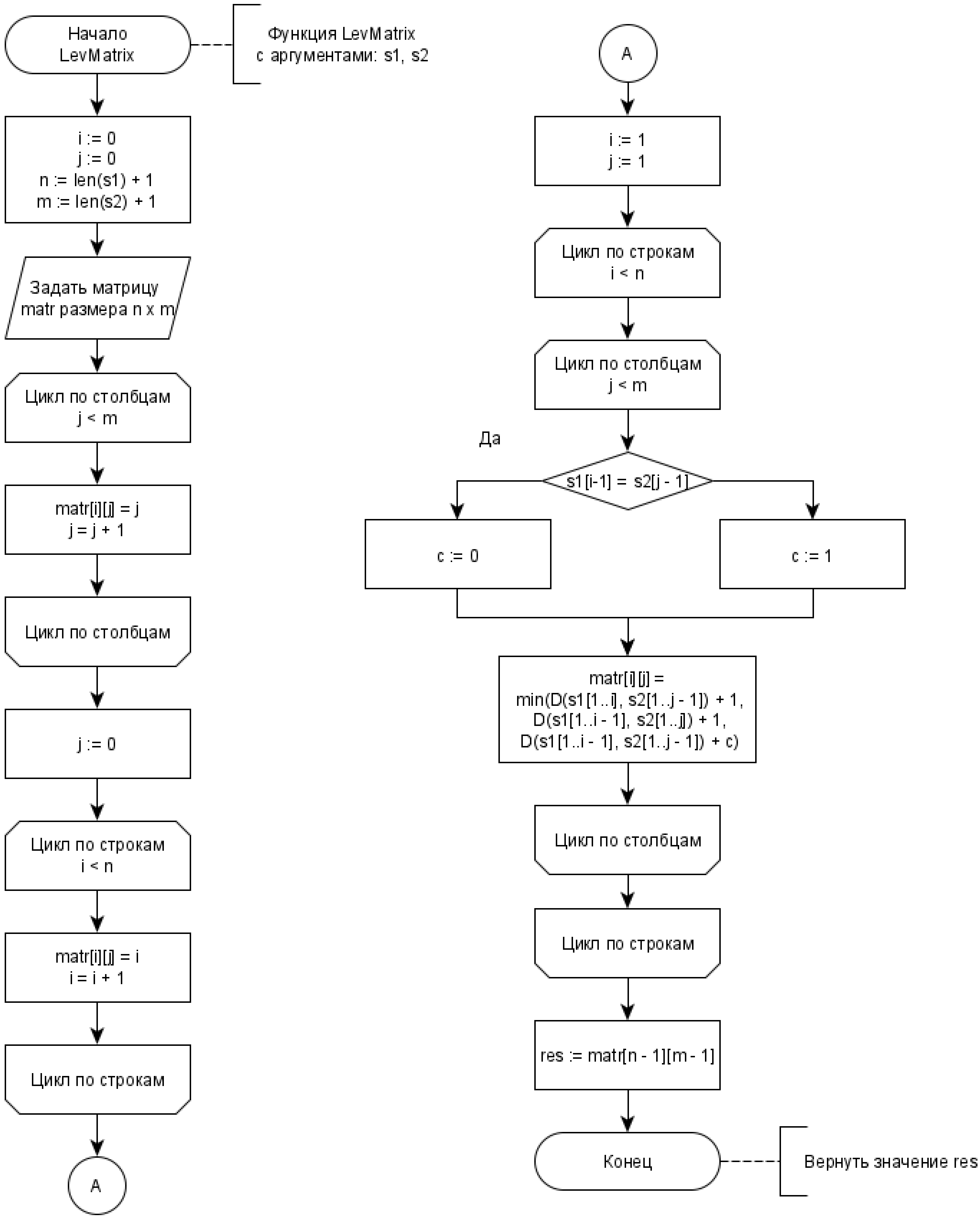

Рис. 2.1: Матричный алгоритм нахождения расстояния Левенштейна

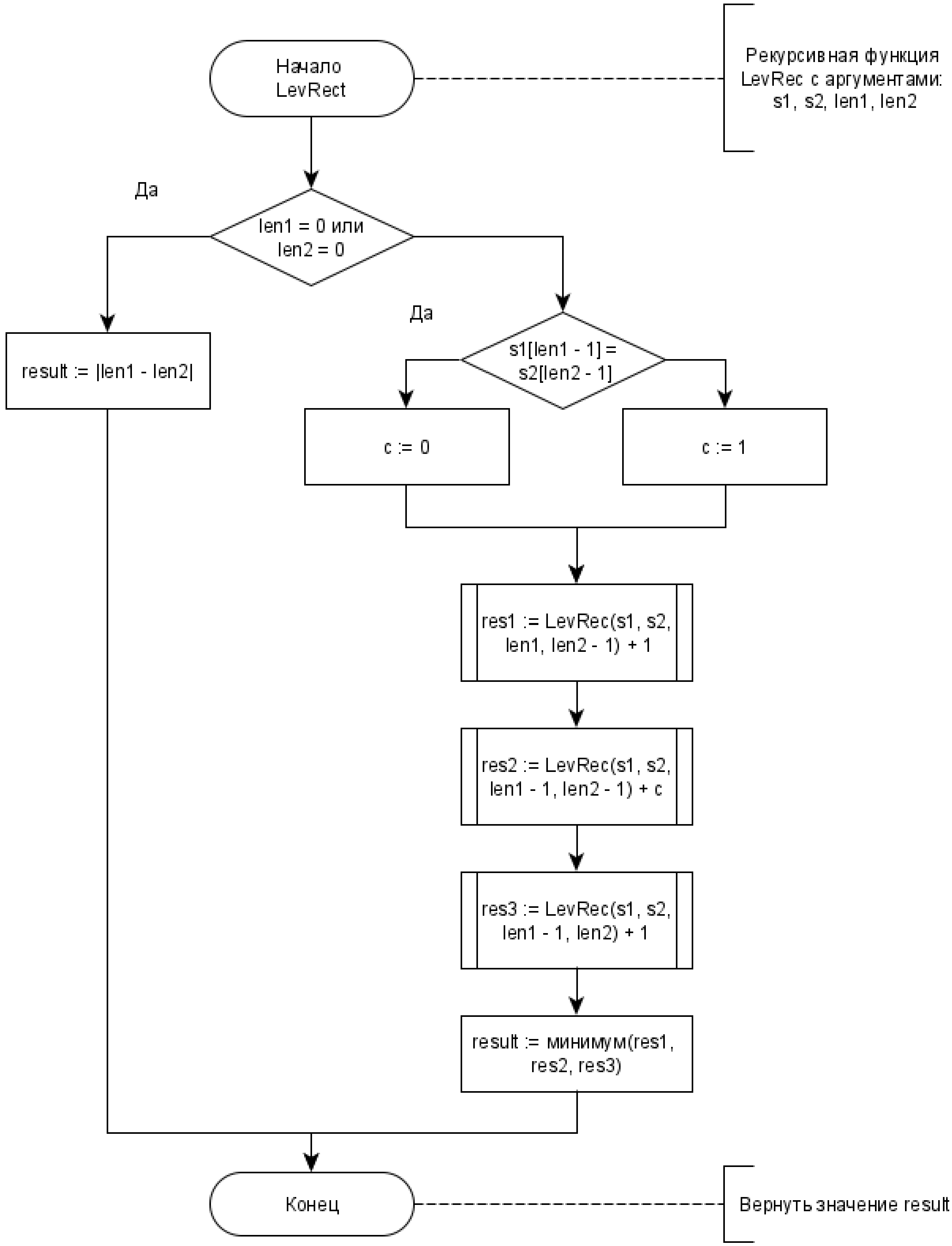

Рис. 2.2: Рекурсивный расчёт

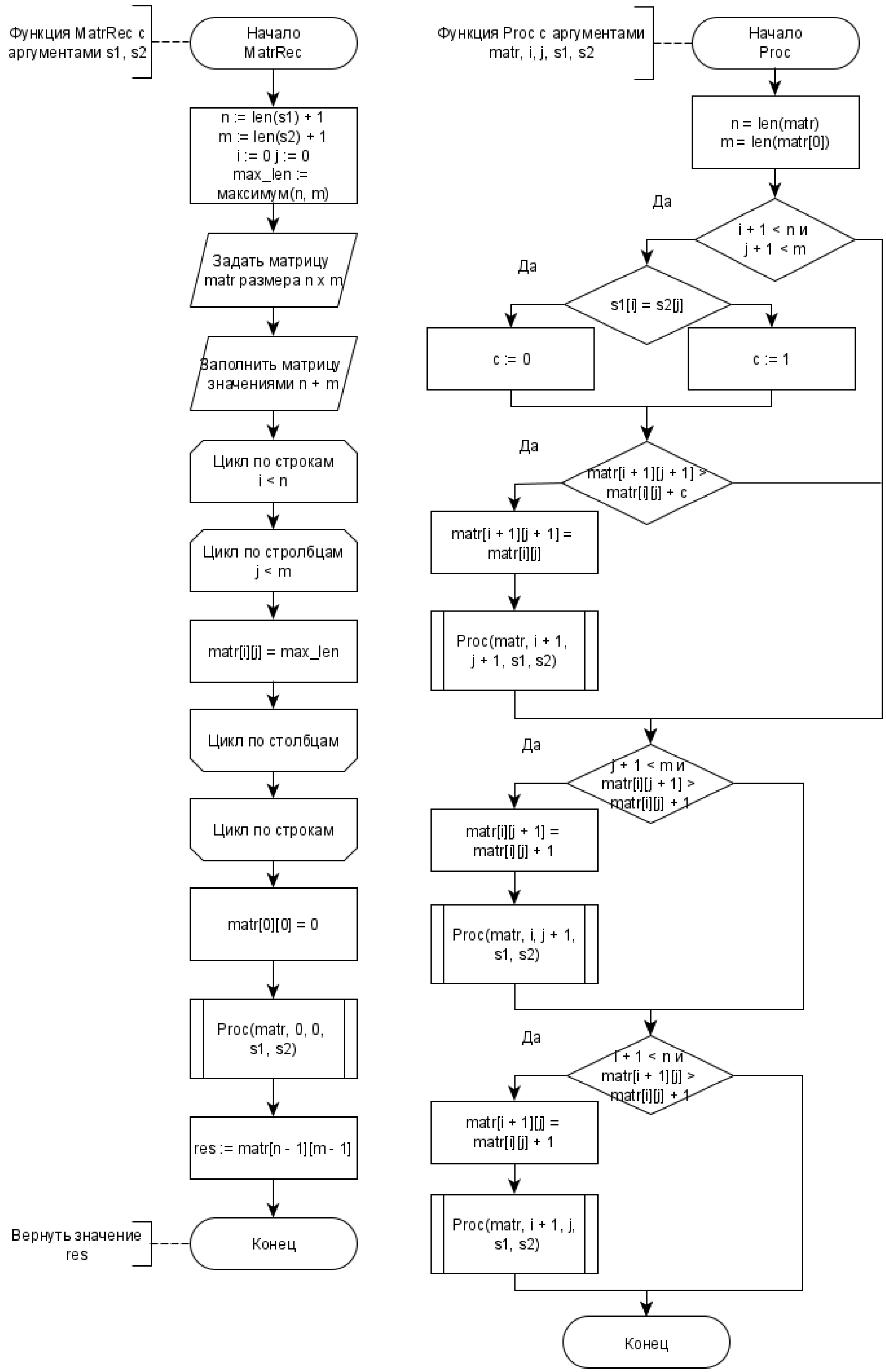

Рис. 2.3: Алгоритм, использующий рекурсию и матрицу

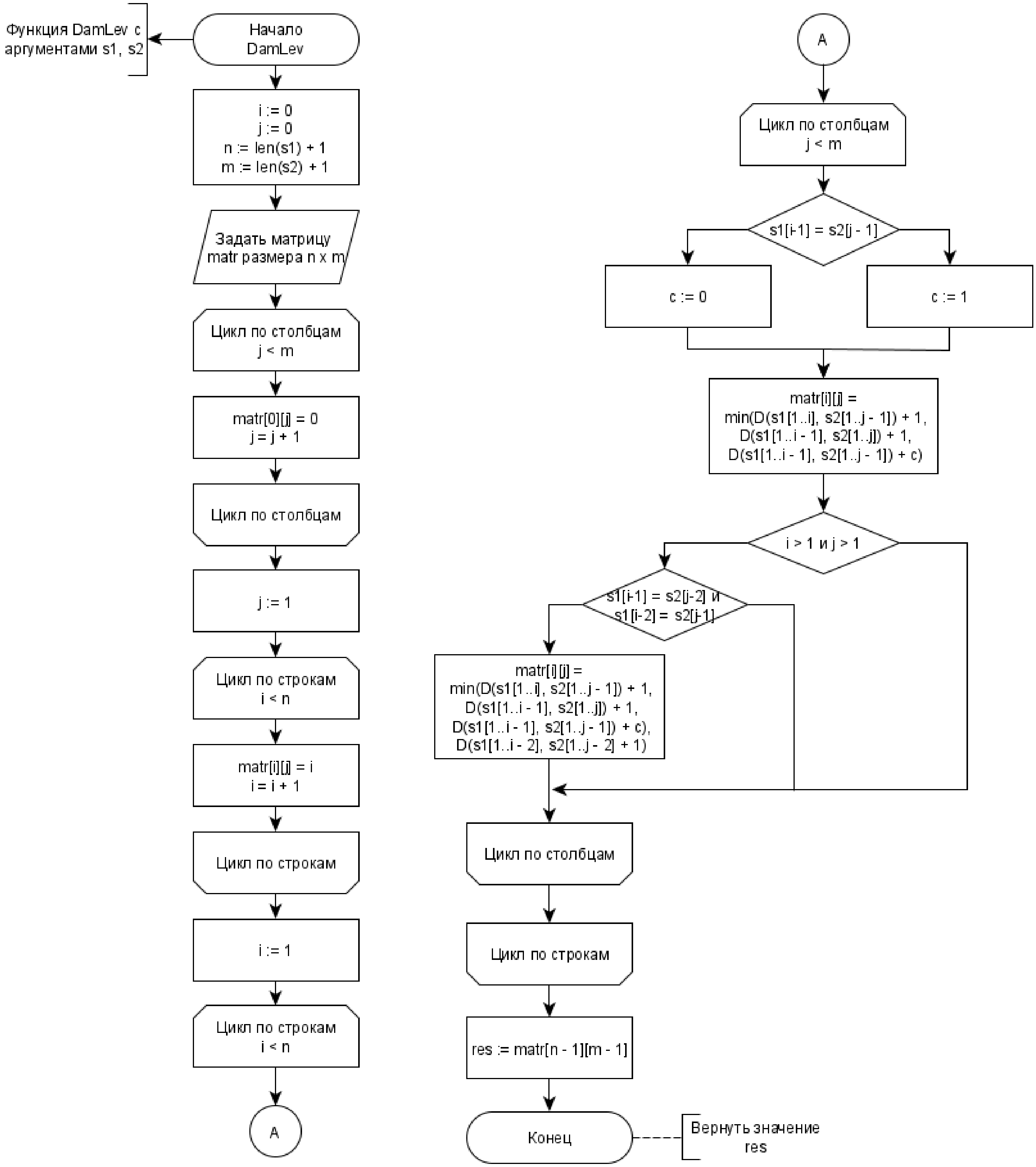

Рис. 2.4: Алгоритм нахождения расстояния Дамерау-Левенштейна

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 Инструменты замеров

3.3 Листинг

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)
```

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)
```

Листинг 3.4: Расстояние Дамерау-Левенштейна

```
    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)
```

4 Исследовательская часть

Заключение