Анализ Алгоритмов

Лабораторная работа №1

Юмаев Артур Русланович

ИУ7-55

Преподаватель: Волкова Л. Л.

Москва, 2019

Оглавление

[Введение 2](#_Toc22660516)

[1. Аналитическая часть 4](#_Toc22660517)

[Метод Левенштейна поиска минимального расстояния 4](#_Toc22660518)

[Расстояние методом Дамерау-Левенштейна 5](#_Toc22660519)

[2. Конструкторская часть 6](#_Toc22660520)

[3. Технологическая часть 12](#_Toc22660521)

[Поиска минимального расстояния методом Левенштейна матрично 12](#_Toc22660522)

[Поиск минимального расстояния методом Дамерау-Левенштейна матрично 13](#_Toc22660523)

[Поиск минимального расстояния методом Левенштейна рекурсивно 13](#_Toc22660524)

[Поиск минимального расстояния методом Дамерау-Левенштейна рекурсивно 14](#_Toc22660525)

[Сравнительный анализ потребляемой памяти 14](#_Toc22660526)

[4. Исследовательская часть 16](#_Toc22660527)

[Заключение 17](#_Toc22660528)

# Введение

Данная работа посвящена исследованию “Редакторского алгоритма” или алгоритма Левенштейна и его дополнения – алгоритма Дамерау-Левенштейна. Исследование является целью изучить математическую часть алгоритмов и запрограммировать их в нескольких вариантах (матричном и рекурсивном). Также будет проведена оценка алгоритмов по скорости работы и подготовлено их сравнение между собой.

# 1. Аналитическая часть

Ставится задача поиска минимального расстояния D(S1, S2) между двумя словами S1 и S2. M = |S1|, N = |S2|.

Метод Левенштейна поиска минимального расстояния

Метод Левенштейна поиска минимального расстояния заключается в поиске минимального количества операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Для расстояние Левенштейна рассмотрим граничные случаи:

D(char, 0) = 1 // Delete

D(0, char) = 1 // Insert

D(0, 0) = 0 // Match,

Где D – расстояние, char – любой символ

Расстояние по методу Левенштейна определяется как: D(S1[1…L(S1)], S2[1…L[S2]])

Общий случай для рекурсивного подхода:

D(S1[1…i], S2[1…j]) = 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]) + {0 if S1[i] = S2[j] else 1}

)

Общий случай для матричного подхода:

A = (L(S1) + 1) x (L(S2) + 1) – матрица

Таблица 1

Общий вид матрицы, используемой при матричной реализации

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | … | L(S1) |
| 1 | … | … | … |
| … | … | … | … |
| L(S2) | … | … | … |

Для i > 0, j > 0:

|  |  |
| --- | --- |
| X | Y |
| Z | Q |

Q = min(

Y + 1,

Z + 1,

X + {0 if S1[i] = S2[j] else 1}

)

## Расстояние методом Дамерау-Левенштейна

Алгоритм Дамерау-Левенштейна заключается в том, что к аргументам функции min добавляется еще одно условие:

D(S1[1…i], S2[1…j]) = 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]) + {0 if S1[i] = S2[j] else 1},

D(S1[1…i-2], S­­2[1…j-2]) + {1 if S1[i] = S2[j - 1] & S2[j] = S1[i - 1] else inf})

# 2. Конструкторская часть

В данном разделе будут приведены блок схемы для каждого из алгоритмов в разных реализациях (рекурсивно, матрично). На рис. 1 в части А показана блок схема заполнения нулевого столбца начальными значениями для дальнейшего расчета для столбцов и строк, начиная со второй позиции по порядку.

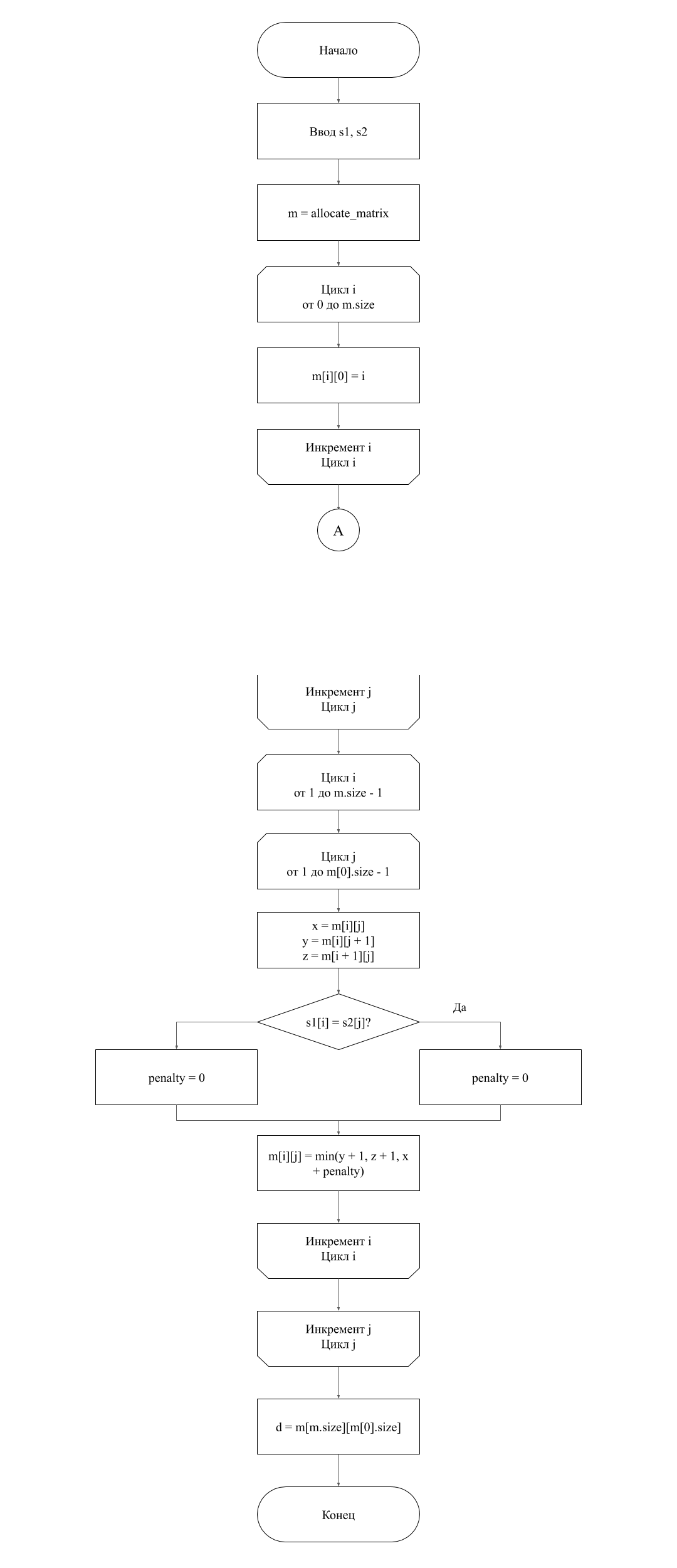


Рис. 1 – Схема алгоритма нахождения расстояния методом Левенштейна. Часть А

На рис. 2 показан поиск минимального расстояния и возврат позиции матрицы, в которой оно находится после конца работы алгоритма.

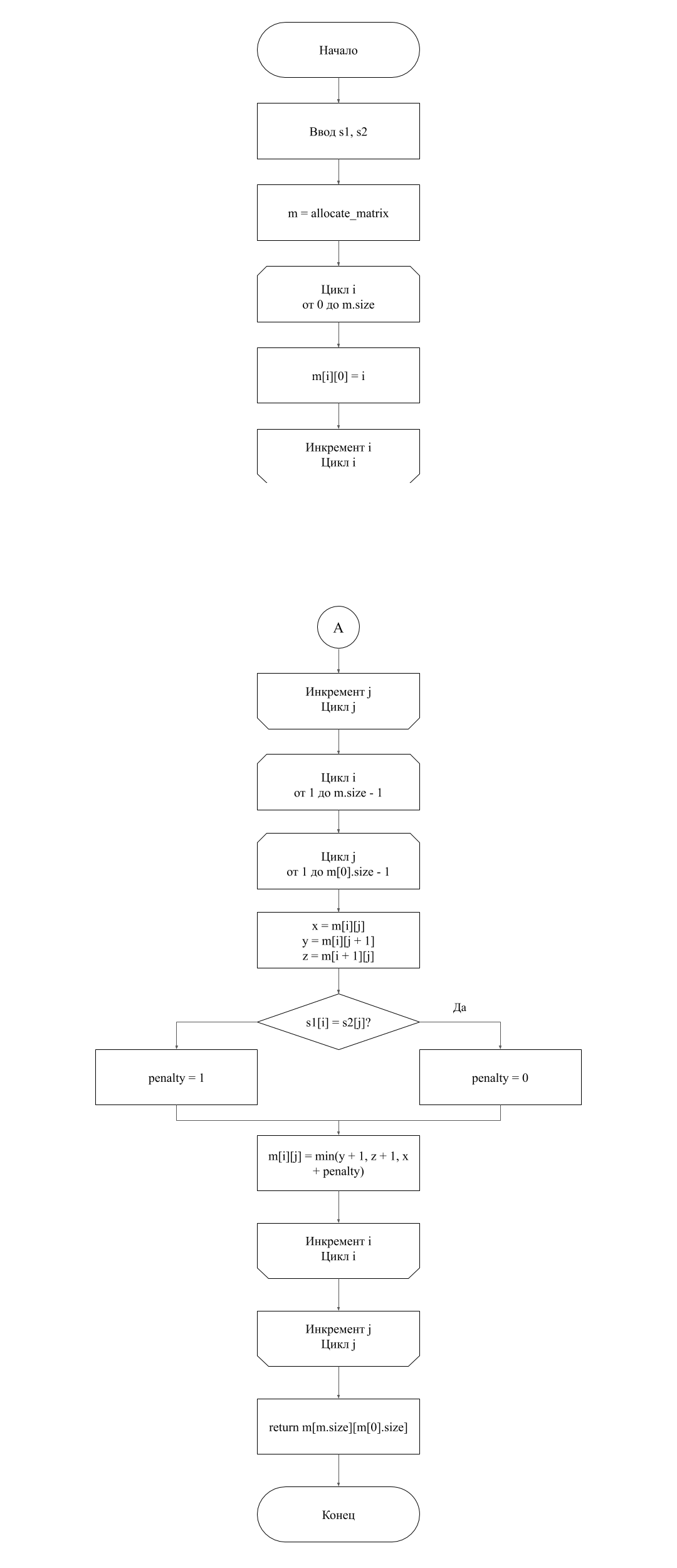


Рис. 2 – Схема алгоритма нахождения расстояния методом Левенштейна. Часть Б

На рис. 3 показана рекурсивная реализация поиска минимального расстояния методом Левенштейна.

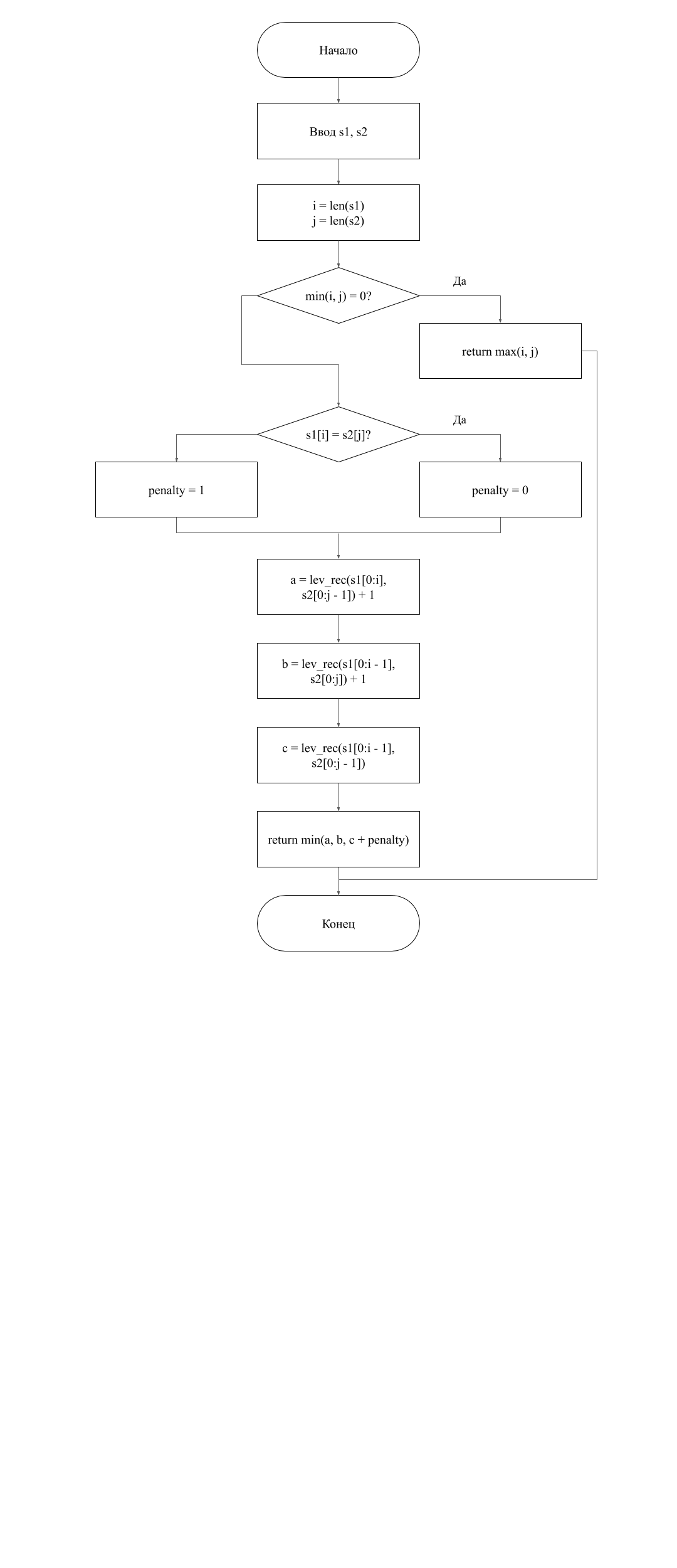


Рис 3. – Схема алгоритма поиска минимального расстояния методом Левенштейна. Рекурсивный подход.

На рис. 4 в части А показана схема поиска минимального расстояния методом Дамерау-Левенштейна. Заполнение матрицы начальными значениями.

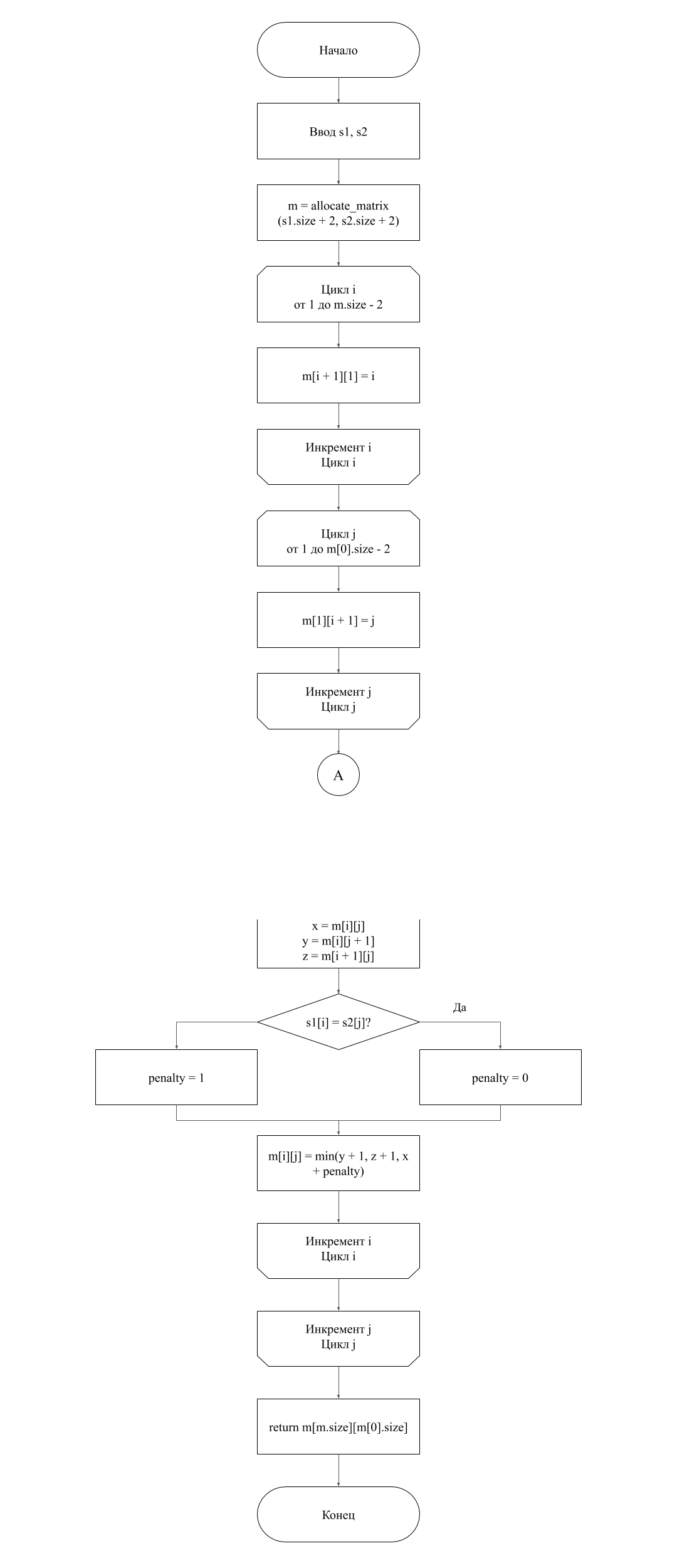


Рис. 4 Схема алгоритма поиска минимального расстояния методом Дамерау-Левенштейна матрично. Часть А.

На рис. 5 показано основное заполнение матрицы и сравнение двух соседних символом на возможность перестановки.

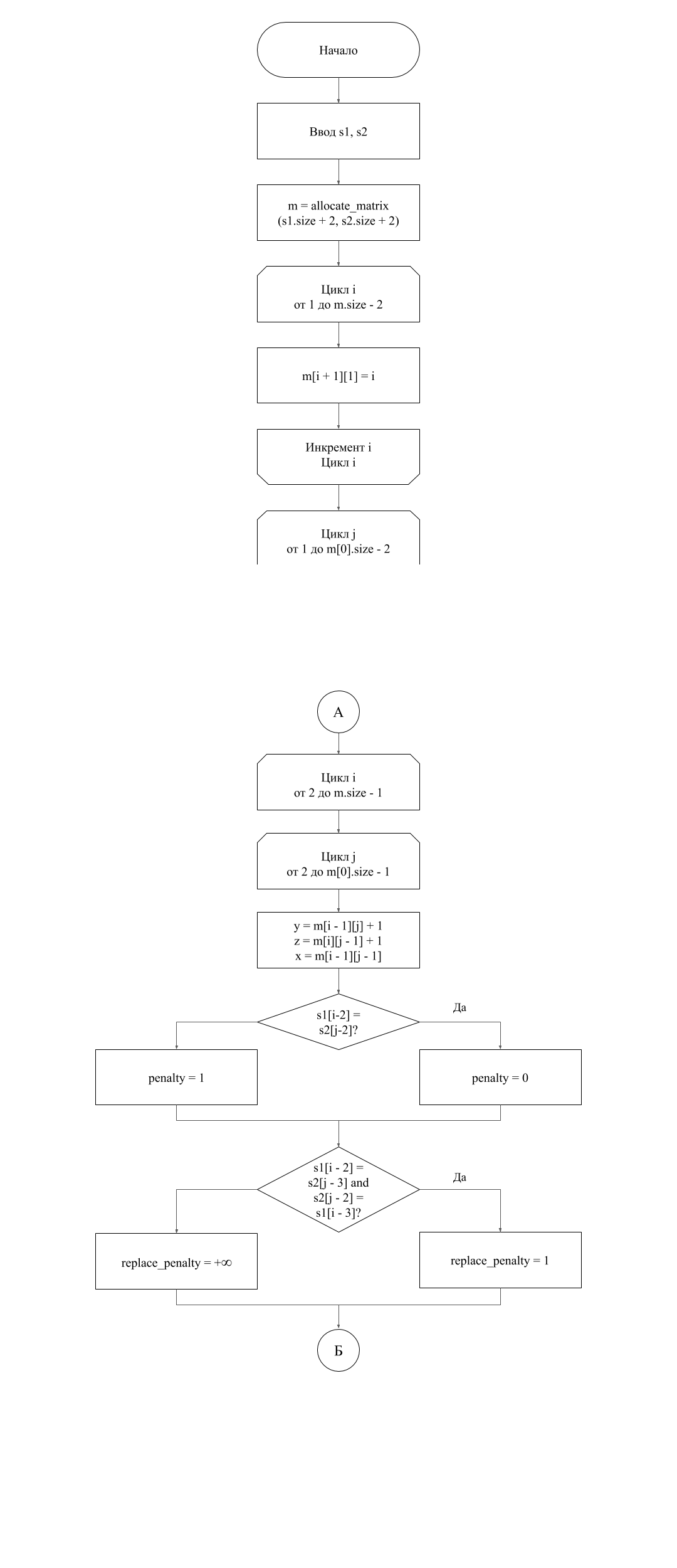


Рис. 5 – Схема алгоритма поиска минимального расстояния методом Дамерау-Левенштейна матрично. Часть Б.

Основное отличие от классического подхода Дамерау – увеличенная на одну строку и столбец матрица и добавление дополнительного аргумента в функцию min. Это показано на рис. 6.

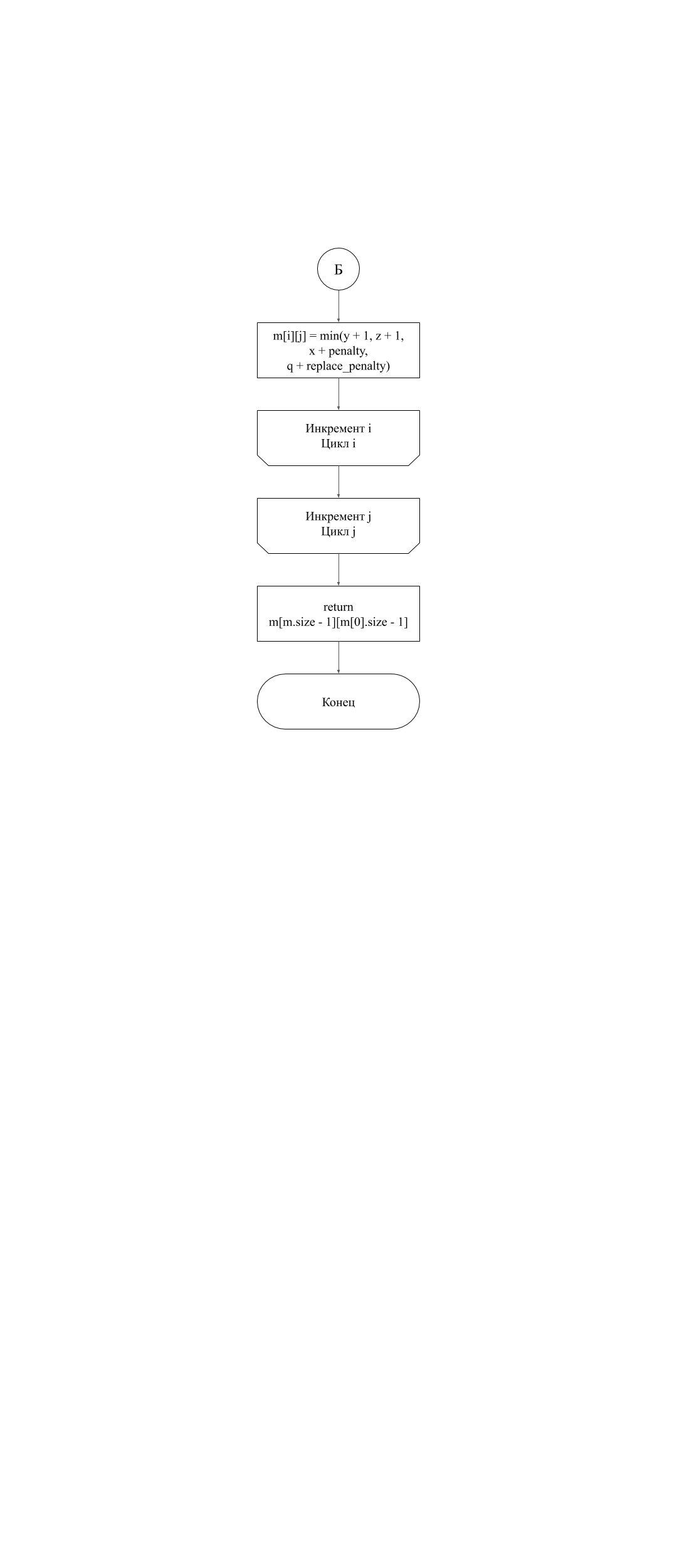


Рис. 6 - Схема алгоритма поиска минимального расстояния методом Дамерау-Левенштейна матрично. Часть В.

# 3. Технологическая часть

В качестве языка программирования был выбран Python из-за того, что для него существуют простые и быстрые библиотеки для работы с матрицами, которые упрощают процесс разработки. Библиотеки для работы с матрицами – Pandas и NumPy. Статистические данные отображаются с помощью библиотеки Matplotlib.

## Поиска минимального расстояния методом Левенштейна матрично

Листинг 1. Алгоритм Левенштейна

|  |
| --- |
| def levenshtein\_matrix(s1, s2, return\_matrix=False):  matrix = alloc\_matrix(s1, s2)  # 1. Simple cases  for i in range(matrix.shape[0]): # Fill the first column  matrix[i][0] = i  for i in range(matrix.shape[1]): # Fill the first row  matrix[0][i] = i  # 2. i > 0, j > 0  for i in range(1, matrix.shape[0]):  for j in range(1, matrix.shape[1]):  x = matrix[i - 1][j - 1]  y = matrix[i - 1][j]  z = matrix[i][j - 1]  matrix[i][j] = min(y + 1,  z + 1,  x + (0 if s1[i - 1] == s2[j - 1]  else 1))  distance = matrix[matrix.shape[0] - 1][matrix.shape[1] - 1]    if return\_matrix:  return matrix, distance  else:  return distance |

## Поиск минимального расстояния методом Дамерау-Левенштейна матрично

Листинг 2. Алгоритм Дамерау-Левенштейна

|  |
| --- |
| def domerau\_levenshtein\_matrix(s1, s2, return\_matrix=False):  matrix = np.zeros((len(s1) + 2, len(s2) + 2))  # 1. Simple cases  for i in range(1, matrix.shape[0] - 1):  matrix[i + 1][1] = i  for i in range(1, matrix.shape[1] - 1):  matrix[1][i + 1] = i  for i in range(2, matrix.shape[0]):  for j in range(2, matrix.shape[1]):  y = matrix[i - 1][j] + 1  z = matrix[i][j - 1] + 1  x = matrix[i - 1][j - 1] + \  (0 if s1[i - 2] == s2[j - 2] else 1)  q = matrix[i - 2][j - 2] + \  (1 if s1[i - 2] == s2[j - 3] and  s2[j - 2] == s1[i - 3]  else np.inf)  matrix[i][j] = min(y, z, x, q)  distance = matrix[matrix.shape[0] - 1][matrix.shape[1] - 1]  if return\_matrix:  return matrix, distance  else:  return distance |

## Поиск минимального расстояния методом Левенштейна рекурсивно

Листинг 3. Рекурсивный алгоритм Левенштейна

|  |
| --- |
| def levenshtein\_rec(s1, s2):  i, j = len(s1), len(s2)  if min(i, j) == 0:  return max(i, j)  return min(levenshtein\_rec(s1[0:i], s2[0:j - 1]) + 1,  levenshtein\_rec(s1[0:i - 1], s2[0:j]) + 1,  levenshtein\_rec(s1[0:i - 1], s2[0:j - 1]) + \  (0 if s1[i - 1] == s2[j - 1] else 1)) |

## Поиск минимального расстояния методом Дамерау-Левенштейна рекурсивно

Листинг 4. Рекурсивный алгоритм Дамерау-Левенштейна

|  |
| --- |
| def domerau\_levenshtein\_rec(s1, s2):  i = len(s1)  j = len(s2)  if min(i, j) == 0:  return max(i, j)  elif (i > 1 and j > 1 and s1[i - 1] == s2[j - 2] and s1[i - 2] == s2[j - 1]):  return min(  domerau\_levenshtein\_rec(s1[:i - 1], s2[:j]) + 1,  domerau\_levenshtein\_rec(s1[:i], s2[:j - 1]) + 1,  domerau\_levenshtein\_rec(s1[:i - 2], s2[:j - 2]) + 1,  domerau\_levenshtein\_rec(s1[:i - 1], s2[:j - 1]) + \  (0 if s1[i - 1] == s2[j - 1] else 1))  else:  return min(  domerau\_levenshtein\_rec(s1[:i - 1], s2[:j]) + 1,  domerau\_levenshtein\_rec(s1[:i], s2[:j - 1]) + 1,  domerau\_levenshtein\_rec(s1[:i - 1], s2[:j - 1]) + \  (0 if s1[i - 1] == s2[j - 1] else 1)) |

## Сравнительный анализ потребляемой памяти

Сравнение будет проводится для языка Python 3.7. Данный ЯП имеет свои особенности хранения объектов в памяти. Типы данных являются классами Python, соответственно каждый объект является экземпляром некого класса, а не просто ячейкой в памяти, что естественным образом увеличивает затраты памяти.

Таблица 2

Количество памяти, занимаемое экземплярами классов в Python 3.7

|  |  |  |
| --- | --- | --- |
| **Класс** | **Представление в коде** | **Занимаемая память (байт)** |
| Пустой список | [ ] | 36 |
| Непустой список | [obj\_1, …, obj\_N] | 36 + 4 \* N |
| Пустой массив типа char | “ ” | 25 |
| Непустой массив типа char | “ABC” | 25 + 1 \* N, N – длина строки |
| Пустая функция | def f( ): pass | 72 |
| Переменная типа int | n = int() | 24 |

Таблица 3

Потребление памяти, занимаемое алгоритмом Левенштейна

|  |  |
| --- | --- |
| **Структура данных** | **Занимаемая память (байт)** |
| Матрица | 40 + (len(s1) + 1) \* (len(s2) + 1) |
| Счетчики цикла типа int | 24 = 2 \* 12 |
| Передача локальных параметров | 25 \* 2 + len(s1) + len(s2) |
| Переменные типа int | 3 \* 24 |
| Итого | 156 + M + N + (M + 1) \* (N + 1), где  M = len(s1), N = len(s2) |

Таблица 2. Количество памяти, занимаемое алгоритмом Левенштейна

Таблица 4

Потребление памяти, занимаемое алгоритмом Дамерау-Левенштейна

|  |  |
| --- | --- |
| **Структура данных** | **Занимаемая память (байт)** |
| Матрица | 40 + (len(s1) + 2) \* (len(s2) + 2) |
| Счетчики цикла типа int | 24 = 2 \* 12 |
| Передача локальных параметров | 25 \* 2 + len(s1) + len(s2) |
| Переменные типа int | 4 \* 24 |
| Итого | 180 + M + N + (M + 2) \* (N + 2), где  M = len(s1), N = len(s2) |

Таблица 5

Количество памяти, занимаемое алгоритмом Дамерау-Левенштейна (рекурсивным)

|  |  |
| --- | --- |
| Структура данных | Занимаемая память (байт) |
| Передача локальных параметров | 25 \* 2 + len(s1) + len(s2) |
| Пять переменных для  подсчета IDRT (I -  insert, D - delete, R -  replace, T – transposing) | 125 |
| Итого (худший случай) | (size(s1) + size(s2)) \*  (175 + len(s1) + len(s2)) |

# 4. Исследовательская часть

В данном разделе будут проведены опыты по замеру времени работы алгоритмов Дамерау и Дамерау-Левенштейна. На рис. 1 приведено сравнение алгоритмов Левенштейна и Дамерау-Левенштейна по времени работы в зависимости от длины слов. Опыт показал, что алгоритм Дамерау-Левенштейна работает медленнее, что было логично, из-за дополнительной операции сравнения букв на неверный порядок. Для каждой пары слов опыт проводился 100 раз и далее считалось среднее время, затраченное на поиск расстояния для каждой конкретной пары. Весь опыт занял около 50 минут.

Замер времени проводился с помощью библиотеки time в Python 3.7 и метода process\_time().

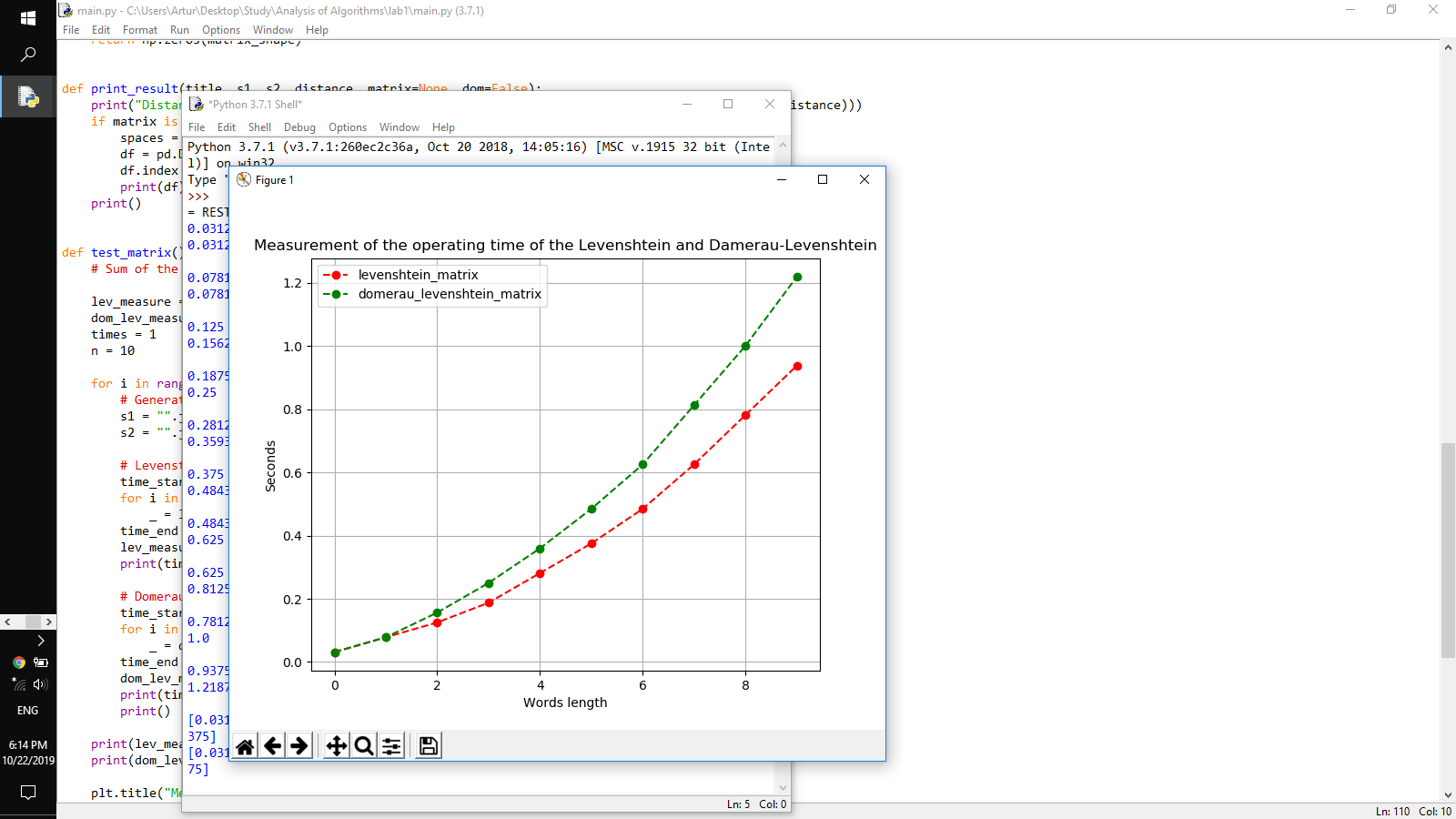


Рис. 1 – Сравнение времени работы алгоритмов Левенштейна и Дамерау-Левенштейна при матричном подходе

Как видно на рис. 1, алгоритм Дамерау-Левенштейна показал неэффективность по времени при росте количества букв в сравниваемых словах.

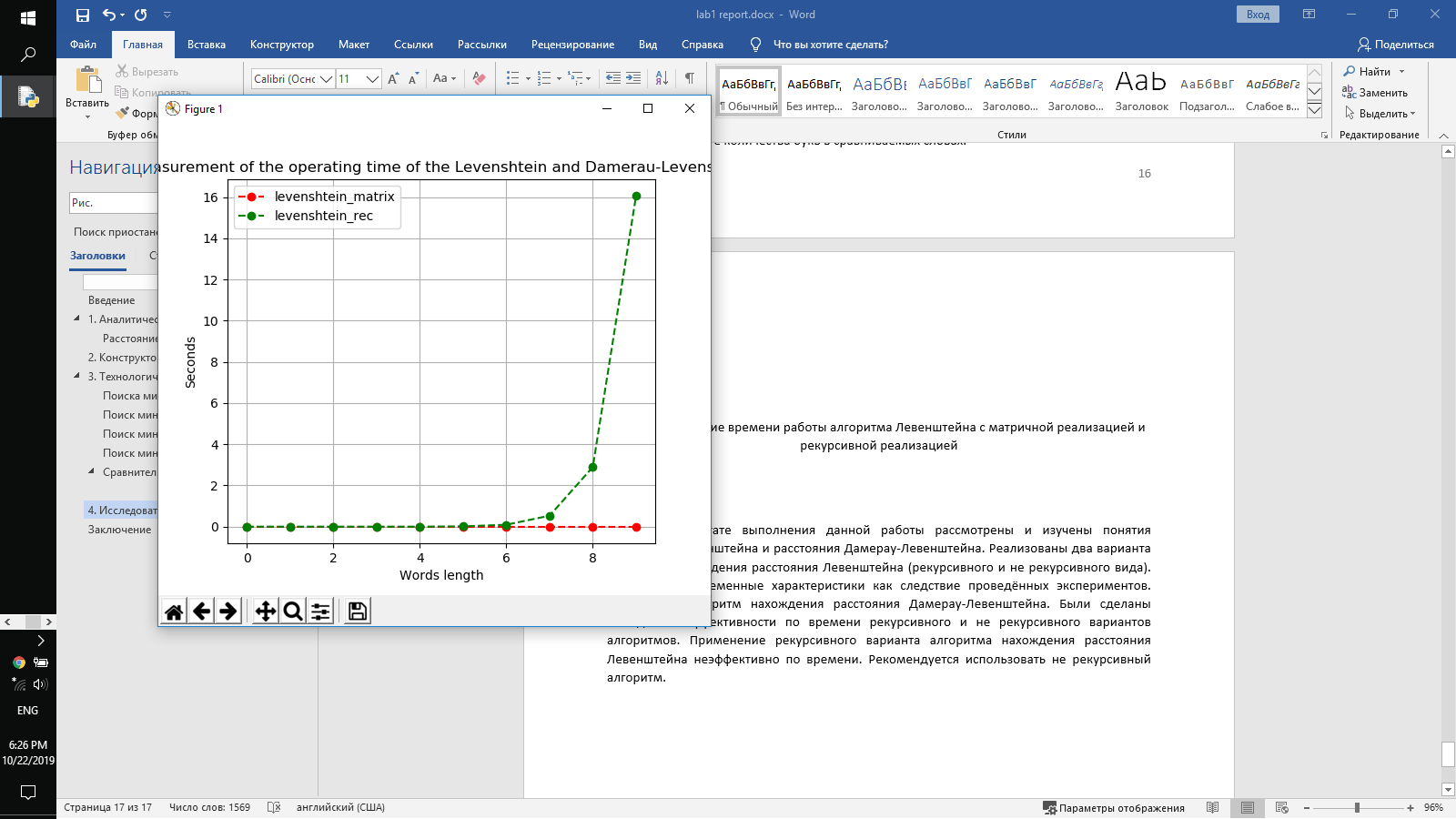


Рис. 2 – Сравнение времени работы алгоритма Левенштейна с матричной реализацией и рекурсивной реализацией

Как показал проведенный опыт, алгоритм Левенштейна, реализованный рекурсивно показал крайнюю неэффективность по сравнению с матричным подходом. Сравнение по времени показано на рис. 2.

# Заключение

В результате выполнения данной работы рассмотрены и изучены понятия расстояния Левенштейна и расстояния Дамерау-Левенштейна. Реализованы два варианта алгоритма нахождения расстояния Левенштейна (рекурсивного и не рекурсивного вида). Сравнены их временные характеристики как следствие проведённых экспериментов. Реализован алгоритм нахождения расстояния Дамерау-Левенштейна. Были сделаны выводы об эффективности по времени рекурсивного и не рекурсивного вариантов алгоритмов. Применение рекурсивного варианта алгоритма нахождения расстояния Левенштейна неэффективно по времени. Рекомендуется использовать не рекурсивный алгоритм.