Анализ Алгоритмов

Лабораторная работа №6

На тему “*Муравьиный алгоритм и задача коммивояжера*”

Студент: Юмаев Артур

Группа: ИУ7-55

Оглавление

[Введение 3](#_Toc26972206)

[1. Аналитическая часть 4](#_Toc26972207)

[Метод Левенштейна поиска минимального расстояния 4](#_Toc26972208)

[Расстояние методом Дамерау-Левенштейна 5](#_Toc26972209)

[2. Конструкторская часть 6](#_Toc26972210)

[3. Технологическая часть 12](#_Toc26972211)

[Поиск минимального расстояния методом Левенштейна матрично 12](#_Toc26972212)

[Поиск минимального расстояния методом Дамерау-Левенштейна матрично. 13](#_Toc26972213)

[Поиск минимального расстояния методом Левенштейна рекурсивно. 13](#_Toc26972214)

[Поиск минимального расстояния методом Дамерау-Левенштейна рекурсивно. 14](#_Toc26972215)

[Сравнительный анализ потребляемой памяти 14](#_Toc26972216)

[4. Исследовательская часть 16](#_Toc26972217)

[Заключение 18](#_Toc26972218)

[Литература 19](#_Toc26972219)

# Введение

Муравьиный алгоритм (алгоритм оптимизации подражанием муравьиной колонии, англ. ant colony optimization, ACO) — один из эффективных полиномиальных алгоритмов для нахождения приближённых решений задачи коммивояжёра, а также решения аналогичных задач поиска маршрутов на графах. Суть подхода заключается в анализе и использовании модели поведения муравьёв, ищущих пути от колонии к источнику питания. Первая версия алгоритма, предложенная доктором наук Марко Дориго[1][2] в 1992 году, была направлена на поиск оптимального пути в графе.

# 1. Аналитическая часть

В данном разделе будет дано полное описание муравьиного алгоритма и его математическое описание.

## Введение

В реальном мире муравьи (первоначально) ходят в случайном порядке и по нахождении продовольствия возвращаются в свою колонию, прокладывая феромонами тропы. Если другие муравьи находят такие тропы, они, вероятнее всего, пойдут по ним. Вместо того, чтобы отслеживать цепочку, они укрепляют её при возвращении, если в конечном итоге находят источник питания. Со временем феромонная тропа начинает испаряться, тем самым уменьшая свою привлекательную силу. Чем больше времени требуется для прохождения пути до цели и обратно, тем сильнее испарится феромонная тропа.

# 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.

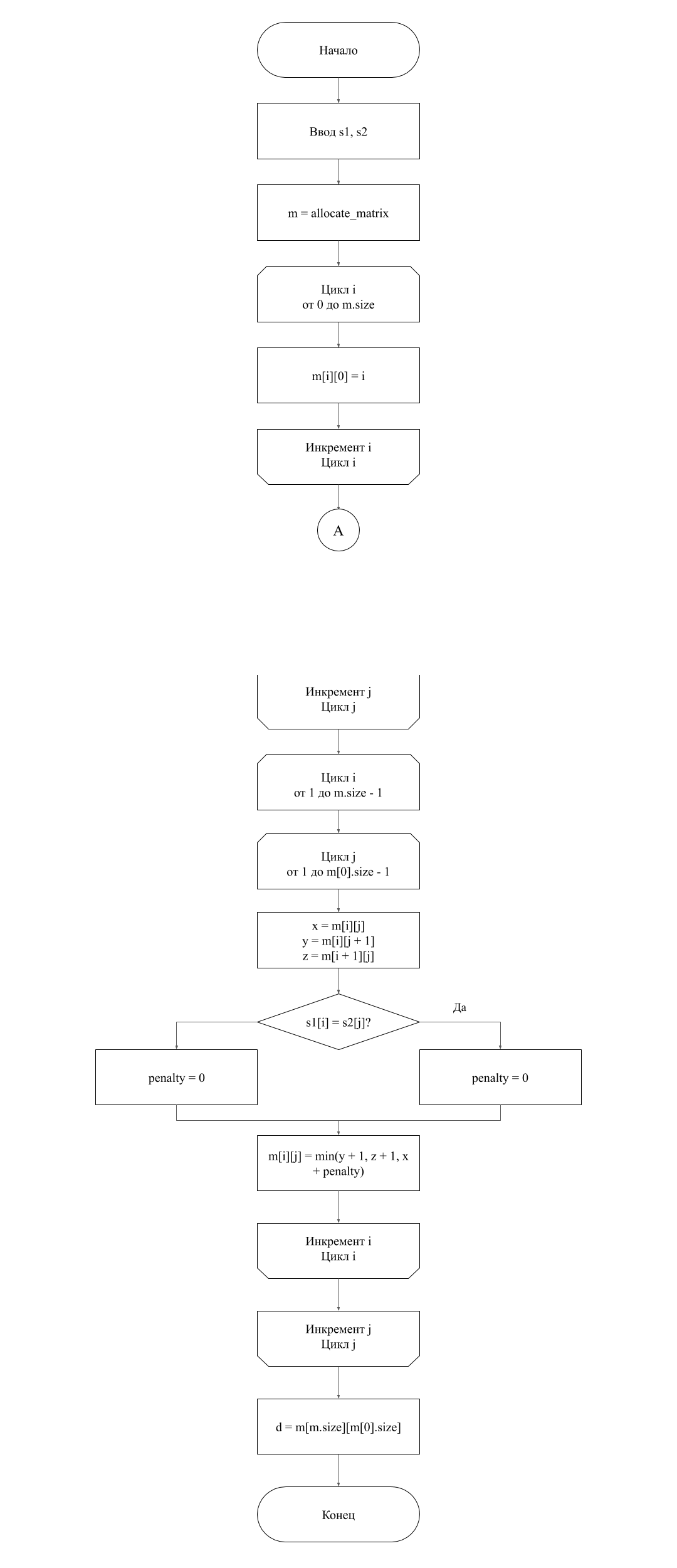


Рис. 1 – Схема алгоритма нахождения расстояния методом Левенштейна. Часть А

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

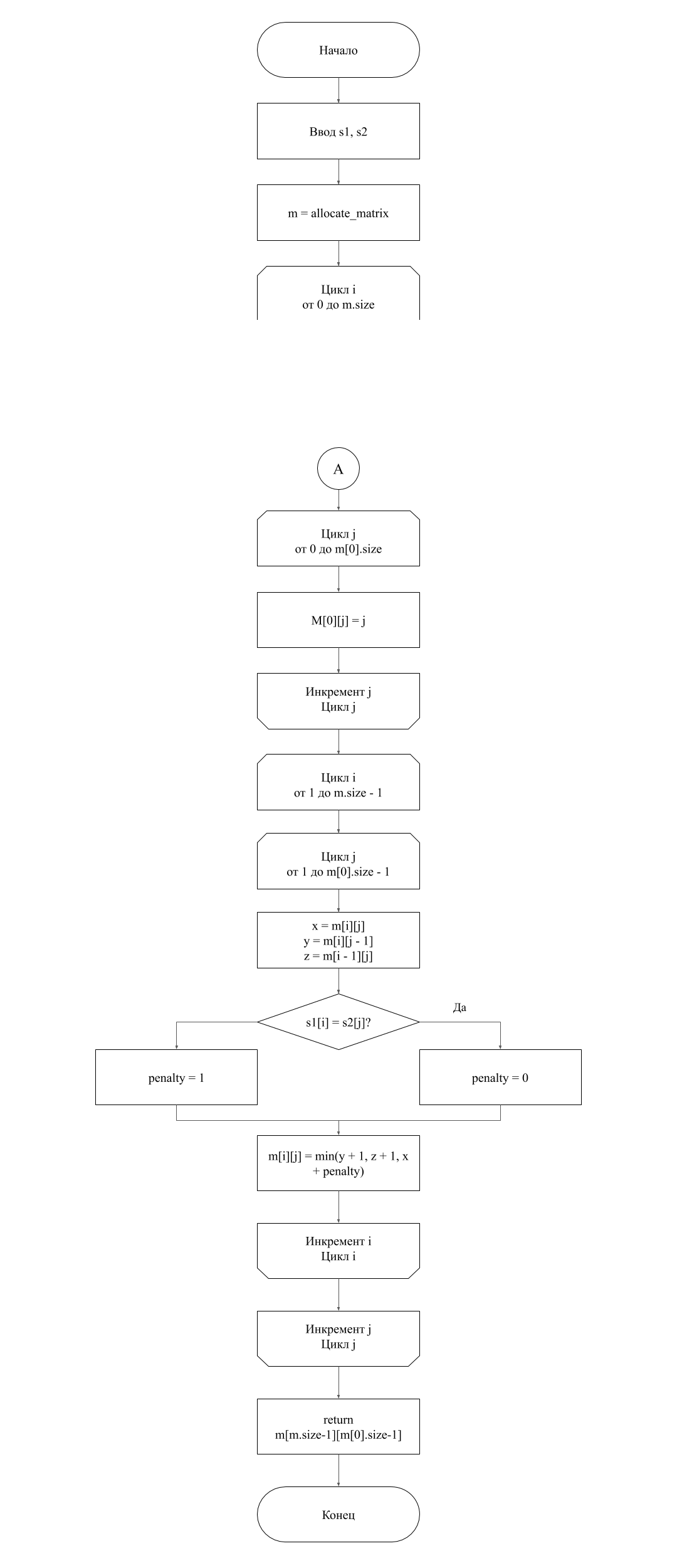


Рис. 2 – Схема алгоритма нахождения расстояния методом Левенштейна. Часть Б

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

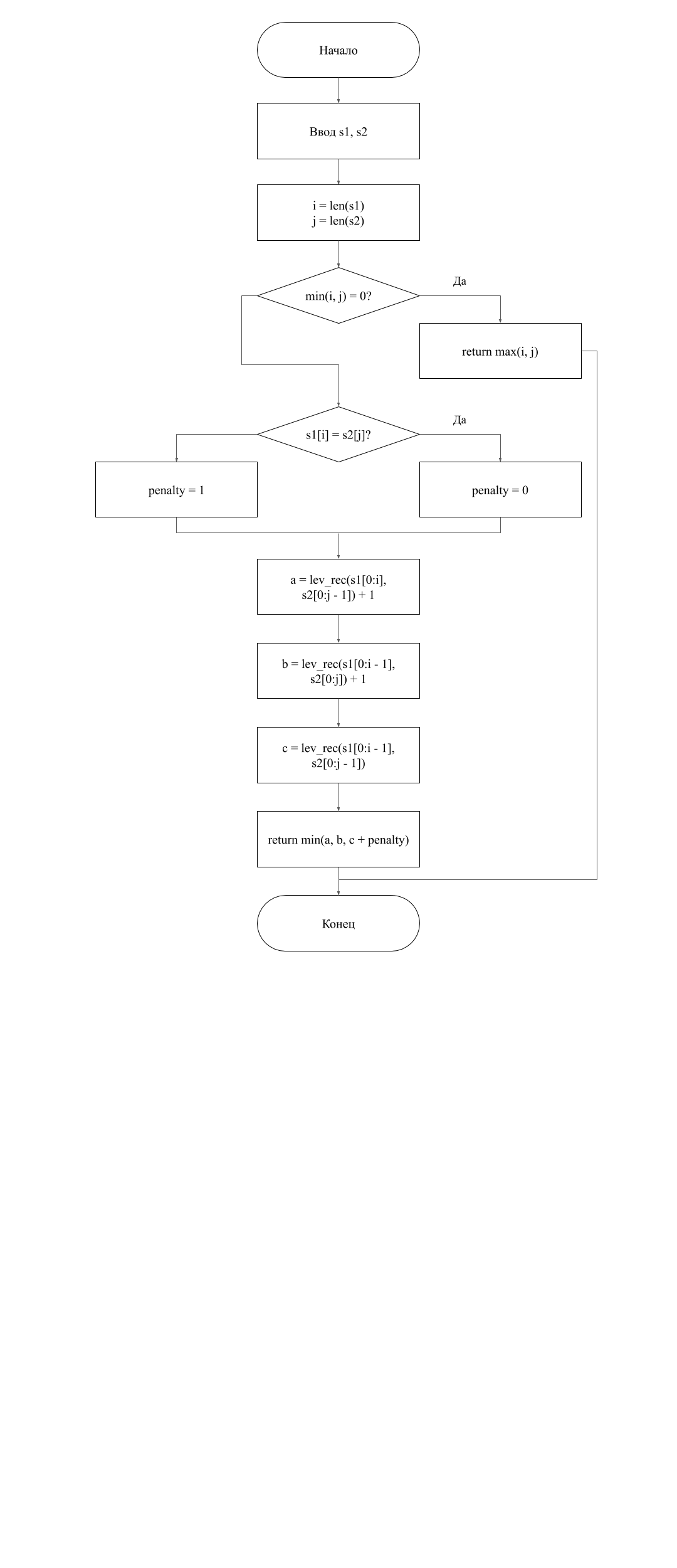


Рис 3. – Схема алгоритма поиска минимального расстояния методом Левенштейна. Рекурсивный подход.

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

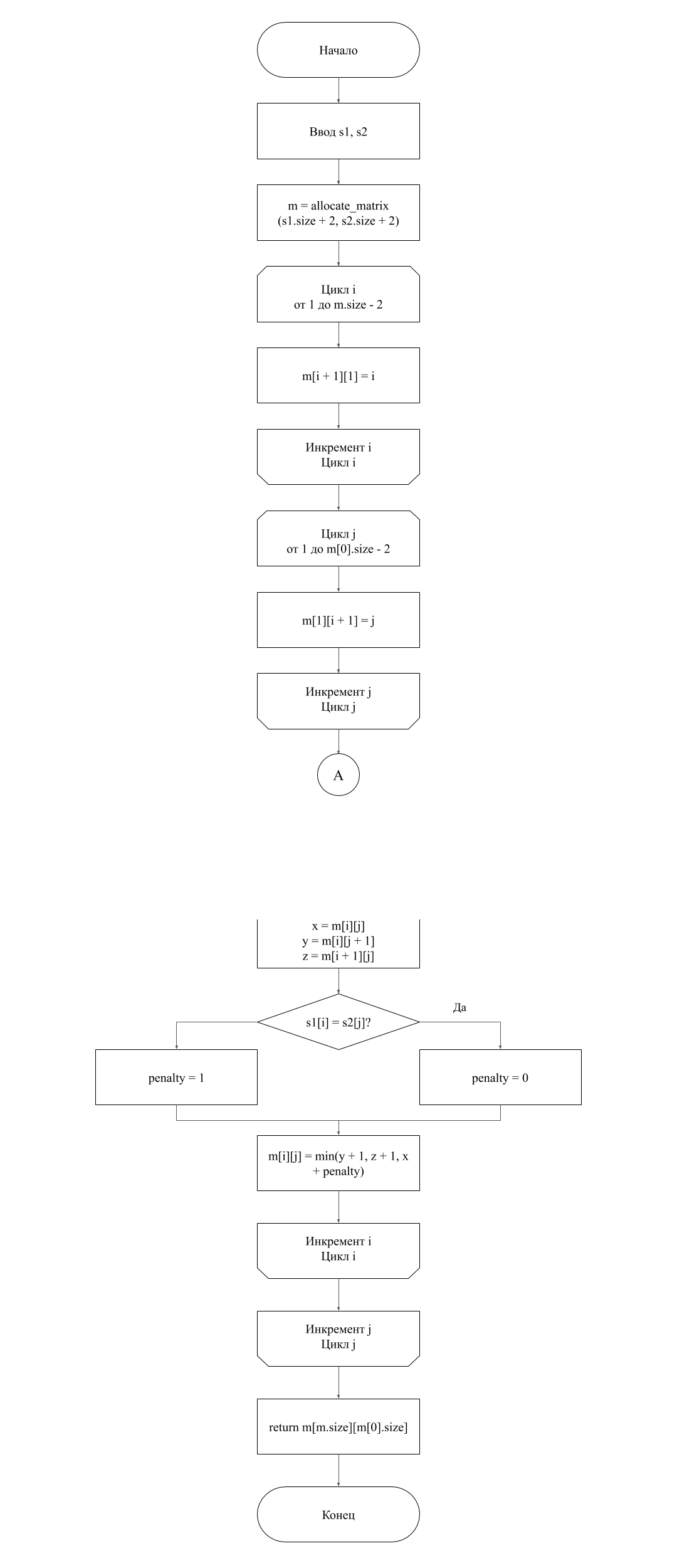


Рис. 4 Схема алгоритма поиска минимального расстояния методом Дамерау-Левенштейна матрично. Часть А.

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

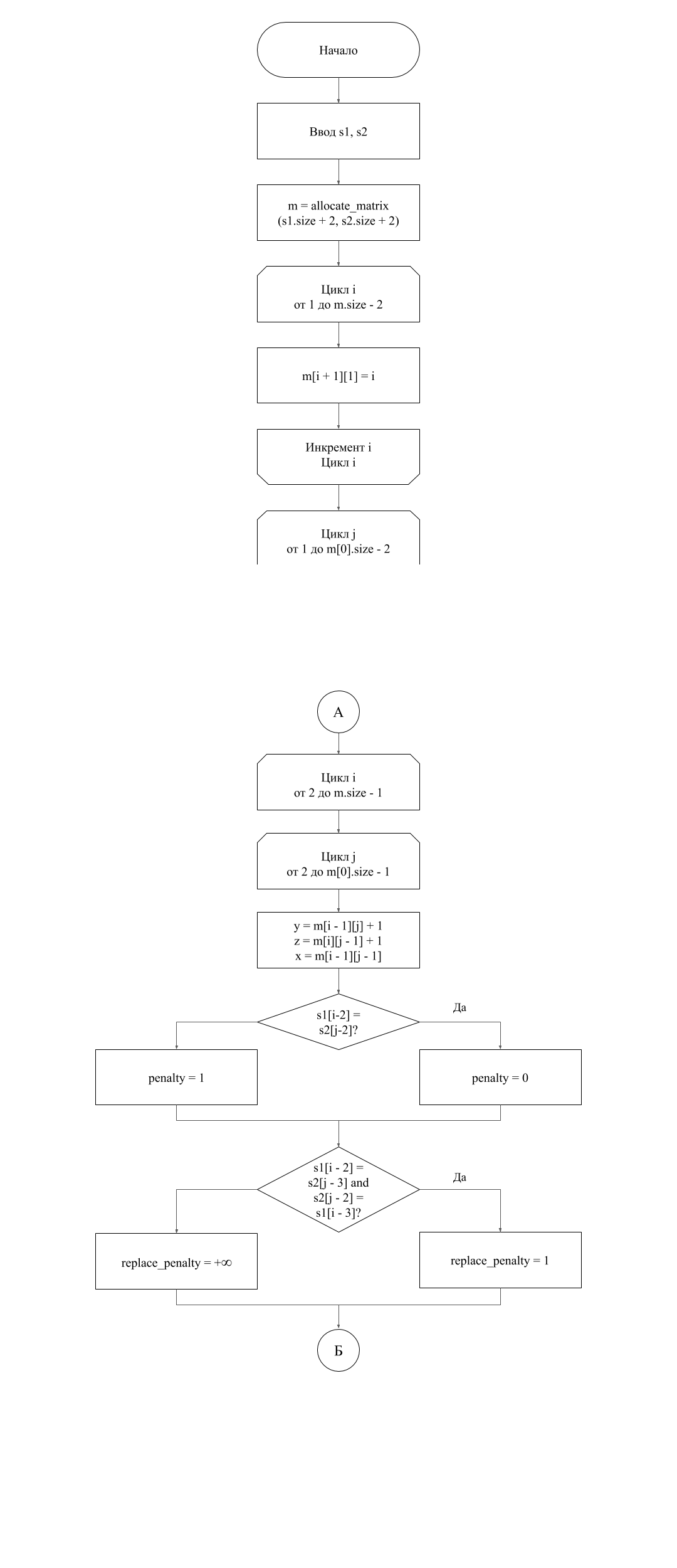


Рис. 5 – Схема алгоритма поиска минимального расстояния методом Дамерау-Левенштейна матрично. Часть Б.

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.

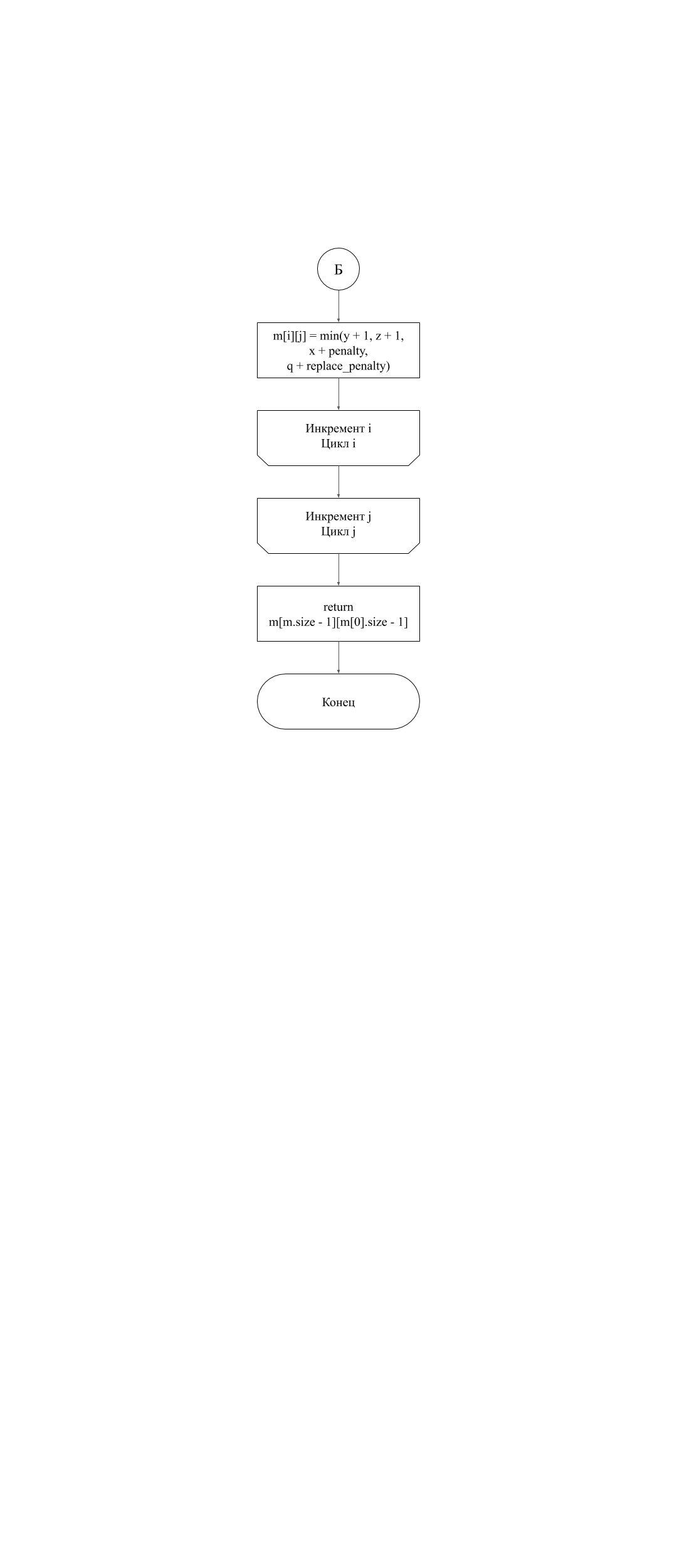


Рис. 6 - Схема алгоритма поиска минимального расстояния методом Дамерау-Левенштейна матрично. Часть В.

# 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.

## Поиск минимального расстояния методом Левенштейна матрично

Листинг 1. Алгоритм Левенштейна

|  |
| --- |
| 1. def levenshtein\_matrix(s1, s2, return\_matrix=False): 2. matrix = alloc\_matrix(s1, s2) 3. # 1. Simple cases 4. for i in range(matrix.shape[0]): # Fill the first column 5. matrix[i][0] = i 6. for i in range(matrix.shape[1]): # Fill the first row 7. matrix[0][i] = i 8. # 2. i > 0, j > 0 9. for i in range(1, matrix.shape[0]): 10. for j in range(1, matrix.shape[1]): 11. x = matrix[i - 1][j - 1] 12. y = matrix[i - 1][j] 13. z = matrix[i][j - 1] 14. matrix[i][j] = min(y + 1, z + 1, 15. x + (0 if s1[i - 1] == s2[j - 1] else 1)) 16. distance = matrix[matrix.shape[0] - 1][matrix.shape[1] - 1] 17. if return\_matrix: 18. return matrix, distance 19. else: 20. return distance |

## Поиск минимального расстояния методом Дамерау-Левенштейна матрично.

Листинг 2. Алгоритм Дамерау-Левенштейна

|  |
| --- |
| 1. def domerau\_levenshtein\_matrix(s1, s2, return\_matrix=False): 2. matrix = np.zeros((len(s1) + 2, len(s2) + 2)) 3. # 1. Simple cases 4. for i in range(1, matrix.shape[0] - 1): 5. matrix[i + 1][1] = i 6. for i in range(1, matrix.shape[1] - 1): 7. matrix[1][i + 1] = i 8. for i in range(2, matrix.shape[0]): 9. for j in range(2, matrix.shape[1]): 10. y = matrix[i - 1][j] + 1 11. z = matrix[i][j - 1] + 1 12. x = matrix[i - 1][j - 1] + \ 13. (0 if s1[i - 2] == s2[j - 2] else 1) 14. q = matrix[i - 2][j - 2] + \ 15. (1 if s1[i - 2] == s2[j - 3] and \ 16. s2[j - 2] == s1[i – 3] else np.inf) 17. matrix[i][j] = min(y, z, x, q) 18. distance = matrix[matrix.shape[0] - 1][matrix.shape[1] - 1] 19. if return\_matrix: 20. return matrix, distance 21. else: 22. return distance |

## Поиск минимального расстояния методом Левенштейна рекурсивно.

Листинг 3. Рекурсивный алгоритм Левенштейна

|  |
| --- |
| 1. def levenshtein\_rec(s1, s2): 2. i, j = len(s1), len(s2) 3. if min(i, j) == 0: 4. return max(i, j) 5. return min(levenshtein\_rec(s1[0:i], s2[0:j - 1]) + 1, 6. levenshtein\_rec(s1[0:i - 1], s2[0:j]) + 1, 7. levenshtein\_rec(s1[0:i - 1], s2[0:j - 1]) + \ 8. (0 if s1[i - 1] == s2[j - 1] else 1)) |

## Поиск минимального расстояния методом Дамерау-Левенштейна рекурсивно.

Листинг 4. Рекурсивный алгоритм Дамерау-Левенштейна

|  |
| --- |
| 1. def domerau\_levenshtein\_rec(s1, s2): 2. i = len(s1) 3. j = len(s2) 4. if min(i, j) == 0: 5. return max(i, j) 6. elif (i > 1 and j > 1 and s1[i - 1] == s2[j - 2] and \ 7. s1[i - 2] == s2[j - 1]): 8. return min( 9. domerau\_levenshtein\_rec(s1[:i - 1], s2[:j]) + 1, 10. domerau\_levenshtein\_rec(s1[:i], s2[:j - 1]) + 1, 11. domerau\_levenshtein\_rec(s1[:i - 2], s2[:j - 2]) + 1, 12. domerau\_levenshtein\_rec(s1[:i - 1], s2[:j - 1]) + \ 13. (0 if s1[i - 1] == s2[j - 1] else 1)) 14. else: 15. return min( 16. domerau\_levenshtein\_rec(s1[:i - 1], s2[:j]) + 1, 17. domerau\_levenshtein\_rec(s1[:i], s2[:j - 1]) + 1, 18. domerau\_levenshtein\_rec(s1[:i - 1], s2[:j - 1]) + 19. (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 | 36 + 4 \* (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 | 152 + M + N + (M + 1) \* (N + 1), где  M = len(s1), N = len(s2) |

Таблица 4

Потребление памяти, занимаемое алгоритмом Дамерау-Левенштейна

|  |  |
| --- | --- |
| **Структура данных** | **Занимаемая память (байт)** |
| Матрица | 36 + 4 \* (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 | 176 + M + N + 4 \* (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().

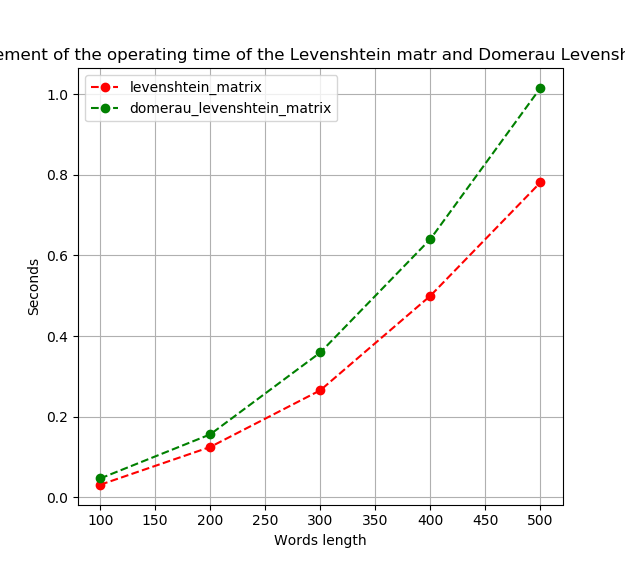


Рис. 1 – Сравнение времени работы алгоритмов Левенштейна и Дамерау-Левенштейна при матричном подходе

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

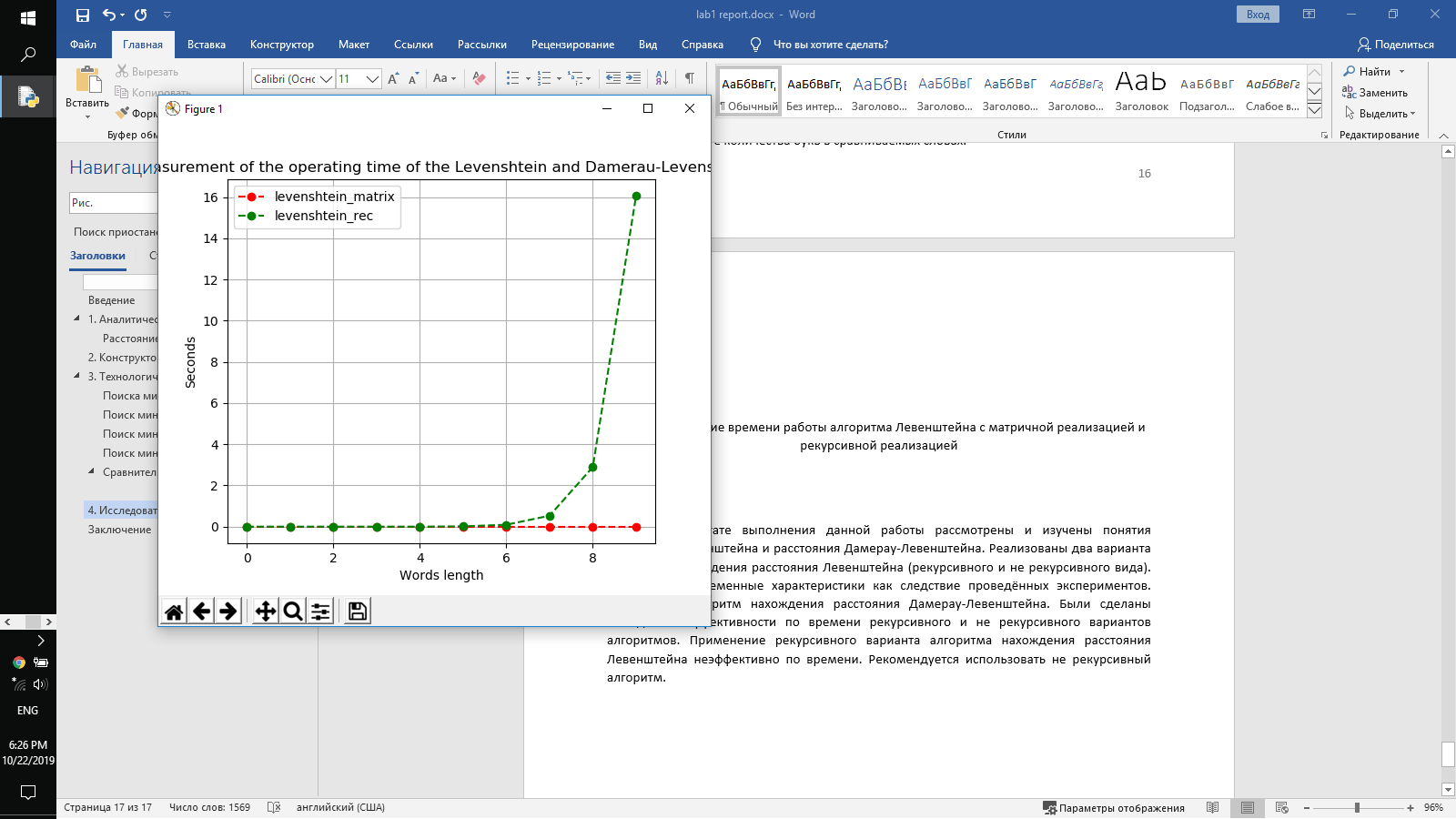


Рис. 2 – Сравнение времени работы алгоритма Левенштейна с матричной реализацией и рекурсивной реализацией

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.

# Литература

[1] A. Colorni, M. Dorigo et V. Maniezzo, Distributed Optimization by Ant Colonies, actes de la première conférence européenne sur la vie artificielle, Paris, France, Elsevier Publishing, 134—142, 1991.

[2] M. Dorigo, Optimization, Learning and Natural Algorithms, PhD thesis, Politecnico di Milano, Italie, 1992.