



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»

Отчёт по лабораторной работе №1

Название: Расстояния Левенштейна и Дamerau-Левенштейна

Дисциплина: Анализ алгоритмов

Студент ИУ7-55Б
(Группа)

Д.Р. Жигалкин
(Подпись, дата) (И.О. Фамилия)

Преподаватель

Л.Л. Волкова
(Подпись, дата) (И.О. Фамилия)

Москва, 2020

Содержание

| | |
|--|----|
| Введение | 3 |
| 1 Аналитическая часть | 4 |
| 1.1 Расстояние Левенштейна и Дамерау-Левенштейна | 4 |
| 1.2 Вывод | 5 |
| 2 Конструкторская часть | 6 |
| 2.1 Требования к функциональности ПО | 6 |
| 2.2 Тесты | 6 |
| 2.3 Схемы алгоритмов | 6 |
| 3 Технологическая часть | 11 |
| 3.1 Средства реализации | 11 |
| 3.2 Листинг программы | 11 |
| 3.3 Тестирование | 13 |
| 3.4 Сравнительный анализ потребляемой памяти | 14 |
| 4 Исследовательская часть | 15 |
| 4.1 Сравнительный анализ на основе замеров времени работы алгоритмов | 15 |
| Заключение | 17 |
| Список использованных источников | 18 |

Введение

Расстояние Левенштейна – минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для решения следующих задач:

- исправления ошибок в слове
- сравнения текстовых файлов утилитой diff
- в биоинформатике для сравнения генов, хромосом и белков

Целью данной лабораторной работы является изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна.

Задачами данной лабораторной являются:

- 1) изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
- 2) применение метода динамического программирования для матричной реализации указанных алгоритмов;
- 3) получение практических навыков реализации указанных алгоритмов: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;
- 4) сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
- 5) экспериментальное подтверждение различий во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;
- 6) описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

1.1 Расстояние Левенштейна и Дameraу-Левенштейна

Задача по нахождению расстояния Левенштейна заключается в поиске минимального количества операций вставки, удаления, замены для превращения одной строки в другую. При нахождении расстояния Дameraу-Левенштейна добавляется операция транспозиции (перестановки соседних символов).

Действия обозначаются так:

- 1) D (англ. delete) — удалить;
- 2) I (англ. insert) — вставить;
- 3) R (replace) — заменить;
- 4) M(match) - совпадение.

Операции I, D, R имеют штраф 1, а операция M - 0.

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ \min(\\ D(i,j-1) + 1, \\ D(i-1,j) + 1, & j > 0, i > 0 \\ D(i-1,j-1) + m(S_1[i], S_2[j]) \\), \end{cases}$$

где $m(a,b)$ равна нулю, если $a = b$ и единице в противном случае; $\min\{a,b,c\}$ возвращает наименьший из аргументов.

Расстояние Дameraу-Левенштейна вычисляется по следующей рекуррентной формуле:

$$D(i, j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & i > 0, j = 0 \\ j, & i = 0, j > 0 \\ \min \begin{cases} D(i, j - 1) + 1, \\ D(i - 1, j) + 1, \\ D(i - 1, j - 1) + m(S_1[i], S_2[i]), \\ D(i - 2, j - 2) + m(S_1[i], S_2[i]), \end{cases} & \begin{array}{l} , \text{ если } i, j > 0 \\ \text{и } S_1[i] = S_2[j - 1] \\ \text{и } S_1[i - 1] = S_2[j] \end{array} \\ \min \begin{cases} D(i, j - 1) + 1, \\ D(i - 1, j) + 1, \\ D(i - 1, j - 1) + m(S_1[i], S_2[i]), \end{cases} & , \text{ иначе} \end{cases}$$

1.2 Вывод

В данном разделе были рассмотрены расстояния Левенштейна и Дамерау-Левенштейна. Расстояние Дамерау-Левенштейна, учитывающее возможность перестановки соседних символов, является модификацией расстояния Левенштейна.

2 Конструкторская часть

В данном разделе будут рассмотрены требования к функциональности ПО, схемы алгоритмов и определены способы тестирования.

2.1 Требования к функциональности ПО

В данной работе требуется обеспечить следующую минимальную функциональность консольного приложения.

- 1) возможность считать две строки;
- 2) вывод расстояний Левенштейна и Дamerau-Левенштейна между строками.

2.2 Тесты

Тестирование ПО будет проводиться методом чёрного ящика. Необходимо проверить работу системы на тривиальных случаях (одна или обе строки пустые, строки полностью совпадают) и несколько нетривиальных случаев.

2.3 Схемы алгоритмов

Ниже будут представлены схемы алгоритмов поиска расстояния Левенштейна:

- 1) нерекурсивного с заполнением матрицы (рисунок 2.1);
- 2) рекурсивного без заполнения матрицы (рисунок 2.2);
- 3) рекурсивного с заполнением матрицы (рисунок 2.3).

Также будет представлена схема нерекурсивного алгоритма поиска расстояния Дamerau-Левенштейна (рисунок 2.4).

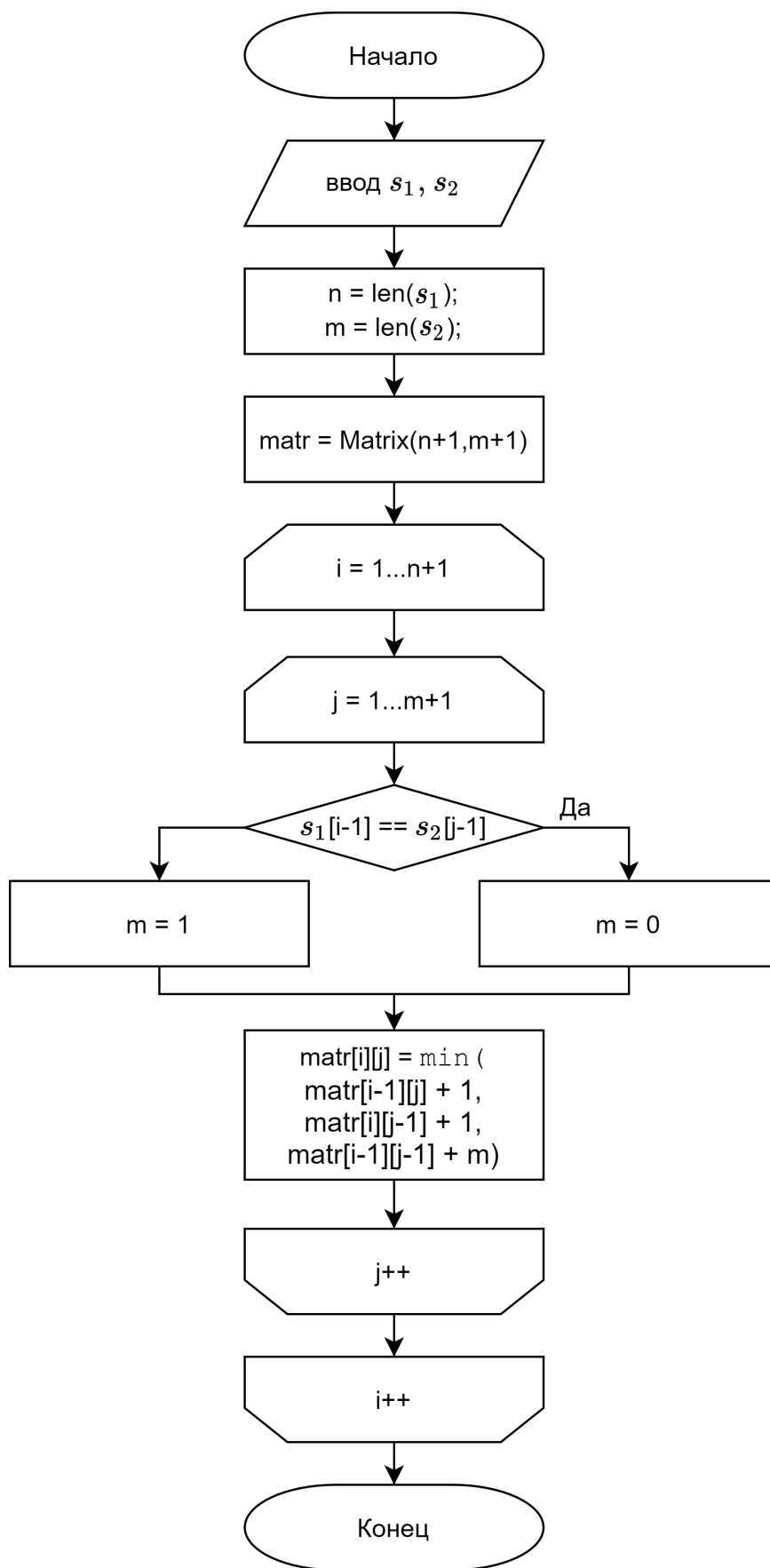


Рисунок 2.1 — Схема нерекурсивного поиска с заполнением матрицы

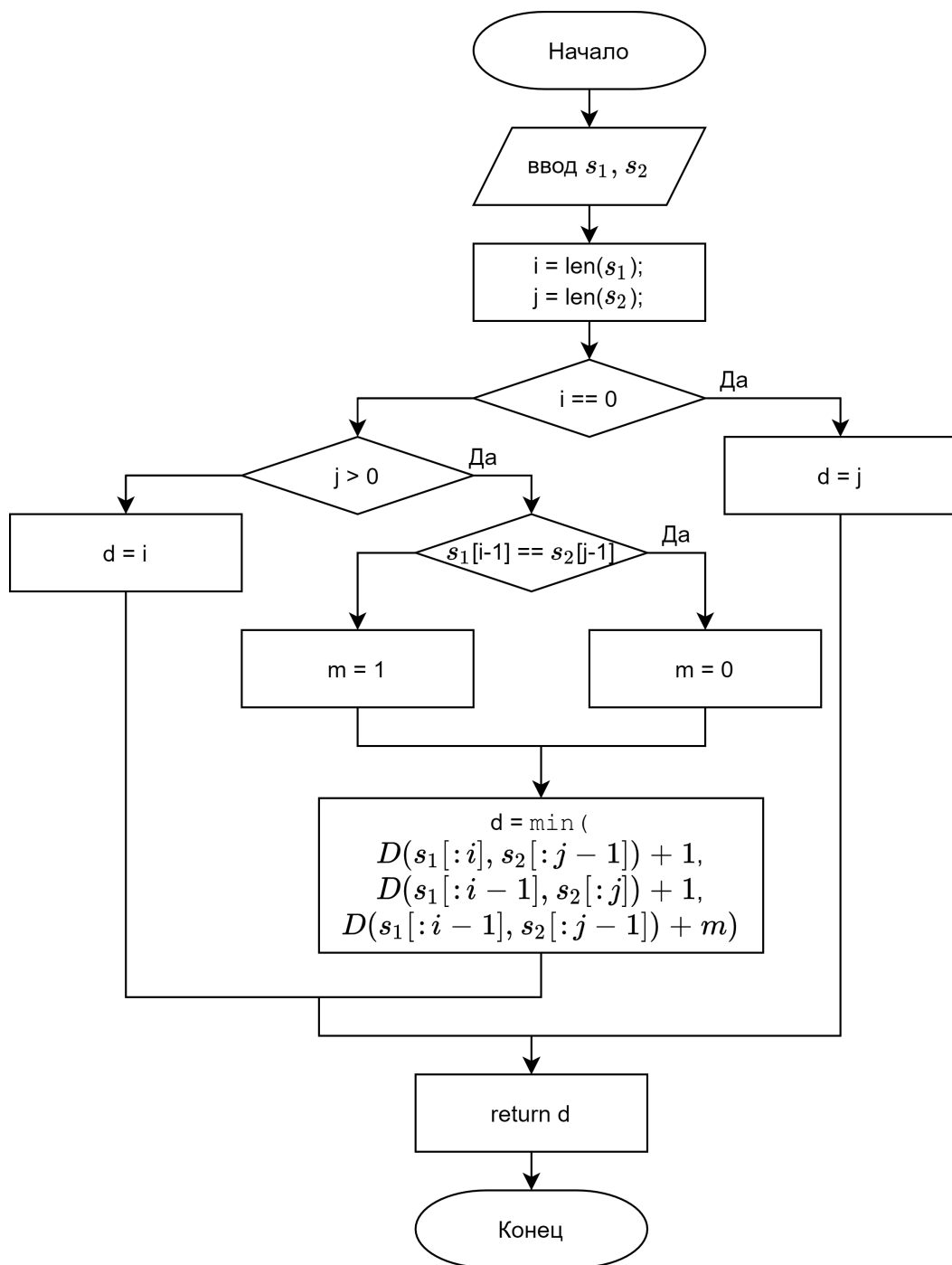


Рисунок 2.2 — Схема рекурсивного поиска без заполнения матрицы

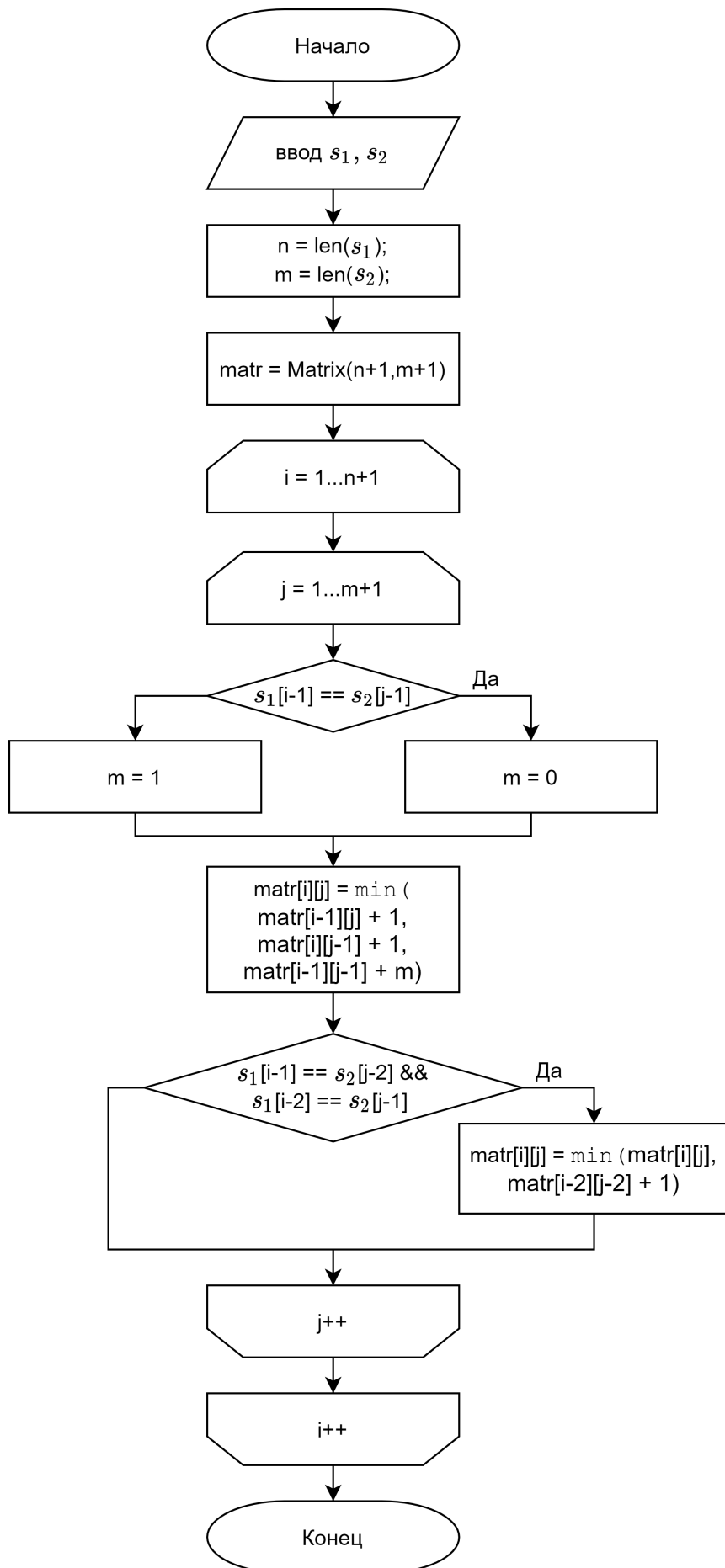


Рисунок 2.3 — Схема рекурсивного поиска с заполнением матрицы

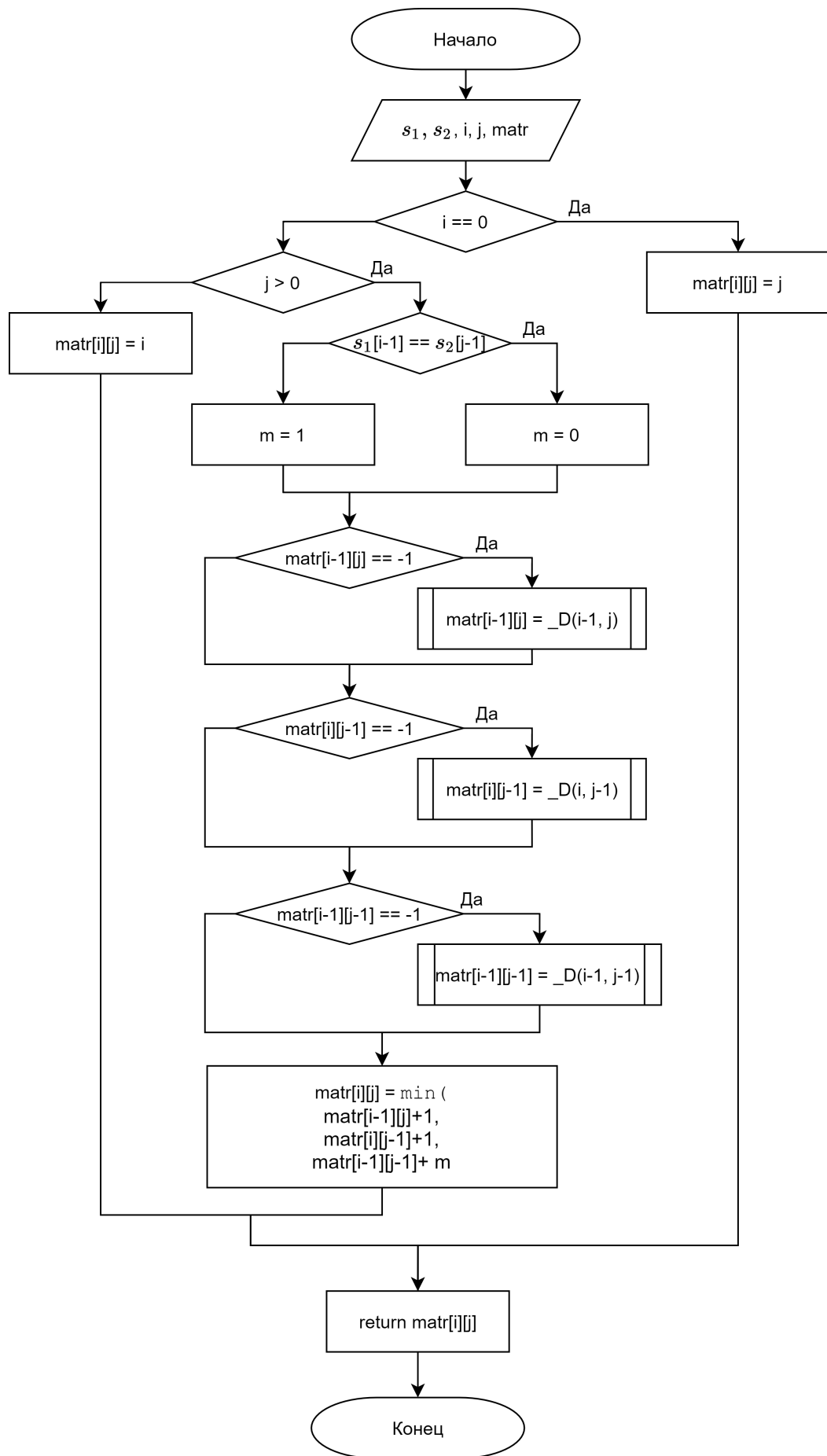


Рисунок 2.4 — Схема нерекурсивного поиска расстояния Дameraу-Левенштейна

3 Технологическая часть

В данном разделе будут выбраны средства репликации ПО, представлен листинг кода и проведён теоритический анализ максимальной затрачиваемой памяти.

3.1 Средства реализации

Для реализации программ я выбрал язык программирования Python, так имею большой опыт работы с ним. Для замера процессорного времени была использована функция `time.timens()` [1]

3.2 Листинг программы

Ниже представлены листинги кода поиска расстояния Левенштейна:

- 1) нерекурсивного с заполнением матрицы (листинг 3.1);
- 2) рекурсивного без заполнения матрицы (листинг 3.2);
- 3) рекурсивного с заполнением матрицы (листинг 3.3);

и код функции поиска расстояния Дameraу-Левенштейна (листинг 3.4).

Листинг 3.1 — Функция нахождения расстояния Левенштейна матрично

```
1 def LevenshteinMatrix(string1 , string2):
2     len_i = len(string1) + 1
3     len_j = len(string2) + 1
4
5     table = [[i + j for j in range(len_j)] for i in range(len_i)]
6
7     for i in range(1, len_i):
8         for j in range(1, len_j):
9             forfeit = 0 if (string1[i - 1] == string2[j - 1]) else 1
10
11             table[i][j] = min(table[i - 1][j] + 1,
12                               table[i][j - 1] + 1,
13                               table[i - 1][j - 1] + forfeit)
14
15     OutputTable(table, string1, string2)
16     return (table[-1][-1])
```

Листинг 3.2 — Функция нахождения расстояния Левенштейна рекурсивно без матрицы

```
1 def LevenshteinRecursion(string1 , string2):
2     if (string1 == "" or string2 == ""):
3         return abs(len(string1) - len(string2))
4
5     forfeit = 0 if (string1[-1] == string2[-1]) else 1
6
```

```

7     return min(LevenshteinRecursion(string1, string2[: -1]) + 1,
8                 LevenshteinRecursion(string1[: -1], string2) + 1,
9                 LevenshteinRecursion(string1[: -1], string2[: -1]) + forfeit)

```

Листинг 3.3 — Функция нахождения расстояния Левенштейна рекурсивно с матрицей

```

1 def GetRecursionLevenshteinMatrix(string1, string2):
2     len_i = len(string1)
3     len_j = len(string2)
4
5     matrix = [[-1 for j in range(len_j + 1)] for i in range(len_i + 1)]
6
7     return RecursionLevenshteinMatrix(string1, len_i, string2, len_j, matrix)
8
9 def RecursionLevenshteinMatrix(string1, i, string2, j, matrix):
10     if (i == 0):
11         matrix[i][j] = j
12     elif (j == 0):
13         matrix[i][j] = i
14     else:
15         if (matrix[i][j - 1] == -1):
16             matrix[i][j - 1] = RecursionLevenshteinMatrix(string1, i, string2, j - 1,
17                                                             matrix)
18
19         if (matrix[i - 1][j] == -1):
20             matrix[i - 1][j] = RecursionLevenshteinMatrix(string1, i - 1, string2,
21                                                             j, matrix)
22
23         if (matrix[i - 1][j - 1] == -1):
24             matrix[i - 1][j - 1] = RecursionLevenshteinMatrix(string1, i - 1,
25                                                             string2, j - 1, matrix)
26
27         forfeit = 0 if (string1[i - 1] == string2[j - 1]) else 1
28         matrix[i][j] = min(min(matrix[i][j - 1], matrix[i - 1][j]) + 1, matrix[i -
29                               1][j - 1] + forfeit)
30
31     return matrix[i][j];

```

Листинг 3.4 — Функция нахождения расстояния Дамерау-Левенштейна матрично

```

1 def DamerauLevenshteinMatrix(string1, string2):
2     len_i = len(string1) + 1
3     len_j = len(string2) + 1
4
5     table = [[i + j for j in range(len_j)] for i in range(len_i)]
6
7     OutputTable(table, string1, string2)
8
9     for i in range(1, len_i):
10        for j in range(1, len_j):
11            forfeit = 0 if (string1[i - 1] == string2[j - 1]) else 1
12
13            table[i][j] = min(table[i - 1][j] + 1,
14                               table[i][j - 1] + 1,
15                               table[i - 1][j - 1] + forfeit)
16
17            if ((i > 1 and j > 1) and string1[i - 1] == string2[j - 2] and string1[i
18                - 2] == string2[j - 1]):
19                table[i][j] = min(table[i][j], table[i - 2][j - 2] + 1)
20
21        OutputTable(table, string1, string2)
22
23     return print(table[-1][-1])

```

3.3 Тестирование

В таблице 3.1 отображён возможный набор тестов для тестирования методом чёрного ящика. В столбцах "Ожидаемый результат" и "Полученный результат"⁴ числа соответствуют рекурсивному алгоритму без матрицы нахождения расстояния Левенштейна, рекурсивному алгоритму с матрицей нахождения расстояния Левенштейна, матричному алгоритму нахождения расстояния Левенштейна, матричному алгоритму нахождения расстояния Дамерау-Левенштейна.

Таблица 3.1 — Таблица тестовых данных

| № | строка 1 | строка 2 | Ожидаемый результат | Фактический результат |
|---|----------|----------|---------------------|-----------------------|
| 1 | | | 0 0 0 0 | 0 0 0 0 |
| 2 | kot | skat | 2 2 2 2 | 2 2 2 2 |
| 3 | | ke | 2 2 2 2 | 2 2 2 2 |
| 4 | qwerty | queue | 4 4 4 4 | 4 4 4 4 |
| 5 | bmstu | utsmb | 4 4 4 4 | 4 4 4 4 |

3.4 Сравнительный анализ потребляемой памяти

С точки зрения использования памяти алгоритмы Левенштейна и Дамерау-Левенштейна не отличаются, следовательно, достаточно рассмотреть лишь разницу рекурсивной и матричной реализаций данных методов.

Использование памяти на строках s_1, s_2 длиной n и m соответственно при использовании матрицы теоритически определяется формулой (3.1):

$$V = (n + 1)(m + 1)\text{sizeof}(\text{int}) + 4\text{sizeof}(\text{size_t}) + 2\text{sizeof}(\text{char*}) + \text{sizeof}(\text{char})(n + m) \quad (3.1)$$

Максимальный расход памяти на строках s_1, s_2 длиной n и m соответственно при использовании рекурсии определяется максимальной глубиной стека вызовов, которая теоритически определяется формулой (3.2):

$$V = \text{sizeof}(\text{char})(n + m) + (n + m)(2\text{sizeof}(\text{char*}) + 3\text{sizeof}(\text{size_t})) \quad (3.2)$$

4 Исследовательская часть

В данном разделе будут проведены эксперименты для проведения сравнительного анализа алгоритмов по затрачиваемому процессорному времени[2] и максимальной используемой памяти.

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведен замер времени работы каждой реализации на строках равной длины, с их случайным заполнением. В таблице 4.1 приняты следующие обозначения:

- 1) LevRecursion – расстояние Левенштейна (рекурсивно, без матрицы) ;
- 2) LevMatrix – расстояние Левенштейна (матрично, без рекурсии) ;
- 3) RecursionLevMatrix – расстояние Левенштейна (матрично, с рекурсией);
- 4) DamLevMatrix – расстояние Дамерау-Левенштейна (матрично).

Таблица 4.1 — Время работы реализации алгоритмов (в наносекундах)

| len | LevRecursion | LevMatrix | RecursionLevMatrix | DamLevMatrix |
|-----|--------------|-----------|--------------------|--------------|
| 0 | 12000000 | 245000000 | 261000000 | 256000000 |
| 1 | 23000000 | 261000000 | 314000000 | 300000000 |
| 2 | 90000000 | 364000000 | 454000000 | 412000000 |
| 3 | 448000000 | 512000000 | 649000000 | 581000000 |
| 4 | 2222000000 | 647000000 | 886000000 | 730000000 |

Наиболее эффективными по времени при маленькой длине слова являются рекурсивные реализации алгоритмов, но как только увеличивается длина слова, их эффективность резко снижается, что обусловлено большим количеством повторных расчетов. Время работы алгоритма, использующего матрицу, намного меньше благодаря тому, что в нем требуется только $(m + 1) * (n + 1)$ операций заполнения ячейки матрицы. Также установлено, что алгоритм Дамерау-Левенштейна работает немного дольше алгоритма Левенштейна, т.к. в нем добавлены дополнительные проверки, однако алгоритмы сравнимы по временной эффективности.

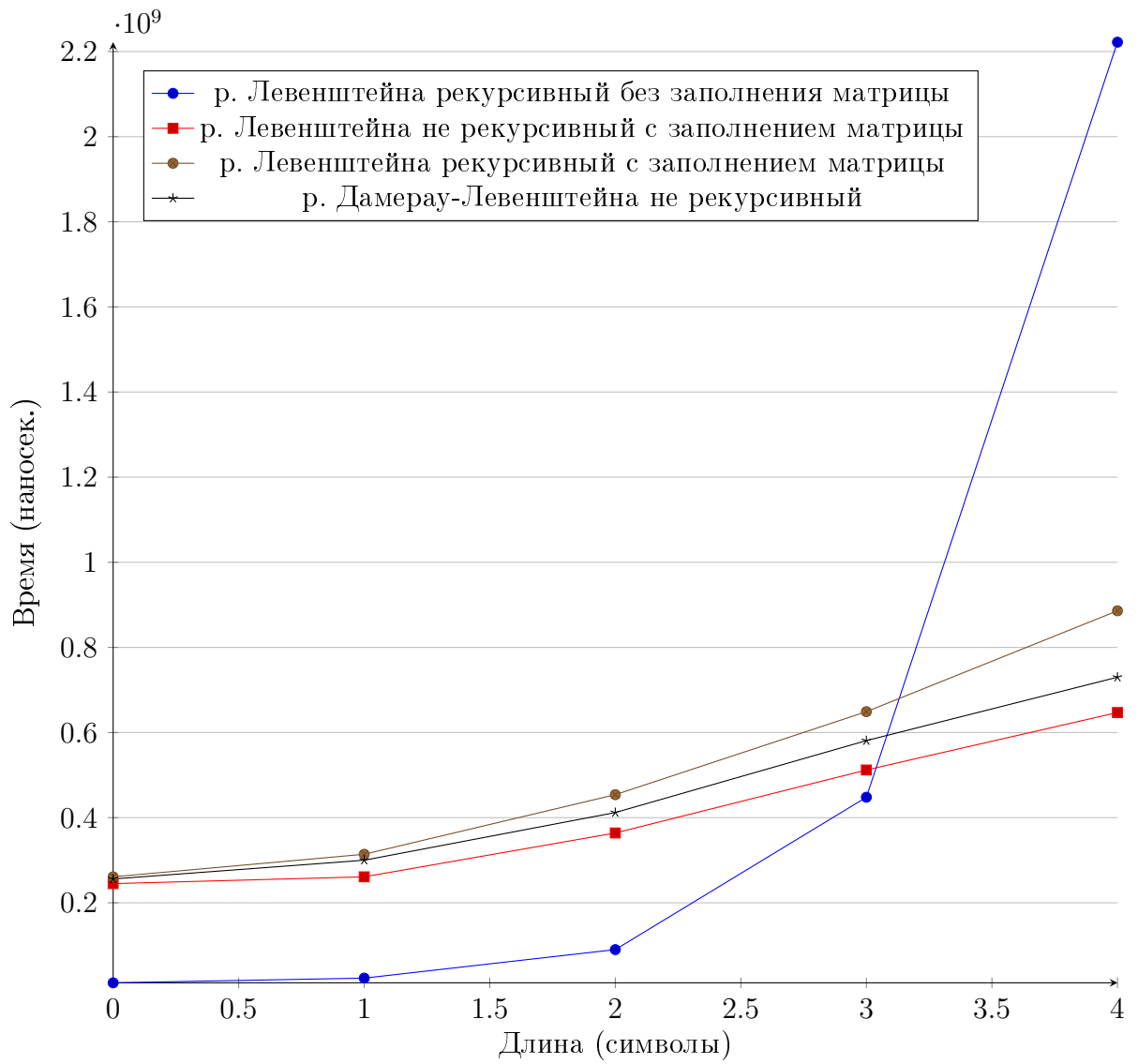


Рисунок 4.1 — График зависимости времени работы алгоритмов от длин строк

Заключение

Был изучен метод динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна. Также изучены алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками, получены практические навыки реализации указанных алгоритмов в матричной и рекурсивных версиях.

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк.

В результате исследований я пришел к выводу, что матричная реализация данных алгоритмов заметно выигрывает по времени при росте длины строк, следовательно более применима в реальных проектах.

Список использованных источников

1. Time. // [Электронный ресурс]. Режим доступа: <https://docs.python.org/3/library/time.html>, (дата обращения: 04.10.2020).
2. Time Basics. // [Электронный ресурс]. Режим доступа: <https://www.gnu.org/software/libc/manual/htmlnode/TimeBasics.html>, (дата обращения: 04.10.2020).