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»

Лабораторная работа № 2
по дисциплине «Анализ алгоритмов»

Тема Алгоритмы умножения матриц

Студент Вавилова В. Л.

Группа ИУ7-54Б

Преподаватели Волкова Л. Л., Строганов Ю. В.

Москва, 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В данной работе рассматриваются и сравниваются различные алгоритмы для вычисления расстояния Левенштейна и Дамерау-Левенштейна. Целью данной лабораторной работы является анализ каждого из алгоритмов, их реализация и оценка эффективности с точки зрения времени выполнения. [1]

В рамках исследования были реализованы следующие алгоритмы:

— итеративный алгоритм поиска расстояния Левенштейна,

— рекурсивный алгоритм поиска расстояния Левенштейна,

— рекурсивный алгоритм поиска расстояния Левенштейна с мемоизацией,

— алгоритм Дамерау-Левенштейна.

Расстояние Левенштейна - это минимальное количество односимвольных операций вставки, удаления и замены одного символа, необходимых для преобразования одной строки в другую.

Расстояние Левенштейна применяется в таких областях, как:

— компьютерная лингвистика (исправление ошибок ввода),

— биоинформатика.

Задачи лабораторной работы:

1) Описать теоретическую основу алгоритмов.
2) Реализовать алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна.
3) Провести тестирование всех реализаций на различных строках и оценить время выполнения в зависимости от длины строк.
4) Замерить время выполнения для каждой реализации и построить график зависимости времени от длины строк.
5) Обосновать выбор языка программирования.

1 Аналитическая часть

Расстояние Левенштейна и Дамерау-Левенштейна

Расстояние Левенштейна представляет собой минимальное количество односимвольных операций (вставки, удаления, замены), необходимых для преобразования одной строки в другую. В то же время расстояние Дамерау-Левенштейна также учитывает операцию транспозиции двух соседних символов, что делает его более гибким в определенных сценариях.

Штрафы операций Для вычисления расстояний между строками используются следующие операции:

— **Вставка (I)** — добавление символа в строку. Штраф операции — 1.

— **Удаление (D)** — удаление символа из строки. Штраф операции — 1.

— **Замена (R)** — замена одного символа на другой. Штраф операции — 1 (если символы разные, если одинаковые — 0).

— **Транспозиция (T)** — перестановка двух соседних символов (только для расстояния Дамерау-Левенштейна). Штраф операции — 1.

Стоимость каждой операции фиксирована и равна 1, что упрощает вычисления.

1.1 Итеративный алгоритм Левенштейна

Итеративный алгоритм использует матрицу для хранения промежуточных значений. Это обеспечивает линейную сложность по времени.

Пусть s_1 и s_2 — две строки длиной M и N соответственно над некоторым алфавитом. Тогда расстояние Левенштейна $d(s_1, s_2)$ можно вычислить по следующей рекуррентной формуле:

$$d(s_1, s_2) = D(M, N), \tag{1.1}$$

где

$$D(i,j) = \begin{cases} 0, & \text{если } i = 0 \text{ и } j = 0, \\ i, & \text{если } j = 0 \text{ и } i > 0, \\ j, & \text{если } i = 0 \text{ и } j > 0, \\ \min\left(D(i,j-1)+1,\right. & \\ \quad D(i-1,j)+1, & \text{если } i > 0 \text{ и } j > 0, \\ \quad \left. D(i-1,j-1)+m(S_1[i],S_2[j])\right) & \end{cases} \tag{1.2}$$

где $m(a,b) = 0$, если $a = b$, и 1 в противном случае. Функция $\min\{a,b,c\}$ возвращает наименьшее из значений.

Шаги алгоритма:

1) Инициализировать матрицу D размером $(m+1) \times (n+1)$, где m — длина строки $s1$, n — длина строки $s2$. Строка $s1$ преобразуется в строку $s2$.

2) Заполнить первую строку и первый столбец значениями от 0 до m и n соответственно. Эти значения соответствуют стоимости преобразования пустой строки в строку с длиной m или n.

3) Вычислять стоимость операций для каждой пары символов $s1[i]$ и $s2[j]$:

$$D[i][j] = \min \begin{cases} D[i-1][j]+1 & \text{(удаление)} \\ D[i][j-1]+1 & \text{(вставка)} \\ D[i-1][j-1]+m(s1[i],s2[j]) & \text{(замена/совпадение)} \end{cases} \tag{1.3}$$

4) Значение в $D[m][n]$ будет равно расстоянию Левенштейна между строками.

Пример: Для строк $s1 =$ `КОТ` и $s2 =$ `СКАТ`, матрица будет выглядеть следующим образом:

		C	K	A	T
	0	1	2	3	4
K	1	1	1	2	3
O	2	2	2	2	3
T	3	3	3	3	2

(1.4)

1.2 Рекурсивный алгоритм Левенштейна

Рекурсивный алгоритм основан на вычислении каждой возможной операции (вставки, удаления или замены) для каждого символа обеих строк. Поскольку на каждом шаге происходит ветвление вызовов для каждой операции, это приводит к значительному увеличению вычислительной сложности.

Шаги алгоритма:

1) Если одна из строк пустая, вернуть длину другой строки (стоимость всех операций вставки/удаления).
2) Для каждой пары символов $s1[i]$ и $s2[j]$, рекурсивно вычислить минимальную стоимость преобразования с помощью следующих операций:
   — вставка,
   — удаление,
   — замена (если символы разные).
3) Возвращаем минимальную стоимость из трех вариантов.

1.3 Рекурсивный алгоритм с мемоизацией

Рекурсивный алгоритм с мемоизацией устраняет недостатки предыдущего подхода, сохраняя результаты вычислений. Это позволяет избежать повторных вызовов.

Шаги алгоритма:

1) Инициализировать таблицу $memo$ размерами $(m+1) \times (n+1)$, заполненную значениями -1.
2) Проверять таблицу на наличие заранее вычисленных значений для каждого вызова функции.
3) Если значение еще не вычислено, рекурсивно вычислить его и сохра-

нить в таблице.

1.4 Алгоритм Дамерау-Левенштейна

Алгоритм Дамерау-Левенштейна использует ту же схему, что и Левенштейн, с добавлением проверки на транспозицию двух соседних символов. Это делает его полезным в задачах, где возможны перестановки символов.

Шаги алгоритма:

1) Инициализировать матрицу D размером $(m+1) \times (n+1)$.
2) Выполнять все операции для расстояния Левенштейна.
3) Если символы $s1[i-1]$ и $s1[i-2]$ равны $s2[j-2]$ и $s2[j-1]$, добавить проверку транспозиции:

$$D[i][j] = \min(D[i][j], D[i-2][j-2] + 1) \tag{1.5}$$

Вывод

В аналитической части были разобраны теоретические основы алгоритмов Левенштейна и Дамерау-Левенштейна. Описаны основные операции, их стоимость.

2 Конструкторская часть

2.1 Схема алгоритмов поиска расстояния Левенштейна

На рисунках 2.1 и 2.2 представлена схема итеративного алгоритма вычисления расстояния Левенштейна, которая состоит из двух частей:

— на рисунке 2.1 показана первая часть,

— на рисунке 2.2 — вторая часть.

На рисунке 2.3 представлена схема рекурсивного алгоритма вычисления расстояния Левенштейна.

На рисунках 2.4 и 2.5 показана схема рекурсивного алгоритма с заполнением матрицы машинной бесконечностью, которая состоит из двух частей:

— на рисунке 2.4 показана первая часть,

— на рисунке 2.5 — вторая часть.

2.2 Схема алгоритма поиска расстояния Дамерау — Левенштейна

На рисунках 2.6 и 2.7 представлена схема алгоритма вычисления расстояния Дамерау — Левенштейна, которая состоит из двух частей:

— на рисунке 2.6 показана первая часть,

— на рисунке 2.7 — вторая часть.

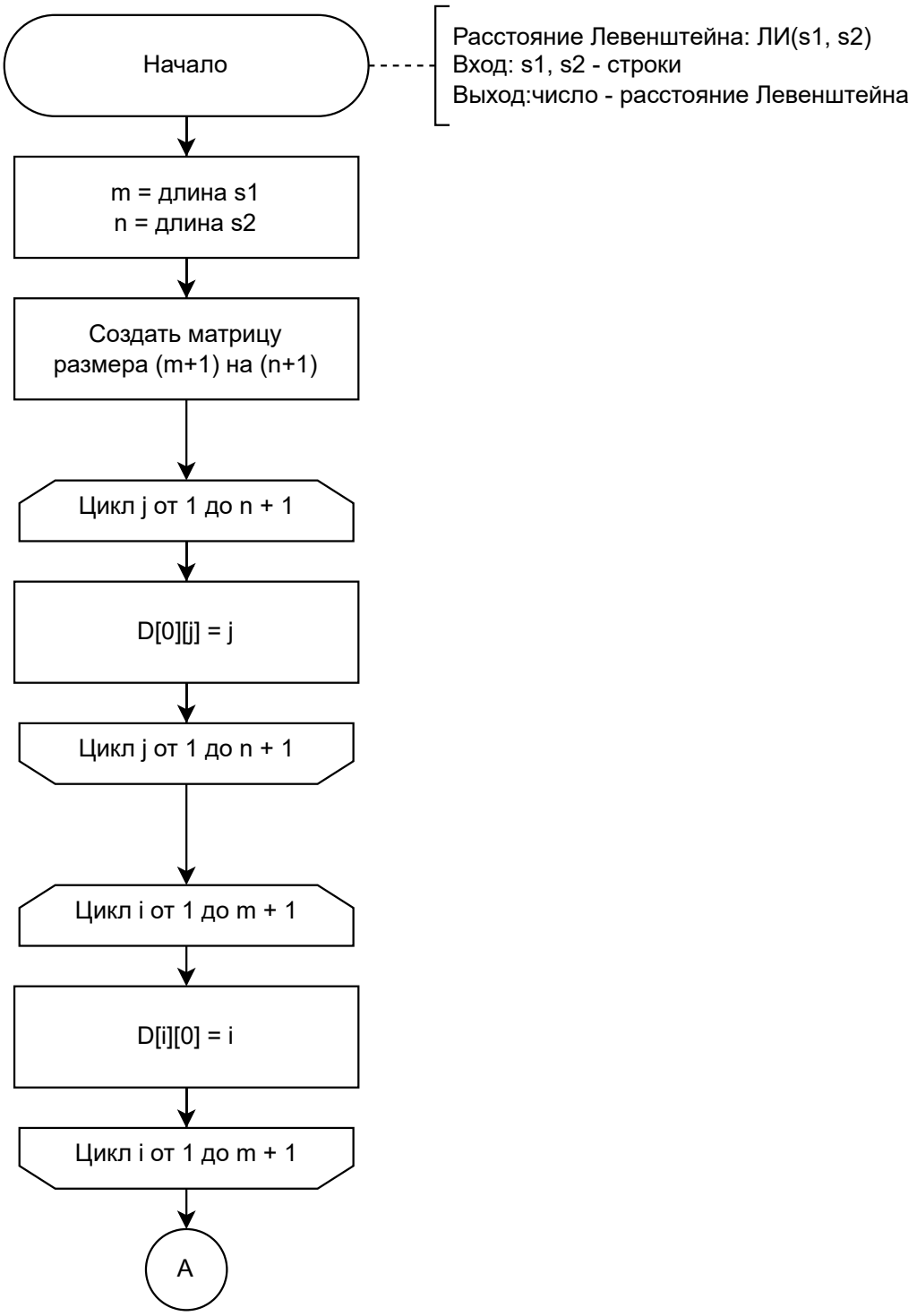

Рисунок 2.1 — Схема итеративного алгоритма вычисления расстояния Левенштейна. Часть 1.

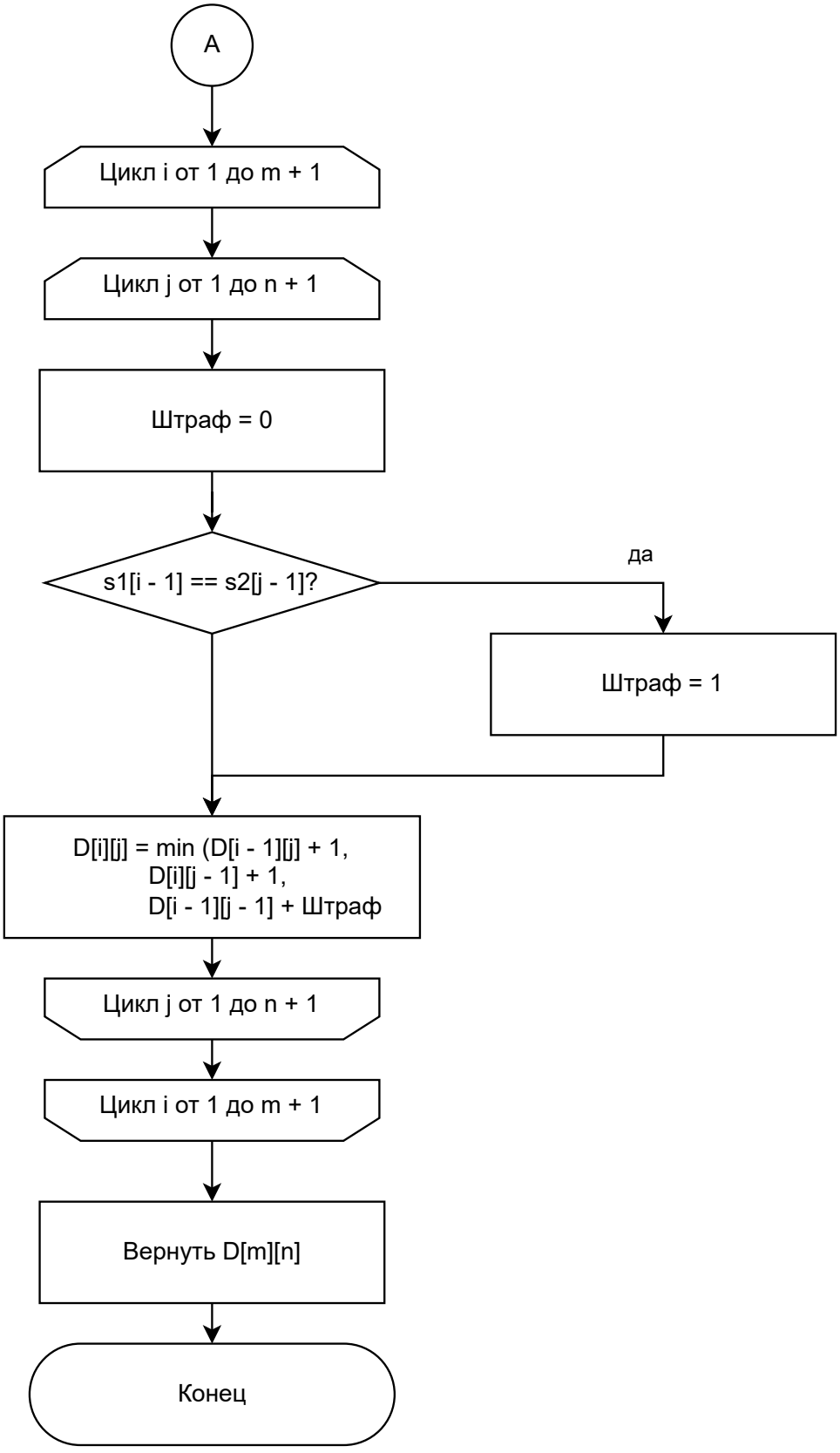

Рисунок 2.2 — Схема итеративного алгоритма вычисления расстояния Левенштейна. Часть 2.

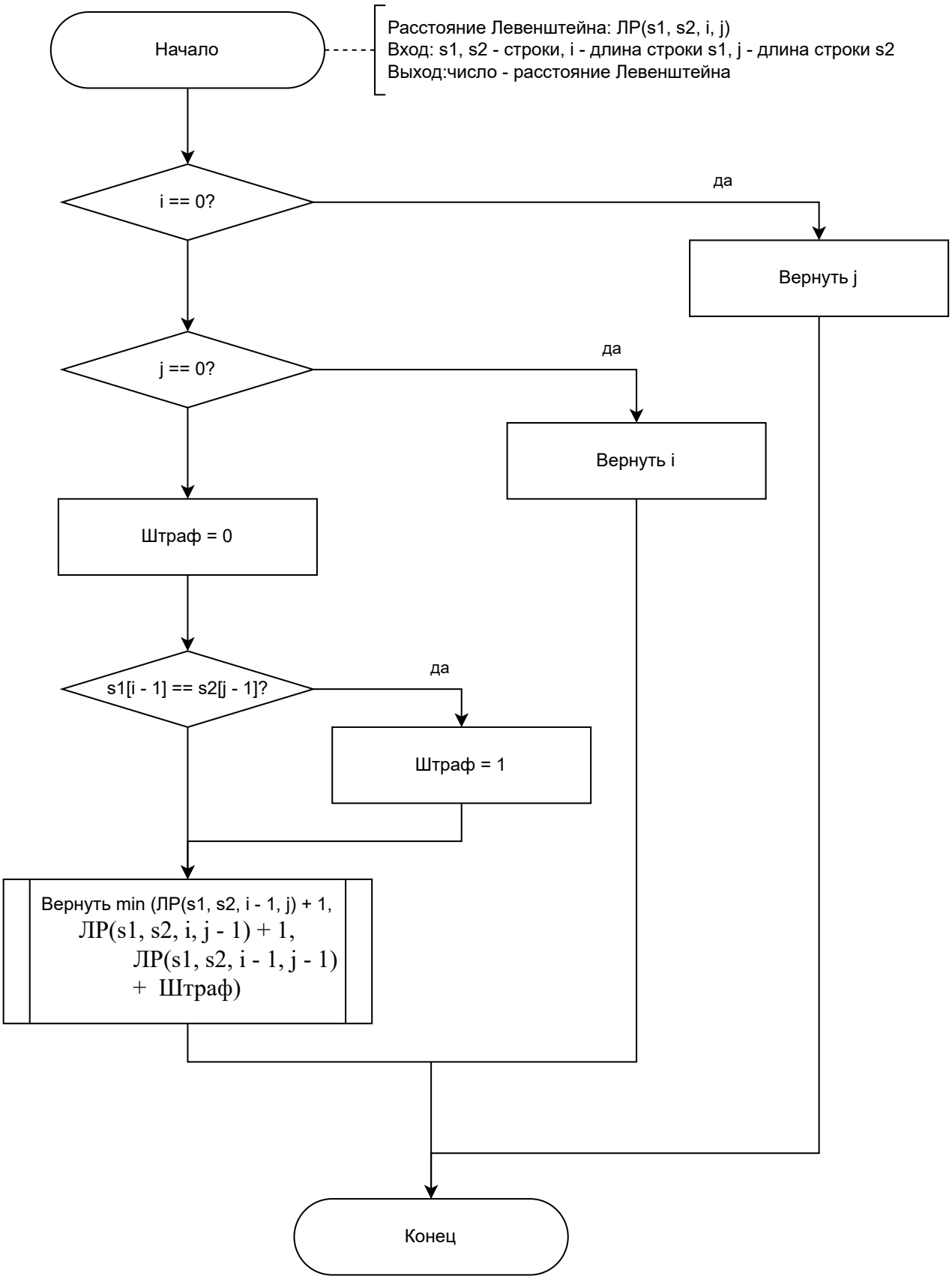

Рисунок 2.3 — Схема рекурсивного алгоритма вычисления расстояния Левенштейна.

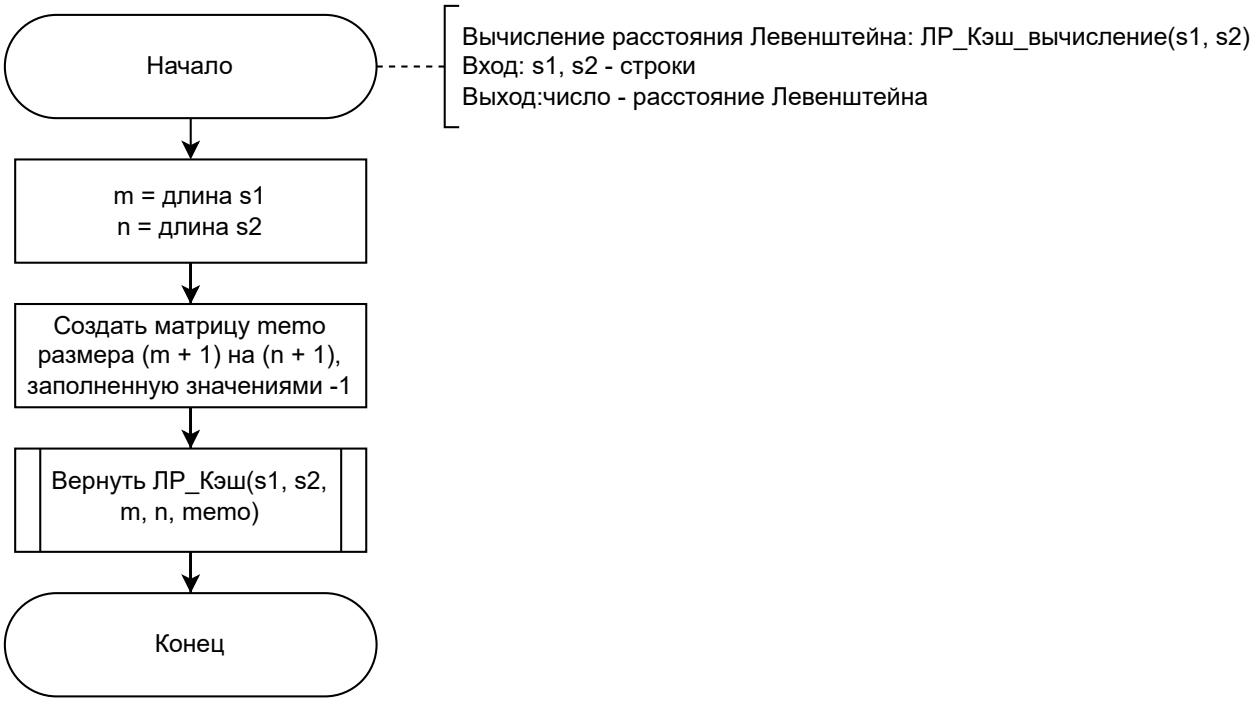

Рисунок 2.4 — Схема рекурсивного алгоритма с заполнением матрицы. Часть 1.

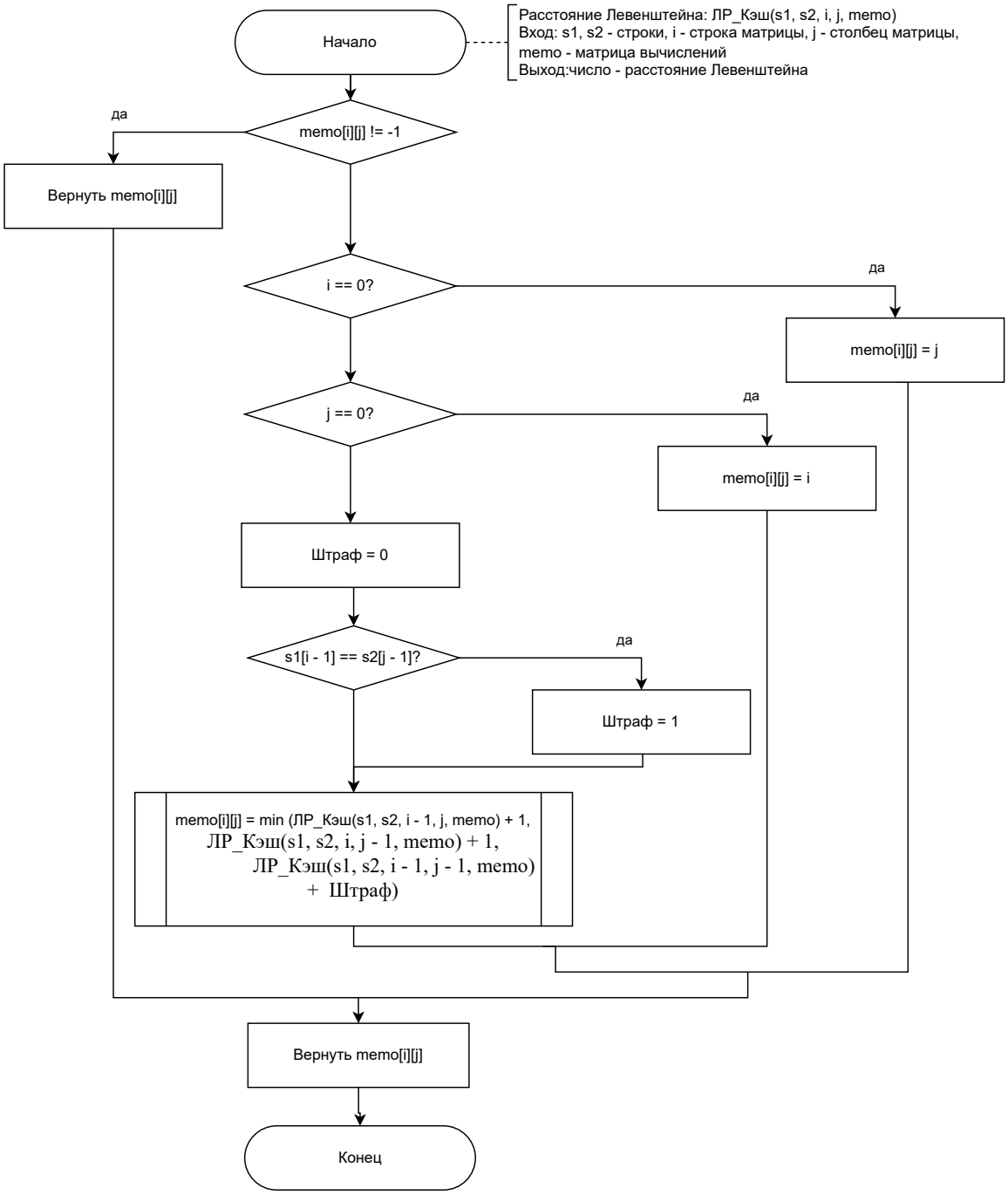

Рисунок 2.5 — Схема рекурсивного алгоритма с заполнением матрицы. Часть 2.

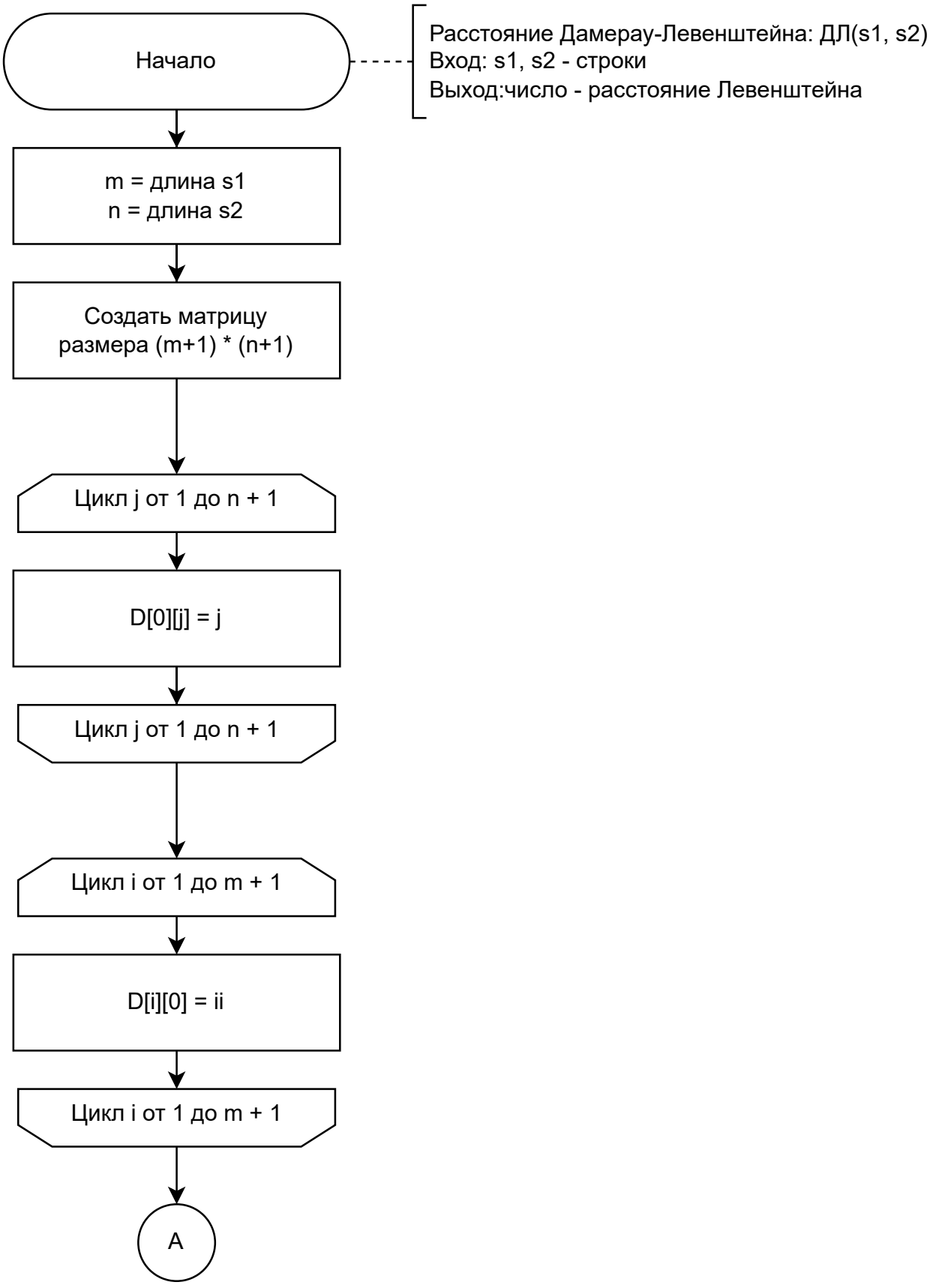

Рисунок 2.6 — Схема алгоритма вычисления расстояния Дамерау — Левенштейна. Часть 1.

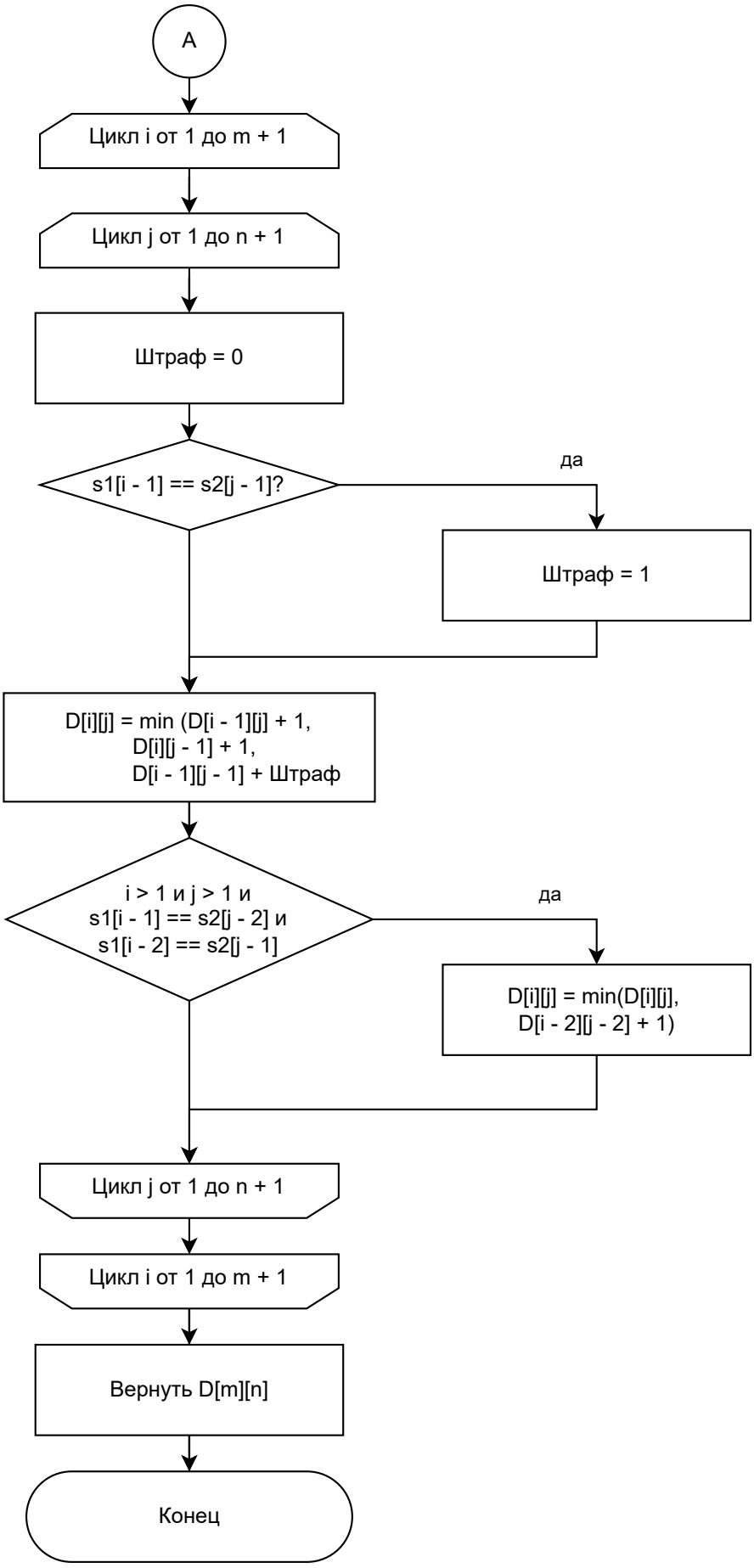

Рисунок 2.7 — Схема алгоритма вычисления расстояния Дамерау — Левенштейна. Часть 2.

Вывод

На основе теоретических данных, полученных в аналитическом разделе, были разработаны схемы алгоритмов, необходимых для решения поставленных задач.

3 Технологическая часть

В данном разделе будут рассмотрены средства, использованные для разработки, а также приведены коды реализованных алгоритмов.

3.1 Средства реализации

Для реализации алгоритмов был выбран MicroPython, так как он обеспечивает эффективную работу на микроконтроллерах. Это позволило выполнить точные замеры времени выполнения.

3.1.1 Итеративный алгоритм поиска расстояния Левенштейна

```
1  def l_iteration_matrix_cost(s1, s2):
2      m, n = len(s1), len(s2)
3      D = [[0] * (n + 1) for _ in range(m + 1)]
4
5      for j in range(1, n + 1):
6          D[0][j] = j
7      for i in range(1, m + 1):
8          D[i][0] = i
9
10     for i in range(1, m + 1):
11         for j in range(1, n + 1):
12             D[i][j] = min(
13                 D[i - 1][j] + 1,
14                 D[i][j - 1] + 1,
15                 D[i - 1][j - 1] + is_replaced(s1[i - 1], s2[j - 1])
16             )
17
18     return D[m][n]
```

Листинг 3.1 — Итеративный алгоритм поиска расстояния Левенштейна

3.1.2 Рекурсивный алгоритм поиска расстояния Левенштейна

```
1  def l_recursion(s1, s2, i, j):
2      if i == 0:
3          return j
```

```
    elif j == 0:
        return i
    else:
        return min(
            l_recursion(s1, s2, i - 1, j) + 1,
            l_recursion(s1, s2, i, j - 1) + 1,
            l_recursion(s1, s2, i - 1, j - 1) + is_replaced(s1[i - 1],
                s2[j - 1])
        )
```

Листинг 3.2 — Рекурсивный алгоритм поиска расстояния Левенштейна

3.1.3 Рекурсивный алгоритм с мемоизацией

```
def l_recursion_cache(s1, s2, i, j, memo):
    if memo[i][j] != -1:
        return memo[i][j]

    if i == 0:
        memo[i][j] = j
    elif j == 0:
        memo[i][j] = i
    else:
        memo[i][j] = min(
            l_recursion_cache(s1, s2, i - 1, j, memo) + 1,
            l_recursion_cache(s1, s2, i, j - 1, memo) + 1,
            l_recursion_cache(s1, s2, i - 1, j - 1, memo) +
                is_replaced(s1[i - 1], s2[j - 1])
        )
    return memo[i][j]
```

Листинг 3.3 — Рекурсивный алгоритм с мемоизацией для поиска расстояния Левенштейна

3.1.4 Алгоритм Дамерау-Левенштейна

```
def dl_matrix_cost(s1, s2):
    m, n = len(s1), len(s2)
    D = [[0] * (n + 1) for _ in range(m + 1)]

    for j in range(1, n + 1):
        D[0][j] = j
    for i in range(1, m + 1):
```

```
8           D[i][0] = i
9
10      for i in range(1, m + 1):
11          for j in range(1, n + 1):
12              D[i][j] = min(
13                  D[i - 1][j] + 1,
14                  D[i][j - 1] + 1,
15                  D[i - 1][j - 1] + is_replaced(s1[i - 1], s2[j - 1])
16              )
17              if i > 1 and j > 1 and s1[i - 1] == s2[j - 2] and s1[i - 2]
                    == s2[j - 1]:
18                  D[i][j] = min(D[i][j], D[i - 2][j - 2] + 1)
19
20      return D[m][n]
```

Листинг 3.4 — Алгоритм поиска расстояния Дамерау-Левенштейна

3.2 Тестовые данные

Для тестирования использовались строки различной длины и содержания, чтобы проверить корректность работы алгоритмов. Все тесты пройдены успешно. Тестовые данные представлены в таблице 3.1.

Таблица 3.1 — Тестовые данные для алгоритмов

Номер теста	Строка 1	Строка 2	Ожидаемое расстояние
1			0
2		"a"	1
3		"ab"	2
4	"a"		1
5	"ab"		2
6	"a"	"a"	0
7	"a"	"b"	1
8	"cat"	"cut"	1
9	"book"	"back"	2
10	"kitten"	"sitting"	3
11	"flaw"	"lawn"	2
12	"intention"	"execution"	5

Все тесты пройдены успешно.

Вывод

В данном разделе были рассмотрены алгоритмы вычисления расстояний Левенштейна и Дамерау-Левенштейна, а также их реализации на языке MicroPython. Программы успешно прошли тестирование на различных строках разной длины и содержания, что подтверждает их корректность и работоспособность.

4 Исследовательская часть

Целью данного исследования является анализ производительности различных алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна на различных строках, в зависимости от их длины и характера. В ходе исследования измерялось время выполнения программ на различных реализациях (итеративная, рекурсивная, рекурсивная с мемоизацией и алгоритм Дамерау-Левенштейна).

Технические характеристики используемого микроконтроллера STM32F767ZIT6 [3]:

— 32 бит,

— 216 МГц,

— ARM Cortex-M7,

— Flash 2 Мбайт,

— RAM 512 Кбайт;

Для исследования использовались строки разной длины (от 3 до 9 (включительно) символов). Алгоритмы выполнялись на микроконтроллере с использованием MicroPython.

4.1 Результаты исследования

Была построена зависимость времени выполнения алгоритмов от длины строк (таблица с результатами замеров представлена под номером 4.1 сравнительный график представлен на рисунке 4.1).

Длина слов	Рекурсивный Л.(с)	Рекурс. + кэш Л.(с)	Итеративный Л.(с)	Итерат. Д.-Л. (с)
1	0.00099945	0.00099945	0.00300550	0.00099683
2	0.00299788	0.00602508	0.00598407	0.00600028
3	0.01799178	0.01100206	0.00499701	0.00600147
4	0.08699870	0.01299977	0.00800014	0.01199961
5	0.45722175	0.01900077	0.01100016	0.01510143
6	2.61085057	0.02775168	0.01549363	0.01852417
7	12.134186980	0.03571510	0.02200031	0.02300000
8	73.384949920	0.05216599	0.02795100	0.03400254
9	375.0456602600	0.05351448	0.03253579	0.03846931

Таблица 4.1 — Время выполнения для различных алгоритмов и длин слов

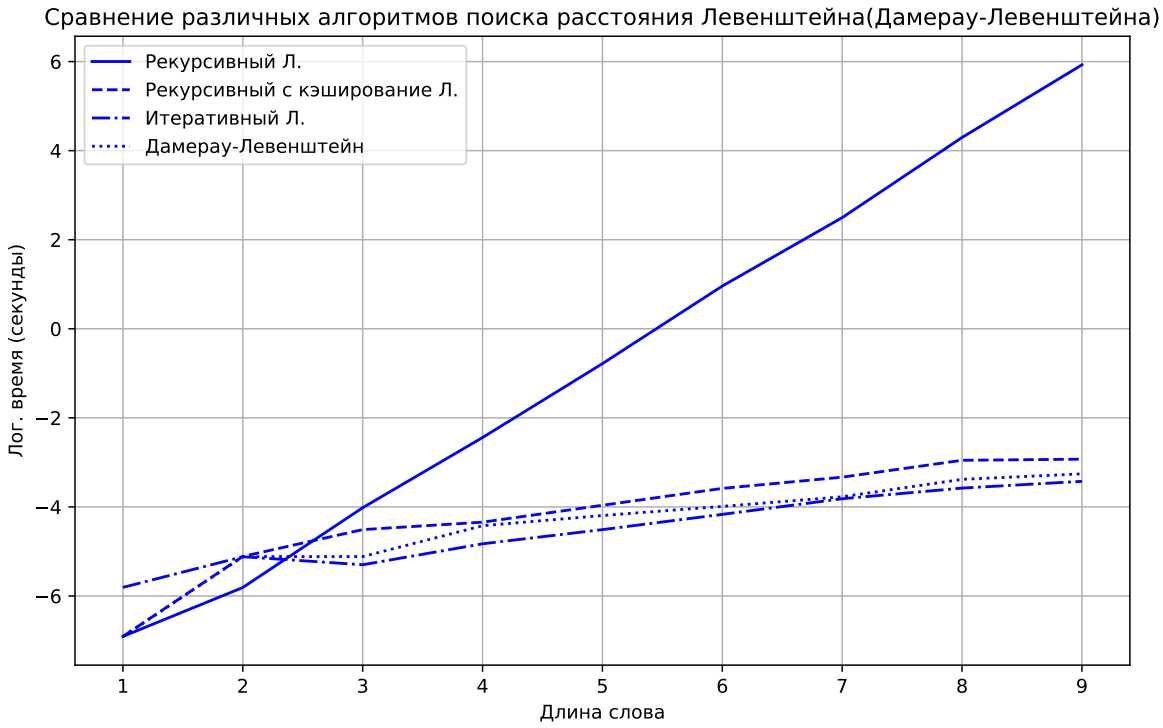

Рисунок 4.1 — Сравнительный график зависимости времени выполнения алгоритмов от длины строк

Вывод

В ходе исследования была проанализирована производительность различных алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна. Итеративные алгоритмы показали лучшее время выполнения по сравнению с рекурсивными. Использование мемоизации улучшило производительность рекурсивного метода, но он все равно оказался менее эффективным по сравнению с итеративным подходом.

ЗАКЛЮЧЕНИЕ

В данной работе была проведена оценка алгоритмов для вычисления расстояний Левенштейна и Дамерау-Левенштейна, с целью анализа их эффективности и времени выполнения. Были реализованы четыре алгоритма: 3 для поиска расстояния Левенштейна (итеративный, рекурсивный, рекурсивный с мемоизацией) и алгоритм поиска расстояния Дамерау-Левенштейна.

В ходе работы были выполнены следующие задачи:

1) Описана теоретическая основа алгоритмов.
2) Реализованы алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна.
3) Проведено тестирование всех реализаций на различных строках, с целью оценки времени выполнения в зависимости от их длины.
4) Замерено время выполнения для каждой реализации, и построен график зависимости времени от длины строк.
5) Обоснован выбор языка программирования для реализации алгоритмов.

Результаты исследования показали, что итеративные алгоритмы имеют лучшее время выполнения по сравнению с рекурсивными. Использование мемоизации значительно улучшило производительность рекурсивного метода, хотя он все равно оказался менее эффективным по сравнению с итеративным подходом.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Л. Л. Волкова "Конспекты лекций и семинаров курса "Анализ алгоритмов"2024г."(дата обращения 08.10.2024) [1]

2. В. М. Черненький, Ю. Е. Гапанюк "МЕТОДИКА ИДЕНТИФИКАЦИИ ПАССАЖИРА ПО УСТАНОВОЧНЫМ ДАННЫМ". Режим доступа: https://engjournal.bmstu.ru/articles/89/89.pdf (дата обращения 10.10.2024) [2]

3. "Руководство Micro Python". Режим доступа: https://micropython-ru.readthedocs.io/ru/latest/tutorial/ (дата обращения 10.10.2024) [3]