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»

Отчет по лабораторной работе № 1 по курсу "Анализ алгоритмов"

Тема Расстояние Левенштейна и Дамерау — Левенштейна

Студент Беляев Н.А.

Группа ИУ7-51Б

Оценка (баллы)

Преподаватель Волкова Л. Л.

Москва — 2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Задача определения редакционного расстояния между двумя символьными строками является актуальной. Соответствующие решения применяются в поисковых системах для обнаружения опечаток в набранном тексте, а также в биоинформатике для определения сходства между последовательностями ДНК.

Цель работы — описать, реализовать и сравнить алгоритмы вычисления редакционного расстояния Левенштейна и Дамерау — Левенштейна между двумя символьными строками.

Для достижения цели необходимо выполнить следующие задачи:

– описать вариации алгоритмов Левенштейна и Дамерау — Левенштейна;
– спроектировать и реализовать описанные вариации алгоритмов;
– для каждого алгоритма провести замер процессорного времени.

1 Аналитический раздел

1.1 Алгоритм Левенштейна

Расстояние Левенштейна — это число, характеризующее минимальное количество операций вставки, удаления или замены, необходимых для приведения двух строк к равенству [1]. Для операций вставки, удаления или замены символа устанавливается единичная стоимость, а для операции сравнения двух символов — стоимость равна нулю.

Для определения расстояния Левенштейна между подстроками $S_1[1...i]$ и $S_2[1...j]$ вводится функция $D(i, j)$, которая определяется соотношением (1.1):

$$D(i,j) = \begin{cases} 0, & \text{если } i = 0 \text{ и } j = 0, \\ i, & \text{если } j = 0 \text{ и } i > 0, \\ j, & \text{если } i = 0 \text{ и } j > 0, \\ \min \begin{cases} D(i, j-1) + 1, \\ D(i-1, j) + 1, \\ D(i-1, j-1) + \delta(S_1[i], S_2[j]), \end{cases} & \text{если } i > 0 \text{ и } j > 0. \end{cases} \tag{1.1}$$

Здесь $\delta(S_1[i], S_2[j])$ определяется соотношением (1.4):

$$\delta(S_1[i], S_2[j]) = \begin{cases} 0, & \text{если } S_1[i] = S_2[j], \\ 1, & \text{иначе.} \end{cases} \tag{1.2}$$

Расстояние Левенштейна между строками S_1 и S_2 равно значению функции $D(L_1, L_2)$, где L_1 и L_2 — длины строк S_1 и S_2 соответственно. Для хранения значений функции $D(i, j)$ используется матрица размером $M \times N$, где M — длина первой строки, а N — длина второй строки.

1.2 Алгоритм Дамерау — Левенштейна

Дамерау модифицировал понятие редакционного расстояния Левенштейна. Он ввел операцию транспозиции двух символов и назначил ей единичную стоимость. Обновленный вид функции $D(i, j)$ имеет вид, представленный формулой (1.3):

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ \min \begin{cases} D(i, j-1) + 1, \\ D(i-1, j) + 1, \\ D(i-1, j-1) + \delta(S_1[i], S_2[j]) \end{cases}, & i > 0,\ j > 0 \\ \min \begin{cases} D(i, j), \\ D(i-2, j-2) + 1 \end{cases}, & \begin{matrix} S_1[i-1] = S_2[j-2], \\ S_1[i-2] = S_2[j-1] \end{matrix} \end{cases} \tag{1.3}$$

где сравнение символов строк $\delta(S_1[i], S_2[j])$ задается как:

$$\delta(S_1[i], S_2[j]) = \begin{cases} 0, & \text{если } S_1[i] = S_2[j] \\ 1, & \text{иначе} \end{cases} \tag{1.4}$$

1.3 Рекурсивный алгоритм Левенштейна

Алгоритм Левенштейна можно реализовать рекурсивно, непосредственно используя формулу (1.1). При этом нет необходимости хранить матрицу значений функции $D(i, j)$, но важно правильно задать условие выхода из рекурсии, чтобы избежать переполнения стека.

1.4 Алгоритм Дамерау — Левенштейна с кешем

В процессе вычисления значений функции $D(i, j)$ по формуле (1.3) результат для конкретной пары (i, j) может быть вычислен несколько раз. Мемоизация вводится для уменьшения количества повторных вычислений. Для мемоизации используется структура данных, например, хеш-таблица, хранящая значения функции D для конкретных пар (i, j). Если для очередной пары значение уже вычислено, то оно берется из структуры, а не пересчитывается.

Вывод

В данном разделе были описаны понятия расстояний Левенштейна и Дамерау — Левенштейна, приведены соотношения для вычисления соответствующих расстояний, а также рассмотрены возможные оптимизации алгоритмов.

2 Конструкторский раздел

В данном разделе приведены схемы различных реализаций алгоритмов Левенштейна и Дамерау — Левенштейна.

2.1 Алгоритм Левенштейна

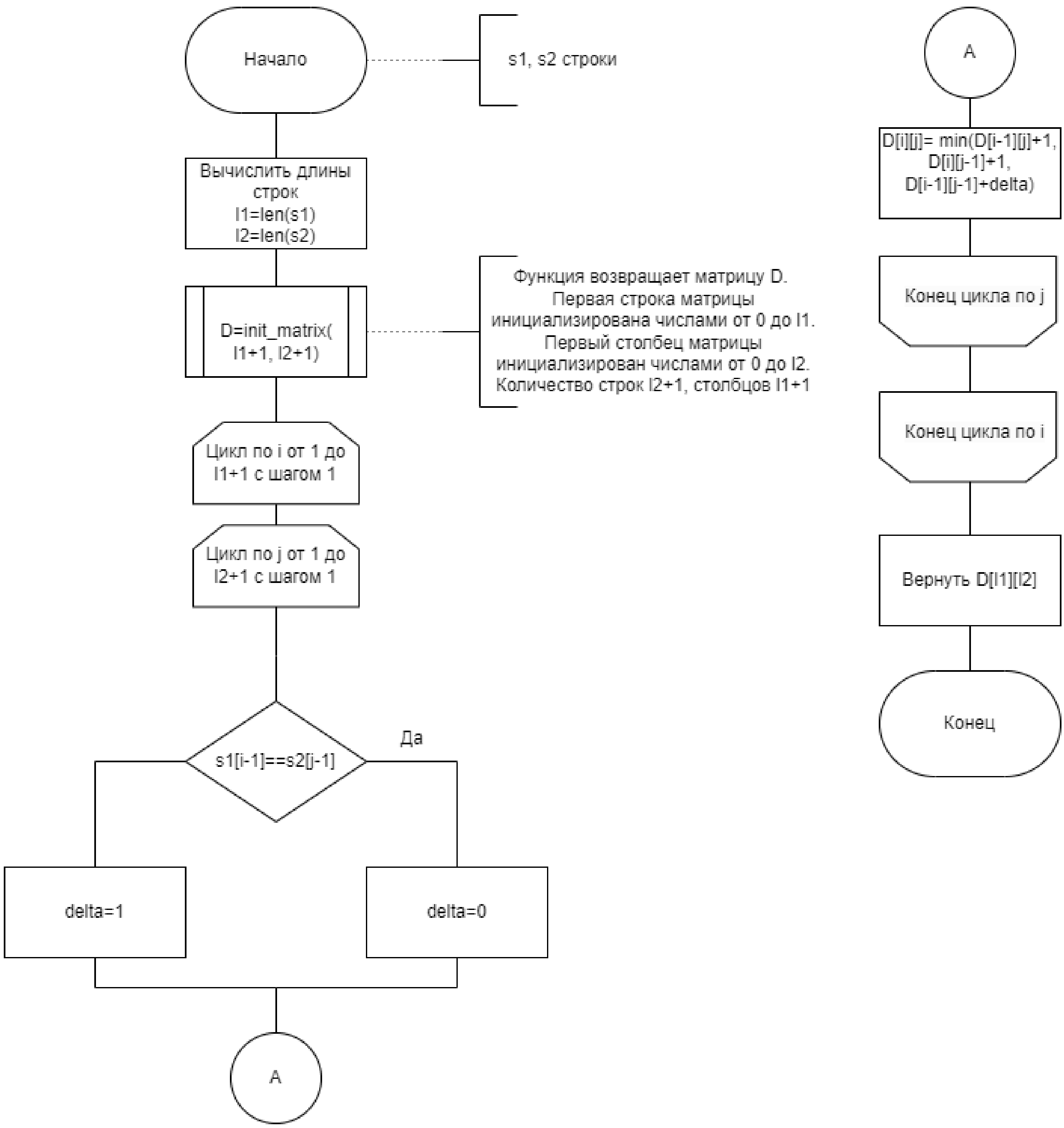

Рисунок 2.1 – Схема нерекурсивного алгоритма Левенштейна

2.2 Нерекурсивный Дамерау — Левенштейн

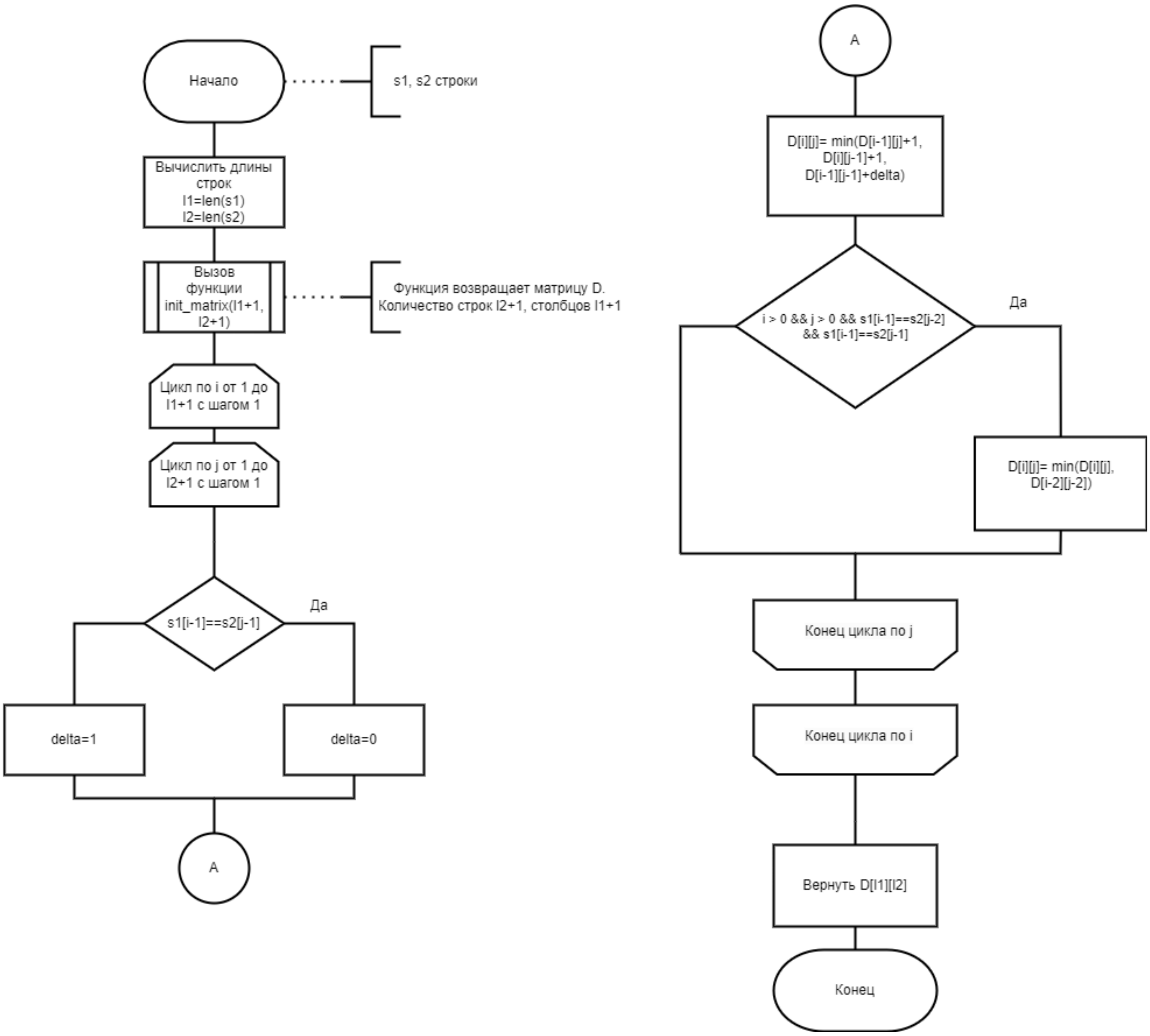

Рисунок 2.2 – Схема нерекурсивного алгоритма Дамерау — Левенштейна

2.3 Рекурсивный алгоритм Левенштейна

На рисунке 2.3 приведена схема рекурсивной реализации алгоритма Левенштейна.

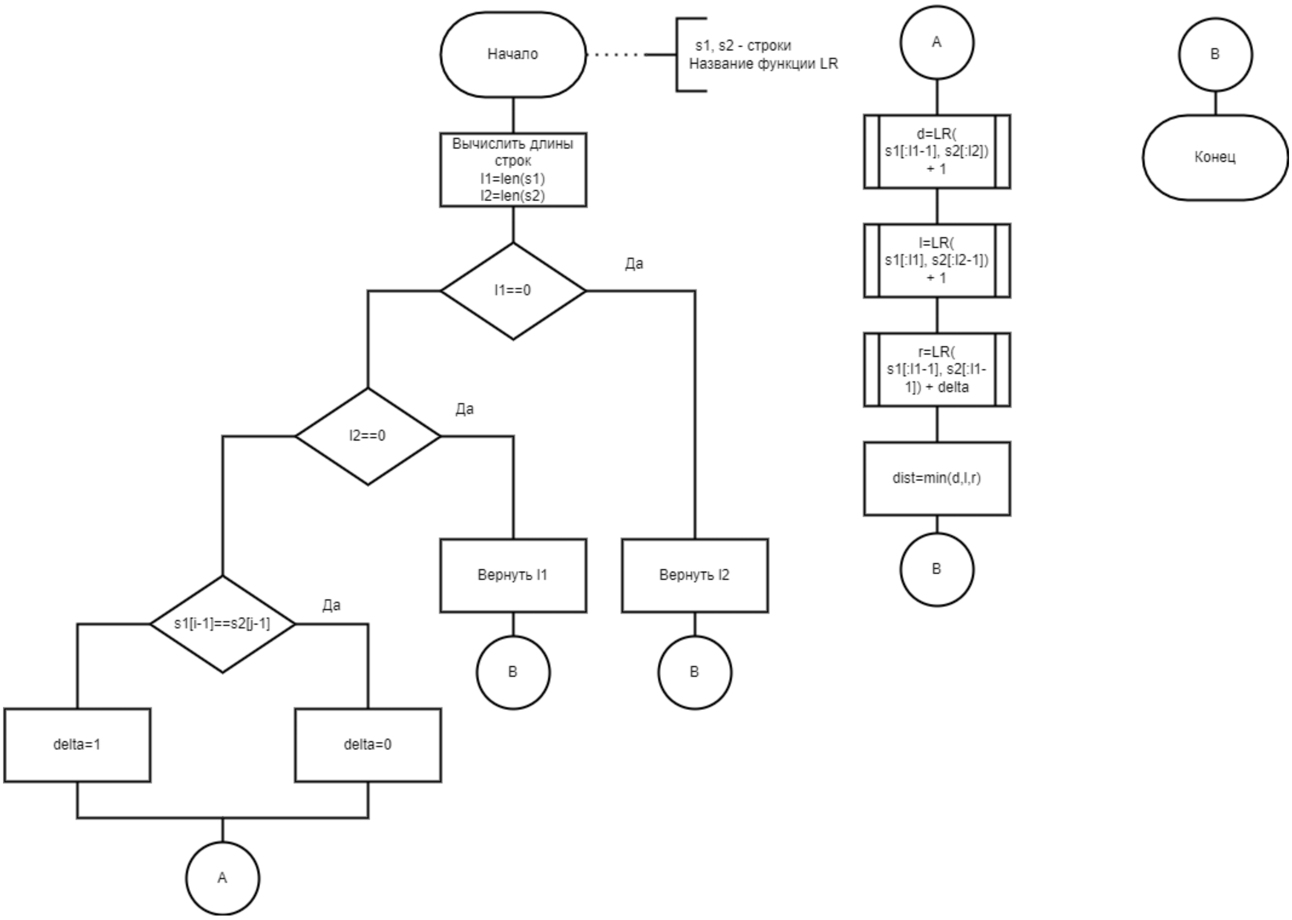

Рисунок 2.3 – Схема рекурсивного алгоритма Левенштейна

2.4 Алгоритм Дамерау — Левенштейна с кешем

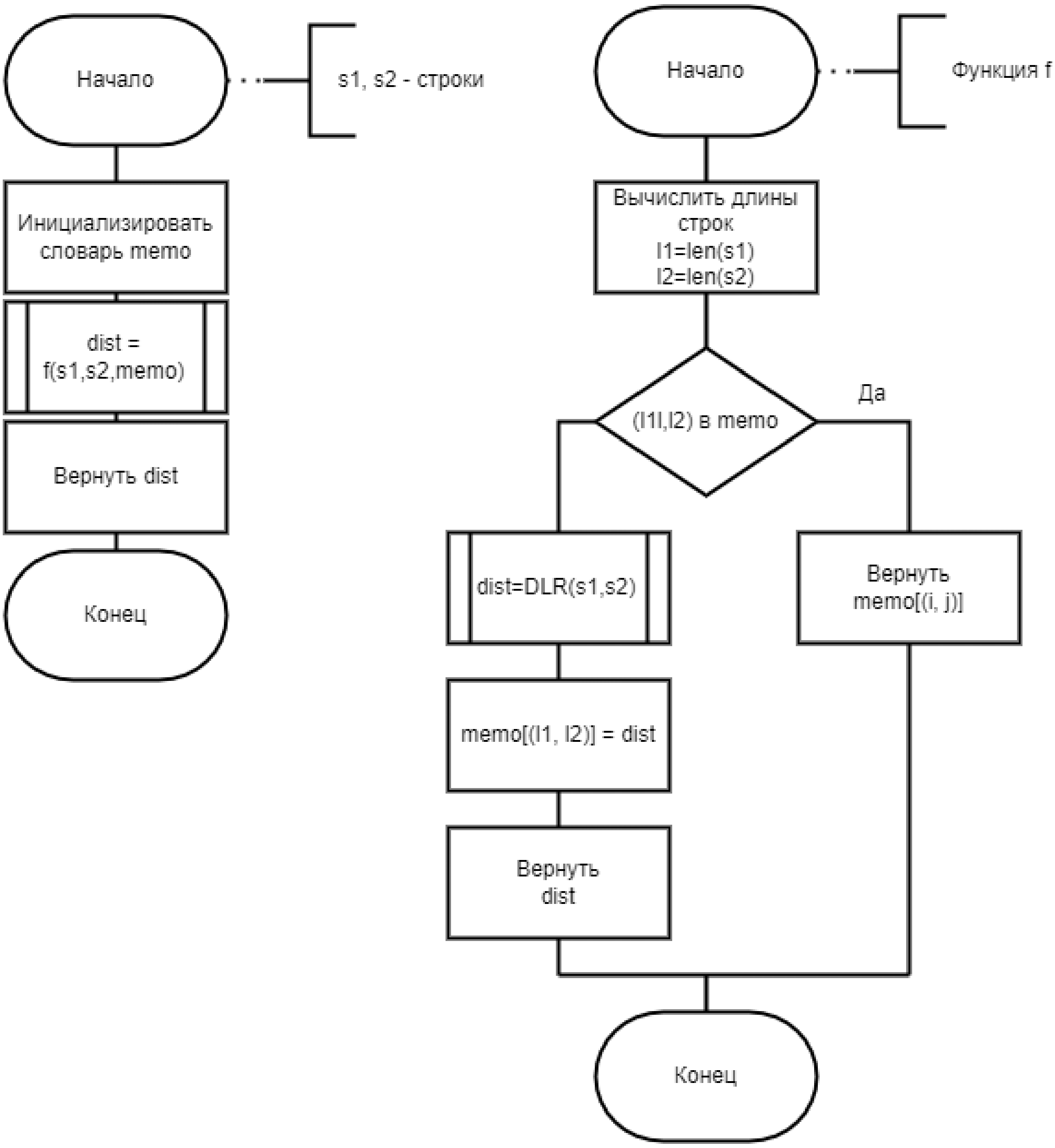

Рисунок 2.4 – Схема рекурсивного алгоритма Дамерау — Левенштейна с мемоизацией, первая часть

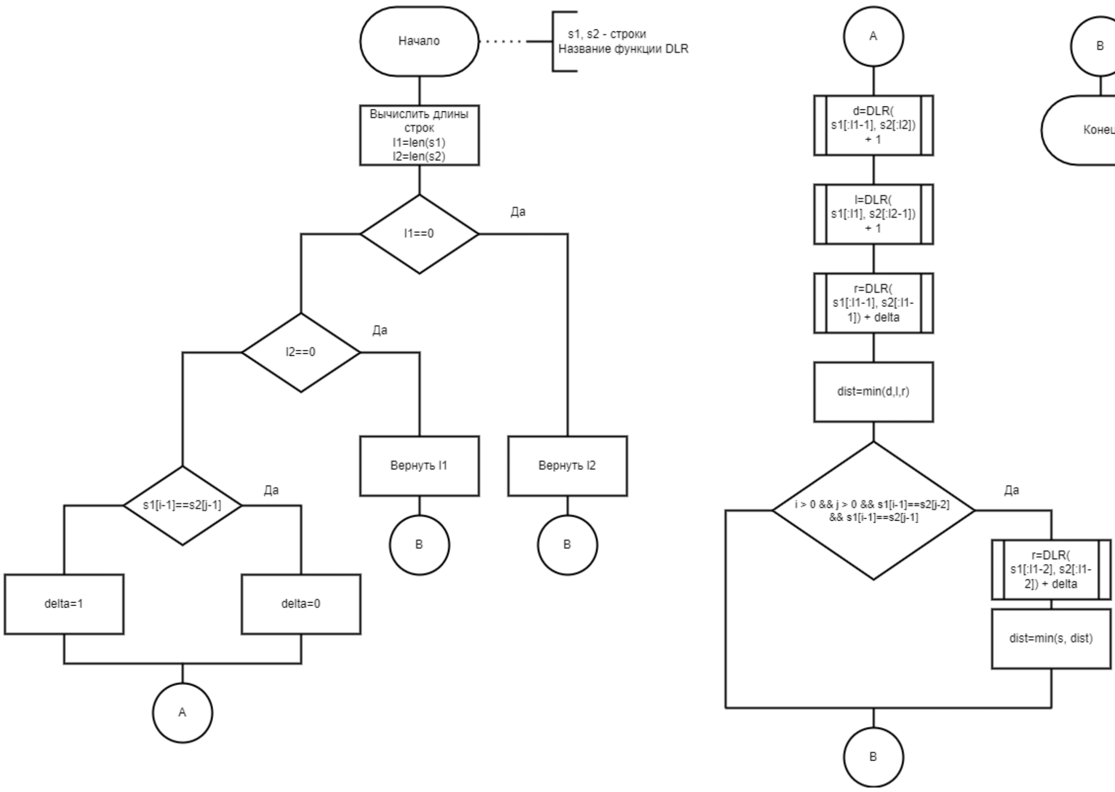

Рисунок 2.5 – Схема рекурсивного алгоритма Дамерау — Левенштейна с мемоизацией, вторая часть

2.5 Алгоритм инициализации матрицы

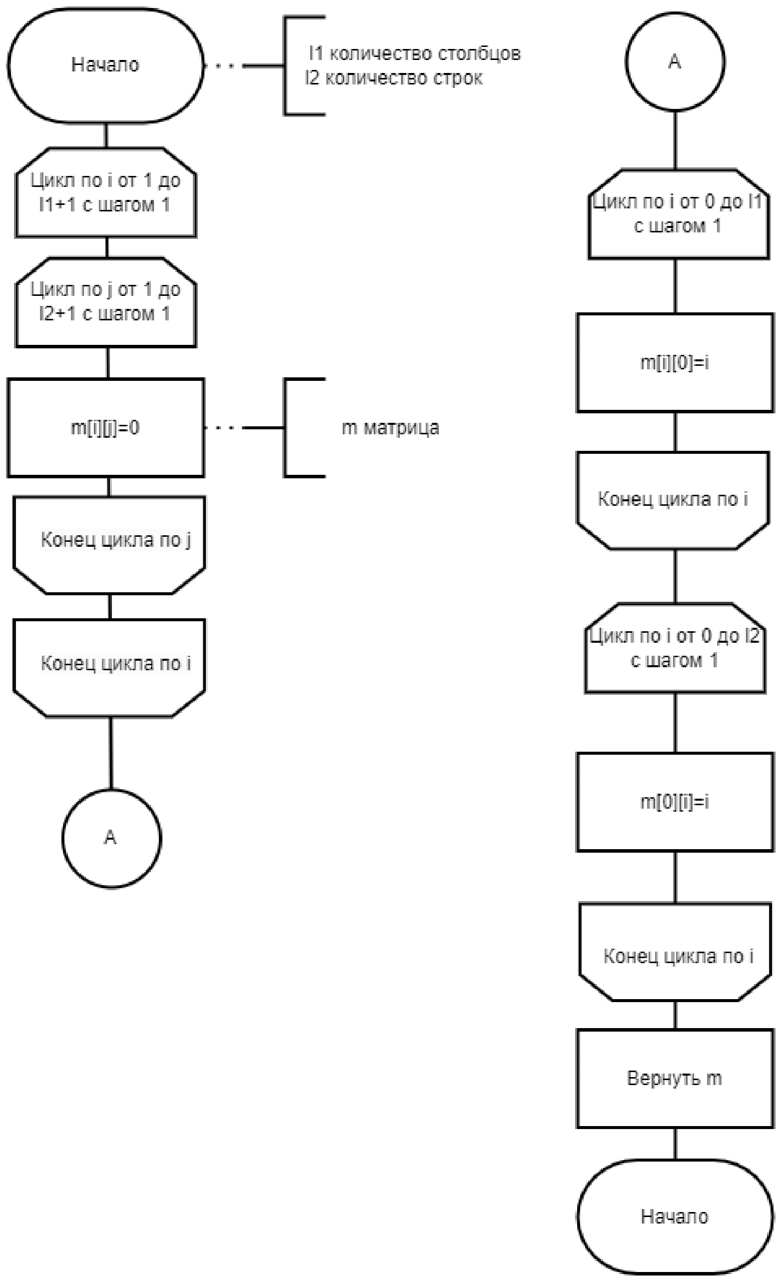

Рисунок 2.6 – Схема алгоритма инициализации матрицы значений функции

Вывод

В данном разделе были приведены схемы различных алгоритмов Левенштейна и Дамерау — Левенштейна.

3 Технологический раздел

Раздел содержит описание средств реализации программы, листинги кода алгоритмов и функциональные тесты.

3.1 Средства реализации

Для реализации программы выбран язык программирования *Python* [2]. Данный язык поддерживает кодировку символов *UTF-8* и позволяет замерить процессорное время работы алгоритма с помощью функции *process_ time* модуля *time* [3].

3.2 Реализация алгоритмов

Листинги 3.1, 3.2, 3.3, 3.4 содержат реализации алгоритмов Левенштейна и Дамерау — Левенштейна. Листинг 3.5 содержит реализацию вспомогательной функции инициализации матрицы.

Листинг 3.1 – Алгоритм Левенштейна

```
def levenstein(s1, s2: str):
    l1, l2 = len(s1), len(s2)
    distance_matrix = matrix_init(l1 + 1, l2 + 1)

    for i in range(1, l1 + 1):
            for j in range(1, l2 + 1):
                delta = 0 if s1[i - 1] == s2[j - 1] else 1
                distance_matrix[i][j] = min(distance_matrix[i -
                   1][j] + 1, distance_matrix[i][j - 1] + 1,
                   distance_matrix[i - 1][j - 1] + delta)

    return distance_matrix[l1][l2], distance_matrix
```

Листинг 3.2 – Алгоритм Дамерау — Левенштейна

```
def damerau_levenstein(s1, s2: str):
    l1, l2 = len(s1), len(s2)
    distance_matrix = matrix_init(l1 + 1, l2 + 1)

    for i in range(1, l1 + 1):
            for j in range(1, l2 + 1):
                delta = 0 if s1[i - 1] == s2[j - 1] else 1
                distance_matrix[i][j] = min(distance_matrix[i -
                   1][j] + 1, distance_matrix[i][j - 1] + 1,
                   distance_matrix[i - 1][j - 1] + delta)

                if i > 1 and j > 1 and s1[i - 1] == s2[j - 2]
                   and s1[i - 2] == s2[j - 1]:
                    distance_matrix[i][j] =
                       min(distance_matrix[i][j],
                       distance_matrix[i - 2][j - 2] + 1)

    return distance_matrix[l1][l2], distance_matrix
```

Листинг 3.3 – Рекурсивный алгоритм Левенштейна

```
def levenstein_recursive(s1, s2: str):
    l1, l2 = len(s1), len(s2)

    if l1 == 0: return l2
    if l2 == 0: return l1

    delta = 0 if s1[l1 - 1] == s2[l2 - 1] else 1

    d = levenstein_recursive(s1[:l1 - 1], s2[:l2]) + 1
    i = levenstein_recursive(s1[:l1], s2[:l2 - 1]) + 1
    r = levenstein_recursive(s1[:l1 - 1], s2[:l2 - 1]) + delta

    dist = min(d, i, r)

    return dist
```

Листинг 3.4 – Рекурсивный алгоритм Дамерау — Левенштейна с мемоизацией

```
def damerau_levenstein_memo(s1, s2: str):
    memo = {}

    def f(s1, s2: str, memo: dict):
        l1, l2 = len(s1), len(s2)
        if (l1, l2) in memo:
             return memo[(l1, l2)]

        dist = damerau_levenstein_recursive(s1, s2)
        memo[(s1,s2)] = dist
        return dist

    return f(s1, s2, memo)

def damerau_levenstein_recursive(s1, s2: str):
    l1, l2 = len(s1), len(s2)

    if l1 == 0: return l2
    if l2 == 0: return l1

    delta = 0 if s1[l1 - 1] == s2[l2 - 1] else 1

    d = damerau_levenstein_recursive(s1[:l1 - 1], s2[:l2]) + 1
    i = damerau_levenstein_recursive(s1[:l1], s2[:l2 - 1]) + 1
    r = damerau_levenstein_recursive(s1[:l1 - 1], s2[:l2 - 1]) +
       delta

    dist = min(d, i, r)
    if l1 > 1 and l2 > 1 and s1[l1 - 1] == s2[l2 - 2] and s1[l1
       - 2] == s2[l2 - 1]:
        s = damerau_levenstein_recursive(s1[:l1 - 2], s2[:l2
           -2]) + 1
        dist = min(dist, s)

    return dist
```

Листинг 3.5 – Алгоритм инициализации матрицы

```
def matrix_init(l1, l2: int) -> List[List[int]]:
    m =[[0 for _ in range(l2)] for _ in range(l1)]

    for i in range(l1):
        m[i][0] = i

    for i in range(l2):
        m[0][i] = i

    return m
```

3.3 Функциональное тестирование

Таблица 3.1 содержит информацию о проведенном функциональном тестировании реализованных алгоритмов. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Входные данные		**Расстояние и алгоритм**			
		Итер. Л	**Итер. Д-Л**	**Рекур. Л**	**Рекур. с мемо Д-Л**
		0	0	0	0
	"qwe"	3	3	3	3
"qwe"		3	3	3	3
"qweR"	"QWER"	3	3	3	3
"qwerty"	"qwertyuiop"	4	4	4	4
"qwer"	"qwre"	2	1	2	1
"qwer"	"qare"	3	2	3	2
"hello"	"привет"	6	6	6	6
"45"	45"	1	1	1	1
"qwer ber"	"qwer"	4	4	4	4

Вывод

В разделе были описаны средства реализации алгоритмов, приведены листинги кода и описание функционального тестирования.

4 Исследовательский раздел

Раздел содержит описание замера процессорного времени и оценку затрачиваемой алгоритмом оперативной памяти.

4.1 Замер процессорного времени

Замер процессорного времени выполнен на микроконтроллере *STM-32 Nucleo-144*. Замер проведен с помощью функции *process_time* модуля *time*. Исследовалась зависимость процессорного времени работы алгоритма от длины входных строк. Размер строк варьировался от 1 до 6 символов с шагом в один символ. Содержимое строк – случайные последовательности символов заданной длины в кодировке *UTF-8*. Длина строк в рамках замера совпадает. Для каждой длины замер проведен $N = 10$ раз. В качестве результата выбрано среднее арифметическое из N замеров.

В таблице 4.1 приведены результаты замера процессорного времени работы алгоритмов.

Таблица 4.1 – Результат замеров процессорного времени работы алгоритмов для строк с длиной от 1 до 10 символов.

	Время, мкс			
	Левенштейн	**Итер. Д-Л**	**Рекур. Л**	**Рекур. с мемо Д-Л**
Длина, символ				
1	0.102485	0.101186	0.088565	0.386414
2	0.172556	0.183024	0.424275	1.047953
3	0.348513	0.354702	2.049482	2.956012
4	0.561802	0.670172	10.499670	4.002789
5	0.715580	1.031140	57.425976	6.211656
6	1.115551	1.244014	291.462974	10.310614

На рисунке 4.1 табличные данные отображены графически.

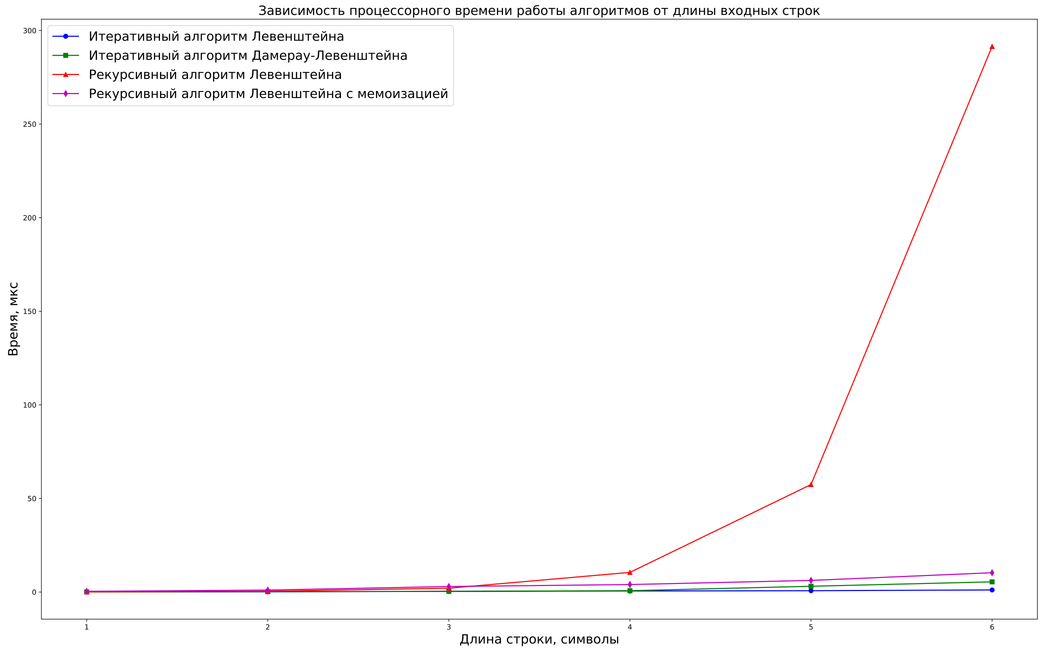

Рисунок 4.1 – Зависимость процессорного времени работы алгоритма от длины входных строк

4.2 Оценка затраченной оперативной памяти

4.2.1 Алгоритм Левенштейна

На хранение локальных переменных длин строк S_1, S_2 и параметра $delta$ затрачено $3 \cdot sizeof(int)$ байт. Для хранения матрицы значений $D(i, j)$ затрачено $(sizeof(S_1) + 1) \cdot (sizeof(S_2) + 1) \cdot sizeof(int)$. Итоговый объем памяти выражен формулой (4.1).

$$(sizeof(S_1) + 1) \cdot (sizeof(S_2) + 1) \cdot sizeof(int) + 3 \cdot sizeof(int) \tag{4.1}$$

4.2.2 Алгоритм Дамерау — Левенштейна

На хранение локальных переменных длин строк S_1, S_2 и параметра $delta$ затрачено $3 \cdot sizeof(int)$ байт. Для хранения матрицы значений $D(i, j)$ затрачено $(sizeof(S_1)+1) \cdot (sizeof(S_2)+1) \cdot sizeof(int)$. Дополнительно, для учета перестановок символов, затраты на память остаются аналогичными. Итоговый объем памяти выражен формулой (4.2).

$$(sizeof(S_1) + 1) \cdot (sizeof(S_2) + 1) \cdot sizeof(int) + 3 \cdot sizeof(int) \tag{4.2}$$

4.2.3 Рекурсивный алгоритм Левенштейна

Для рекурсивного алгоритма на каждом уровне рекурсии затрачивается память на хранение локальных переменных длин строк S_1, S_2 и параметра $delta$, что составляет $3 \cdot sizeof(int)$ байт. Также на каждом уровне создаются новые строки, что в худшем случае требует $sizeof(S_1) + sizeof(S_2)$ памяти на каждом уровне рекурсии. Глубина рекурсии в худшем случае будет $sizeof(S_1) \cdot sizeof(S_2)$, что дает итоговый объем памяти, выраженный формулой (4.3).

$$\begin{split}(sizeof(S_1) + sizeof(S_2)) \cdot sizeof(int) \cdot (sizeof(S_1) \cdot sizeof(S_2)) +\\ 3 \cdot sizeof(int) \cdot (sizeof(S_1) \cdot sizeof(S_2))\end{split} \tag{4.3}$$

4.2.4 Алгоритм Дамерау — Левенштейна с кешем

Для хранения локальных переменных длин строк S_1, S_2, параметра $delta$ и хеш-таблицы (мемоизация) затрачивается $3 \cdot sizeof(int)$ байт. Объем памяти, необходимый для хранения всех возможных промежуточных результатов, составляет $sizeof(S_1) \cdot sizeof(S_2) \cdot sizeof(int)$ для хранения ключей и значений хеш-таблицы. Итоговый объем памяти выражен формулой (4.4).

$$(sizeof(S_1) \cdot sizeof(S_2) \cdot sizeof(int)) + 3 \cdot sizeof(int) \tag{4.4}$$

Вывод

Алгоритмы Левенштейна и Дамерау — Левенштейна имеют схожую производительность как по памяти, так и по времени работы. Выбор алгоритма стоит делать опираясь на то, нужно ли учитывать транспозицию символов в решаемой задаче. Рекурсивная реализация алгоритма Левенштейна имеет худшее время работы. Стоит отметить, что такая реализация имеет аппаратное ограничение, так как ввод длинных строк может привести к переполнению стека из-за рекурсивного спуска. Наиболее оптимальным по времени и памяти оказался алгоритм Дамерау — Левенштейна с кешированием. Рекурсия обеспечивает малое количество затрачиваемой памяти, а мемоизация сокращает количество повторных вычислений. В отсутствие нехватки вычислительных мощностей предпочтение стоит отдать именно этой реализации.

ЗАКЛЮЧЕНИЕ

Задачи лабораторной работы выполнены:

- описаны алгоритмы поиска расстояния Левенштейна и Дамерау — Левенштейна;
- спроектированы и реализованы описанные алгоритмы поиска редакционных расстояний;
- проведен анализ эффективности реализаций алгоритмов по памяти и процессорному времени.

Цель лабораторной работы достигнута: алгоритмы реализованы и спроектированы. Проведен замер процессорного времени работы реализаций алгоритмов и анализ количества затрачиваемой ими памяти.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Haldar R.*, *Mukhopadhyay D.* Levenshtein Distance Technique in Dictionary Lookup Methods: An Improved Approach. —.
2. Python Documentation [Электронный ресурс]. — Режим доступа: `https://www.python.org` (дата обращения: 20.09.2024).
3. Time access and conversions [Электронный ресурс]. — Режим доступа: `https://docs.python.org/3/library/time.html#functions` (дата обращения: 20.09.2024).