



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»

Студент:

Климов И. С.

Группа:

ИУ7-52Б

Преподаватель:

Волкова Л. Л.

Оценка:

\_\_\_\_\_

Москва

2021

# Содержание

<b>Введение</b>	<b>4</b>
<b>1 Аналитический раздел</b>	<b>5</b>
1.1 Определение . . . . .	5
1.2 Матричный способ нахождения расстояния Левенштейна .	5
1.3 Рекурсивный способ нахождения расстояния Левенштейна	6
1.4 Рекурсивный способ нахождения расстояния Левенштейна с использованием кэширования . . . . .	8
1.5 Рекурсивный способ нахождения расстояния Дамерау- Левенштейна . . . . .	8
1.6 Вывод . . . . .	8
<b>2 Конструкторский раздел</b>	<b>10</b>
2.1 Алгоритм нахождения расстояния Левенштейна - матрично	10
2.2 Алгоритм нахождения расстояния Левенштейна - рекурсивно	12
2.3 Алгоритм нахождения расстояния Левенштейна - рекурсив- но с использованием кэша . . . . .	14
2.4 Алгоритм нахождения расстояния Дамерау-Левенштейна - рекурсивно . . . . .	17
2.5 Вывод . . . . .	20
<b>3 Технологический раздел</b>	<b>21</b>
3.1 Выбор языка программирования . . . . .	21
3.2 Реализация алгоритма нахождения расстояния Левенштей- на - матрично . . . . .	22
3.3 Реализация алгоритма нахождения расстояния Левенштей- на - рекурсивно . . . . .	22

3.4	Реализация алгоритма нахождения расстояния Левенштейна - рекурсивно с использованием кэша . . . . .	23
3.5	Реализация алгоритма нахождения расстояния Дамерау-Левенштейна - рекурсивно . . . . .	24
3.6	Вывод . . . . .	25
<b>4</b>	<b>Исследовательский раздел</b>	<b>26</b>
4.1	Тестовые наборы . . . . .	26
	<b>Заключение</b>	<b>27</b>
	<b>Список использованных источников</b>	<b>28</b>
	<b>Приложение</b>	<b>29</b>

# Введение

Почти каждый день встречается ситуация, когда слово, написанное в поисковике, введено ошибочно, и предлагается замена на схожее с ним, или в текстовом редакторе происходит автозамена ввиду наличия опечатки. Позволить решить эту проблему компьютером позволяет нахождение редакционного расстояния - минимальное количество операций, которые надо совершить, чтобы перевести исходную строку в конечную. [1] Благодаря ему можно найти "ближайшее" слово. Одним из базовых видов такого расстояния является *расстояние Левенштейна* (также может использоваться схожее с ним *расстояние Дамерау-Левенштейна*). Помимо этого, оно используется в биоинформатике для определения схожести друг с другом разных участков ДНК или РНК. [2]

**Цель работы:** получить навык динамического программирования на материале алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна и оценить полученные реализации по памяти и времени. Для достижения цели были поставлены следующие **задачи**:

- изучить алгоритмы нахождения расстояния Левенштейна матричным способом, рекурсивным с использованием кэширования и без и расстояния Дамерау-Левенштейна рекурсивным методом;
- разработать алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна перечисленными способами;
- реализовать разработанные алгоритмы;
- провести сравнительный анализ процессорного времени выполнения реализации каждого алгоритма;
- провести сравнительный анализ пиковой затрачиваемой реализациями алгоритмов памяти.

# 1 Аналитический раздел

В данном разделе рассматриваются различные методы нахождения расстояния Левенштейна (матричный, рекурсивный, рекурсивный с использованием кэширования), рекурсивный способ поиска расстояния Дамерау-Левенштейна.

## 1.1 Определение

Расстояния Левенштейна, как упоминалось ранее, это базовый вид редакционного расстояния, а точнее - минимальное количество редакторских операций, необходимых для превращения одной строки в другую, - операций вставки (I - insert), удаления (D - delete) и замены (R - replace). [3] Каждая из них имеет цену величиной в 1, и путем посимвольного преобразования необходимо найти такую последовательность операций, чтобы суммарная цена было наименьшей. Для симметричности сравнения еще вводится операция соответствия - match (M). В дальнейшем Ф. Дамерау доказал, что следует добавить еще одну операцию - операцию перестановки двух символов - совокупность этих четырех операций смогут покрыть большинство ошибок при письме, и его способ определения расстояния был назван расстоянием Дамерау-Левенштейна.

## 1.2 Матричный способ нахождения расстояния Левенштейна

Для поиска расстояния Левенштейна чаще всего используют матрицу D размерами  $n + 1$  и  $m + 1$ , где  $n$ ,  $m$  - длины сравниваемых строк  $s_1$  и  $s_2$ . Первая строка и первый столбец заполняются как тривиальные случаи, так как можно однозначно понять, сколько потребуется операций,

чтобы превратить пустую строку в строку с одним символом, двумя и т.д. (соответственно одна операция вставки, две и т.д.) и наоборот. Далее каждая ячейка  $D_{i,j}$  находится по правилу 1.1.

$$D_{i,j} = \min \begin{cases} (D) D_{i-1,j} + 1, \\ (I) D_{i,j-1} + 1, \\ (R) D_{i-1,j-1} + \begin{cases} 1, & \text{if } s1[i] == s2[j]; \\ 0, & \text{else} \end{cases} \end{cases} \quad (1.1)$$

Результатом будет являться правая нижняя ячейка в получившейся матрице. Можно заметить, что в выполнении этих действий участвуют только две строки: заполняемая и предыдущая. Поэтому для экономии памяти можно не хранить всю матрицу, а работать только с ними.

Далее приводится пример матрицы 1.2, составленной при сравнении двух строк: КОТ и СКАТ.

$$\begin{pmatrix} \dots & 0 & C & K & A & T \\ 0 & 0 & 1 & 2 & 3 & 4 \\ K & 1 & 1 & 1 & 2 & 3 \\ O & 2 & 2 & 2 & 2 & 3 \\ T & 3 & 3 & 3 & 3 & 2 \end{pmatrix} \quad (1.2)$$

Расстояние Левенштейна равняется двум. Действительно: 1) добавление в начало буквы 'С', 2) замена 'О' на 'А'.

### 1.3 Рекурсивный способ нахождения расстояния Левенштейна

Рекурсивный способ нахождения расстояния Левенштейна схож с матричным за исключением того, что используется рекурсивная формула

1.3 нахождения результата.

$$D(s1[1..i], s2[1..j]) = \begin{cases} 0, & \text{if } i == 0, j == 0; \\ i, & \text{if } i > 0, j == 0; \\ j, & \text{if } i == 0, j > 0; \\ \min \begin{cases} D(s1[1..i], s2[1..j-1]) + 1, \\ D(s1[1..i-1], s2[1..j]) + 1, \\ D(s1[1..i-1], s2[1..j-1]) + \begin{cases} 1, & \text{if } s1[i] == s2[j], \\ 0, & \text{else} \end{cases} \end{cases} \end{cases} \quad (1.3)$$

В функции 1.3:

- если обе строки пустые, то требуется 0 операций;
- если вторая строка пустая, то требуется удалить все символы первой строки;
- если первая строка пустая, то требуется вставить в пустоту все символы второй строки;
- иначе находится минимум среди:
  - суммы расстояния между первой строкой и второй, уменьшенной на 1, и единицы;
  - суммы расстояния между второй строкой и первой, уменьшенной на 1, и единицы;
  - суммы расстояния между первой и второй строками, уменьшенными на 1, и единицы в случае совпадения текущих рассматриваемых символов или нуля иначе.

К существенному недостатку использования данного метода можно отнести нерациональные затраты по времени: сложность алгоритма будет иметь экспоненциальную зависимость, при этом параметры в получающихся функциях могут повторяться, то есть будут повторно пересчитываться уже известные значения.

## 1.4 Рекурсивный способ нахождения расстояния Левенштейна с использованием кэширования

Решить проблему неэффективного использования рекурсивной формулы нахождения расстояния Левенштейна в виде повторного пересчитывания поможет кэширование. Кэширование - это высокоскоростной уровень хранения, на котором требуемый набор данных временного характера. [4] Благодаря наличию кэша, можно будет подставлять в формулу уже вычисленное ранее значение, если такое имеется. Существует множество способов кэширования, а также уже готовые решения.

## 1.5 Рекурсивный способ нахождения расстояния Дамерау-Левенштейна

Способ нахождения расстояний Дамерау-Левенштейна и Левенштейна аналогичны. В функции 1.3 в формулу нахождения минимума добавляется еще одна строка 1.4

$$\left[ \begin{array}{l} D(s1[1..i - 2], s2[1..j - 2]) + 1, \text{ if } i > 2, j > 2, s1[i - 2] == s2[j - 2], \\ \infty, \text{ else} \end{array} \right. \quad (1.4)$$

При невыполнении заданного условия будет присваиваться значение бесконечности, которая заведомо больше любого числа, то есть никак не повлияет на результат.

## 1.6 Вывод

Таким образом, разобраны способы нахождения расстояний Левенштейна и Дамерау-Левенштейна (отличие последнего состоит в добавлении одного условия), получены формулы. Редакционное расстояние можно получить как матрично, то есть итерационно, так и рекурсивно. Есть возможность сделать эффективней каждый из этих методов:



первый - путем хранения только двух строк матрицы, второй - путем кэширования.

## 2 Конструкторский раздел

В данном разделе представлены схемы алгоритмов нахождения редакционного расстояния: Левенштейна - с использованием матрицы, Левенштейна - рекурсивно, Левенштейна - рекурсивно с использованием кэша, Дамерау-Левенштейна - рекурсивно.

### 2.1 Алгоритм нахождения расстояния Левенштейна - матрично

На рисунках 2.1 - 2.2 представлена схема алгоритма нахождения расстояния Левенштейна с использованием матрицы.

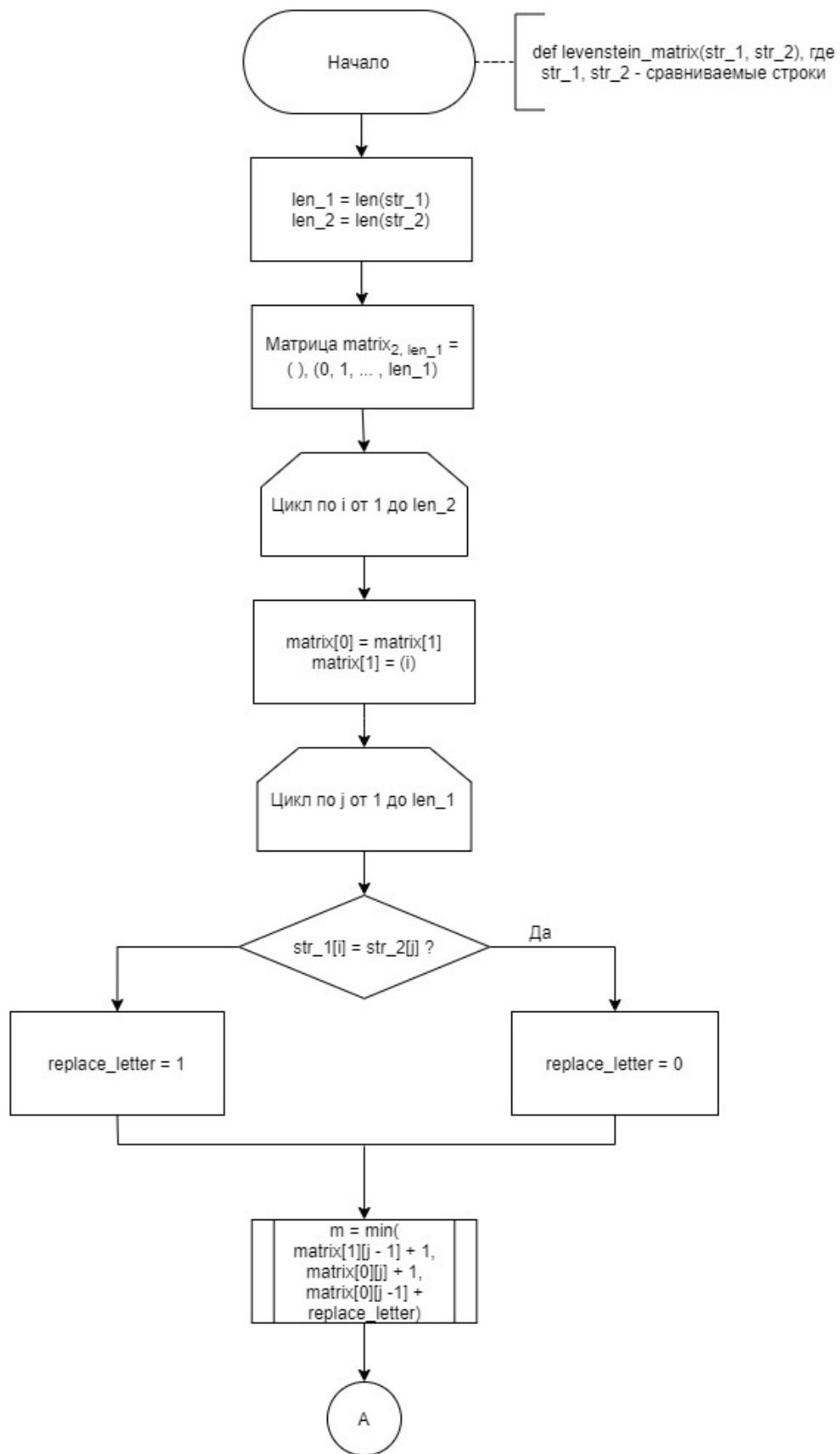


Рис. 2.1: Схема алгоритма нахождения расстояния Левенштейна - матричным способом (часть 1)

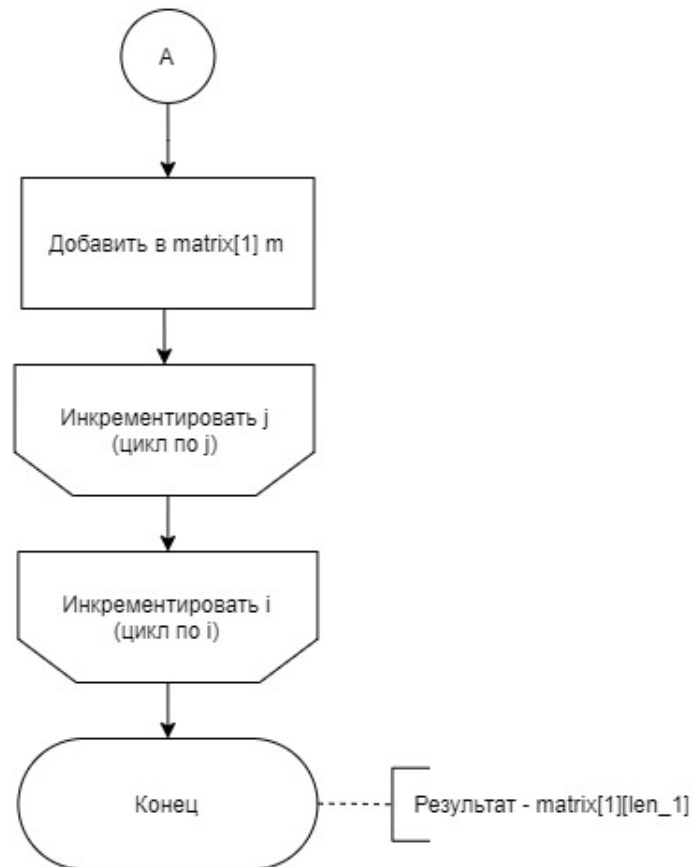


Рис. 2.2: Схема алгоритма нахождения расстояния Левенштейна - матричным способом (часть 2)

## 2.2 Алгоритм нахождения расстояния Левенштейна - рекурсивно

На рисунках 2.3 - 2.4 представлена схема алгоритма нахождения расстояния Левенштейна рекурсивно.

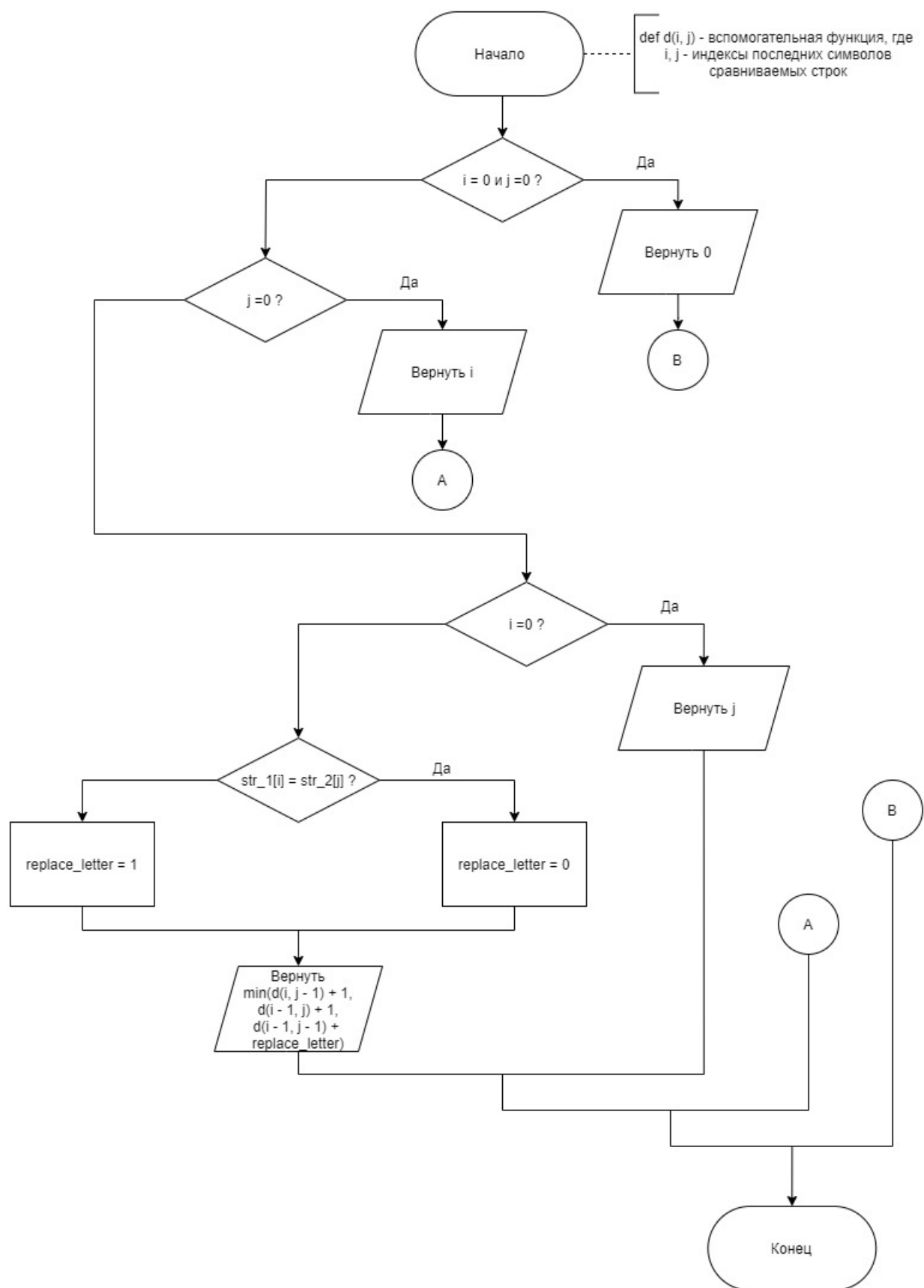


Рис. 2.3: Схема алгоритма нахождения расстояния Левенштейна рекурсивно (часть 1)

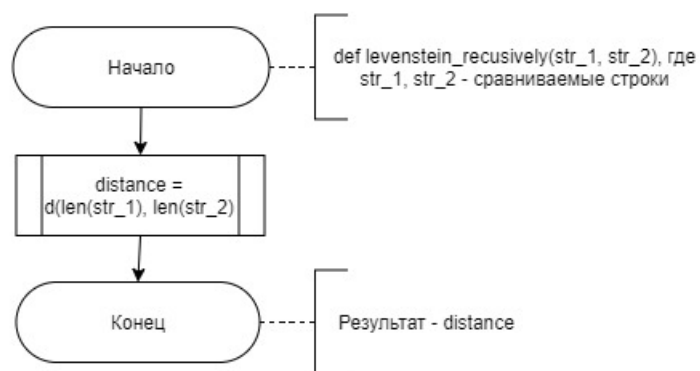


Рис. 2.4: Схема алгоритма нахождения расстояния Левенштейна рекурсивно (часть 2)

## 2.3 Алгоритм нахождения расстояния Левенштейна - рекурсивно с использованием кэша

На рисунках 2.5 - 2.7 представлена схема алгоритма нахождения расстояния Левенштейна рекурсивно с использованием кэширования.

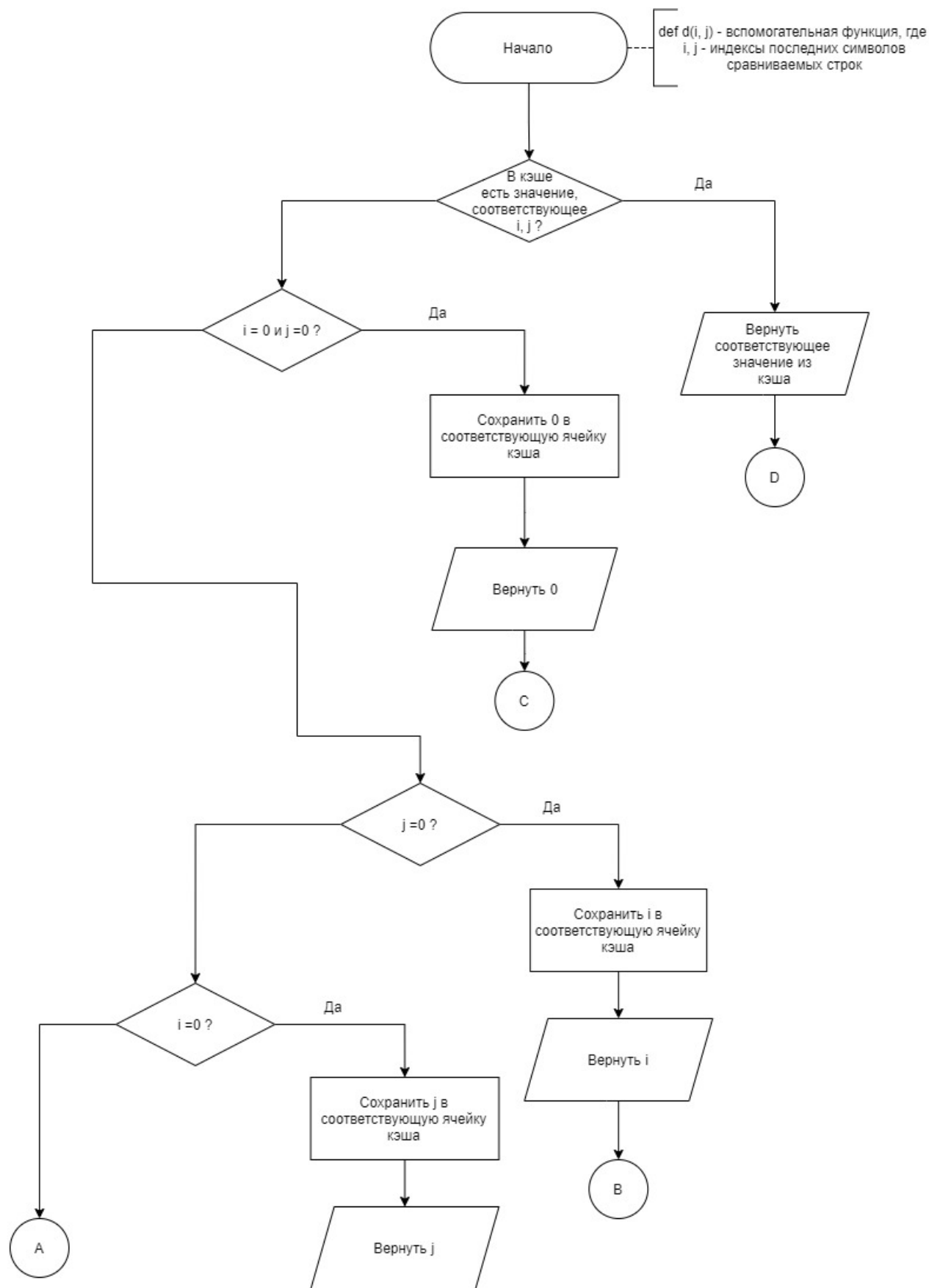


Рис. 2.5: Схема алгоритма нахождения расстояния Левенштейна рекурсивно с использованием кэша (часть 1)

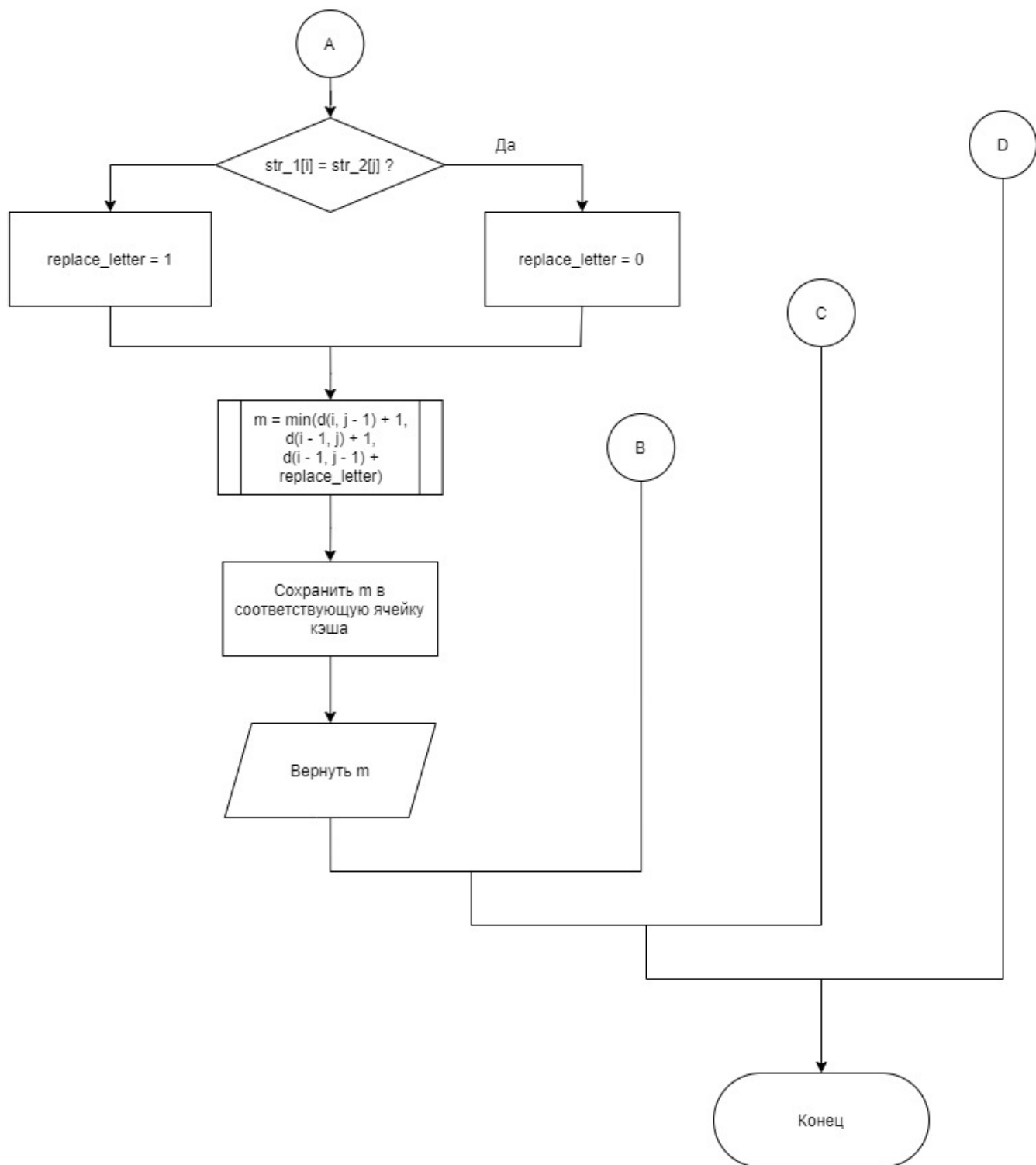


Рис. 2.6: Схема алгоритма нахождения расстояния Левенштейна рекурсивно с использованием кэша (часть 2)



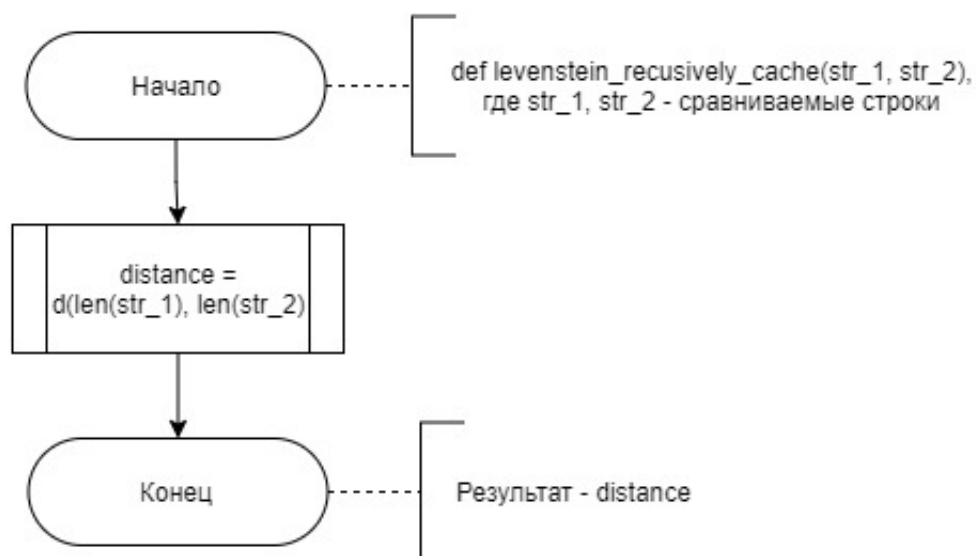


Рис. 2.7: Схема алгоритма нахождения расстояния Левенштейна рекурсивно с использованием кэша (часть 3)

## 2.4 Алгоритм нахождения расстояния Дameraу-Левенштейна - рекурсивно

На рисунках 2.8 - 2.10 представлена схема алгоритма нахождения расстояния Дameraу-Левенштейна рекурсивно.

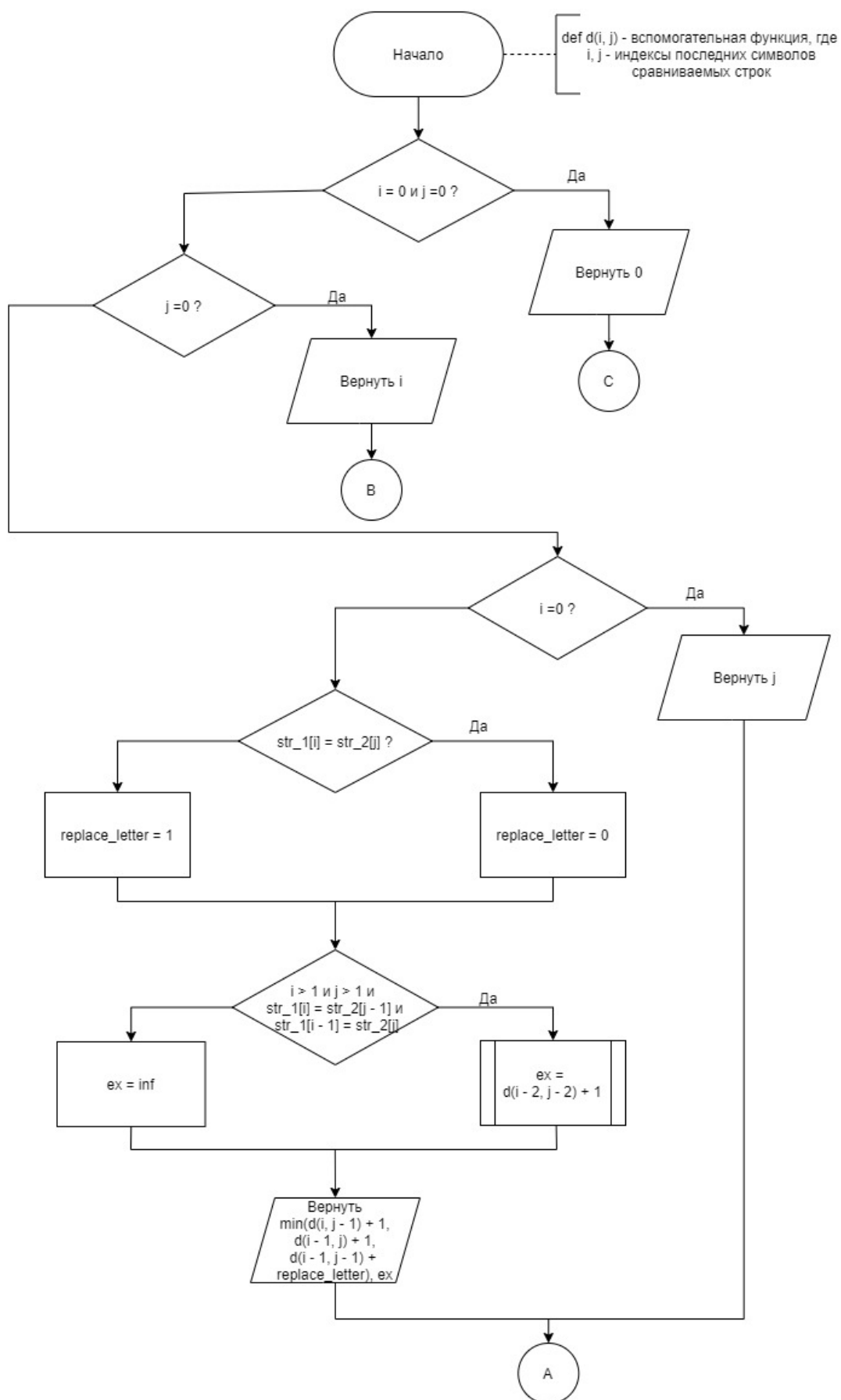


Рис. 2.8: Схема алгоритма нахождения расстояния Дameraу-Левенштейна рекурсивно (часть 1)

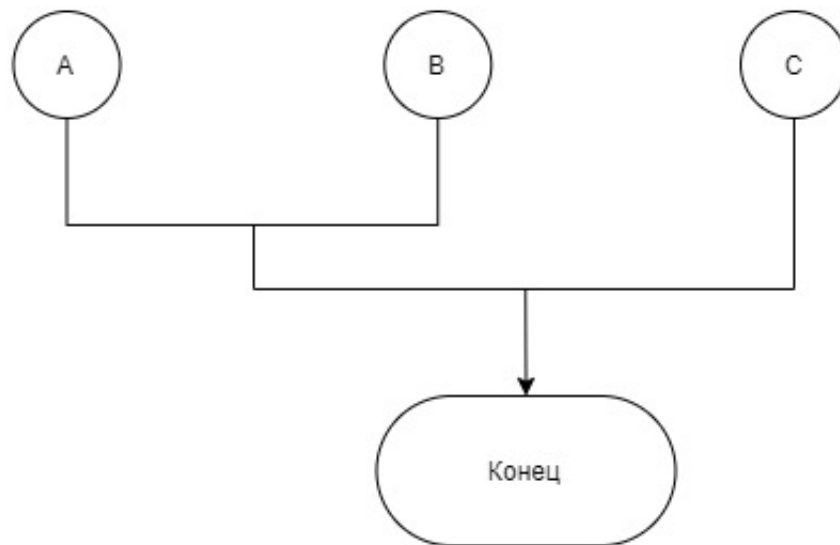


Рис. 2.9: Схема алгоритма нахождения расстояния Дameraу-Левенштейна рекурсивно (часть 2)

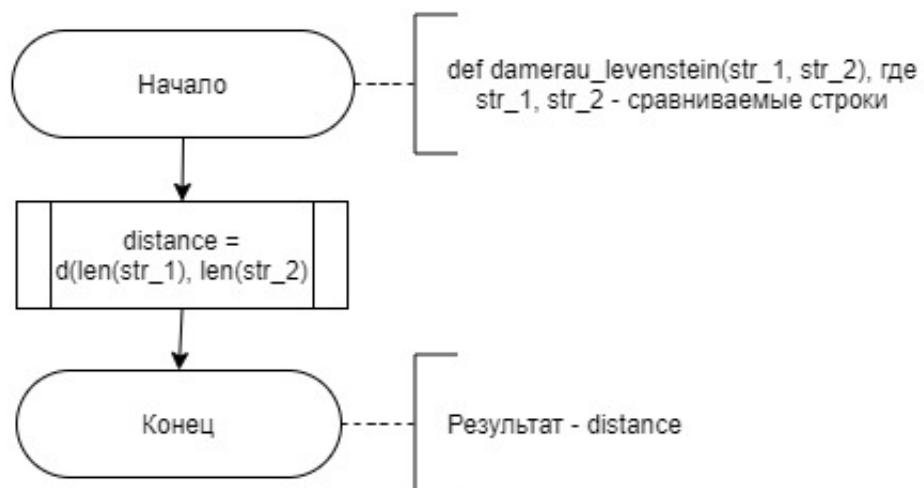


Рис. 2.10: Схема алгоритма нахождения расстояния Дameraу-Левенштейна рекурсивно (часть 3)

## 2.5 Вывод

Таким образом, были разработаны схемы алгоритмов нахождения расстояния Левенштейна тремя разными способами и расстояния Дамерау-Левенштейна - рекурсивно.

## 3 Технологический раздел

В данном разделе выбирается язык программирования и обонывается его выбор, вместе с этим подбираются необходимые библиотеки. Также предоставлены листинги реализованных алгоритмов.

### 3.1 Выбор языка программирования

В качестве языка программирования был выбран язык программирования Python. Благодаря динамической типизации и простому синтаксису, Python позволяет писать быстро и элегантно, позволяя программисту сосредоточиться на реализации самого алгоритма. Также имеется огромное количество библиотек, в том числе и для реализации графического интерфейса, например, PyQt5, на которой выполнен интерфейс реализуемой программы. Для построения графиков используется библиотека matplotlib.

Для реализации кэширования использовалась модуль библиотеки `functools - cache` (является декоратором, то есть функцией, который принимает в качестве аргумента другую функцию). Также на вход подается максимальный размер кэша. Если вызов функции с данными параметрами уже совершался, то возвращается значение из кэша. [5]

Для достижения задач, связанных с замером эффективности, выбраны библиотеки `time` (функция `process_time_ns()` - процессорное время в наносекундах [6]) и `tracemalloc` (позволяет узнать пиковое значение памяти с момента старта [7]).

## 3.2 Реализация алгоритма нахождения расстояния Левенштейна - матрично

На листинге 3.1 предоставлены реализации алгоритмов нахождения расстояния Левенштейна матричным способом.

Листинг 3.1: Реализация алгоритма Левенштейна матричным способом

```
1 def levenshtein_matrix(str_1, str_2):
2     len_1, len_2 = len(str_1), len(str_2)
3     matrix = [[], [i for i in range(len_1 + 1)]]
4
5     for i in range(1, len_2 + 1):
6         matrix[0], matrix[1] = matrix[1], [i]
7
8         for j in range(1, len_1 + 1):
9             replace_letter = 0 if str_2[i - 1] == str_1[j - 1] else 1
10            matrix[1].append(
11                min(
12                    matrix[1][j - 1] + 1,
13                    matrix[0][j] + 1,
14                    matrix[0][j - 1] + replace_letter
15                )
16            )
17
18     return matrix[1][len_1]
```

## 3.3 Реализация алгоритма нахождения расстояния Левенштейна - рекурсивно

На листинге 3.2 предоставлены реализации алгоритмов нахождения расстояния Левенштейна рекурсивно.

Листинг 3.2: Реализация алгоритма Левенштейна рекурсивным способом

```
1 def levenshtein_recursively(str_1, str_2):
2     def d(i, j):
3         if i == 0 and j == 0:
4             return 0
5         elif j == 0:
```

```

6         return i
7     elif i == 0:
8         return j
9     else:
10        replace_letter = 0 if str_1[i - 1] == str_2[j - 1] else 1
11        return min(
12            d(i, j - 1) + 1,
13            d(i - 1, j) + 1,
14            d(i - 1, j - 1) + replace_letter
15        )
16
17 distance = d(len(str_1), len(str_2))
18 return distance

```

### 3.4 Реализация алгоритма нахождения расстояния Левенштейна - рекурсивно с использованием кэша

На листинге 3.3 предоставлены реализации алгоритмов нахождения расстояния Левенштейна рекурсивно с использованием кэша.

Листинг 3.3: Реализация алгоритма Левенштейна рекурсивным способом с использованием кэширования

```

1 def levenshtein_recursively_cache(str_1, str_2):
2     @lru_cache(maxsize=len(str_1) * len(str_2))
3     def d(i, j):
4         if i == 0 and j == 0:
5             return 0
6         elif j == 0:
7             return i
8         elif i == 0:
9             return j
10        else:
11            replace_letter = 0 if str_1[i - 1] == str_2[j - 1] else 1
12            return min(
13                d(i, j - 1) + 1,
14                d(i - 1, j) + 1,
15                d(i - 1, j - 1) + replace_letter

```

```

16     )
17
18     distance = d(len(str_1), len(str_2))
19     return distance

```

### 3.5 Реализация алгоритма нахождения расстояния Дамерау-Левенштейна - рекурсивно

На листинге 3.4 предоставлены реализации алгоритмов нахождения расстояния Дамерау-Левенштейна рекурсивно.

Листинг 3.4: Реализация алгоритма Дамерау-Левенштейна рекурсивным способом

```

1 def damerau_levenshtein(str_1, str_2):
2     def d(i, j):
3         if i == 0 and j == 0:
4             return 0
5         elif j == 0:
6             return i
7         elif i == 0:
8             return j
9         else:
10            replace_letter = 0 if str_1[i - 1] == str_2[j - 1] else 1
11            if i > 1 and j > 1 and str_1[i - 1] == str_2[j - 2] and
               str_1[i - 2] == str_2[j - 1]:
12                exchange = d(i - 2, j - 2) + 1
13            else:
14                exchange = float('inf')
15
16            return min(
17                d(i, j - 1) + 1,
18                d(i - 1, j) + 1,
19                d(i - 1, j - 1) + replace_letter,
20                exchange
21            )
22
23     distance = d(len(str_1), len(str_2))

```



## 3.6 Вывод

Таким образом, был выбран язык программирования Python для реализации программы, выбраны соответствующие библиотеки, реализованы заданные алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна.

## 4 Исследовательский раздел

В данном разделе представлены тестовые наборы данных, примеры работы программы, замеры времени и пикового значения памяти для разного размера входных строк и для разных алгоритмов.

### 4.1 Тестовые наборы

В качестве демонстрации правильности работы алгоритмов протестирова

1. скот скат 2. домино дом 3. проект преок 4. программирование  
ппрrogrammovaин 5. дверь дверь 6. объект бъек 7. телефон тмлтфо 8.  
браво аво 9. поезд поезд 10. гарантия гарантия

## Заключение

## Список использованных источников

- [1] Редакционное расстояние [Электронный ресурс]. Режим доступа: <https://zadachi.mccme.ru/plak/plak-levenstein2.pdf>. Дата обращения: 07.09.2021.
- [2] Что такое редакционное расстояние [Электронный ресурс]. Режим доступа: <https://sysblok.ru/knowhow/что-такое-редакционное-rasstoianie/>. Дата обращения: 07.09.2021.
- [3] В.И. Левенштейн. *Двоичные коды с исправлением выпадений, вставок и замещений символов*. Доклады АН СССР, 1965.
- [4] Обзор кэширования. Режим доступа: <https://aws.amazon.com/ru/caching/>. Дата обращения: 08.09.2021.
- [5] Кэширование на python. Режим доступа: "<https://docs.python.org/3/library/functools.html>", note=.
- [6] Процессорное время на python. Режим доступа: [https://docs.python.org/3/library/time.html#time.process\\_time](https://docs.python.org/3/library/time.html#time.process_time). Дата обращения: 12.09.2021.
- [7] Отслеживание распределения памяти на python. Режим доступа: <https://docs.python.org/3/library/tracemalloc.html>. Дата обращения: 12.09.2021.

# Приложение