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»

«Расстояние Левенштейна и Дамерау-Левенштейна»

Студент Маслова Марина Дмитриевна

Группа ИУ7-53Б

Оценка (баллы) _____

Преподаватель Волкова Лилия Леонидовна

2021 г.

Оглавление

Введение

Расстояние Левенштейна — минимальное количество операций вставки, удаления и замены символа, необходимых для превращения одной строки в другую. Если к указанным операциям добавить перестановку двух соседних символов, получим определение *расстояния Дамерау-Левенштейна* [1]. Поиск каждой из этих характеристик основан на рекуррентных вычислениях, то есть на вычислениях, которые используют уже вычисленные значения для вычисления новых.

Таким образом, задача поиска данных расстояний основывается на методе динамического программирования — разбиении задач на более мелкие и простые подзадачи такого же вида, решение которых проводится один раз и далее используется при решении других задач и подзадач [2]. Поэтому изучение, разработка и реализация алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна позволит получить навыки использования данного метода.

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

Для достижения поставленной цели необходимо выполнить следующие **задачи**:

- изучить алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
- разработать алгоритмы поиска расстояния между строками;
- реализовать указанные алгоритмы;
- провести тестирование реализованных алгоритмов;
- провести сравнительный анализ алгоритмов по затрачиваемой памяти и процессорному времени работы реализации.

1 Аналитическая часть

В данном разделе представлено теоретическое описание алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна, а также рассмотрены области их применения.

1.1 Расстояние Левенштейна

Расстояние Левенштейна[1] между двумя строками — это минимальное количество операций вставки, удаления и замены символа, необходимых для превращения одной строки в другую.

Цена операции может зависеть от её вида и/или от участвующх в ней символов, что отражает разную вероятность различных ошибок при вводе текста и т. п. Для решения задачи поиска расстояния между двумя стракаами необходимо найти последовательность применяющихся операций, такую, что суммарная их цена будет минимальной. При вычислении расстояния Левенштейна используются следующие цены:

- $w(a, a) = 0$ — цена совпадения двух символов;
- $w(a, b) = 1,\ a \neq b$ — цена замены символа a на символ b;
- $w(\lambda, a) = 1$ — цена вставки символа a;
- $w(a, \lambda) = 1$ — цена удаления символа a.

1.1.1 Рекурсивный алгоритм

В рекурсивном алгоритме поиска расстояния Левенштейна между двумя строками искомая величина вычисляется через соответсвующие величины подстрок, а рекуррентная формула выводится из следующих рассуждений:

1) для перевода пустой строки в пустую требуется ноль операций;

2) для перевода пустой строки в строку s требуется $|s|$ операций вставки (здесь и далее $|s|$ обозначает длину строки);

3) для перевода строки s в пустую строку требуется $|s|$ операций удаления;

4) для перевода строки s_1 в строку s_2 требуется выполнить некоторую последовательность операций удаления, вставки или замены, при этом операции в оптимальной последовательности можно произвольно менять места-

ми, так как две последовательные операции любых видов можно переставить, что доказывается простым перебором вариантов возможных пар, поэтому без ограничения общности можно считать, что операция над последним символом была произведена последней и цена преобразования строки s_1 в строку s_2 будет являться минимальной ценой из цен, полученных одним из следующих способов (пусть при этом s_1' и s_2' — строки s_1 и s_2 без последнего символа, соответственно):

– сумма цены преобразования строки s_1 в s_2' и цены проведения операции вставки, которая необходима для преобразования s_2' в s_2;

– сумма цены преобразования строки s_1' в s_2 и цены проведения операции удаления, которая необходима для преобразования s_1 в s_1';

– сумма цены преобразования из s_1' в s_2' и операции замены, предполагая, что s_1 и s_2 оканчиваются на разные символы;

– цена преобразования из s_1' в s_2', предполагая, что s_1 и s_2 оканчиваются на один и тот же символ.

Таким образом, для расчета расстояния Левенштейна между двумя строками s_1 и s_2 используется рекуррентная формула для расчета через подстроки:

$$D(s_1[1..i], s_2[1..j]) = \begin{cases} 0, & i = j = 0 \\ j, & i = 0,\ j > 0 \\ i, & j = 0,\ i > 0 \\ \min\{ & \\ \quad D(s_1[1..i], s_2[1..j-1]) + 1, & \\ \quad D(s_1[1..i-1], s_2[1..j]) + 1, & \\ \quad D(s_1[1..i-1], s_2[1..j-1]) + l(s_1[i], s_2[j]) & \\ \} & \end{cases}, \tag{1.1}$$

где величина $l(a, b)$ выражется формулой:

$$l(a, b) = \begin{cases} 0, & \text{если } a = b \\ 1, & \text{иначе} \end{cases}. \tag{1.2}$$

1.1.2 Матричный алгоритм

При явной реализации формулы 1.1 через рекурсию многие вызовы будут производиться при одних и тех же значениях параметров, то есть большое количество вычислений будет повторяться и не один раз. Данную проблему решает матричный алгоритм поиска расстояния Левенштейна, который представляет собой построчное заполнение матрицы с размерами $N + 1$ на $M + 1$, где N и M — размеры исходной s_1 и получаемой s_2 строк соответственно, а в ячейке с координатами (i, j) находится расстояние между подстрокой исходной строки длины i, идущей от начала строки, и подстрокой получаемой строки длины j, также идущей от начала строки.

1.1.3 Рекурсивный алгоритм с кэшем

Проблему повторяющихся вычислений можно решить и при использовании рекурсии. Для этого достаточно создать кэш в виде матрицы, где будут храниться уже вычисленные значения. Если при выполении рекурсии происходит вызов с теми данными, которые ещё не были обработаны, то необходимое значение вычисляется и заносится в соответствующую ячейку матрицы. Если же данные уже были обработаны в дальнейших вычислениях участвует значение из матрицы. Таким образом, повторных вычислений не происходит.

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна[1] между двумя строками — это минимальное количество операций ставки, удаления, замены и транспозиции (перестановки двух соседних) символов, необходимых для превращения одной строки в другую.

1.2.1 Рекурсивный алгоритм

Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна полностью аналогичен алгоритму поиска расстояния Левенштейна, за исключением дополнительной операции транспозиции. Для её учета в формулу 1.1 добавляется рассмотрение случаев с возможной транспозицией последних символов. После добавления выражений, учитывающих дополнительную опера-

ции, получаем математическое представление поиска расстояния Дамерау-Левенштейна:

$$D(s_1[1..i], s_2[1..j]) = \begin{cases} 0, \quad i = j = 0 \\ j, \quad i = 0,\ j > 0 \\ i, \quad j = 0,\ i > 0 \\ \min\{ \\ \qquad D(s_1[1..i], s_2[1..j-1]) + 1, \\ \qquad D(s_1[1..i-1], s_2[1..j]) + 1, \\ \qquad D(s_1[1..i-1], s_2[1..j-1]) + l(s_1[i], s_2[i]), \\ \qquad D(s_1[1..i-2], s_2[1..j-2]) + 1, \\ \}, \text{ если } i, j > 1 \text{ и } s_1[i] = s_2[j-1] \text{ и } s_1[i-1] = s_2[j] \\ \min\{ \\ \qquad D(s_1[1..i], s_2[1..j-1]) + 1, \\ \qquad D(s_1[1..i-1], s_2[1..j]) + 1, \\ \qquad D(s_1[1..i-1], s_2[1..j-1]) + l(s_1[i], s_2[j]) \\ \}, \text{ иначе} \end{cases} . \tag{1.3}$$

1.3 Области применения алгоритмов

Поиск расстояний Левенштейна и Дамерау-Левенштейна широко используется в теории информации и компьютерной лингвистике. Так, он применяется:

– для автозамены, исправленя ошибок в словах (поисковые системы, базы данных, ввод текста, автоматическое распознавание отсканированного текса или речи);

– для сравнения текстовых файлов утилитой *diff*;

– в биоинформатике для сравнения генов, хромосом и белков.

2 Конструкторская часть

В данном разделе разрабатываются алгоритмы, а также производится сравнительный анализ рекурсивной и нерекурсивной их реализации на примере алгоритма поиска расстояния Левенштейна.

2.1 Разработка алгоритмов

В данном подразделе приводятся схемы разработанных алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна.

2.1.1 Схемы алгоритмов поиска расстояния Левенштейна

Схема рекурсивного алгоритма поиска расстояния Левенштейна представлена на рисунке 2.1.

В двух следующих алгоритмах используется матрица для сохранения результатов вычислений, так как в обоих матрица инициализируется одним и тем же способом на рисунке 2.2 представен алгоритм инициализации матрицы.

Матричный алгоритм и рекурсивный алгоритм с кэшем для поиска расстояния Левенштейна приведены на рисунках 2.3 и 2.4-2.5 соответственно.

2.1.2 Схема алгоритма поиска расстояния Дамерау-Левенштейна

Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна представлена на рисунке 2.6.

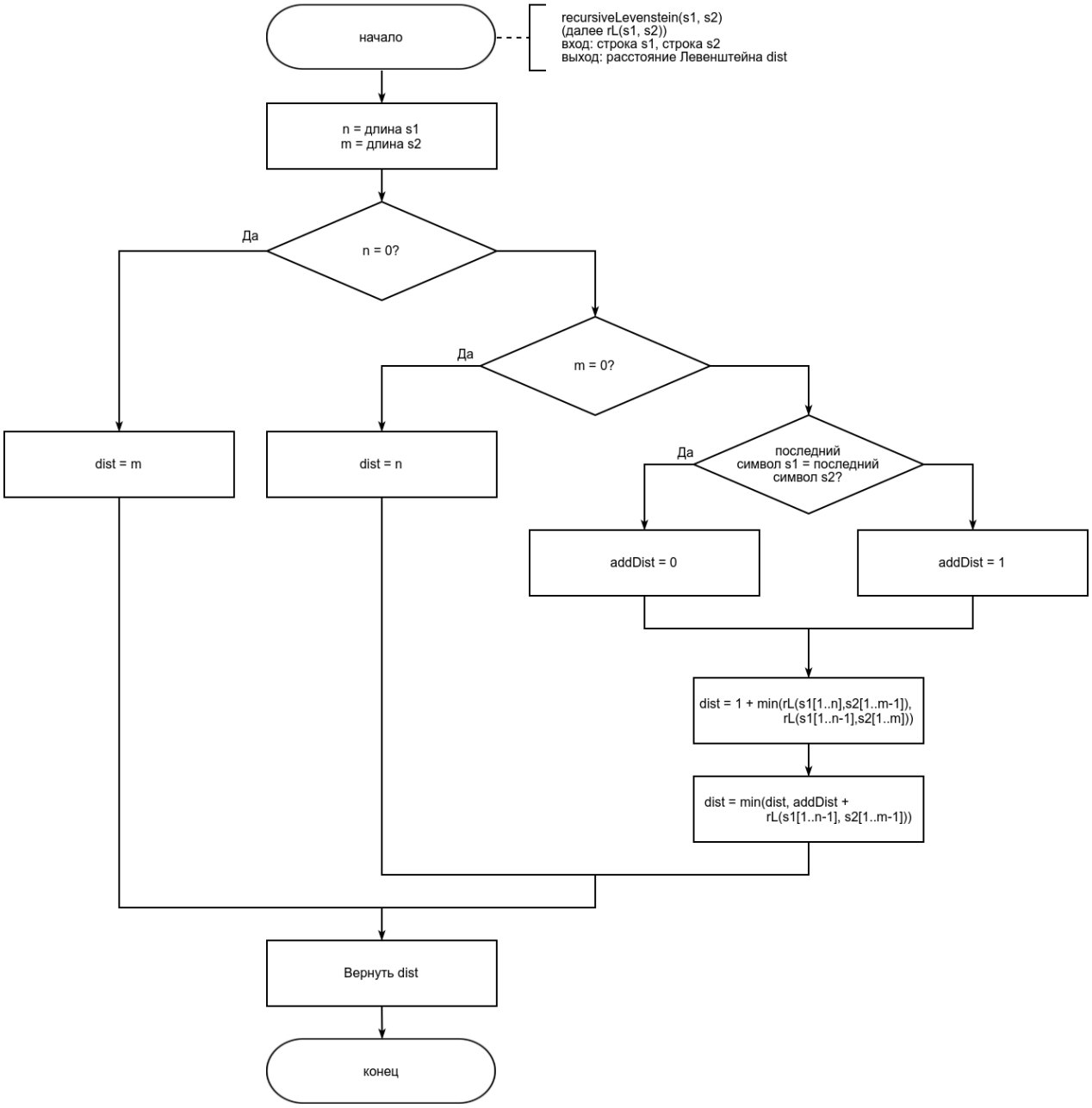

Рисунок 2.1 – Схема рекурсивного алгоритма поиска расстояния Левенштейна

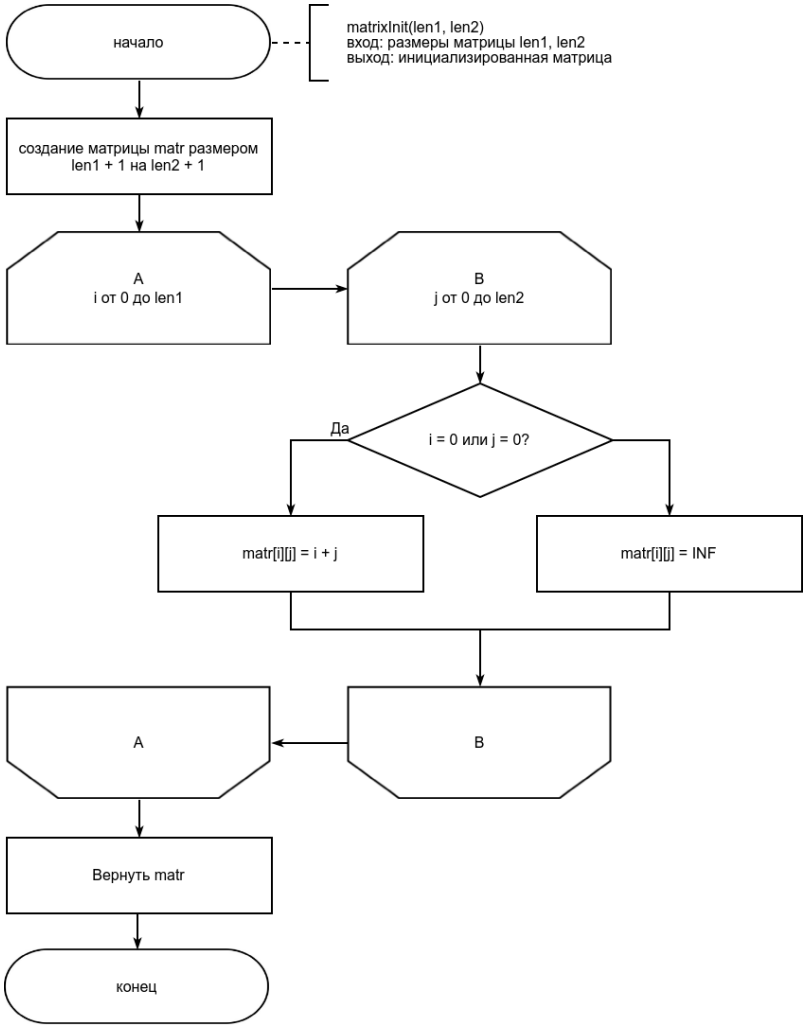

Рисунок 2.2 – Схема алгоритма инициализации матрицы

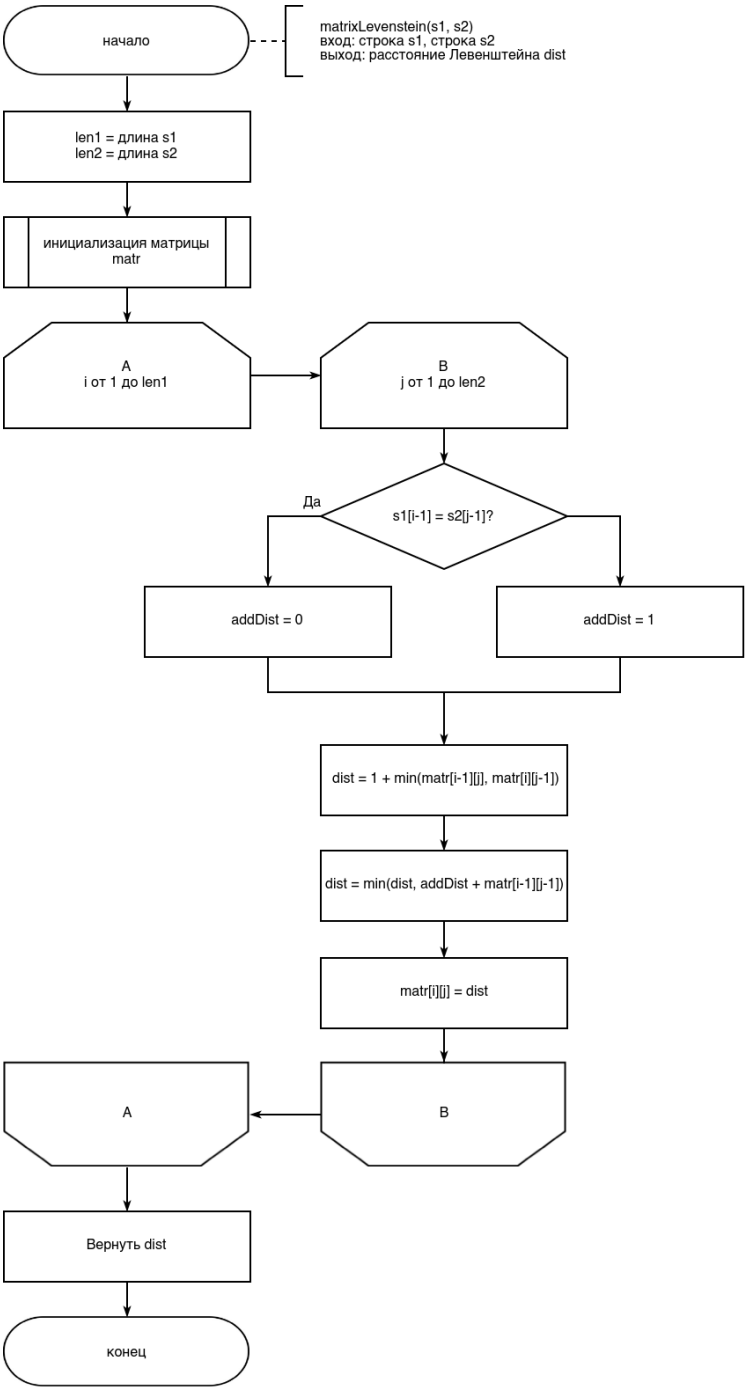

Рисунок 2.3 – Схема матричного алгоритма поиска расстояния Левенштейна

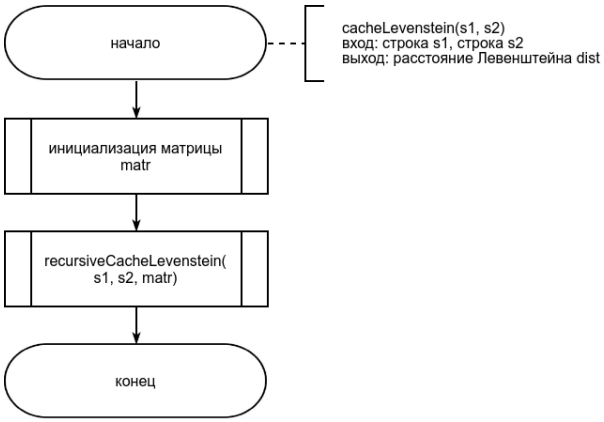

Рисунок 2.4 – Схема рекурсивного алгоритма поиска расстояния Левенштейна с кэшем

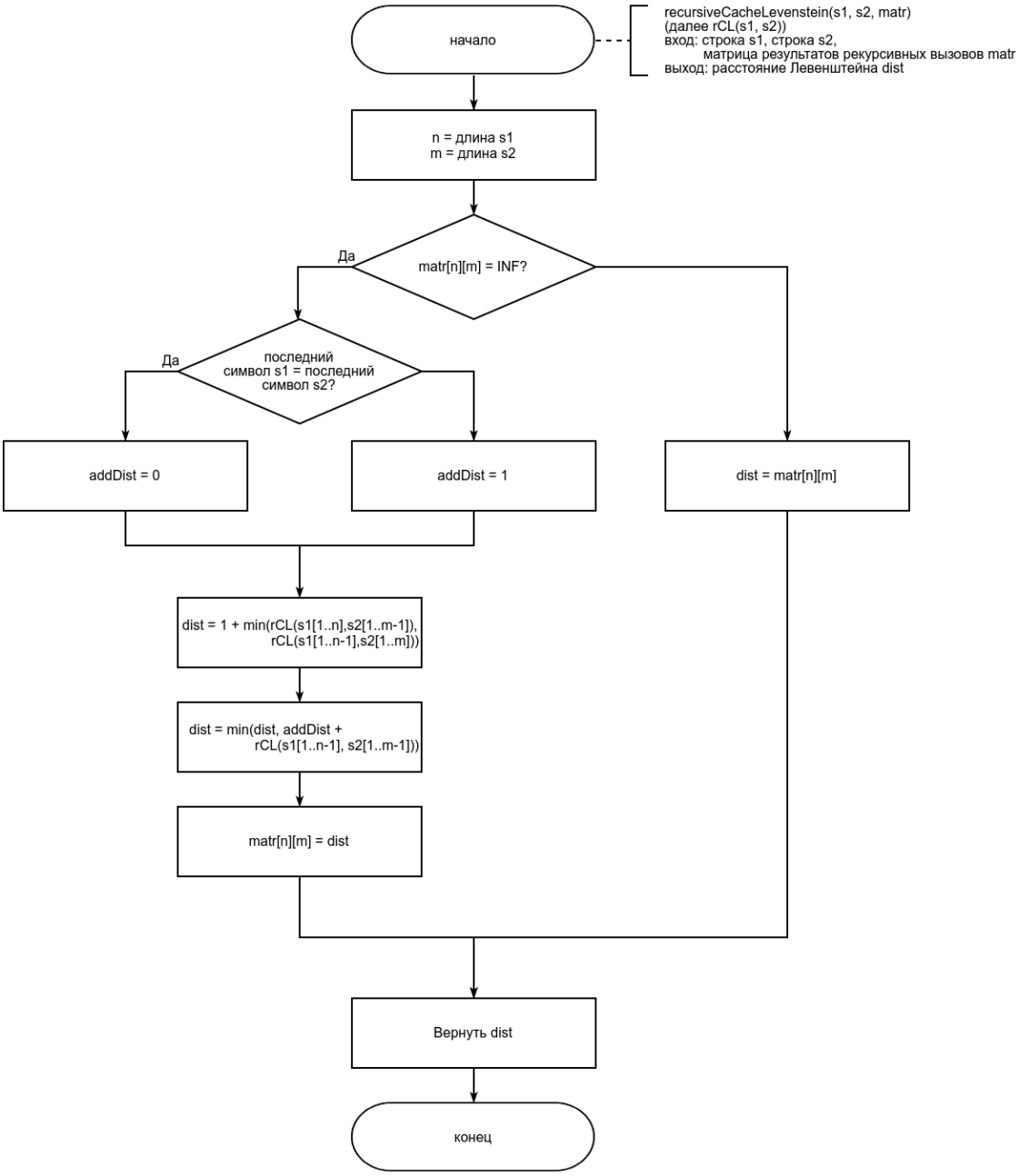

Рисунок 2.5 – Схема рекурсивной подпрограммы алгоритма поиска расстояния Левенштейна с кэшем

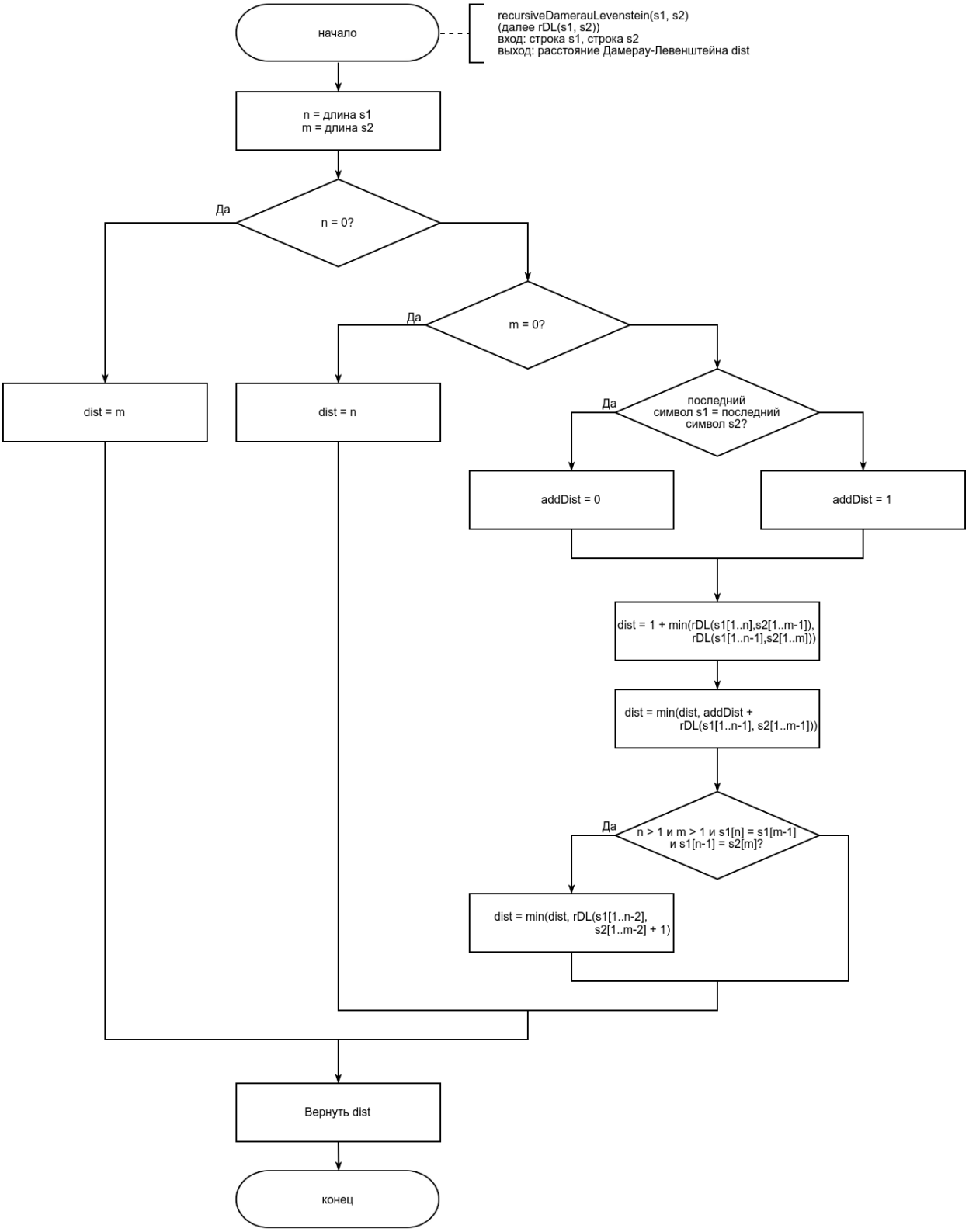

Рисунок 2.6 – Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна

3 Технологическая часть

В данном разделе описаны требования к программному обеспечению, средства реализации, приведены листинги кода и данные, на которых будет проводиться тестирование.

3.1 Требования к программному обеспечению

Программа должна предоставлять следующие возможности:

- выбор режима работы: для единичного эксперимента и для массовых экспериментов;
- в режиме единичного эксперимента ввод двух строк на русском или английском языках и вывод полученных разными реализациями расстояний;
- в режиме массовых экспериментов измерение времени при различных длинах строк и построение графиков по полученным данным.

3.2 Средства реализации

Для реализации данной лабораторной работы выбран интерпретируемый язык программирования высокого уровня Python, так как он позволяет реализовывать сложные задачи за кратчайшие сроки за счет простоты синтаксиса и наличия большого количества подключаемых библиотек.

В качестве среды разработки выбран текстовый редактор vim с установленными плагинами автодополнения и поиска ошибок в процессе написания, так как он реализует быстрое перемещение по тексту программы и простое взаимодействие с командной строкой.

3.3 Листинги кода

В данном подразделе представлены листинги кода ранее описанных алгоритмов:

- рекурсивный алгоритм поиска расстояния Левенштейна (листинг 3.1);
- матричный алгоритм поиска расстояния Левенштейна (листинги 3.2-3.3);

– рекурсивный алгоритм поиска расстояния Левенштейна с кэшем (листинги 3.2, 3.4-3.5);
– рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна (листинг 3.6).

Листинг 3.1 – Реализация рекурсивного алгоритма поиска расстояния Левенштейна

```
def recursiveLevenstein(str1, str2):
    len1 = len(str1)
    len2 = len(str2)

    dist = 0
    if len1 == 0:
        dist = len2
    elif len2 == 0:
        dist = len1
    else:
        addDist = 0 if str1[-1] == str2[-1] else 1
        dist = 1 + min(recursiveLevenstein(str1, str2[:-1]),
                       recursiveLevenstein(str1[:-1], str2))
        dist = min(dist, addDist + recursiveLevenstein(str1[:-1], str2[:-1]))

    return dist
```

Листинг 3.2 – Реализация инициализации матрицы

```
def matrixInit(len1, len2):
    matrix = [[i + j if i * j == 0 else INF for j in range(len2 + 1)]
              for i in range(len1 + 1)]

    return matrix
```

Листинг 3.3 – Реализация матричного алгоритма поиска расстояния Левенштейна

```
def matrixLevenstein(str1, str2):
    len1 = len(str1)
    len2 = len(str2)

    matr = matrixInit(len1, len2)
    dist = 0

    for i in range(1, len1 + 1):
        for j in range(1, len2 + 1):
            addDist = 0 if str1[i - 1] == str2[j - 1] else 1
            dist = 1 + min(matr[i - 1][j], matr[i][j - 1])
            dist = min(dist, addDist + matr[i - 1][j - 1])
```

```
13              matr[i][j] = dist
14
15      return dist, matr
```

Листинг 3.4 – Реализация инициализации данных для рекурсивного алгоритма поиска расстояния Левенштейна с кэшем

```
1  def cacheLevenstein(str1, str2):
2      matr = matrixInit(len(str1), len(str2))
3
4      dist = recursiveCacheLevenstein(str1, str2, matr)
5
6      return dist
```

Листинг 3.5 – Реализация рекурсивного алгоритма поиска расстояния Левенштейна с кэшем

```
1  def recursiveCacheLevenstein(str1, str2, matr):
2      len1 = len(str1)
3      len2 = len(str2)
4
5      if matr[len1][len2] == INF:
6          addDist = 0 if str1[-1] == str2[-1] else 1
7          dist = 1 + min(
8                          recursiveCacheLevenstein(str1, str2[:-1], matr),
9                          recursiveCacheLevenstein(str1[:-1], str2, matr),
10                     )
11         dist = min(
12                     dist,
13                     addDist
14                     + recursiveCacheLevenstein(str1[:-1], str2[:-1], matr),
15                 )
16         matr[len1][len2] = dist
17     else:
18         dist = matr[len1][len2]
19
20     return dist
```

Листинг 3.6 – Реализация рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна

```
1  def recursiveDamerauLevenstein(str1, str2):
2      len1 = len(str1)
3      len2 = len(str2)
4      dist = 0
5
6      if len1 == 0:
7          dist = len2
```

```
    elif len2 == 0:
        dist = len1
    else:
        addDist = 0 if str1[-1] == str2[-1] else 1
        dist = 1 + min(recursiveDamerauLevenstein(str1, str2[:-1]),
                       recursiveDamerauLevenstein(str1[:-1], str2))
        dist = min(dist, addDist
                         + recursiveDamerauLevenstein(str1[:-1], str2[:-1]))

        if (len1 > 1 and len2 > 1
                and str1[-1] == str2[-2] and str1[-2] == str2[-1]):
            dist = min(
                       dist,
                       recursiveDamerauLevenstein(str1[:-2], str2[:-2]) + 1
                   )

    return dist
```

3.4 Описание тестирования

В таблице 3.1 приведены функциональные тесты для алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

Таблица 3.1 – Функциональные тесты

Строка 1	Строка 2	Ожидаемый результат	
		Левенштейн	Дамерау-Левенштейн
’hello’	’hello’	0	0
”	”	0	0
”	’string’	6	6
’string’	”	6	6
”	’строка’	6	6
’строка’	”	6	6
’dif’	’str’	3	3
’разные’	’строки’	6	6
’len’	’diff’	4	4
’количество’	’разн’	6	6
’swap’	’sawp’	2	1
’wspa’	’swap’	4	2
’смена’	’мсена’	2	1
’мсеан’	’смена’	4	2
’add’	’adds’	1	1
’delete’	’dele’	2	2
’insrt’	’insert’	1	1
’insert’	’insrt’	1	1
’добав’	’добавить’	3	3
’удалитьош’	’удалить’	2	2
’вствить’	’вставить’	1	1
’внутери’	’внутри’	1	1
’роза’	’поза’	1	1
’bed’	’bad’	1	1
’лиса’	’сила’	2	2
’principal’	’principle’	2	2
’РеГисТР’	’регистр’	0	0
’LOW’	’low’	0	0

Литература

[1] Черненький В. М., Гапанюк Ю. Е. Методика идентификации пассажира по установочным данным // Инженерный журнал: наука и инновации. 2012. № 3. С. 30–39.

[2] Окулов С. М., Пестов О. А. Динамическое программирование. М.: БИНОМ. Лаборатория знаний, 2012. с. 296.