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 Расстояния Левенштейна и Дамерау-Левенштейна

Студент Артюхин Н.П.

Группа ИУ7-51Б

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

Москва — 2022 г.

Оглавление

Введение

Цель лабораторной работы - изучение метода динамического программирования на материале расстояний Дамерау-Левенштейна и Левенштейна.

Расстояние Левенштейна (редакционное расстояние) — метрика, измеряющая по модулю разность между двумя последовательностями символов. Оно определяется как минимальное количество редакторских операций, необходимых для преобразования одной строки в другую.

Редакторские операции:

1) вставка одного символа;
2) удаление одного символа;
3) замена 1 символа.

Операциям, используемым в преобразовании, можно назначить свои цены (штрафы).

Расстояние Дамерау — Левенштейна является модификацией расстояния Левенштейна, а именно к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции, то есть перестановки даух соседних символов.

Применение редакционных расстояний:

- компьютерная лингвистика (например, автозамена в поисковых запросах);
- биоинформатика (например, анализ иммунитета, сравнение генов).

Для достижения поставленной цели необходимо решить следующие задачи:

1) изучение расстояний Левенштейна и Дамерау-Левенштейна;
2) разработка алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна;

3) реализация одного алгоритма поиска расстояния Левенштейна (нерекурсивный с заполнением матрицы расстояний), трех алгоритмов поиска расстояния Дамерау-Левенштейна (нерекурсивный, рекурсивный без кэширования, рекурсивный с кэшированием);
4) выполнение оценки затрат алгоритмов по памяти;
5) выполнение замеров процессорного времени работы реализаций алгоритмов: поиска расстояния Левенштейна (нерекурсивный), поиска расстояния Дамерау-Левенштейна (нерекурсивный, рекурсивный без кэширования, рекурсивный с кэшированием);
6) сравнительный анализ нерекурсивных алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна, рекурсивных алгоритмов поиска расстояния Дамерау-Левенштейна (с кэшированием и без) по затрачиваемым ресурсам.

1 Аналитическая часть

В данном разделе будут представлены описания алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

Расстояния Левенштейна и Дамерау–Левенштейна – это минимальное количество редакторских операций, необходимых для преобразования одной строки в другую. Различаются данные расстояния только набором допустимых операций.

Введем следующие обозначения операций (в скобках указана цена операции (штраф)):

- D (delete) — удаление одного символа (штраф 1);
- I (insert) — вставка одного символа (штраф 1);
- R (replace) — замена одного символа (штраф 1);
- X (exchange) — перестановка соседних символов (штраф 1) – только для расстояния Дамерау-Левенштейна.

Также введем еще одно обозначение M (match) – совпадение (штраф 0).

Задача нахождения расстояний Левенштейна и Дамерау–Левенштейна сводится к поиску последовательности операций, дающих в сумме минимально возможный штраф. Данную задачу можно решить с помощью рекуррентных формул, которые будут рассмотрены далее в текущем разделе.

1.1 Расстояние Левенштейна

Пусть дано две строки S_1 и S_2 длиной L_1 и L_2 соответственно. Тогда расстояние Левенштейна можно найти по рекуррентной формуле (1.1):

$$D(S_1[1...i], S_2[1...j]) = \begin{cases} \text{0, i = 0, j = 0} \\ \text{j, i = 0, j > 0} \\ \text{i, i > 0, j = 0} \\ min(\\ D(S_1[1...i], S_2[1...j-1]) + 1, \\ D(S_1[1...i-1], S_2[1...j]) + 1, \\ D(S_1[1...i-1], S_2[1...j-1]) + \\ \left[\begin{array}{l} \text{0, } S_1\text{[i] == } S_2\text{[j]} \\ \quad\text{1, иначе} \end{array}\right. \\ \text{), иначе} \end{cases} \tag{1.1}$$

Первые три формулы в системе (1.1) описывают тривиальные случаи:

- совпадение строк, так как обе строки пусты - M (match);
- вставка j символов в пустую строку S_1 для создания строки-копии S_2 длиной j;
- удаление всех (i символов) из строки S_1 для совпадения с пустой строкой S_2.

В последней формуле из системы (1.1) необходимо выбрать минимум из штрафов:

- операция вставки символа (I) в S_1,
- операция удаления символа (D) из S_1,
- совпадение (M, штраф отсутствует) или операция замены (R), в зависимости от равенства рассматриваемых на данном этапе символов строк [1].

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна между строками S_1 и S_2 (длиной L_1 и L_2 соответственно) рассчитывается по схожей с (1.1) рекуррентной формуле, добавится еще один возможный вариант в группу min (1.2):

$$\left[\begin{array}{c} D(S_1[1...i-2], S_2[1...j-2]) + 1, \text{ если i, j>1, } a_{i-1} = b_j, a_i = b_{j-1} \\ \infty \text{ , иначе} \end{array} \right. \tag{1.2}$$

Формула (1.2) означает перестановку двух соседних символов в S_1, если длины обеих строк $L_1, L_2 > 1$, и соседние рассматриваемые символы в строках S_1 и S_2 крест-накрест равны. Иначе, если хотя бы одно из условий не выполняется, данная оперция не учитывается при поиске минимума, что и обозначет ∞ в формуле (1.2).

Полученная формула для поиска расстояния Дамерау-Левенштейна (1.3):

$$D(S_1[1...i], S_2[1...j]) = \begin{cases} 0, \text{ i} = 0, \text{ j} = 0 \\ \text{j, i} = 0, \text{ j} > 0 \\ \text{i, i} > 0, \text{ j} = 0 \\ min(\\ D(S_1[1...i], S_2[1...j-1]) + 1, \\ D(S_1[1...i-1], S_2[1...j]) + 1, \\ D(S_1[1...i-1], S_2[1...j-1]) + \\ \left[\begin{array}{c} 0, \; S_1[\text{i}] == S_2[\text{j}] \\ 1, \text{ иначе} \end{array} \right. \\ , \\ \left[\begin{array}{c} D(S_1[1...i-2], S_2[1...j-2]) + 1, \\ i, j > 1, \text{ a}_i = b_{j-1}, b_j = a_{i-1}; \\ \infty \text{ , иначе} \end{array} \right. \\ ), \text{ иначе} \end{cases} \tag{1.3}$$

Вывод

В данном разделе были рассмотрены основные материалы и формулы, которые потребуются далее при разработке и реализации алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

2 Конструкторская часть

При разработке алгоритмов, решающих задачи поиска расстояний Левенштейна и Дамерау-Левенштейна, можно использовать несколько различных подходов: итеративный, рекурсивный с кешированием, рекурсивный без кеширования, которые будут рассмотрены в текущем разделе.

2.1 Алгоритм поиска расстояния Левенштейна

На рисунке 2.1 приведена схема итеративного алгоритма поиска расстояния Левенштейна с заполнением матрицы расстояний.

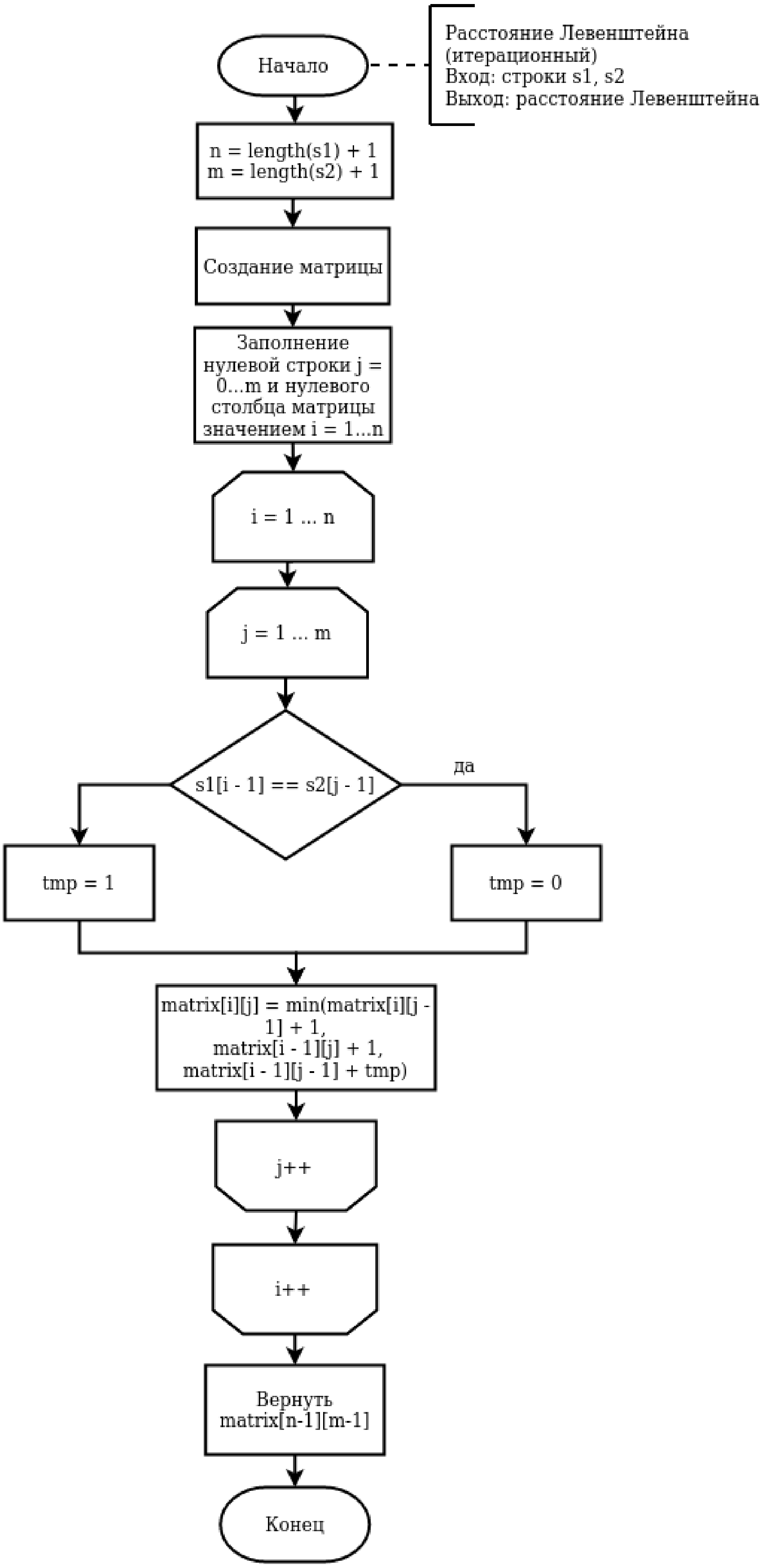


Рисунок 2.1 – Схема итеративного алгоритма поиска расстояния Левенштейна

2.2 Алгоритмы поиска расстояния Дамерау - Левенштейна

На рисунке 2.2 приведена схема итеративного алгоритма поиска расстояния Дамерау-Левенштейна с заполнением матрицы расстояний, на рисунке 2.3 – схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна без кеширования и на рисунках 2.4 − 2.6 – схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кешированием.

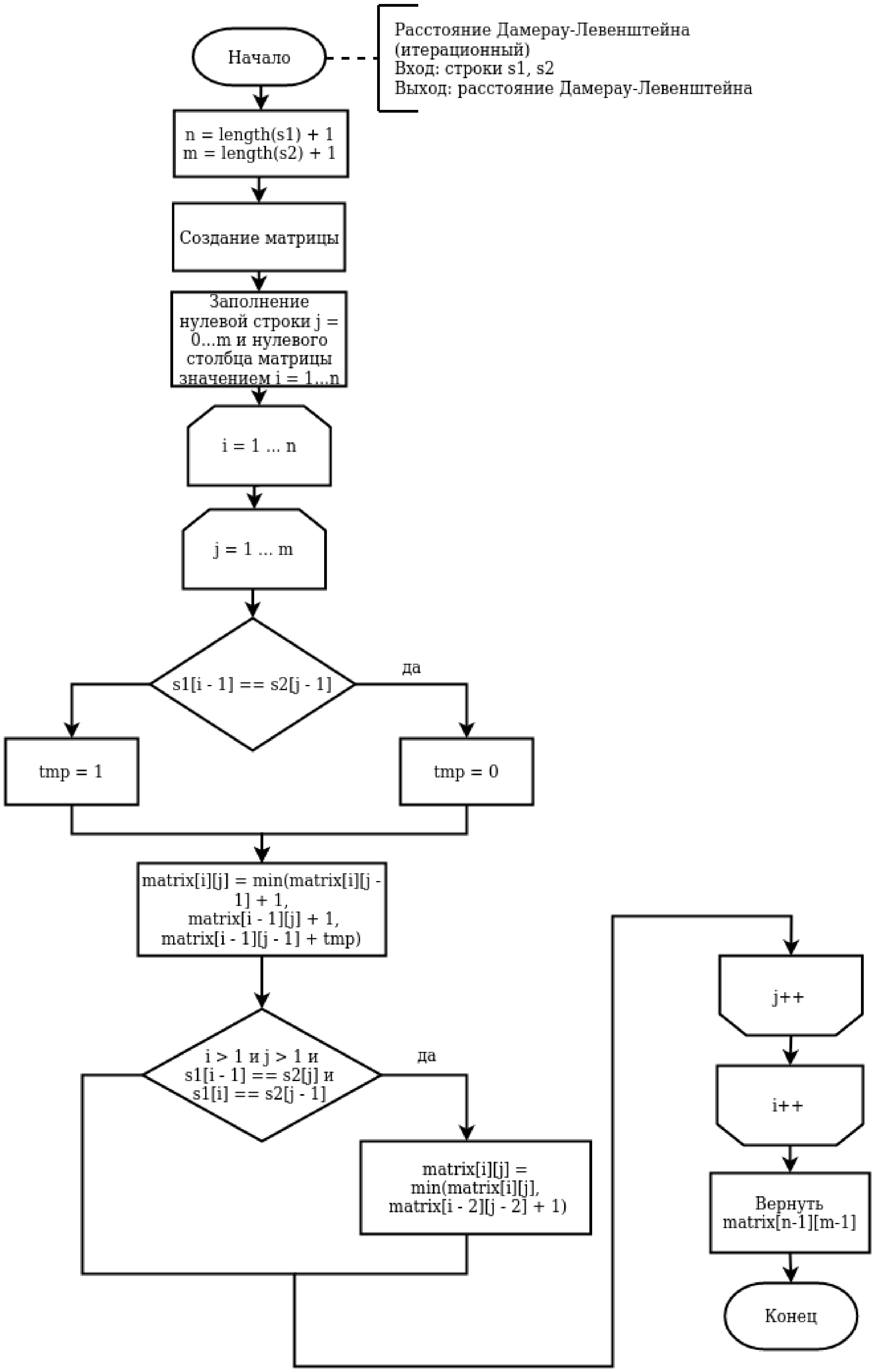


Рисунок 2.2 – Схема итеративного алгоритма поиска расстояния Дамерау-Левенштейна

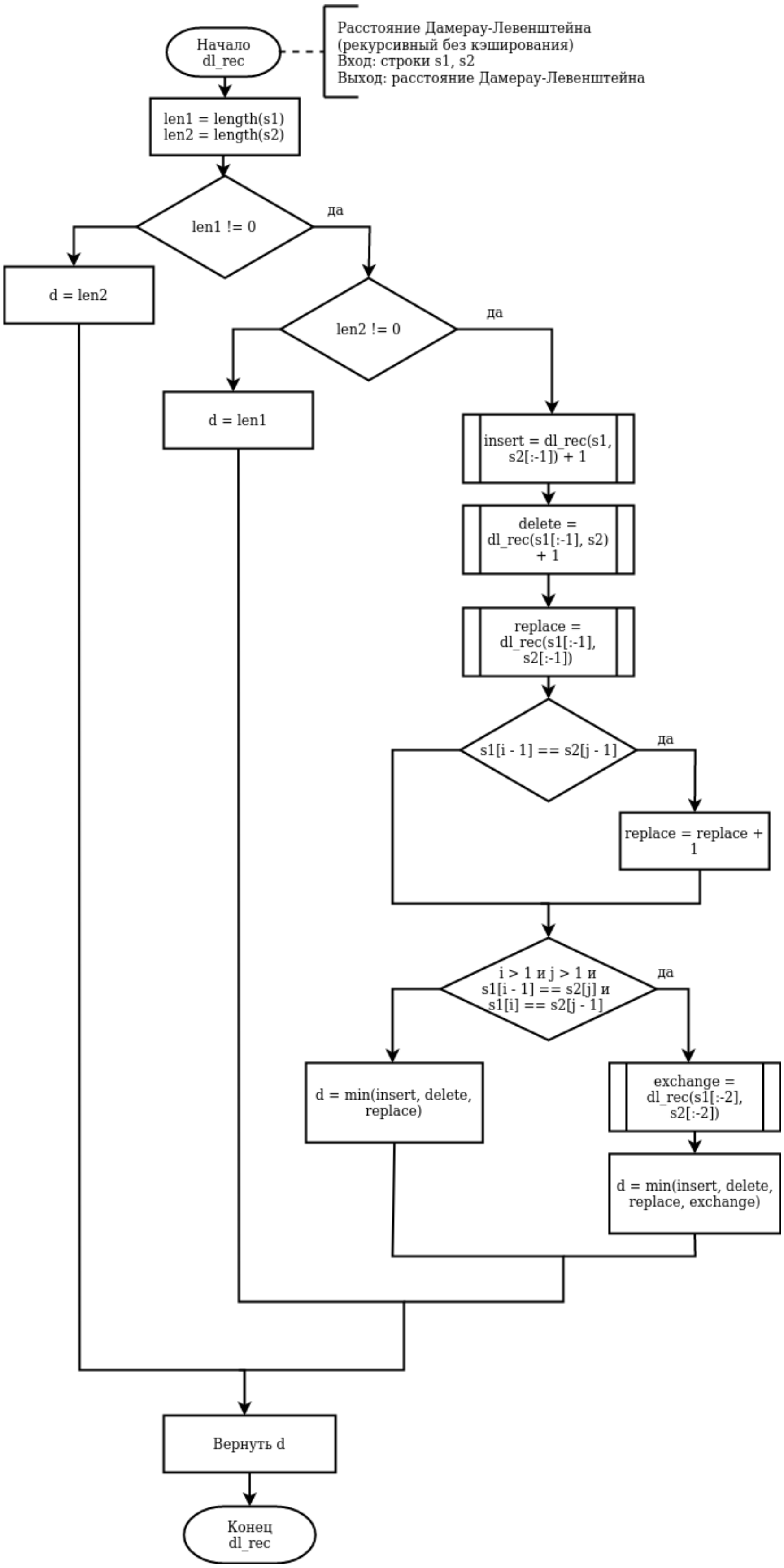


Рисунок 2.3 – Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна без кеширования

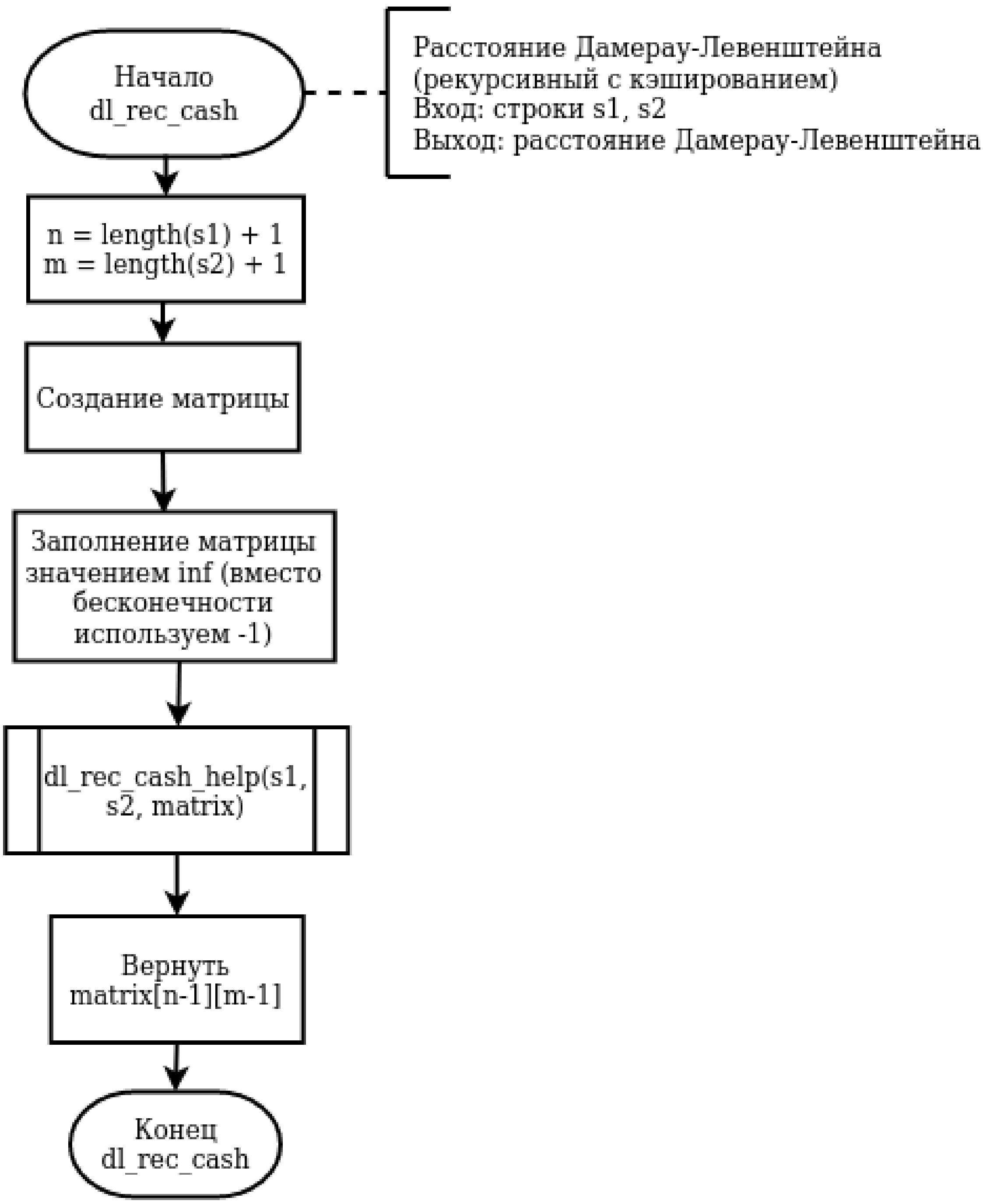


Рисунок 2.4 – Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кешированием

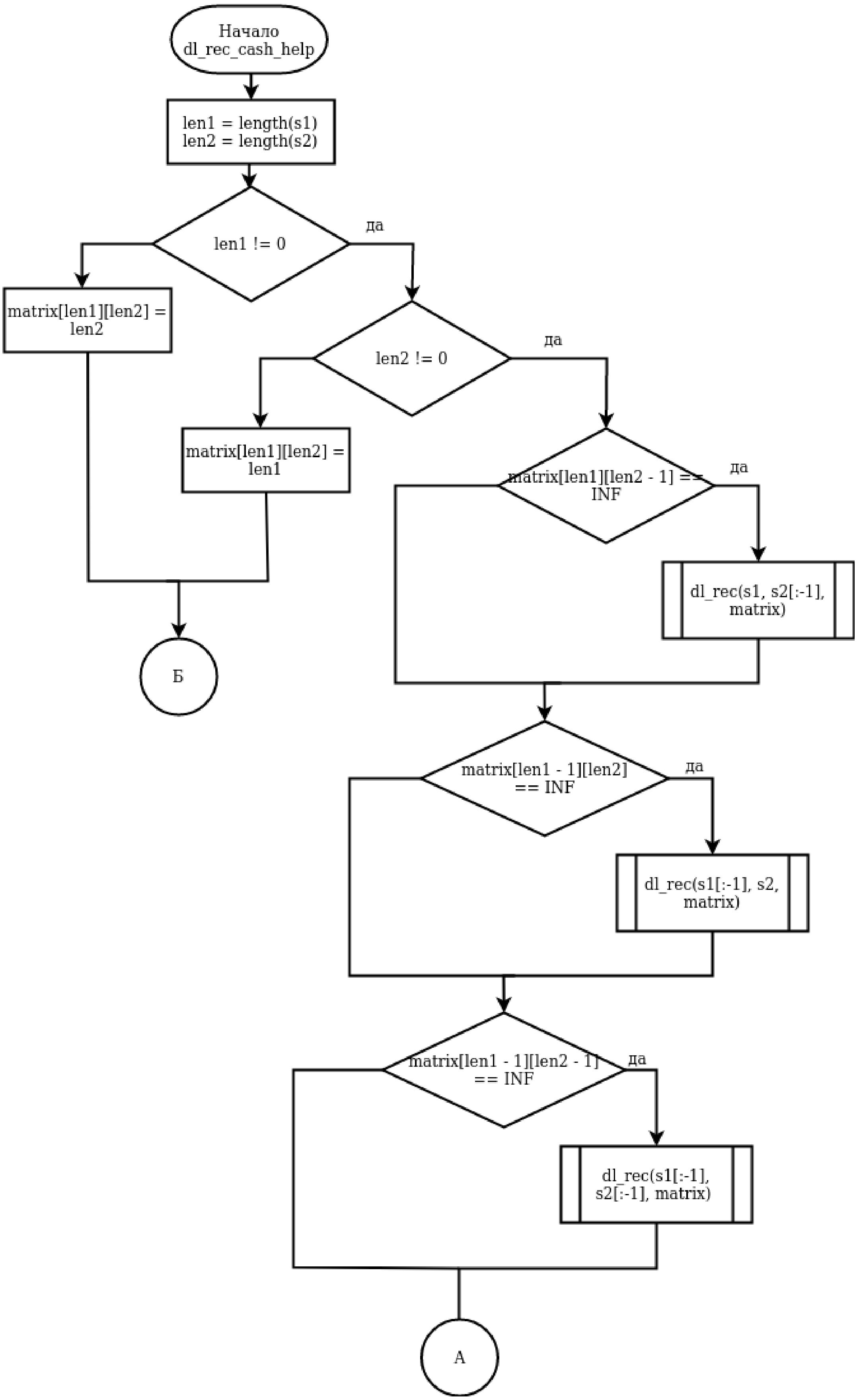


Рисунок 2.5 – Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кешированием

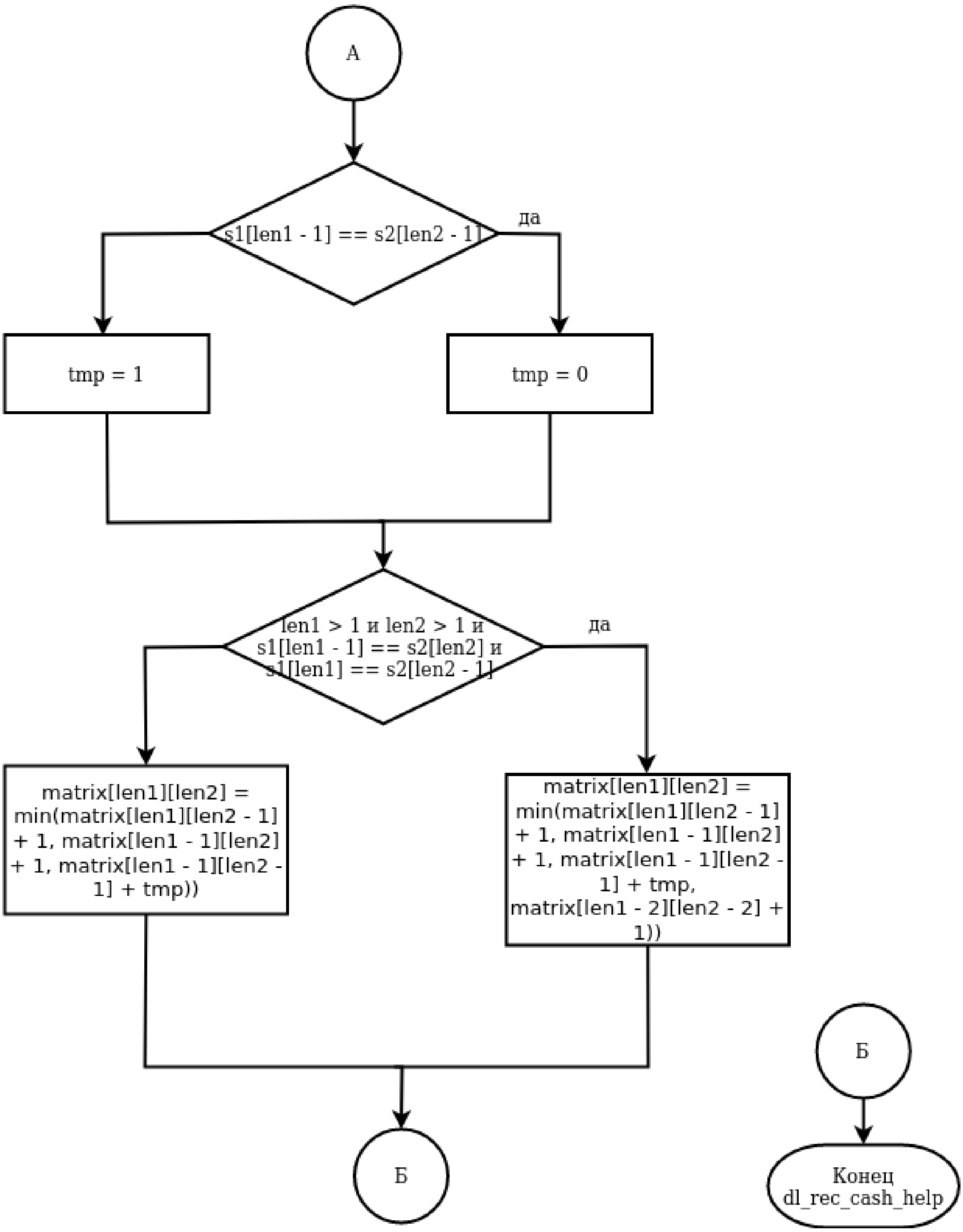


Рисунок 2.6 – Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кешированием

Вывод

В данном разделе были разработаны схемы алгоритмов, которые позволяютс помощью различных подходов находить расстояния Левенштейна и Дамерау-Левенштейна.

3 Технологическая часть

В данном разделе будут представлены требования к программному обеспечению, средства реализации, листинги кода и тесты.

3.1 Требования к программному обеспечению

Вход: две строки (регистрозависимые);

Выход: искомое расстояние, посчитанное с помощью реализованных алгоритмов: для расстояния Левенштейна - итерационный, а для расстояния Дамерау-Левенштейна - итерационный, рекурсивный (с кешированием и без кеширования).

Если на вход программы подаются 2 пустые строки - это корректный ввод, программа должна завершаться без ошибок. В результате работы программа должна вывести число - расстояние Левенштейна или Дамерау-Левенштейна в зависимости от алгоритма, матрицу при необходимости.

3.2 Выбор средств реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык программирования Python [2]. Данный язык ускоряет процесс разработки и удобен в использовании.

Процессорное время реализованных алгоритмов было замерено с помощью функции process_time() из библиотеки time [3].

В качестве среды разработки был выбран PyCharm Professional [4]. Данная среда разработки является кросс-платформенной, предоставляет функциональный отладчик, средства для рефакторинга кода и возможность быстрой установки необходимых библиотек при необходимости.

3.3 Реализация алгоритмов

В листингах 3.1 - 3.4 представлены реализации алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Функция итеративного алгоритма поиска расстояния Левенштейна с заполнением матрицы расстояний

```
def lowenstein_dist_non_recursive(s1, s2, flag=False):
    # len of string + 1 empty symbol
    n = len(s1) + 1
    m = len(s2) + 1

    matrix = [[0 for i in range(m)] for j in range(n)]

    # trivial cases
    for j in range(1, m):
        matrix[0][j] = j  # INSERT
    for i in range(1, n):
        matrix[i][0] = i  # DELETE

    # filling the rest part of the matrix
    for i in range(1, n):
        for j in range(1, m):
            insert = matrix[i][j - 1] + 1
            delete = matrix[i - 1][j] + 1
            tmp = 1  # REPLACE
            if s1[i - 1] == s2[j - 1]:
                tmp = 0  # MATCH
            replace = matrix[i - 1][j - 1] + tmp
            matrix[i][j] = min(insert, delete, replace)

    if flag:
        print('Matrix:')
        for row in matrix:
            print(row)

    return matrix[n - 1][m - 1]  # Result
```

Листинг 3.2 – Функция итеративного алгоритма поиска расстояния Дамерау-Левенштейна с заполнением матрицы расстояний

```
def damerau_lowenstein_dist_non_recursive(s1, s2, flag=False):
    # len of string + 1 empty symbol
    n = len(s1) + 1
    m = len(s2) + 1

    matrix = [[0 for i in range(m)] for j in range(n)]

    # trivial cases
    for j in range(1, m):
        matrix[0][j] = j  # INSERT
    for i in range(1, n):
        matrix[i][0] = i  # DELETE

    # filling the rest part of the matrix
    for i in range(1, n):
        for j in range(1, m):
            insert = matrix[i][j - 1] + 1
            delete = matrix[i - 1][j] + 1
            tmp = 1  # REPLACE
            if s1[i - 1] == s2[j - 1]:
                tmp = 0  # MATCH
            replace = matrix[i - 1][j - 1] + tmp
            matrix[i][j] = min(insert, delete, replace)
            # i -> i-1, i-1 -> i-2 because string don't have zero
                symbol in begin
            if i > 1 and j > 1 and s1[i - 1] == s2[j - 2] and s1[i
                - 2] == s2[j - 1]:
                exchange = matrix[i - 2][j - 2] + 1  # EXCHANGE
                matrix[i][j] = min(matrix[i][j], exchange)

    if flag:
        print('Matrix:')
        for row in matrix:
            print(row)

    return matrix[n - 1][m - 1]  # Result
```

Листинг 3.3 – Функция рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна без кеширования

```
def damerau_lowenstein_dist_recursive(s1, s2):
    # trivial cases (exit)
```

```
    if not s1:
        return len(s2)
    elif not s2:
        return len(s1)

    insert = damerau_lowenstein_dist_recursive(s1, s2[:-1]) + 1
    delete = damerau_lowenstein_dist_recursive(s1[:-1], s2) + 1
    replace = damerau_lowenstein_dist_recursive(s1[:-1], s2[:-1])
        + int(s1[-1] != s2[-1])

    if len(s1) > 1 and len(s2) > 1 and s1[-1] == s2[-2] and s1[-2]
        == s2[-1]:
        exchange = damerau_lowenstein_dist_recursive(s1[:-2], s2
            [:-2]) + 1
        return min(insert, delete, replace, exchange)
    else:
        return min(insert, delete, replace)
```

Листинг 3.4 – Функция рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кешированием

```
INF = -1

def damerau_lowenstein_dist_recursive_cache_helper(s1, s2, matrix)
    :
    # len of string
    len1 = len(s1)
    len2 = len(s2)

    # trivial cases
    if not len1:
        matrix[len1][len2] = len2
    elif not len2:
        matrix[len1][len2] = len1
    else:
        # insert
        if matrix[len1][len2 - 1] == INF:
            damerau_lowenstein_dist_recursive_cache_helper(s1, s2
                [:-1], matrix)
        # delete
        if matrix[len1 - 1][len2] == INF:
            damerau_lowenstein_dist_recursive_cache_helper(s1
```

```
                [:-1], s2, matrix)
        # replace
        if matrix[len1 - 1][len2 - 1] == INF:
            damerau_lowenstein_dist_recursive_cache_helper(s1[:-1], s2[:-1], matrix)
        # exchange
        #if matrix[len1 - 2][len2 - 2] == INF:
           #damerau_lowenstein_dist_recursive_cache_helper(s1[:-2], s2[:-2], matrix)

        matrix[len1][len2] = min(matrix[len1][len2 - 1] + 1,
                                 matrix[len1 - 1][len2] + 1,
                                 matrix[len1 - 1][len2 - 1] + int(s1[-1] != s2[-1]))

        if len1 > 1 and len2 > 1 and s1[-1] == s2[-2] and s1[-2] == s2[-1]:
            matrix[len1][len2] = min(matrix[len1][len2],
                                     matrix[len1 - 2][len2 - 2] + 1)

        return

def damerau_lowenstein_dist_recursive_cache(s1, s2, flag=False):
    # len of string + 1 empty symbol
    n = len(s1) + 1
    m = len(s2) + 1
    matrix = [[-1 for i in range(m)] for j in range(n)]
    damerau_lowenstein_dist_recursive_cache_helper(s1, s2, matrix)

    if flag:
        print('Matrix:')
        for row in matrix:
            print(row)

    return matrix[n - 1][m - 1]
```

3.4 Тестирование

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояний Левенштейна и Дамерау — Левенштейна. В таблице приняты следующие обозначения: Расст. Л - результат алгоритма поиска расстояния Левенштейна, Расст. Д-Л - результат алгоритма поиска рассостяния Дамерау-Левенштейна). Все тесты были пройдены успешно.

Таблица 3.1 – Тесты

			Ожидаемый результат	
№	Строка 1	Строка 2	Расст. Л	Расст. Д-Л
1	"пустая строка"	"пустая строка"	0	0
2	"пустая строка"	abc	3	3
3	abc	"пустая строка"	3	3
4	art	art	0	0
5	af	a	1	1
6	f	af	1	1
7	гора	горы	1	1
8	12345	12435	2	1
9	солнце	солнцестояние	7	7
10	солнцестояние	солнце	7	7
11	кот	скат	2	2
12	нитситту	институт	4	2
13	клсас	класс	2	1
14	аргон	арнон	1	1

Вывод

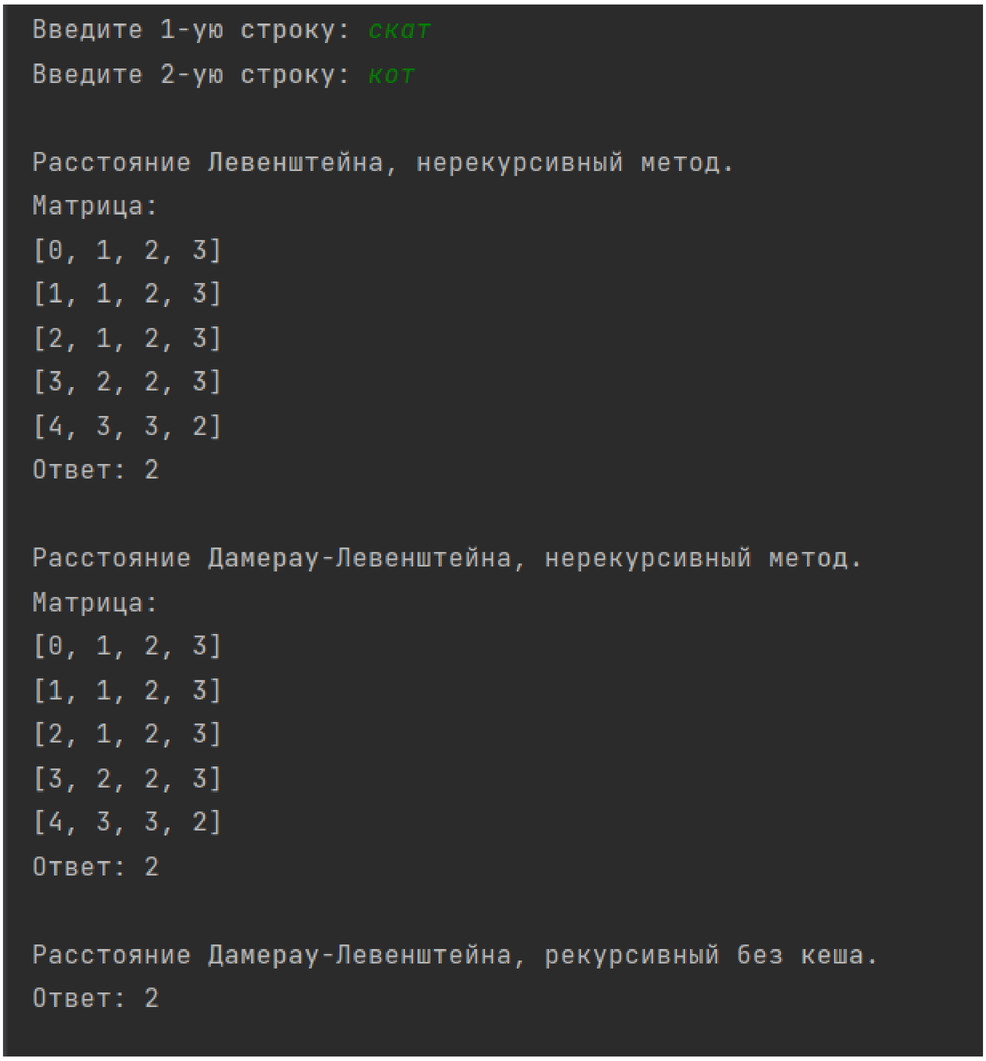
В данном разделе были представлены требования к программному обеспечению и средства реализации, реализованы и протестированы алгоритмы поиска расстояний: Левенштейна - итерационный, Дамерау-Левенштейна - итерационный, рекурсивный (с кешированием и без)

4 Исследовательская часть

В текущем разделе будут представлены примеры работы разработанного программного обеспечения, постановка эксперимента и сравнительный анализ реализованных алгоритмов.

4.1 Пример работы программного обеспечения

На рисунках 4.1, 4.2 представлен результат работы программы.



```
Введите 1-ую строку: скат
Введите 2-ую строку: кот

Расстояние Левенштейна, нерекурсивный метод.
Матрица:
[0, 1, 2, 3]
[1, 1, 2, 3]
[2, 1, 2, 3]
[3, 2, 2, 3]
[4, 3, 3, 2]
Ответ: 2

Расстояние Дамерау-Левенштейна, нерекурсивный метод.
Матрица:
[0, 1, 2, 3]
[1, 1, 2, 3]
[2, 1, 2, 3]
[3, 2, 2, 3]
[4, 3, 3, 2]
Ответ: 2

Расстояние Дамерау-Левенштейна, рекурсивный без кеша.
Ответ: 2
```

Рисунок 4.1 – Пример работы программы

```
Расстояние Дамерау-Левенштейна, рекурсивный с кешем (матрицей).
Матрица:
[0, 1, 2, 3]
[1, 1, 2, 3]
[2, 1, 2, 3]
[3, 2, 2, 3]
[4, 3, 3, 2]
Ответ: 2
```

Рисунок 4.2 – Пример работы программы

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система: Windows 10 [5];
- оперативная память: 16 Гб;
- процессор: Intel® Core™ i5 10300H 2.5 ГГц.

Во время тестирования ноутбук был включен в сеть питания и нагружен только встроенными приложениями окружения и системой тестирования.

4.3 Время выполнения реализаций алгоритмов

Замеры процессорного времени реализованных алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна проводились с помощью функции process_time() из библиотеки time языка Python.

Функция process_time() возвращает время в секундах (сумму системного и пользовательского процессорного времени).

Замеры времени для каждой длины слов (от 0 до 9 символов) проводились 1e5 раз для нерекурсивных алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна, 100 раз для рекурсивных (с кешированием и без) алгоритмов поиска расстояния Дамерау-Левенштейна. В качестве результата бралось среднее время работы алгоритма на каждой длине слова.

На рисунке 4.3 представлено сравнение процессорного времени работы реализаций нерекурсивных алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна. На графике видно, что оба алгоритма практически одинаково эффективны по времени, но чуть менее эффективен алгоритм поиска расстояния Дамерау-Левенштейна (с длины слова равной 4) из-за дополнительной операции обмена двух соседних символов.

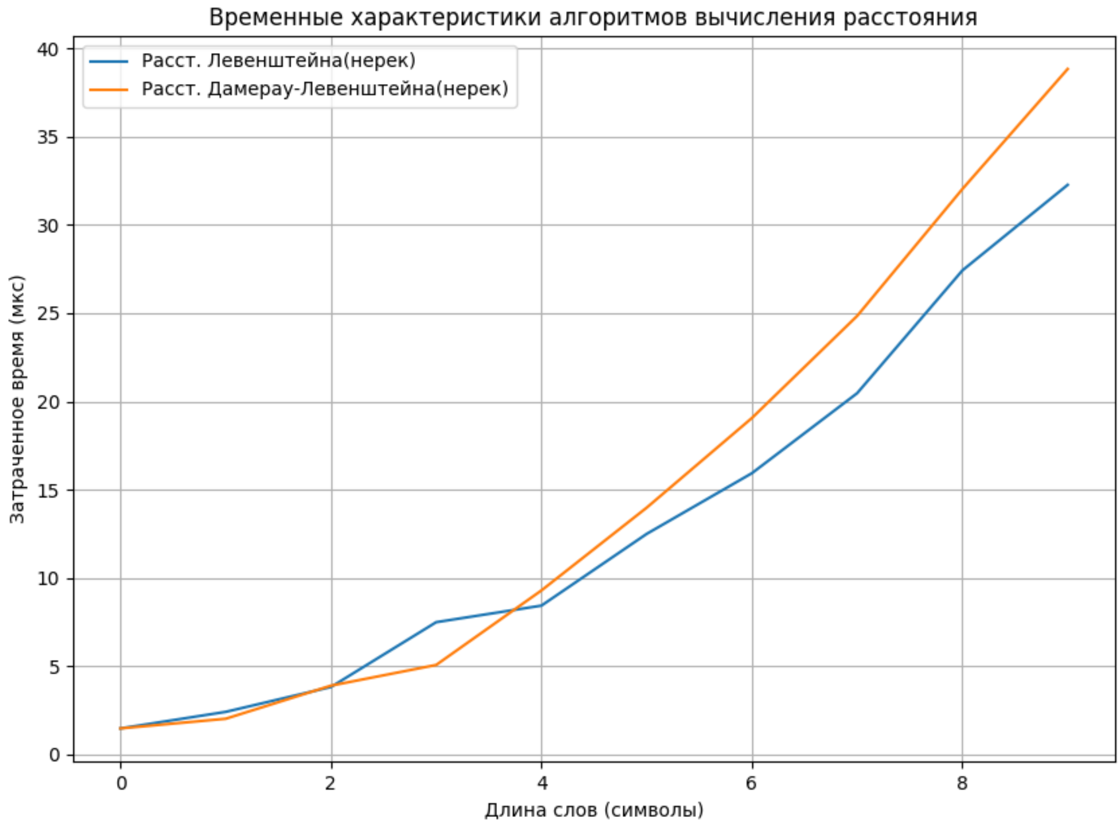


Рисунок 4.3 – Сравнение процессорного времени работы реализаций нерекурсивных алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна

На рисунке 4.4 представлено сравнение процессорного времени работы реализаций рекурсивных алгоритмов (с кешированием и без) поиска расстояния Дамерау-Левенштейна. На графике видно, что полученные результаты почти накладываются друг на друга до длины слова равной 6, но при большей длине слова рекурсивный алгоритм Дамерау-Левенштейна с кешированием значительно эффективнее по времени, так как за счет кеша в виде матрицы не производится повторных вычислений (отсутствие вызова функций для вычисления значений, которые уже были посчитаны ранее).

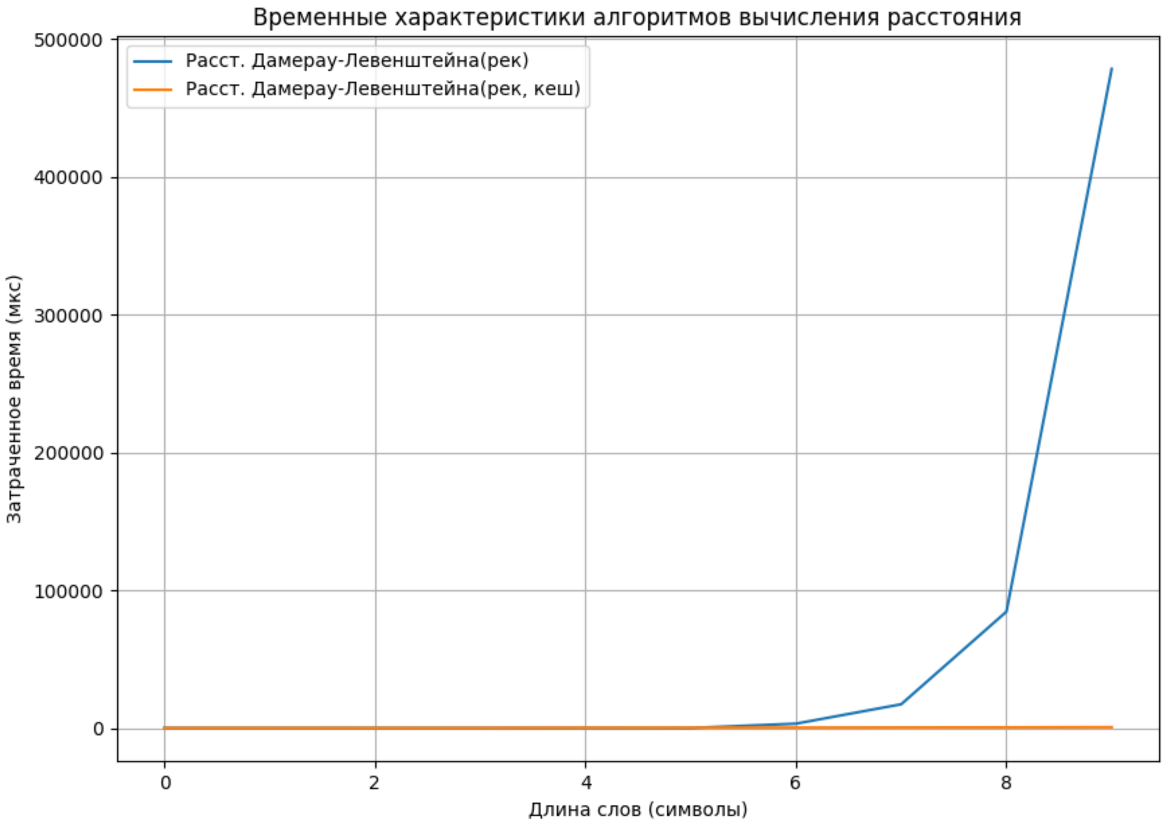


Рисунок 4.4 – Сравнение процессорного времени работы реализаций рекурсивных алгоритмов (с кешированием и без) поиска расстояния Дамерау-Левенштейна

На рисунке 4.5 представлено сравнение всех реализованных в рамках лабораторной работы алгоритмов поиска расстония Левенштейна и Дамерау-Левенштейна. На графике видно, что при длине слова больше 6 символов самым неэффективным по времени алгоритмом является рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна без кеширования из-за большого количества повторных вычислений.

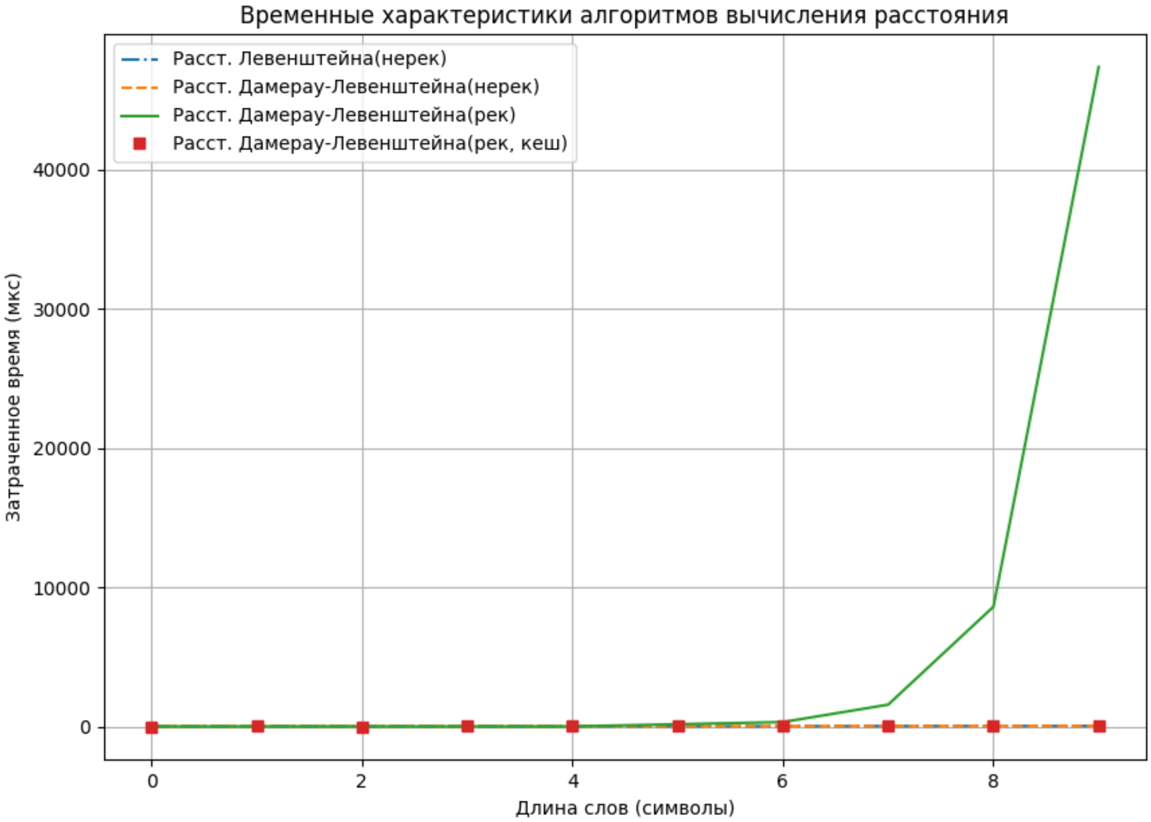


Рисунок 4.5 – Сравнение процессорного времени работы реализаций поиска расстоний Левенштейна и Дамерау-Левенштейна

4.4 Оценка затрат алгоритмов по памяти

Алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна затрачивают схожее количество памяти, но оно будет варьироваться при использовании разных подходов (итеративный, рекурсивный).

Пусть длина строки S_1 - m, длина строки S_2 - n, тогда затраты памяти на приведенные выше алгоритмы будут следующими.

Итерационный алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна:

- матрица - ((m + 1) * (n + 1)) * sizeof(int);

- строки S_1, S_2 - (m + n) * sizeof(char);
- длины строк - 2 * sizeof(int);
- вспомогательные переменные - 3 * sizeof(int); (для алгоритма поиска расстояния Дамерау-Левенштейна - 4 * sizeof(int))

Суммарные затраты памяти итерационного алгоритма поиска расстояния Левенштейна: ((m + 1) * (n + 1) + 5) * sizeof(int) + (m + n) * sizeof(char) байт.

Суммарные затраты памяти итерационного алгоритма поиска расстояния Дамерау-Левенштейна: ((m + 1) * (n + 1) + 6) * sizeof(int) + (m + n) * sizeof(char) байт.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк (m + n).

Затраты по памяти для каждого рекурсивного вызова:

- рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна без кеширования:
  - строки S_1, S_2 - (m + n) * sizeof(char);
  - длины строк S_1, S_2 - 2 * sizeof(int);
  - вспомогательные переменные - 4 * sizeof(int);
  - адрес возврата - address_size;
- рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кешированием:
  - матрица - ((m + 1) * (n + 1)) * sizeof(int); (1 раз)
  - строки S_1, S_2 - (m + n) * sizeof(char);
  - длины строк S_1, S_2 - 2 * sizeof(int);
  - вспомогательная переменная - 1 * sizeof(int);
  - указатель на матрицу - 1 * sizeof(int*);
  - адрес возврата - address_size;

Максимальные суммарные затраты памяти рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна без кеширования: (m + n) * sizeof(char) + (m + n) * (6 * sizeof(int) + address_size) байт.

Максимальные суммарные затраты памяти рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кешированием: (m + 1) * (n + 1) * sizeof(int) + (m + n) * sizeof(char) + (m + n) * (4 * sizeof(int) + address_size) байт.

Вывод

Реализации итерационных алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна почти одинаково эффективны по времени, но при увеличении длины слова незначительно больше времени затрачивает алгоритм поиска расстояния Дамерау-Левенштейна из-за дополнительной операции обмена двух соседних символов, но она довольно часто позволяет найти более короткое расстояние между строками.

Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна без кеша работает намного дольше рекурсивного алгоритма с кешем и итеративного алгоритма поиска расстояния Дамерау-Левенштейна. Рекурсивный алгоритм с кешированием требует меньше времени, чем без кеширования, однако больше, чем итерационный алгоритм поиска расстояния Дамерау-Левенштейна, причем чем больше длина строки, тем больше разрыв по времени.

По количеству затрачиваемой памяти итеративные алгоритмы проигрывают рекурсивным без кеширования (максимальный размер используемой памяти в итеративных алгоритмах пропорционален произведению длин строк, в рекурсивных без кеширования — сумме длин строк), но рекурсивный алгоритм с кешированием является самым затратным по памяти, из-за того, что в отличие от рекурсивного алгоритма без кеширования нужно еще хранить матрицу расстояний.

Заключение

В результате выполнения лабораторной работы цель достигнута, а именно при исследовании алгоритмов поиска расстояний Дамерау-Левенштейна и Левенштейна был изучен и применен метод динамического программирования.

В ходе выполнения данной работы были решены следующие задачи:

- изучены алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна;
- разработаны алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна;
- реализованы алгоритмы поиска расстояния Левенштейна с заполнением матрицы, Дамерау-Левенштейна с использованием рекурсии и с помощью рекурсивного заполнения матрицы;
- выполнена оценка затрат алгоритмов поиска расстояний Левенштейна (итеративный), Дамерау-Левенштейна (итеративный, рекурсивный с кешем, рекурсивный без кеша) по памяти;
- выполнены замеры процессорного времени работы реализаций алгоритмов поиска расстояний Левенштейна (итеративный), Дамерау-Левенштейна (итеративный, рекурсивный с кешем, рекурсивный без кеша);
- проведен сравнительный анализ нерекурсивной и рекурсивной реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна по времени и памяти;

В результате лабораторной работы можно сделать вывод, что итеративная реализация алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна существенно выигрывает по времени с увеличением длины строк, но проигрывает по количеству затрачиваемой памяти рекурсивной реализации алгоритмов с кешированием.

Литература

[1] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. – М.: Доклады АН СССР, 1965. Т. 163. С. 845–848.

[2] Лутц Марк. Изучаем Python, том 1, 5-е изд. Пер. с англ. — СПб.: ООО “Диалектика”, 2019. Т. 832.

[3] time — Time access and conversions [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html (дата обращения: 20.09.2021).

[4] Узнайте все о PyCharm [Электронный ресурс]. Режим доступа: https://www.jetbrains.com/ru-ru/pycharm/learn/ (дата обращения: 20.09.2022).

[5] Windows 10 [Электронный ресурс]. Режим доступа: https://learn.microsoft.com/ru-ru/windows/ (дата обращения: 20.09.2022).