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 Зайцева А.А.

Группа ИУ7-52Б

Преподаватель Волкова Л.Л.

Москва — 2021 г.

Оглавление

Введение

Целью данной лабораторной работы является применение навыков динамического программирования в алгоритмах нахождения расстояний Левенштейна и Дамерау-Левенштейна.

Определение расстояния Леввенштейна (редакционного расстояния) основано на понятии «редакционное предписание».

Редакционное предписание – последовательность действий, необходимых для получения из первой строки второй кратчайшим способом.

Расстояние Левенштейна – минимальное количество действий (вставка, удаление, замена символа), необходимых для преобразования одного слова в другое.

Если текст был набран с клавиатуры, то вместо расстояния Левенштейна чаще используют расстояние Дамерау – Левенштейна, в котором добавляется еще одно возможное действие - перестановка двух соседних символов. [1]

Расстояние Левенштейна и Дамерау – Левенштейна применяется в таких сферах, как:

- компьютерная лингвистика (автозамена в посиковых запросах, текстовая редактура);
- биоинформатика (сравнение генов, хромосом и белков);
- нечеткий поиск записей в базах (борьба с мошенниками и опечатками);

В рамках выполнения работы необходимо решить следующие задачи:

1) изучить расстояния Левенштейна и Дамерау-Левенштейна;
2) разработать алгоритмы поиска этих расстояний;
3) реализовать разработанные алгоритмы;
4) провести сравнительный анализ процессорного времени выполнения реализаций этих алгоритмов;
5) провести сравнительный анализ затраченной реализованными алгоритмами пиковой памяти.

1 Аналитическая часть

Расстояния Левенштейна и Дамерау–Левенштейна – это минимальное количество действий, необходимых для преобразования одной строки в другую. Различие между этими расстояниями - в наборе допустимых операций.

В расстоянии Левенштейна расссматриваются такие действия над символами, как I-insert (вставка), D-delete (удаление) и R-replace (замена). Также вводится операция, которая не требует никаких действий: M-match(совпадение)

В расстоянии Дамерау–Левенштейна в дополнение к перечисленным операциям вводится действие X-xchange (перестановка соседних символов).

Данным операциям можно назначить цену (штраф). Часто ипользуется следующий набор штрафов: для операции M он равен нулю, а для остальных (I, D, R, X) - единице.

Тогда задача нахождения расстояний Левенштейна и Дамерау–Левенштейна сводится к поиску последовательности действий, минимизирующих суммарный штраф. Это можно сделать с помощью рекуррентных формул, которые будут рассмотрены в этом разделе.

1.1 Нахождение расстояния Левенштейна

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

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

Первые три уравнения в системе (1.1) являются тривиальными и подразумевают, соответственно, отсутствие действий (совпадение, так как обе строки пусты), вставки j символов в пустую S_1 для создания строки-копии S_2, длиной j, и удаления всех i символов из строки S_1 для совпадения с пустой строкой S_2.

В дальнейшем необходимо выбирать минимум из штрафов, которые будут порождены операциями вставки символа в S_1 (первое уравнение в группе min), удаления символа из S_1, (второе уравнение в группе min), совпадения или замены, в зависимости от равенства рассматриваемых на данном этапе символов строк (третье уравнение в группе min). [2]

1.2 Нахождение расстояния Дамерау-Левенштей

Расстояние Дамерау-Левенштейна между строками S_1 и S_2 рассчитывается по схожей с (1.1) рекуррентной формуле. Отличие состоит лишь в добавлении четвертого возможного уравнения (1.2) в группу min:

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

Это уравнение подразумевает перестановку соседних символов в S_1, ес-

ли длины обеих строк больше единицы и соседние рассматриваемые символы в S_1 и S_2 крест-накрест равны. Если же хотя бы одно из условий не выполняется, то данное уравнение не учитывается при поиске минимума.

1.3 Вывод

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

2 Конструкторская часть

Рекуррентные формулы, рассмотренные в предыдущем разделе, позволяют находить расстояния Левенштейна и Дамерау-Левенштейна. Однако при разработке алгоритмов, решающих эти задачи, можно использовать различные подходы (циклы, рекурсия с кешированием, рекурсия без кеширования), которые будут рассмотрены в данном разделе.

2.1 Алгоритмы нахождения расстояния Левенштейна

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

На рисунке 2.2 приведена схема рекурсивного алгоритма поиска расстояния Левенштейна без кеширования.

На рисунке 2.3 приведена схема рекурсивного алгоритма поиска расстояния Левенштейна с кешированием.

2.2 Алгоритмы нахождения расстояния Дамерау Левенштейна

На рисунке 2.4 приведена схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна.

Вывод

На основе теоретических данных, полученных из аналитического раздела были построены схемы требуемых алгоритмов.

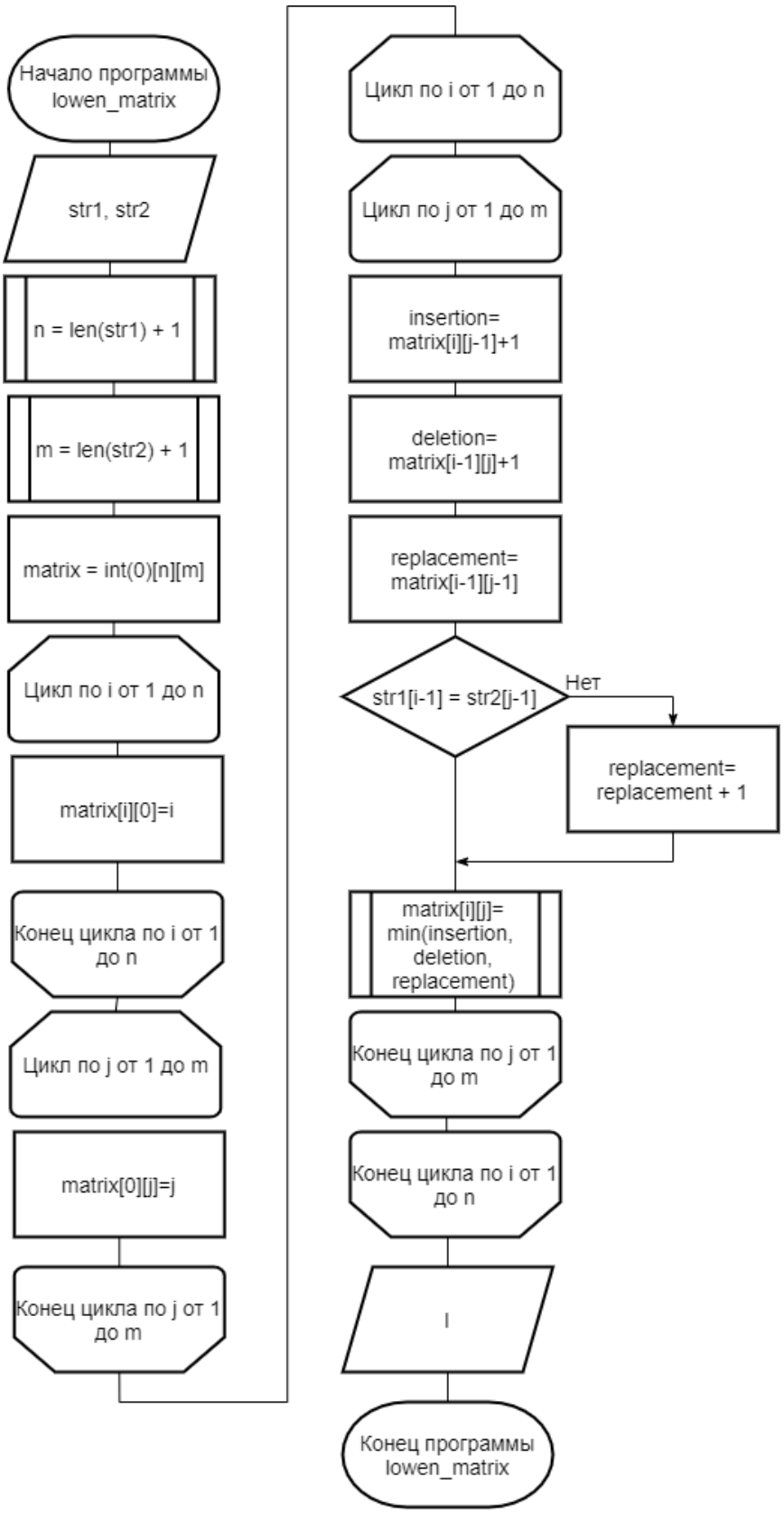

Рисунок 2.1 – Схема итеративного алгоритма поиска расстояния Левенштейна с заполнением матрицы расстояний

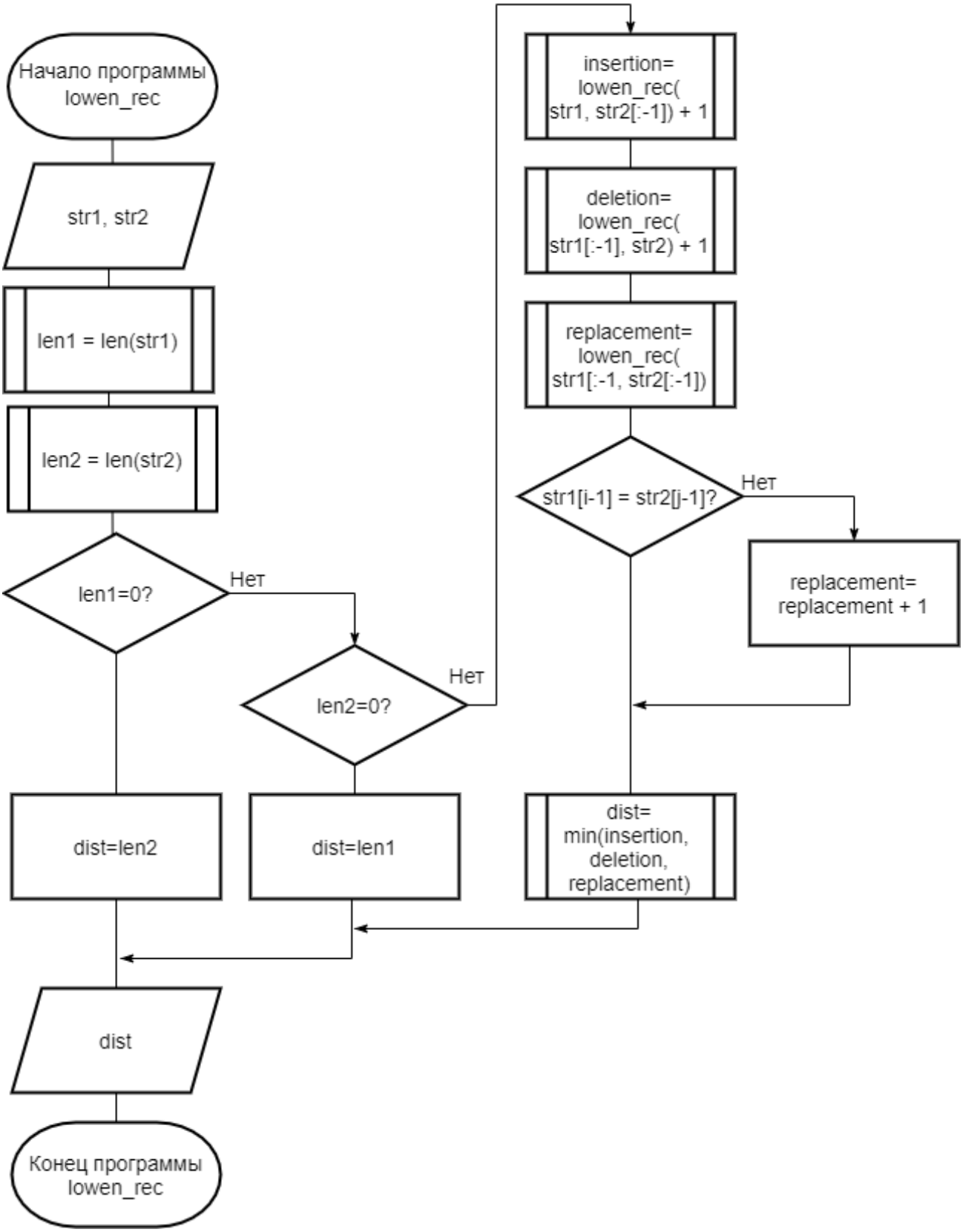

Рисунок 2.2 – Схема рекурсивного алгоритма поиска расстояния Левенштейна без кеширования

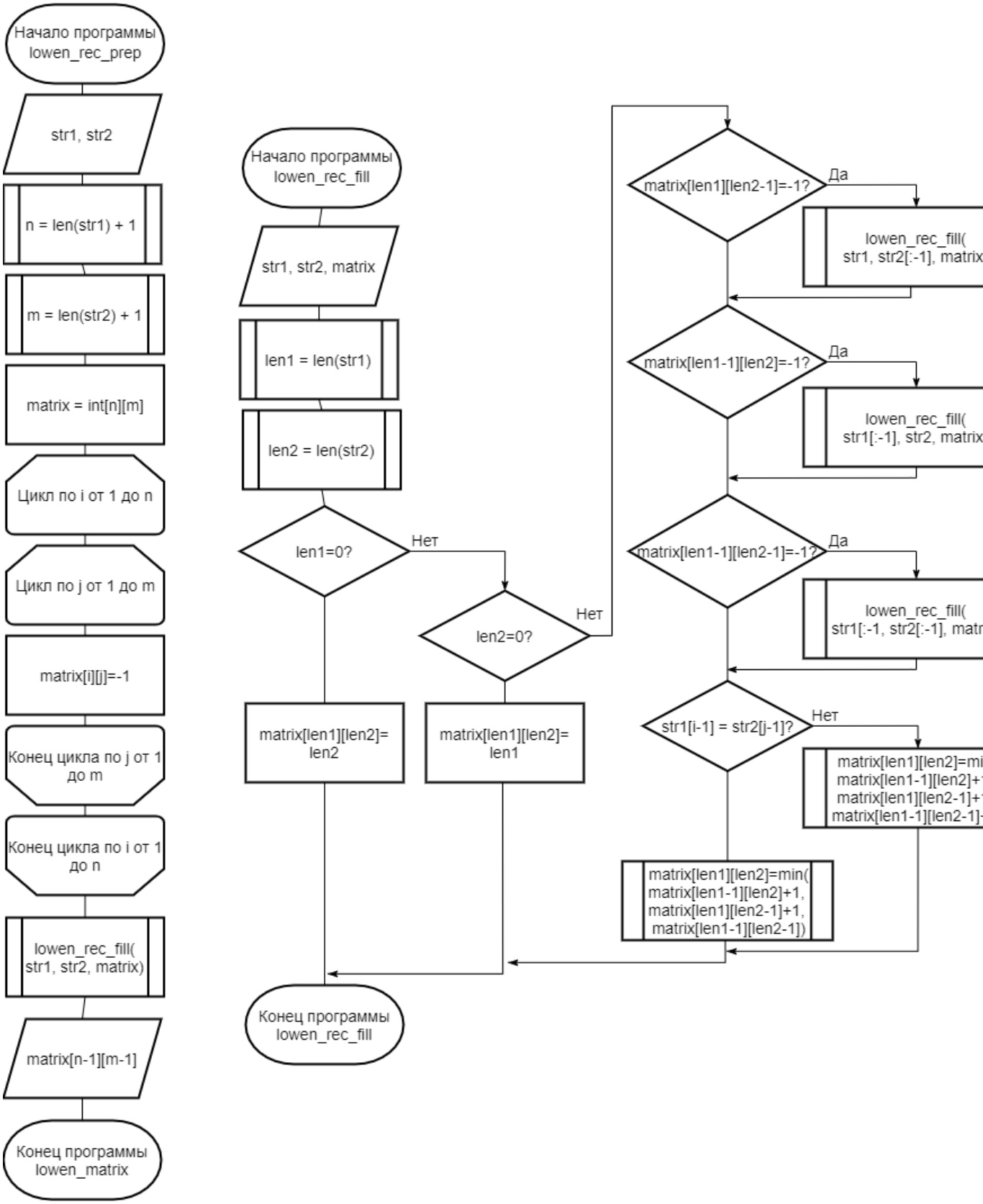

Рисунок 2.3 – Схема рекурсивного алгоритма поиска расстояния Левенштейна с кешированием

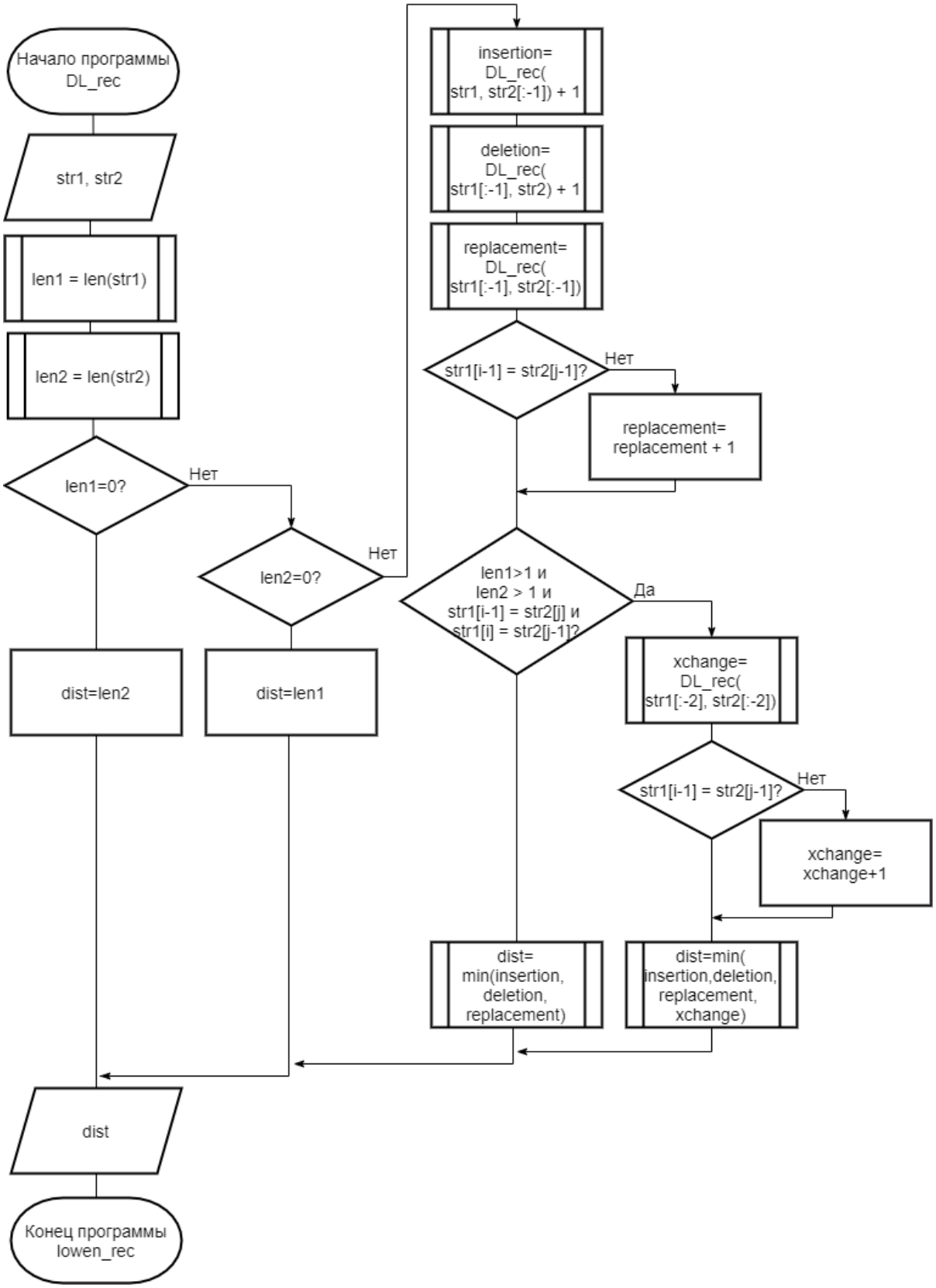

Рисунок 2.4 – Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна

3 Технологическая часть

В данном разделе производится выбор средств реализации, а также приводятся требования к программному обеспечению (ПО), листинги реализованных алгоритмов и тесты для программы.

3.1 Требования к ПО

На вход программе подаются две строки (регистрозависимые), а на выходе должно быть получено искомое расстояние, посчитанное с помощью каждого реализованного алгоритма: для расстояния Левенштейна - итерационный и рекурсивный (с кешем и без), а для расстояния Дамерау-Левенштейна - рекурсивный без кеша. Также необходимо вывести затраченное каждым алгоритмом процессорное время и пиковый объем выделенной памяти.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык Python [3]. Он позволяет быстро реализовывать различные алгоритмы без выделения большого времени на проектирование сруктуры программы и выбор типов данных.

Кроме того, в Python есть библиотека time, которая предоставляет функцию process_time для замра процессорного времени [4], а также библиотека memory_profiler, предоставляющая функцию memory_usage, которая позволяет замерить пиковый объем памяти, выделенный при работе функции [5].

В качестве среды разработки выбран PyCharm. Он является кроссплатформенным, а также предоставляет удобный и функциональнаый отладчик и средства для рефакторинга кода, что позволяет быстро находить и исправлять ошибки [6].

3.3 Листинг кода

В листингах 3.1 - 3.4 представлены реализации рассматриваемых алгоритмов.

Листинг 3.1 – Функция поиска расстояния Левенштейна с заполнением матрицы расстояний

```
def lowenstein_dist_matrix_classic(str1, str2):
    # +1 because of an empty string
    n = len(str1) + 1
    m = len(str2) + 1
      matrix = [[0 for i in range(m)] for j in range(n)]  # MATCH

    # fill with trivial rules
    for i in range(1, n):
      matrix[i][0] = i  # DELETION
    for j in range(1, m):
      matrix[0][j] = j  # INSERTION

    # fill the rest of the matrix
    for i in range(1, n):
      for j in range(1, m):
        insertion = matrix[i][j - 1] + 1
        deletion = matrix[i - 1][j] + 1
        replacement = matrix[i - 1][j - 1] + int(str1[i - 1] !=
            str2[j - 1])

        matrix[i][j] = min(insertion, deletion, replacement)


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

Листинг 3.2 – Функция рекурсивного алгоритма поиска расстояния Левенштейна без кеширования

```
def lowenstein_dist_recursion_classic(str1, str2):
    # trivial rules
    if not str1:
      return len(str2)
    elif not str2:
      return len(str1)
```

```
7
8       insertion = lowenstein_dist_recursion_classic(str1, str2[:-1])
            + 1
9       deletion = lowenstein_dist_recursion_classic(str1[:-1], str2)
            + 1
10      replacement = lowenstein_dist_recursion_classic(str1[:-1],
            str2[:-1]) + int(str1[-1] != str2[-1])
11
12      return min(insertion, deletion, replacement)
```

Листинг 3.3 – Функция рекурсивного алгоритма поиска расстояния Левенштейна с кешированием

```
1   def lowenstein_dist_recursion_optimized(str1, str2):
2       def _lowenstein_dist_recursion_optimized(str1, str2, matrix):
3           len1 = len(str1)
4           len2 = len(str2)
5
6           # trivial rules
7           if not len1:
8               matrix[len1][len2] = len2
9           elif not len2:
10              matrix[len1][len2] = len1
11          else:
12              # insertion
13              if matrix[len1][len2 - 1] == -1:
14                  _lowenstein_dist_recursion_optimized(str1, str2[:-1],
                        matrix)
15              # deletion
16              if matrix[len1 - 1][len2] == -1:
17                  _lowenstein_dist_recursion_optimized(str1[:-1], str2,
                        matrix)
18              # replacement
19              if matrix[len1 - 1][len2 - 1] == -1:
20                  _lowenstein_dist_recursion_optimized(str1[:-1], str2
                        [:-1], matrix)
21
22              matrix[len1][len2] = min(matrix[len1][len2 - 1] + 1,
23              matrix[len1 - 1][len2] + 1,
24              matrix[len1 - 1][len2 - 1] + int(str1[-1] != str2[-1]))
25
26      # +1 because of an empty string
```

```
    n = len(str1) + 1
    m = len(str2) + 1
    matrix = [[-1 for i in range(m)] for j in range(n)]
    _lowenstein_dist_recursion_optimized(str1, str2, matrix)

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

Листинг 3.4 – Функция рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна

```
def damerau_lowenstein_dist_recursion(str1, str2):
    # trivial rules
    if not str1:
        return len(str2)
    elif not str2:
        return len(str1)

    insertion = lowenstein_dist_recursion_classic(str1, str2[:-1])
        + 1
    deletion = lowenstein_dist_recursion_classic(str1[:-1], str2)
        + 1
    replacement = lowenstein_dist_recursion_classic(str1[:-1],
        str2[:-1]) + int(str1[-1] != str2[-1])

    if len(str1) > 1 and len(str2) > 1 and str1[-1] == str2[-2]
        and str1[-2] == str2[-1]:
        xchange = lowenstein_dist_recursion_classic(str1[:-2], str2
            [:-2]) + int(str1[-1] != str2[-1])
        return min(insertion, deletion, replacement, xchange)
    else:
        return min(insertion, deletion, replacement)
```

3.4 Тестирование

В таблице **??** приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна и Дамерау — Левенштейна. Тесты пройдены успешно.

Таблица 3.1 – Тесты

		Ожидаемый результат	
Строка 1	Строка 2	Алг. Левенштейна	Алг. Дамерау-Левенштейна
		0	0
abc	abc	0	0
ab	a	1	1
a	ab	1	1
see	sea	1	1
1234	1324	2	1
hello	ehlla	3	2
cat	pop	3	3
кот	скат	2	2

Вывод

Был производен выбор средств реализации, реализованы и протестированы алгоритмы поиска расстояний: Левенштейна - итерационный и рекурсивный (с кешем и без), Дамерау-Левенштейна - рекурсивный без кеша

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Manjaro [7] Linux [8] x86_64.
- Память: 8 GiB.
- Процессор: Intel® Core™ i7-8550U[9].

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, окружением, а также непосредственно системой тестирования.

4.2 Время выполнения алгоритмов

Алгоритмы тестировались при помощи написания «бенчмарков» [?], предоставляемых встроенными в Rust средствами. Такие бенчмарки делают за нас некоторое кол-во замеров (достаточное, чтобы считать результат стабильным), предоставляя затем результат с некоторой погрешностью. Также мною были написаны тесты, прогоняющие алгоритмы Z раз, где Z можно выбирать.

В листинге 4.1 пример реализации бенчмарка.

Листинг 4.1 – Пример бенчмарка

```
#[cfg(test)]
mod benchs {
    use super::*;

    #[bench]
    fn iterative10(b: &mut Bencher) {
        let s1 = generate_string_of_size(10);
        let s2 = generate_string_of_size(10);
        b.iter(|| algorithms::iterative(&s1, &s2));
```

```
10      }
11  }
```

Результаты замеров приведены в таблице 4.1. В данной таблице для значений, для которых тестирование не выполнялось, в поле результата находится NaN. На рисунках 4.1 и 4.2 приведены графики зависимостей времени работы алгоритмов от длины строк.

Таблица 4.1 – Замер времени для строк, размером от 10 до 200

Длина строк	**Время, нс**			
	Recursive	**RecMem**	**Iterative**	**IterativeDL**
10	32766430	1313	634	681
20	NaN	5157	2367	2582
30	NaN	11342	4813	5207
50	NaN	30066	12518	13533
100	NaN	116134	48111	52078
200	NaN	529335	249057	527797

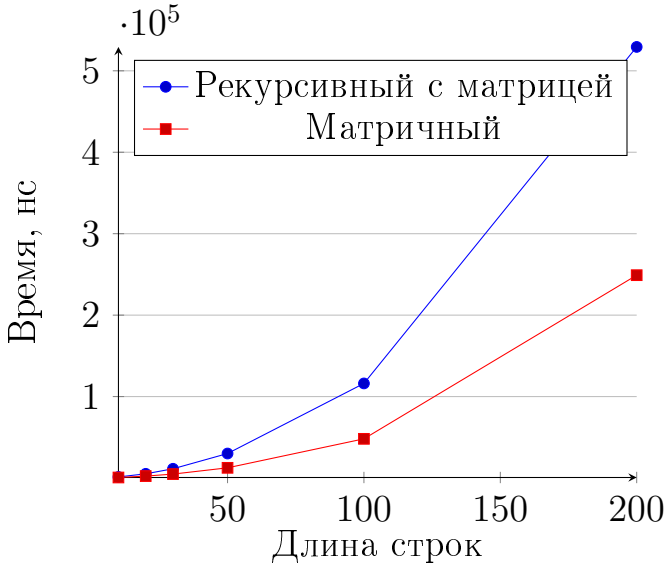

Рисунок 4.1 – Зависимость времени работы алгоритма вычисления расстояния Левенштейна от длины строк (рекурсивная с заполнением матрицы и матричная реализации)

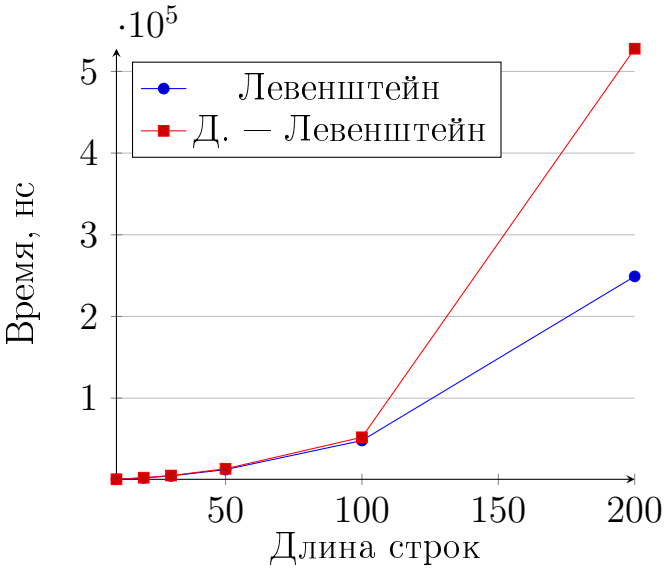

Рисунок 4.2 – Зависимость времени работы матричных реализаций алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна

4.3 Использование памяти

Алгоритмы нахождения расстояний Левенштейна и Дамерау — Левенштейна не отличаются друг от друга с точки зрения использования памяти, следовательно, достаточно рассмотреть лишь разницу рекурсивной и матричной реализаций этих алгоритмов.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, при этом для каждого вызова рекурсии в моей реализации требуется:

- 4 локальные переменные беззнакового типа, в моем случае: $4 \cdot 8 = 32$ байта;
- 2 аргумента типа строка: $2 \cdot 16 = 32$ байта;
- адрес возврата: 8 байт;
- место для записи возвращаемого функцией значения: 8 байт.

Таким образом получается, что при обычной рекурсии на один вызов требуется (4.1):

$$M_{percall} = 32 + 32 + 8 + 8 = 80 \tag{4.1}$$

Следовательно память, расходуемая в момент, когда стек вызовов максимален, равна (4.2):

$$M_{recursive} = 80 \cdot depth \tag{4.2}$$

где $depth$ - максимальная глубина стека вызовов, которая равна (4.3):

$$depth = |S_1| + |S_2| \tag{4.3}$$

где S_1, S_2 - строки.

Если мы используем рекурсивный алгоритм с заполнением матрицы матрицы, то для каждого вызова рекурсии добавляется новый аргумент - ссылка на матрицу - размером 8 байт. Также в данном алгоритме требуется память на саму матрицу, размеры которой: $m = |S_1|+1, n = |S_2|+1$. Размер элемента матрицы равен размеру беззнакового целого числа, используемого в моей реализации, то есть 8 байт. Отсюда выходит, что память, которая тратится на хранение матрицы (4.4):

$$M_{Matrix} = (|S_1| + 1) \cdot (|S_2| + 1) \cdot 8 \tag{4.4}$$

Таким образом, при рекурсивной реализации требуемая память равна (4.5):

$$M_{recursive} = 88 \cdot depth + M_{Matrix} \tag{4.5}$$

где M_{Matrix} взято из соотношения 4.4.

Память, требуемая для при итеративной реализации, состоит из следующего:

- 4 локальные переменные беззнакового типа, в моем случае: $4 \cdot 8 = 32$ байта;
- 2 аргумента типа строка: $2 \cdot 16 = 32$ байта;
- адрес возврата: 8 байт;
- место для записи возвращаемого функцией значения: 8 байт;
- матрица: M_{Matrix} из соотношения 4.4.

Таким образом общая расходуемая память итеративных алгоритмов (4.6):

$$M_{iter} = M_{Matrix} + 80 \tag{4.6}$$

где M_{Matrix} определяется из соотношения 4.4.

Вывод

Рекурсивный алгоритм нахождения расстояния Левенштейна работает на порядок дольше итеративных реализаций, время его работы увеличивается в геометрической прогрессии. На словах длиной 10 символов, матричная реализация алгоритма нахождения расстояния Левенштейна превосходит по времени работы рекурсивную на несколько порядков. Рекурсивный алгоритм с заполнением матрицы превосходит простой рекурсивный и сравним по времени работы с матричными алгоритмами. Алгоритм нахождения расстояния Дамерау — Левенштейна по времени выполнения сопоставим с алгоритмом нахождения расстояния Левенштейна. В нём добавлена дополнительная проверка, позволяющая находить ошибки пользователя, связанные с неверным порядком букв, в связи с чем он работает незначительно дольше, чем алгоритм нахождения расстояния Левенштейна.

Но по расходу памяти итеративные алгоритмы проигрывают рекурсивному: максимальный размер используемой памяти в них растёт как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк.

Заключение

В ходе выполнения лабораторной работы была проделана следующая работа:

- были теоретически изучены алгоритмы нахождения расстояний Левенштейна и Дамерау–Левенштейна;
- для некоторых реализаций были применены методы динамического программирования, что позволило сделать алгоритмы быстрее;
- были практически реализованы алгоритмы в 2 вариантах: рекурсивном и итеративном;
- на основе полученных в ходе экспериментов данных были сделаны выводы по поводу эффективности всех реализованных алгоритмов;
- был подготовлен отчет по ЛР.

Литература

[1] В. М. Черненький Ю. Е. Гапанюк. МЕТОДИКА ИДЕНТИФИКАЦИИ ПАССАЖИРА ПО УСТАНОВОЧНЫМ ДАННЫМ // Вестник МГТУ им. Н.Э. Баумана. Сер. “Приборостроение”. 2012. Т. 39. С. 31–35.

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

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

[4] time — Time access and conversions [Электронный ресурс]. Режим доступа: `https://docs.python.org/3/library/time.html` (дата обращения: 05.09.2021).

[5] memory-profiler [Электронный ресурс]. Режим доступа: `https://pypi.org/project/memory-profiler/` (дата обращения: 05.09.2021).

[6] Python и Pycharm [Электронный ресурс]. Режим доступа: `https://py-charm.blogspot.com/2017/09/pycharm.html` (дата обращения: 05.09.2021).

[7] Manjaro – enjoy the simplicity [Электронный ресурс]. Режим доступа: `https://manjaro.org/` (дата обращения: 14.09.2020).

[8] Linux – Википедия [Электронный ресурс]. Режим доступа: `https://ru.wikipedia.org/wiki/Linux` (дата обращения: 14.09.2020).

[9] Процессор Intel® Core™ i7-8550U [Электронный ресурс]. Режим доступа: `https://ark.intel.com/content/www/ru/ru/ark/products/122589/intel-core-i7-8550u-processor-8m-cache-up-to-4-00-ghz.html` (дата обращения: 14.09.2020).