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 дисциплине "Анализ алгоритмов"

Тема Расстояние Левенштейна и Дамерау-Левенштейна

Студент Андрич К.

Группа ИУ7И-56Б

Оценка (баллы) ________________

Преподаватели Волкова Л.Л.

Москва — 2021 г.

Оглавление

Введение **2**

1 Аналитическая часть **3**
1.1 Расстояние Левенштейна . . . 3
1.2 Расстояние Дамерау-Левенштейна . . . 3
1.3 Вывод . . . 4

2 Конструкторская часть **5**
2.1 Схемы алгоритмов . . . 5
2.2 Вывод . . . 8

3 Технологическая часть **9**
3.1 Выбор ЯП . . . 9
3.2 Требование к ПО . . . 9
3.3 Реализация алгоритмов . . . 9
3.4 Тестовые данные . . . 11
3.5 Вывод . . . 11

4 Исследовательская часть **12**
4.1 Пример работы . . . 12
4.2 Время выполнения алгоритмов . . . 13
4.3 Использование памяти . . . 14
4.4 Вывод . . . 14

Заключение **15**

Список литературы **15**

Введение

Расстояние Левенштейна - минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистики для:

- исправления ошибок в слове
- сравнения текстовых файлов утилитой diff
- в биоинформатике для сравнения генов, хромосом и белков

Целью работы является изучение метода динамического программирования на материале алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна и оценка их реализаций.

Задачи лабораторной работы:

1. Изучение алгоритмов Левенштейна и Дамерау-Левенштейна;
2. Получение практических навыков реализации указанных алгоритмов: матричные и рекурсивные версии;
3. Сравнительный анализ реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам;
4. Экспериментальное подтверждение различий во временно́й эффективности реализаций выбранного алгоритма;
5. Описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

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

При преобразовании одного слова в другое можно использовать следующие операции:

- D (от англ. delete) - удаление.
- I (от англ. insert) - вставка.
- R (от англ. replace) - замена.

Будем считать стоимость каждой вышеизложенной операции - 1. Введем понятие совпадения - M (от англ. match). Его стоимость будет равна 0.

1.1 Расстояние Левенштейна

Имеем две строки S1 и S2, длинной M и N соответственно. Расстояние Левенштейна рассчитывается по рекуррентной формуле:

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

1.2 Расстояние Дамерау-Левенштейна

Как было написано выше, в расстоянии Дамерау-Левенштейна задействует еще одну редакторскую операцию - транспозицию T (*от англ. transposition*). Расстояние Дамерау-Левенштейна рассчитывается по рекуррентной формуле:

$$
D(S_1[1...i], S_2[1...j]) = \begin{cases} \text{j, если i} == 0 \\ \text{i, если j} == 0 \\ min(\\ D(S_1[1...i], S_2[1...j-1]) + 1, \\ D(S_1[1...i-1], S_2[1...j]) + 1, & j > 0, i > 0 \\ D(S_1[1...i-1], S_2[1...j-1]) + \\ \left[\begin{array}{l} \text{0, если } S_1\text{[i]} == S_2\text{[j]} \\ \qquad \text{1, иначе,} \end{array} \right. \\ \left[\begin{array}{l} D(S_1[1...i-2], S_2[1...j-2]) + 1, \text{ если i, j>1, } a_i = b_{j-1}, b_j = a_{i-1} \\ \qquad\qquad\qquad\qquad \infty \text{ , иначе)} \end{array} \right. \end{cases} \tag{1.2}
$$

1.3 Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который является модификаций первого, учитывающего возможность перестановки соседних символов. Алгоритмы могут быть реализованы рекурсивно или матрично.

2 | Конструкторская часть

2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы алгоритмов нахождения расстояние Левештейна и Дамерау - Левенштейна. На рисунках 2.1 - 2.4 представлены рассматриваемые алгоритмы.

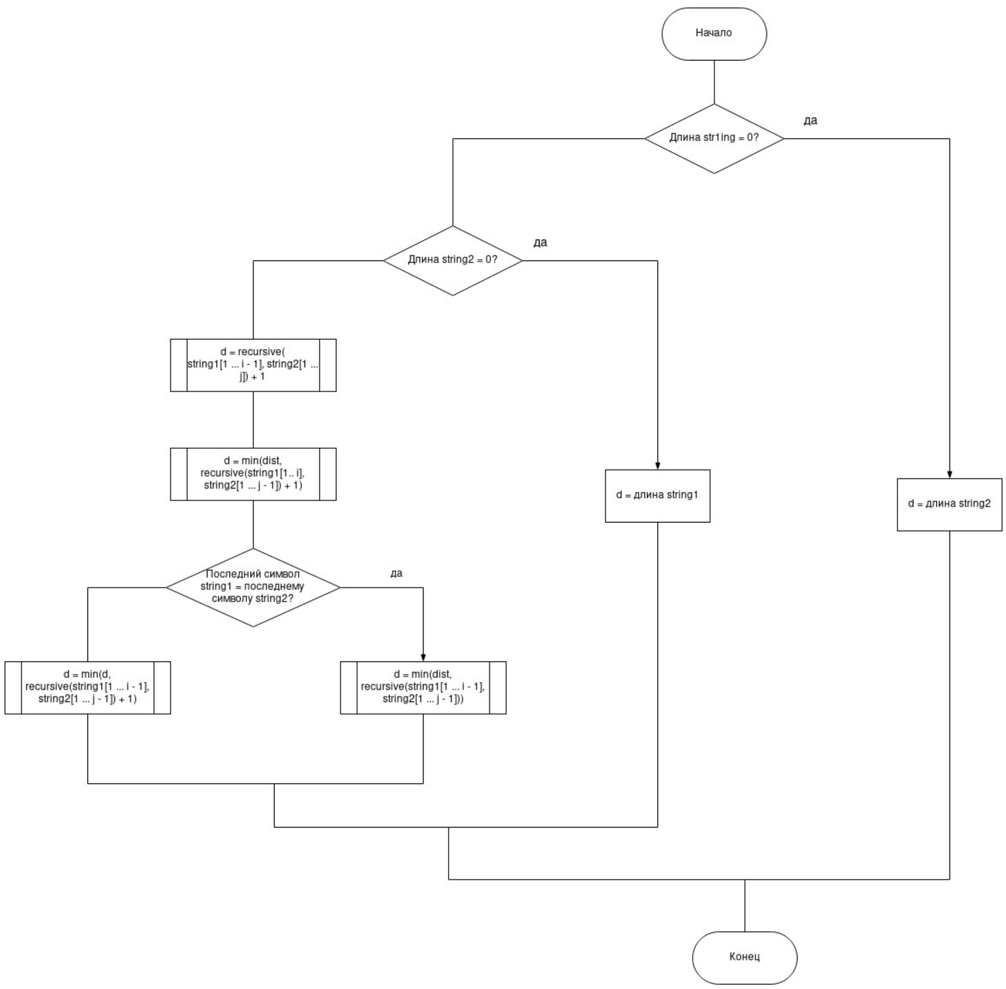

Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

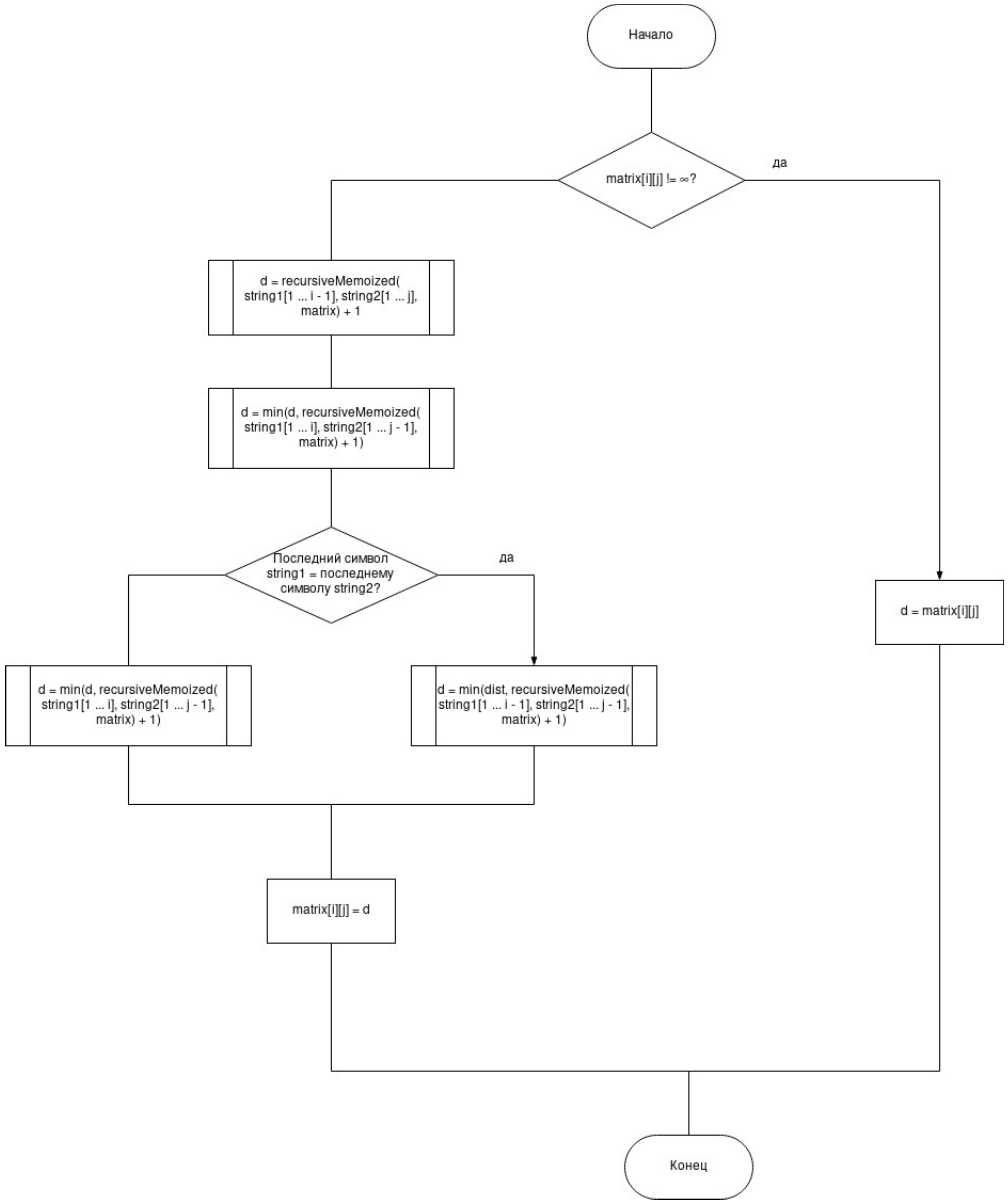

Рис. 2.2: Схема рекурсивного алгоритма Левенштейна с кэшем

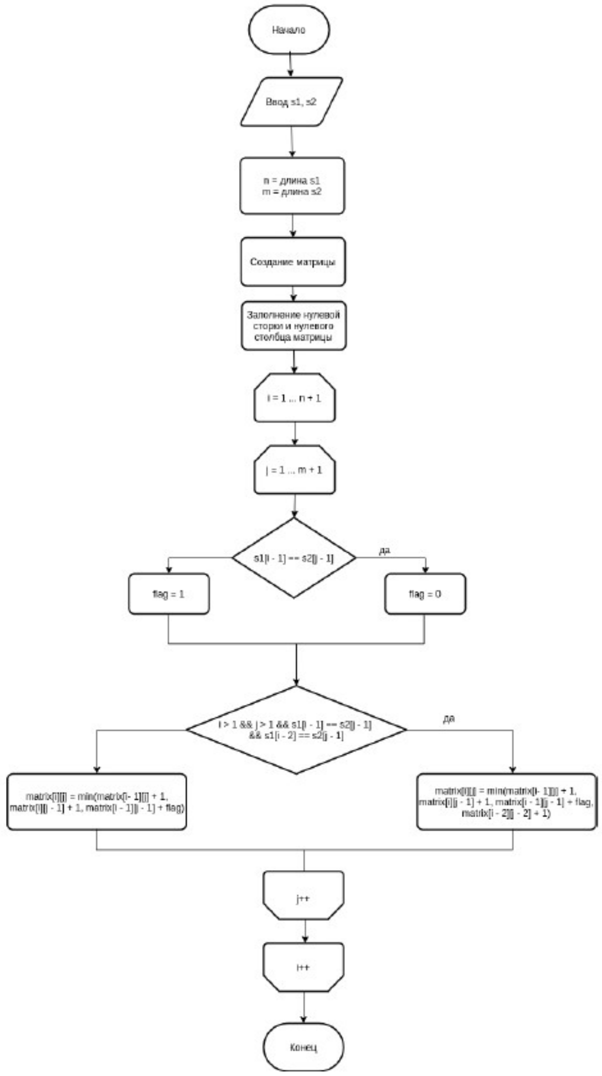

Рис. 2.3: Схема алгоритма поиска расстояния Дамерау–Левенштейна с помощью матрицы(без рекурсии)

2.2 Вывод

На основе теоретических данных, полученные в аналитическом разделе были построены схемы иследуеммых алгоритмов.

3 Технологическая часть

3.1 Выбор ЯП

В данной лабораторной работе использовался язык программирования - python. Данный выбор обусловлен тем, что этот язык наиболее удобен для работы со строками, а также тем, что в нём присутсвует функция для измерения процессорного времени. В качестве среды разработки я использовала Visual Studio Code, так как считаю его достаточно удобным.

3.2 Требование к ПО

Требования к вводу:

1. На вход подаются две строки в любой раскладке (в том числе и пустые);
2. ПО должно выводить полученное расстояние;
3. ПО должно выводить потраченное время;

3.3 Реализация алгоритмов

В листингах 3.1 - 3.4 приведена реализация алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Листинг 3.1: Функция нахождения расстояния Левенштейна рекурсивно

```
def lev_recursion(str1, str2, len1, len2):
  if (len1 == len2) and len1 == 0:
    return 0
  elif len1 == 0:
    return len2
  elif len2 == 0:
    return len1
  else:
  flag = bool(not (str1[len1 - 1] == str2[len2 - 1]))
  return min(min(lev_recursion(str1, str2, len1 - 1, len2) + 1,
          lev_recursion(str1, str2, len1, len2 - 1) + 1),
          lev_recursion(str1, str2, len1 - 1, len2 - 1) + flag)
```

Листинг 3.2: Функция нахождения расстояние Левенштейна рекурсивно с помощью кэша

```
def lev_cache(str1, str2, len1, len2, mtx):
  if not mtx[len1][len2] == 0:
    return mtx[len1][len2]
  elif (len1 == len2) and (len1 == 0):
    mtx[len1][len2] = 0
  elif len1 == 0:
    mtx[len1][len2] = len2
  elif len2 == 0:
    mtx[len1][len2] = len1
  else:
    flag = bool(not(str1[len1 - 1] == str2[len2 - 1]))
    mtx[len1][len2] = min(min(lev_cache(str1, str2, len1 - 1, len2, mtx) +
        1,
                   lev_cache(str1, str2, len1, len2 - 1, mtx) + 1),
                   lev_cache(str1, str2,  len1 - 1, len2 - 1, mtx) + flag)
  return mtx[len1][len2]

def rec_lev_cache(str1, str2, len1, len2):
  mtx = [[0 for x in range(len2 + 1)] for y in range(len1 + 1)]
  return lev_cache(str1, str2, len1, len2, mtx)
```

Листинг 3.3: Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
  def lev_damau_recursion(str1, str2, len1, len2):
    if (len1 == len2) and len1 == 0:
      return 0
    elif len1 == 0:
      return len2
    elif len2 == 0:
      return len1
    else:
      flag = bool(not(str1[len1 - 1] == str2[len2 - 1]))
      res = min(lev_damau_recursion(str1, str2, len1 - 1, len2) + 1,
          lev_damau_recursion(str1, str2, len1, len2 - 1) + 1,
          lev_damau_recursion(str1, str2, len1 - 1, len2 - 1) + flag)
      if (len1 >= 2 and len2 >= 2 and str1[len1 - 1] == str2[len2 - 2] and
         str1[len1 - 2] == str2[len2 - 1]):
        res = min(res, lev_damau_recursion(str1, str2, len1 - 2, len2 - 2) +
            1)
      return res
```

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
def lev_damau_matrix(str1, str2, len1, len2):
  mtx = [[0 for x in range(len2 + 1)] for y in range(len1 + 1)]
  for i in range(len2 + 1):
    mtx[0][i] = i
  for i in range(1, len1 + 1):
    mtx[i][0] = i
  for i in range(1, len1 + 1):
    for j in range(1, len2 + 1):
      add, delete, change = mtx[i - 1][j] + 1, mtx[i][j - 1] + 1, mtx[i -
          1][j - 1]
      if str2[j - 1] != str1[i - 1]:
        change += 1
      mtx[i][j] = min(add, delete, change)
      if ((i > 1 and j > 1) and str1[i - 1] == str2[j - 2] and str1[i - 2]
          == str2[j - 1]):
        mtx[i][j] = min(mtx[i][j], mtx[i - 2][j - 2] + 1)
  return mtx[len1][len2]
```

3.4 Тестовые данные

В таблице 3.1 приведены тестовые данные, на которых было протестированно разработанное ПО. Как видно из этой таблицы, все тесты были успешно пройдены, что означает, что программа работает правильно.

Таблица 3.1: Таблица тестовых данных

№	Первая строка	Вторая строка	Ожидаемый результат	Полученный результат
1	color	colour	1 1 1 1	1 1 1 1
2	padding	touchpad	8 8 8 8	8 8 8 8
3	kaska	taksa	3 3 2 2	3 3 2 2
4	lover	moped	3 3 3 3	3 3 3 3
5	lolly	lolyl	2 2 1 1	2 2 1 1
6	qwerty	queue	4 4 4 4	4 4 4 4
7	мама	папа	2 2 2 2	2 2 2 2
8		pat	3 3 3 3	3 3 3 3
9	baby		4 4 4 4	4 4 4 4
10			0 0 0 0	0 0 0 0

3.5 Вывод

В данном разделе были разработаны исходные коды четырех алгоритмов: вычисления расстояния Левенштейна рекурсивно и рекурсивно с использованием кэша, а также вычисления расстояния Дамерау — Левенштейна рекурсивно и с помощью матрицы.

4 Исследовательская часть

4.1 Пример работы

Демонстрация работы программы приведена на рисунке 4.1.

```
Menu:
1 - enter by hand
2 - compare time
0 - exit

Choice: 1

Input first string: color
Input second string: colour

Levenstein using recursion: 1
Levenstein using recursion witch cache: 1
Levenstein-Damau using recursion: 1
Levenstein-Damau method without using recursion (using matrix): 1

Menu:
1 - enter by hand
2 - compare time
0 - exit

Choice: 1

Input first string: мама
Input second string: папа

Levenstein using recursion: 2
Levenstein using recursion witch cache: 2
Levenstein-Damau using recursion: 2
Levenstein-Damau method without using recursion (using matrix): 2

Menu:
1 - enter by hand
2 - compare time
0 - exit

Choice: 0
```

Рис. 4.1: Работа алгоритмов нахождения расстояния Левенштейна и Дамерау – Левенштейна.

4.2 Время выполнения алгоритмов

Время выполнения алгоритмов замерялось с помощью функции process_time модуля time в Python [3].

В таблице 4.1. представлены замеры времени работы для каждого из алгоритмов.

Таблица 4.1: Таблица времени выполнения алгоритмов

Длина строк	RecLev	RecLevCache	RecDam	MtxDam
3	0.031250	0.031250	0.015625	0.015625
4	0.109375	0.031250	0.093750	0.015625
5	0.484375	0.031250	0.531250	0.015625
6	2.546875	0.046875	2.578125	0.031250

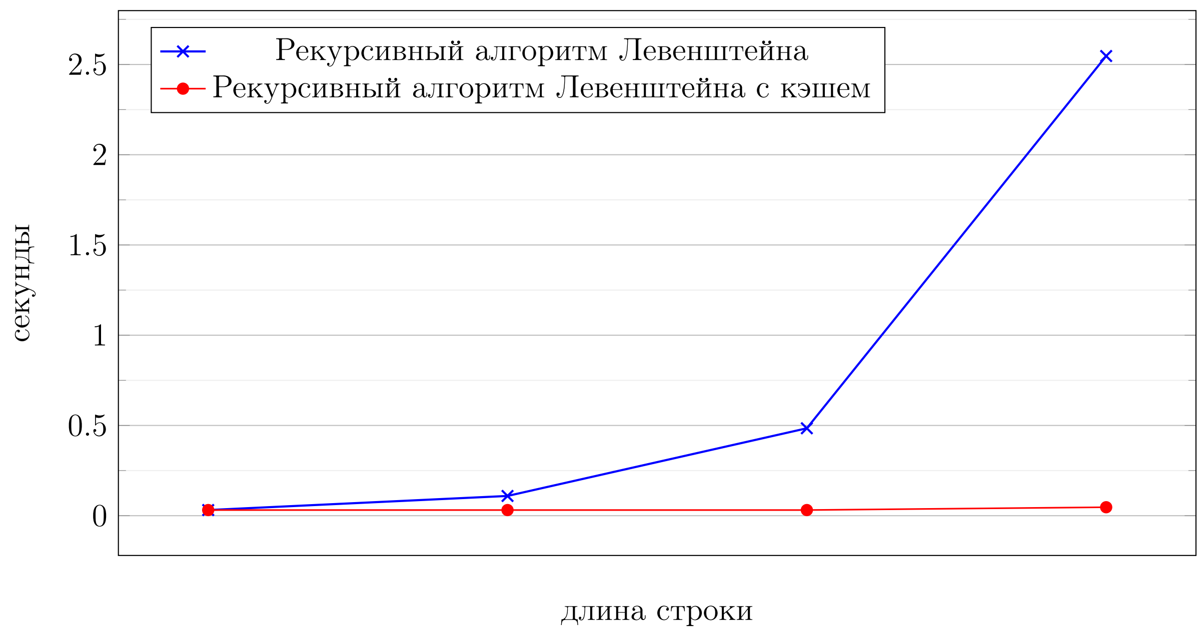

Рис. 4.2: Сравнение рекурсивного алгоритма Левенштейна и рекурсивного с кэшем

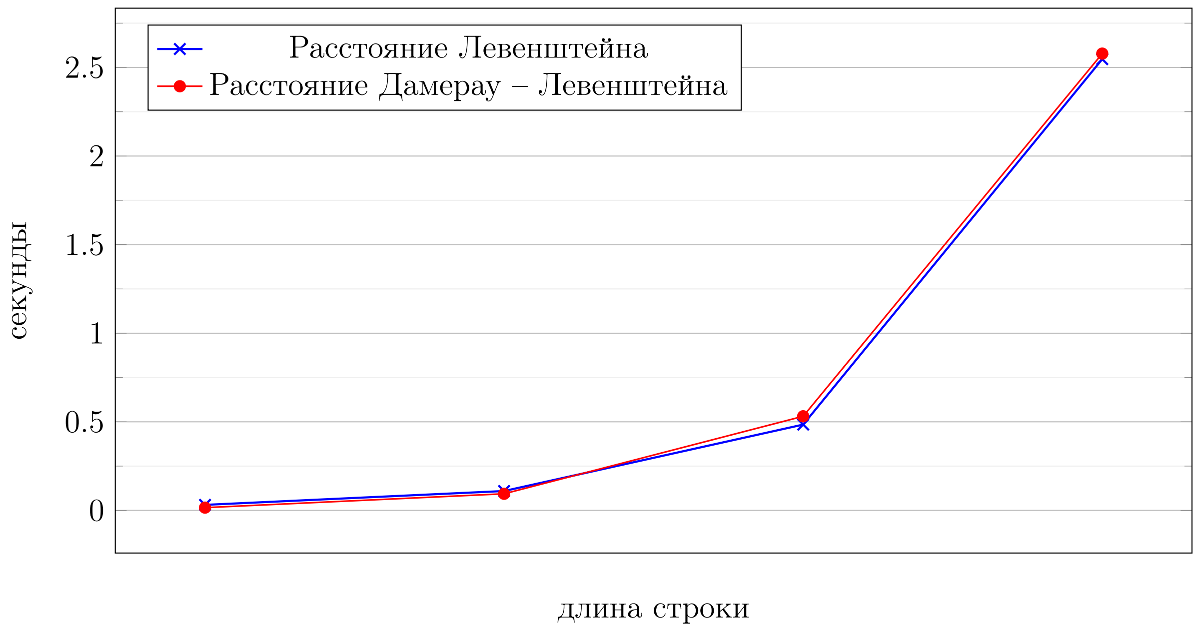

Рис. 4.3: Сравнение рекурсивных алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна

4.3 Использование памяти

Алгоритмы нахождения расстояния Левенштейна и Дамерау — Левенштейна не отличаются друг от друга с точки зрения использования памяти.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк. Поэтому, максимальный расход памяти равен:

$$(\mathcal{S}(STR_1) + \mathcal{S}(STR_2)) \cdot (2 \cdot \mathcal{S}(\text{string}) + 2 \cdot \mathcal{S}(\text{integer})), \tag{4.1}$$

где $\mathcal{S}$ — оператор вычисления размера, STR_1, STR_2 — строки, string — строковый тип, integer — целочисленный тип.

4.4 Вывод

Обычная рекурсивная реализация алгоритма нахождения расстояния Левенштейна работает дольше реализации с кэшем, время работы этой реализации увеличивается в геометрической прогрессии. Рекурсивный метод при этом использует больше памяти.

Заключение

В ходе проделанной работы был изучен метод динамического программирования на материале реализации алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна. Также были изучены алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками и получены практические навыки раелизации указанных алгоритмов в матричной и рекурсивных версиях, а так же в версиях с мемоизацией.

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработаного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк.

Теоретически было рассчитано использование памяти в каждой из реализаций алгоритмов нахождения расстояния Левенштейна и Дамерау - Левенштейна.

Литература

[1] В. И. Левенштейн, Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965. Т. 163 С. 845-848.

[2] Intel Atom x7-E3950: технические характеристики и тесты [Электронный ресурс]. Режим доступа: https://technical.city/ru/cpu/Atom-x7-E3950. Дата обращения: 7.10.2021.

[3] Функция process_time() модуля time в Python [Электронный ресурс]. Режим доступа: https://docs-python.ru/standart-library/modul-time-python/funktsija-process-time-modulja-time/. Дата обращения: 5.10.2021.