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

Преподаватели Волкова Л.Л.

Москва — 2021 г.

Оглавление

Введение

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

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

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

Расстояние Левенштейна – минимальное количество действий (вставка, удаление, замена символа), необходимых для преобразования одной строки в другую.

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]

Расстояния Левенштейна и Дамерау – Левенштейна применяются 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 затрачиваемой 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 вставка (I-insert), удаление (D-delete) и замена (R-replace). Также вводится операция, которая не требует никаких действий - совпадение (M-match).

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

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 [2]. Это можно сделать с помощью рекуррентных формул, которые будут рассмотрены в этом разделе.

1.1 Расстояние 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 формулы в системе (1.1) являются тривиальными и подразумевают, соответственно: отсутствие действий (совпадение, так как обе строки пусты), вставку j символов в пустую 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 дальнейшем необходимо выбирать минимум из штрафов, которые будут порождены операциями вставки символа в S_1 (первая формула в группе min), удаления символа из S_1, (вторая форумла в группе min), а также совпадения или замены, в зависимости от равенства рассматриваемых на данном этапе символов строк (третья формула в группе min) [2].

1.2 Расстояние Дамерау-Левенштейна

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 варианта (1.2) в группу min:

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

Этот вариант подразумевает перестановку соседних символов в S_1, если

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 данная оперция не учитывается при поиске минимума.

Итоговая же формула для поиска расстояния Дамерау-Левенштейна имеет следующий вид (1.3):

$$D(S_1[1...i], S_2[1...j]) = \begin{cases} 0, \text{ если i} == 0, \text{ j} == 0 \\ \text{j, если i} == 0, \text{ j} > 0 \\ \text{i, если j} == 0, \text{ i} > 0 \\ min(\\ D(S_1[1...i], S_2[1...j-1]) + 1, \\ D(S_1[1...i-1], S_2[1...j]) + 1, & \text{j}>0, \text{ i}>0 \\ D(S_1[1...i-1], S_2[1...j-1]) + \\ \begin{bmatrix} 0, \text{ если } S_1[\text{i}] == S_2[\text{j}] \\ 1, \text{ иначе} \end{bmatrix} \\ , \\ \begin{bmatrix} D(S_1[1...i-2], S_2[1...j-2]) + 1, \\ \text{если i, j}>1,\ a_i = b_{j-1}, b_j = a_{i-1}; \\ \infty \text{ , иначе} \end{bmatrix} \\ ) \end{cases} \tag{1.3}$$

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 поиска 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 Алгоритм поиска расстояния Дамерау - Левенштейна

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

Вывод

Были разработаны схемы алгоритмов, позволяющих с помощью различных подходов находить расстояния Левенштейна и Дамерау-Левенштейна.

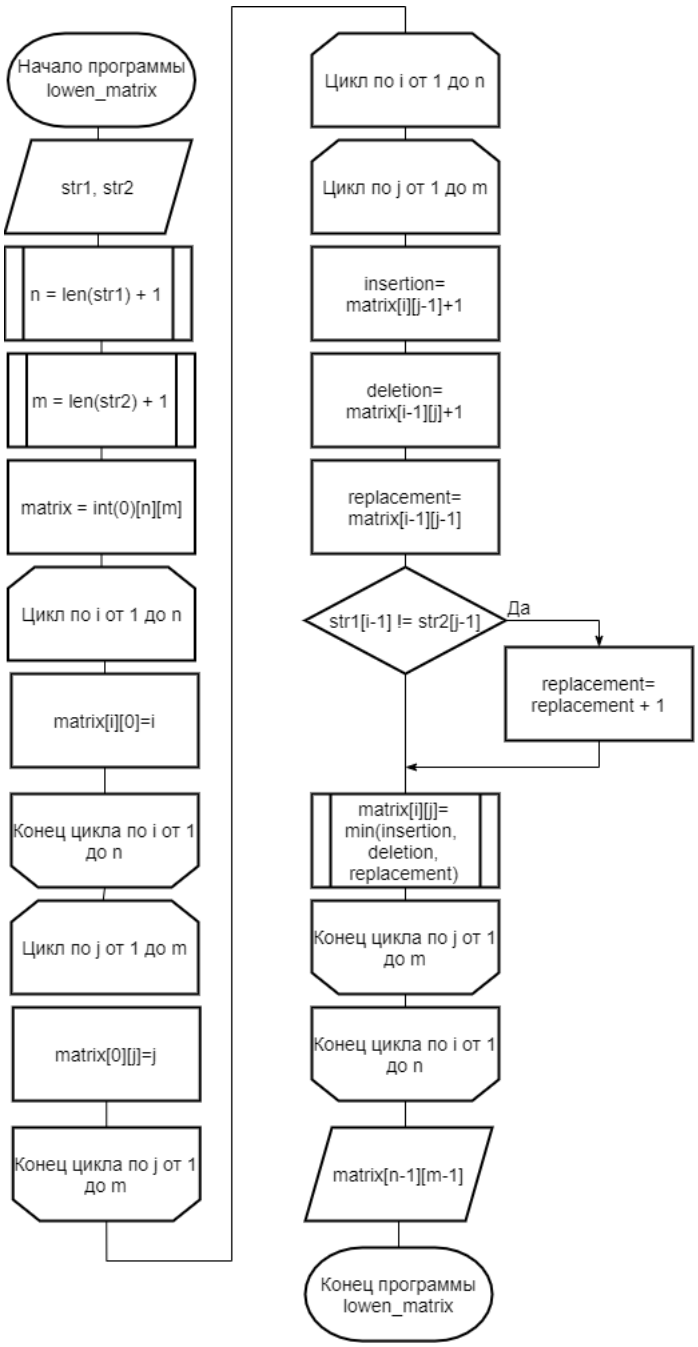

Рисунок 2.1 – Схема итеративного алгоритма поиска расстояния Левенштейна с заполнением матрицы расстояний

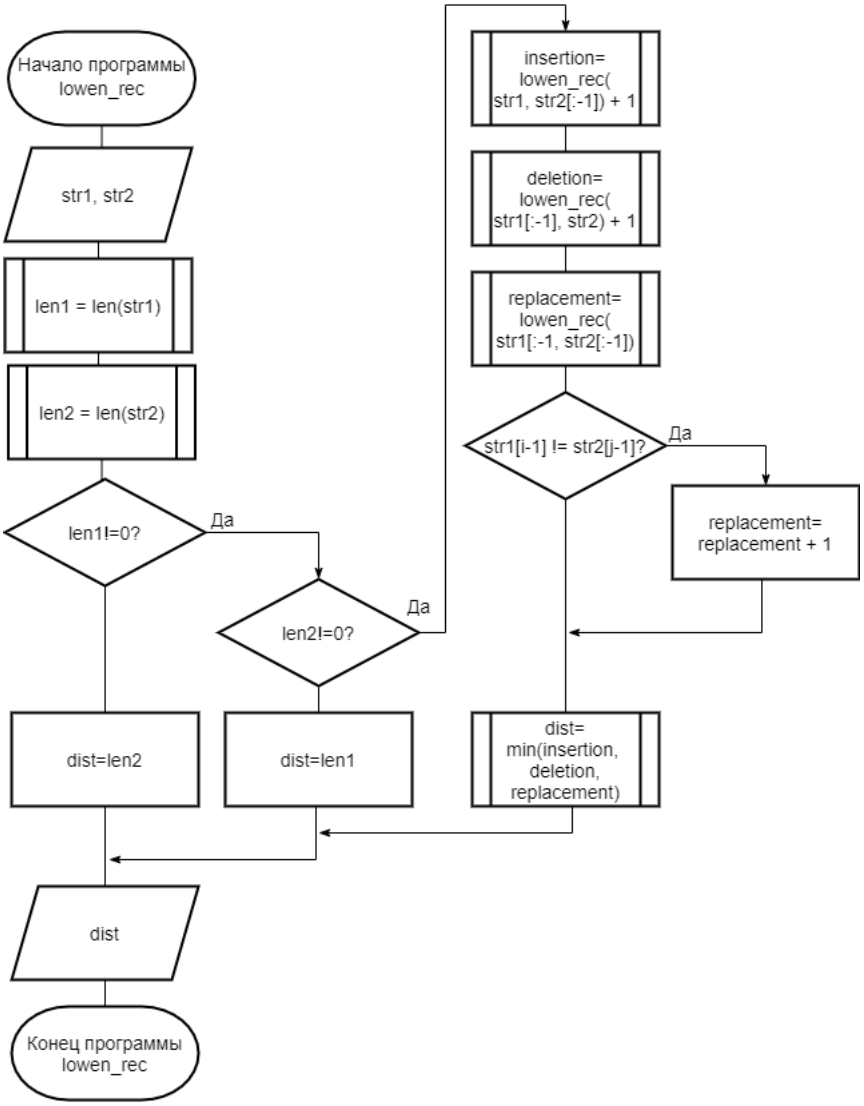

Рисунок 2.2 – Схема рекурсивного алгоритма поиска расстояния Левенштейна без кеширования

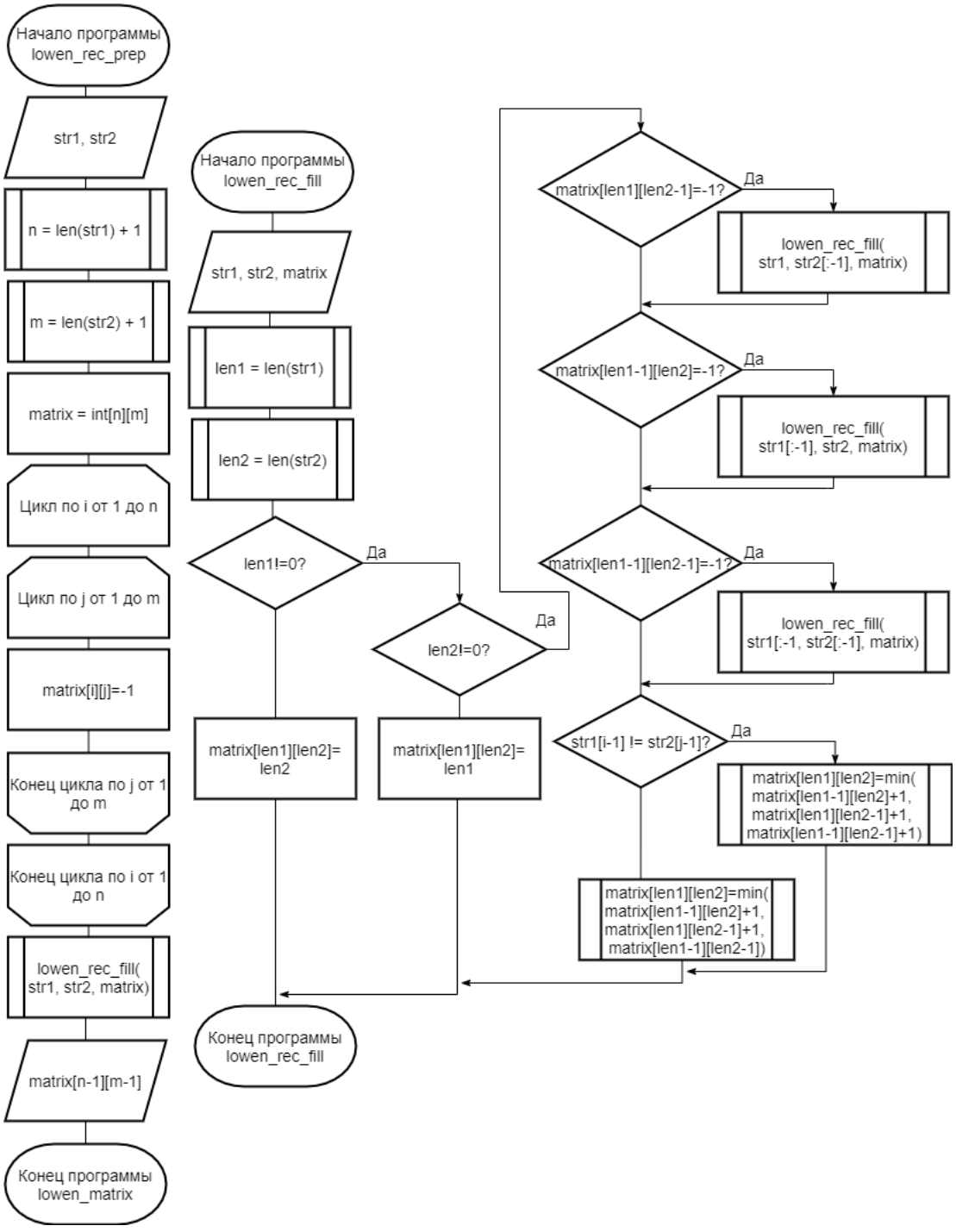

Рисунок 2.3 – Схема рекурсивного алгоритма поиска расстояния Левенштейна с кешированием

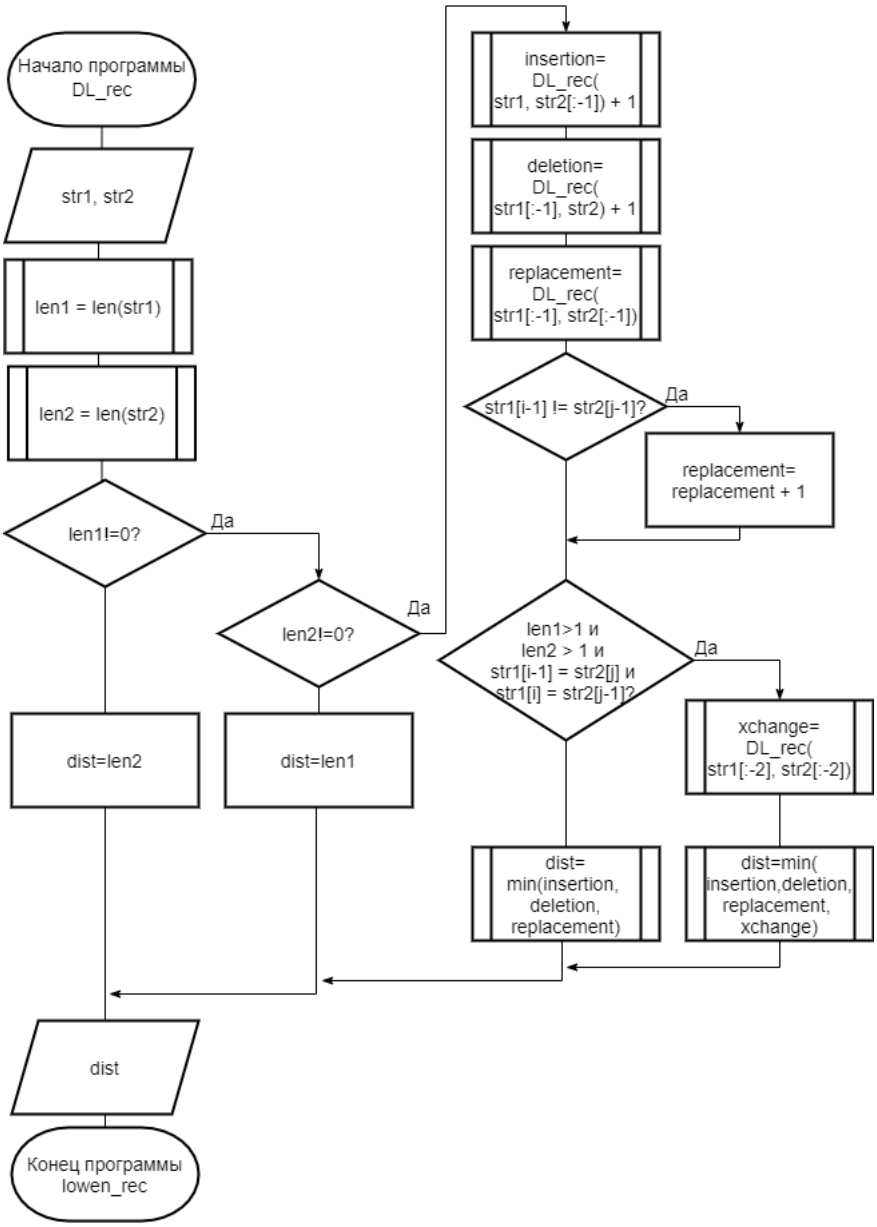

Рисунок 2.4 – Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна без кеширования

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.

3.2 Выбор средств 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 замера процессорного времени [4].

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 [5].

3.3 Листинги 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)

  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])

    return min(insertion, deletion, replacement)
```

Листинг 3.3 – Функция рекурсивного алгоритма поиска расстояния Левенштейна с кешированием

```
def lowenstein_dist_recursion_optimized(str1, str2):
    def _lowenstein_dist_recursion_optimized(str1, str2, matrix):
        len1 = len(str1)
        len2 = len(str2)

        # trivial rules
        if not len1:
            matrix[len1][len2] = len2
        elif not len2:
            matrix[len1][len2] = len1
        else:
            # insertion
            if matrix[len1][len2 - 1] == -1:
                _lowenstein_dist_recursion_optimized(str1, str2[:-1],
                    matrix)
            # deletion
            if matrix[len1 - 1][len2] == -1:
                _lowenstein_dist_recursion_optimized(str1[:-1], str2,
                    matrix)
            # replacement
            if matrix[len1 - 1][len2 - 1] == -1:
                _lowenstein_dist_recursion_optimized(str1[:-1], str2
                    [:-1], matrix)

            matrix[len1][len2] = min(matrix[len1][len2 - 1] + 1,
            matrix[len1 - 1][len2] + 1,
            matrix[len1 - 1][len2 - 1] + int(str1[-1] != str2[-1]))

    # +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]) + 1
    return min(insertion, deletion, replacement, xchange)
  else:
    return min(insertion, deletion, replacement)
```

3.4 Тестирование

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояний Левенштейна и Дамерау — Левенштейна (в таблице приняты обозначения: РЛ - алгоритм поиска расстояния Левенштейна, РДЛ - алгоритм поиска рассотяния Дамерау-Левенштейна). Все тесты пройдены успешно.

Вывод

Был производен выбор средств реализации, реализованы и протестированы алгоритмы поиска расстояний: Левенштейна - итерационный и ре-

Таблица 3.1 – Тесты

		Ожидаемый результат	
Строка 1	Строка 2	РЛ	РДЛ
		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

курсивный (с кешем и без), Дамерау-Левенштейна - рекурсивный без кеша

4 Исследовательская часть

4.1 Пример работы

На рисунке 4.1 приведен пример работы программы.

```
Введите первую строку: 1234567
Введите вторую строку: 1235467

Левенштейн, итерационный.
Матрица:
[0, 1, 2, 3, 4, 5, 6, 7]
[1, 0, 1, 2, 3, 4, 5, 6]
[2, 1, 0, 1, 2, 3, 4, 5]
[3, 2, 1, 0, 1, 2, 3, 4]
[4, 3, 2, 1, 1, 1, 2, 3]
[5, 4, 3, 2, 1, 2, 2, 3]
[6, 5, 4, 3, 2, 2, 2, 3]
[7, 6, 5, 4, 3, 3, 3, 2]
Ответ: 2, время: 0.0

Левенштейн, рекурсивный без кеша.
Ответ: 2, время: 0.03125

Левенштейн, рекурсивный с кешем.
Матрица:
[0, 1, 2, 3, 4, 5, 6, 7]
[1, 0, 1, 2, 3, 4, 5, 6]
[2, 1, 0, 1, 2, 3, 4, 5]
[3, 2, 1, 0, 1, 2, 3, 4]
[4, 3, 2, 1, 1, 1, 2, 3]
[5, 4, 3, 2, 1, 2, 2, 3]
[6, 5, 4, 3, 2, 2, 2, 3]
[7, 6, 5, 4, 3, 3, 3, 2]
Ответ: 2, время: 0.0

Дамерау-Левенштейн, рекурсивный без кеша.
Ответ: 2, время: 0.03125
```

Рисунок 4.1 – Пример работы программы

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система: Windows 10;
- оперативная память: 16 Гб;
- процессор: Intel® Core™ i5-8259U.

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

4.3 Время выполнения реализаций алгоритмов

Все реализации алгоритмов сравнивались на случайно сгенерированных строках длиной от 0 до 10 с шагом 1, а те, что используют матрицы расстояний, еще и на строках длиной от 20 до 100 с шагом 10. Так как замеры времени имеют некоторую погрешность, они для каждой длины строк и каждой реализации алгоритма производились 10 раз, а затем вычислялось среднее время работы реализации со строкой.

На рисунке 4.2 приведены результаты сравнения времени работы всех реализаций. Как видно на графике, реализации, использующие матрицы рассотяний, работают значительно быстрее рекурсивных реализаций без кеширования. Это обусловлено отсутвием в первых вызова функций для вычисления значений, которые уже были подсчитаны ранее.

Чтобы получить полную картину, сравним отдельно реализации, использующие матрицы рассотяний, и те, которые их не используют.

На рисунке 4.3 приведено сравнение времени выполнения реализаций итерационного и рекурсивного (с кешированием) алгоритмов поика расстояния Левенштейна. Как видно, время их работы растет соизмеримо, что

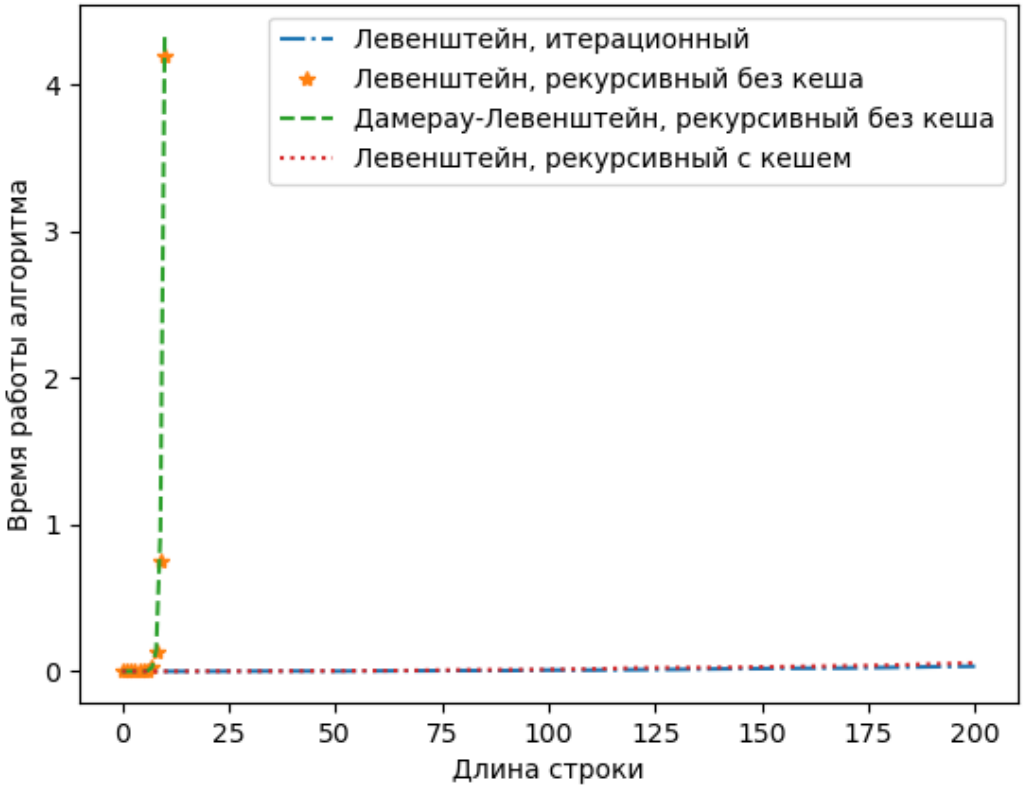

Рисунок 4.2 – Сравнение времени работы реализаций алгоритмов

обусловлено их схожестью в отсутствии вызова функций для вычисления значений, которые уже были подсчитаны ранее. Однако вторая реализация все же работает дольше, так как в ней тратится время на рекурсивный вызов самой себя.

На рисунке 4.4 приведено сравнение времени выполнения реализаций рекурсивных алгоритмов (без кеширования) поика расстояний Левенштейна и Дамерау-Левенштейна. Эти реализации работали примерно одинаковое количество времени, однако для Дамерау-Левенштейна - немного дольше, так как ему требуется время на подсчет штрафа для дополнительной операции - обмена местами соседних символов.

4.4 Пиковая требуемая алгоритмами память

Реализации алгоритмов нахождения расстояний Левенштейна и Дамерау—Левенштейна имеют одинаковый подход к использованию памяти.

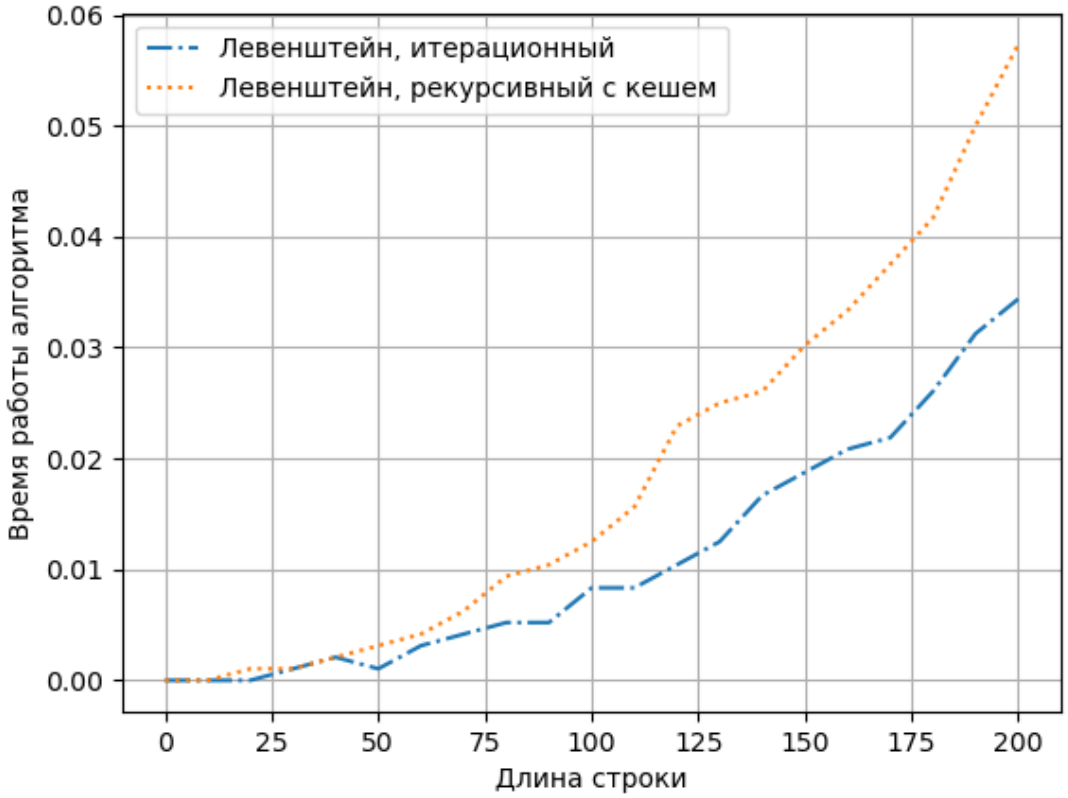

Рисунок 4.3 – Сравнение времени работы реализациий рекурсивных алгоритмов, использующих матрицы растояний

Следовательно, достаточно сравнить, как с увеличением длин строк изменяется пиковая затрачиваемая память в рекурсивных и итерационных реализациях.

При использовании рекурсивной реализации без кеширования для каждого рекурсивного вызова функции будет необходимо выделять память под локальные переменные, возвращаемое функцией значение, адрес возврата. Максимальная глубина стека вызовов на 2 больше суммы длин исходных строк. Следовательно, пиковая затрачиваемая память растет пропорционально сумме длин строк.

При использовании итерационной реализайии память в большей степени затрачивается на матрицу расстояний размером $\text{len}(S_1) + 1$ на $\text{len}(S_2) + 1$. Следовательно, пиковая затрачиваемая память растет пропорционально произведению длин строк, что больше, чем при рекурсивной реализации.

Однако итерационную реализацию можно оптимизировать по памяти, храня лишь последние две строки матрицы расстояний, и тогда пиковая

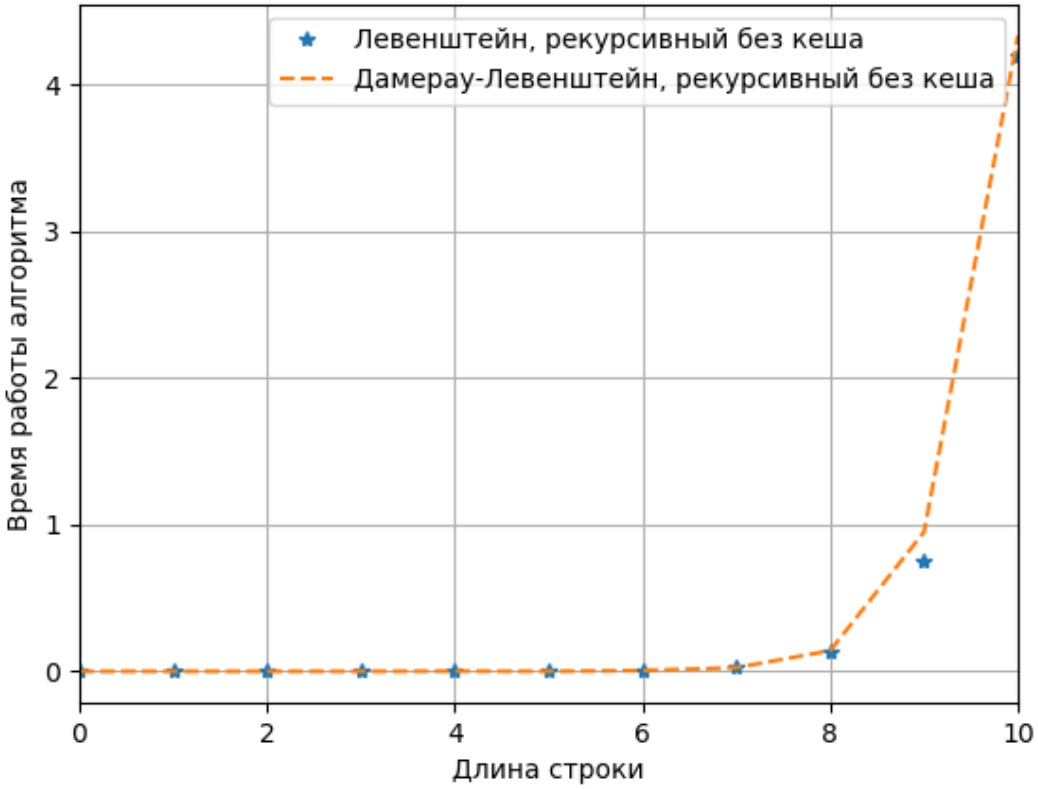

Рисунок 4.4 – Сравнение времени работы реализаций рекурсивных алгоритмов, не использующих кеш

затрачиваемая память будет расти пропорционально изменению длины одной из строк, что меньше, чем при рекурсивной реализации.

Рекурсивные функции с кешированием по этому показателю - худший вариант, так как они потребуют память как под матрицу расстояний (хотя бы в одном экземпляре), так и под стек вызовов.

При небольших длинах строк затрачиваемая память сильно зависит от самой реализации каждого алгоритма (количество локальных переменных, подходы к хранению строк и матриц)

Вывод

Реализации алгоритмов нахождения расстояния Дамерау — Левенштейна по времени выполнения сопоставимы с реализациями алгоритмом нахождения расстояния Левенштейна, хотя и немного уступают вторым в

связи с дополнительной проверкой, позволяющей находить ошибки пользователя, связанные с неверным порядком букв. Однако эта операция зачастую позволяет найти более короткое расстояние между строками.

Рекурсивные реализации алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна, не использующие кеширование, работают на порядок дольше итеративных реализаций. При применении кеширования они требуют меньше времени, однако все равно уступают по производительности итеративным алгоритмам, особенно при большой длине строк.

Но по расходу памяти итеративные реализации проигрывают рекурсивным: максимальный размер используемой памяти в них пропорционален произведению длин строк, в то время как в рекурсивных — сумме длин строк.

Если же применить к итеративным реализациям оптимизацию по памяти, то они будут выигрывать как по пиковой затрачиваемой памяти, так и по времени выполнения.

Заключение

В результате выполнения лабораторной работы при исследовании алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна были применены и отработаны навыки динамического программирования.

В ходе выполнения лабораторной работы были выполнены следующие задачи:

1) изучены расстояния Левенштейна и Дамерау-Левенштейна;
2) разработаны и рнализованы алгоритмы поиска этих расстояний;
3) проведен сравнительный анализ процессорного времени выполнения реализаций этих алгоритмов, а также затрачиваемой ими пиковой памяти;
4) был подготовлен отчет по лабораторной работе.

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