Государственное образовательное учреждение высшего профессионального образования «Московский государственный технический университет имени Н.Э.Баумана» (МГТУ им. Н.Э.Баумана)

₷

2024 год

Отчёт по лабораторной работе №1

По курсу «Анализ алгоритмов»

**Расстояние Левенштейна и Дамерау-Левенштейна**



Выполнила: О.Н. Талышева ИУ7-55Б

Преподаватель: Ю.В. Строганов

Оглавление

[Введение 2](#_Toc176299598)

[Аналитическая часть 3](#_Toc176299599)

[Цель 3](#_Toc176299600)

[Задачи 3](#_Toc176299601)

[Описание алгоритмов 4](#_Toc176299602)

[Расстояние Левенштейна 4](#_Toc176299603)

[Расстояние Дамерау-Левенштейна 6](#_Toc176299604)

[Конструкторская часть 7](#_Toc176299605)

[Схемы алгоритмов 7](#_Toc176299606)

[Технологическая часть 14](#_Toc176299607)

[Требования к программному обеспечению 14](#_Toc176299608)

[Средства реализации 14](#_Toc176299609)

[Реализации алгоритмов 14](#_Toc176299610)

[Тесты 18](#_Toc176299611)

[Экспериментальная часть 19](#_Toc176299612)

[Примеры работы программы 19](#_Toc176299613)

[Сравнение работы реализаций алгоритмов 20](#_Toc176299614)

[Сравнение работы матричной, рекурсивной и рекурсивно-матричной реализаций алгоритмов 21](#_Toc176299615)

[Сравнение работы алгоритмов Левенштейна и Дамерау-Левенштейна 21](#_Toc176299616)

[Сравнение работы матричных и рекурсивно-матричных алгоритмов Левенштейна и Дамерау-Левенштейна 24](#_Toc176299617)

[Заключение 27](#_Toc176299618)

[Источники 28](#_Toc176299619)

# Введение

## Цель

Цель лабораторной работы: исследовать алгоритмы вычисления расстояния Левенштейна и Дамерау-Левенштейна в матричной, рекурсивно-матричной и рекурсивной реализациях.

## Задачи

Для достижения этой цели были поставлены следующие задачи:

* изучить алгоритм вычисления расстояния Левенштейна
* изучить алгоритм вычисления расстояния Дамерау-Левенштейна
* применить метод динамического программирования для матричных реализаций алгоритмов
* сравнить матричную, рекурсивно-матричную и рекурсивную реализации алгоритмов
* сравнить алгоритмы вычисления расстояния Левенштейна и Дамерау-Левенштейна

# Аналитическая часть

## Редакционное расстояние между двумя строками

Часто требуется измерить различие или расстояние между двумя строками (например, в эволюционных, структуральных или функциональных исследованиях биологических строк, в хранении текстовых баз данных, в методах проверки правописания). Есть несколько способов формализации понятия расстояния между строками. Одна общая и простая, формализация называется редакционным расстоянием; она основана на преобразовании (или редактировании) одной строки в другую серией операций редактирования, выполняемых над отдельными символами. Разрешенные операции редактирования — это вставка (I - insertion) символа в первую строку, удаление (D - deletion) символа из первой строки и подстановка или замена (substitution или, лучше, R - replace) символа из первой строки символом из второй строки. Обозначим M — “не-операцию“ над правильной буквой (от match).

Строка над алфавитом 1, D, R , М, которая описывает преобразование одной строки в другую, называется редакционным предписанием (предписанием) этих двух строк.

Редакционное расстояние между двумя строками определяется как минимальное число редакционных операций — вставок, удалений и подстановок, необходимое для преобразования первой строки во вторую.

Подчеркнем, что совпадения операциями не являются и не засчитываются.

Редакционное расстояние иногда называют расстоянием Левенштейна по статье В. Левенштейна, где оно рассматривалось, вероятно, впервые.

## Выравнивание строк

Редакционное предписание — это способ представления конкретного преобразования одной строки в другую. Альтернативный (и часто предпочтительный) способ заключается в показе явного выравнивания (alignment) этих двух строк.

(Глобальное) выравнивание двух строк, Sl и S2, получается вставкой пробелов в строки S1 и S2 (возможно, на их концах) и размещением двух получившихся строк друг над другом так, чтобы каждый символ или пробел одной строки оказался напротив одного символа или пробела другой строки.

Термин «глобальный» подчёркивает, что обе строки участвуют в выравнивании полностью. [3]

## Расстояние Левенштейна

Расстояние Левенштейна, или редакционное расстояние, — метрика cходства между двумя строковыми последовательностями. Чем больше расстояние, тем более различны строки. По сути, это минимальное число односимвольных преобразований (удаления, вставки или замены), необходимых, чтобы превратить одну последовательность в другую.

Цены операций могут зависеть от вида операции (вставка, удаление, замена) и/или от участвующих в ней символов, отражая разную вероятность мутаций в биологии, разную вероятность разных ошибок при вводе текста и т. д. В общем случае:

* Дл (a, b) — цена замены символа a на символ b
* Дл (λ, b) — цена вставки символа b
* Дл (a, λ) — цена удаления символа a

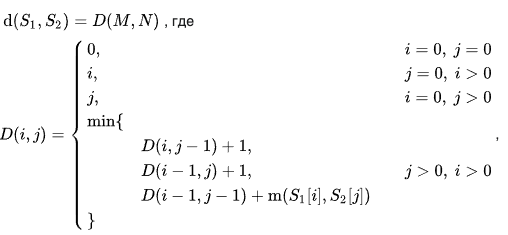
Необходимо найти последовательность замен, минимизирующую суммарную цену. Расстояние Левенштейна является частным случаем этой задачи при ценах:

* Дл (a, а) = 0
* Дл (a, b) = 1 при a≠b
* Дл (λ, b) = 1
* Дл (a, λ) = 1

Как частный случай, так и задачу для произвольных Дл, решает алгоритм Вагнера — Фишера, приведённый ниже. Здесь и ниже считается, что все Дл неотрицательны, и действует неравенство треугольника: замена двух последовательных операций одной не увеличит общую цену (например, замена символа x на y, а потом y на z не лучше, чем сразу x на z).

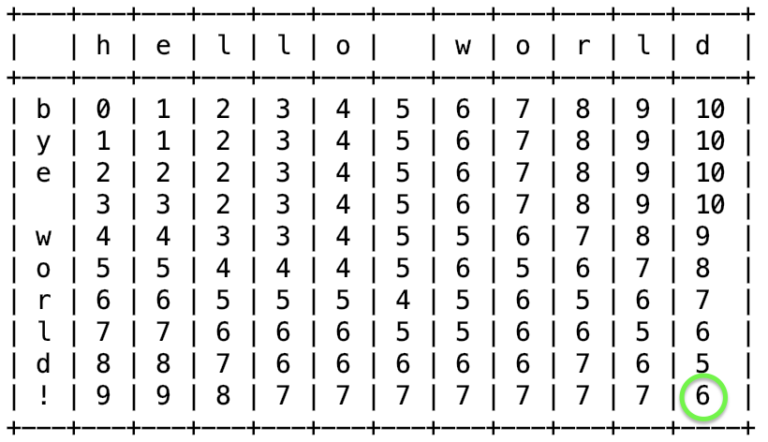
Например, Дл (’hello’, ‘hallo’) = 1, так как потребуется провести одну замену ‘e’ на ‘a’.

Алгоритм реализуется по следующей формуле:



Функция **m** – возвращает единицу, если символы не равны, иначе 0.

Таким образом, требуется вычислить матрицу расстояний размерностью len(str1) \* len(str2), следовательно, объем требуемой памяти растет как O(len(str1) \* len(str2)). Иными словами, для двух мегабайтных строк потребуются гигабайты памяти. Фактически в кэше будет хранится почти все матрица редактирований, а она не нужна целиком. Искомая цель – правый нижний элемент.



Для его поиска можно обойтись лишь парой рядов: текущим и предыдущим. А остальные ряды не хранить в памяти. Так будет достигнут конец таблицы, и нижний правый угол и будет искомым значением.

Чтобы использовать еще меньше памяти, можно поменять местами строки, чтобы длина рядов была минимальна. Это существенно экономит память, если одна из строк длинная, а другая короткая.

## Расстояние Дамерау-Левенштейна

Если к списку разрешённых операций добавить транспозицию (два соседних символа меняются местами), получается расстояние Дамерау — Левенштейна. Для неё также существует алгоритм, требующий O(len(str1) \* len(str2)) операций. Дамерау показал, что 80 % ошибок при наборе текста человеком являются транспозициями. Кроме того, расстояние Дамерау-Левенштейна используется и в биоинформатике.

Цена операции транспозиция также равна 1. При работе алгоритма Левенштейна эта операция реализовалась бы двумя заменами и стоила бы 2. Таким образом, расстояние Дамерау-Левенштейна в некоторых случаях даёт меньший результат, чем расстояние Левенштейна.

В алгоритм добавляется следующая формула:

i > 1

j > 1

str1[i - 1] = str2[j]

str1[i] = str2[j – 1]

# Конструкторская часть

## Схемы алгоритмов

На основании теоретических измышлений были разработаны алгоритмы, вычисляющие расстояние Левенштейна и Дамерау-Левенштейна тремя способами:

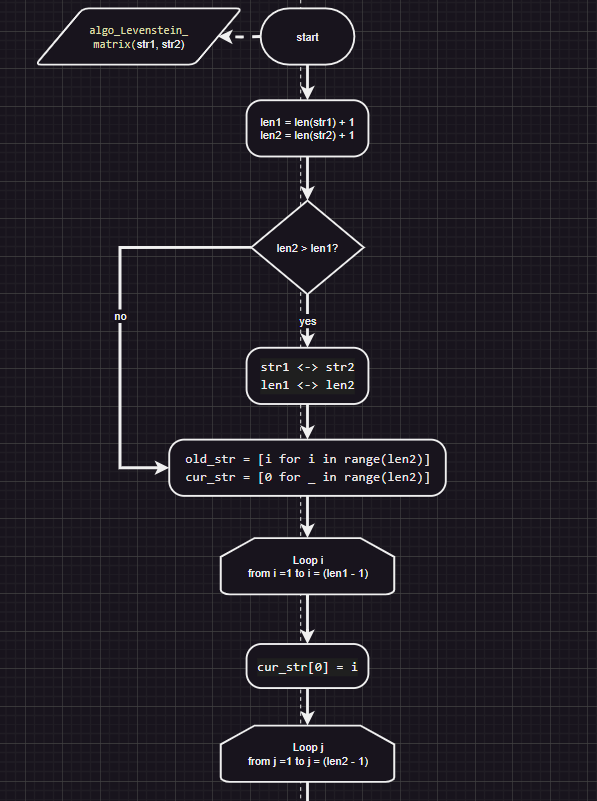


Рисунок 1.1.1 Блоксхема алгоритма Левенштейна (матричная реализация)

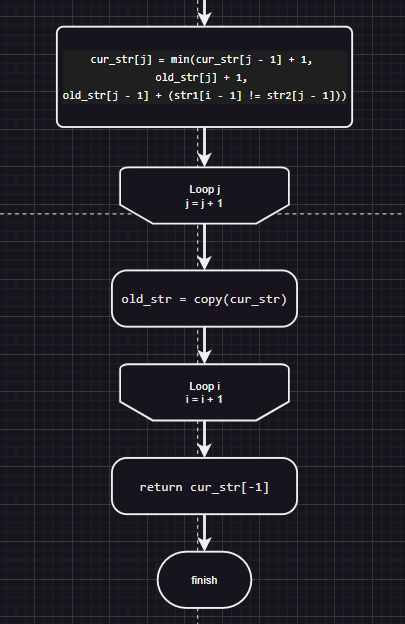


Рисунок 1.1.2 Блоксхема алгоритма Левенштейна (матричная реализация) (продолжение)

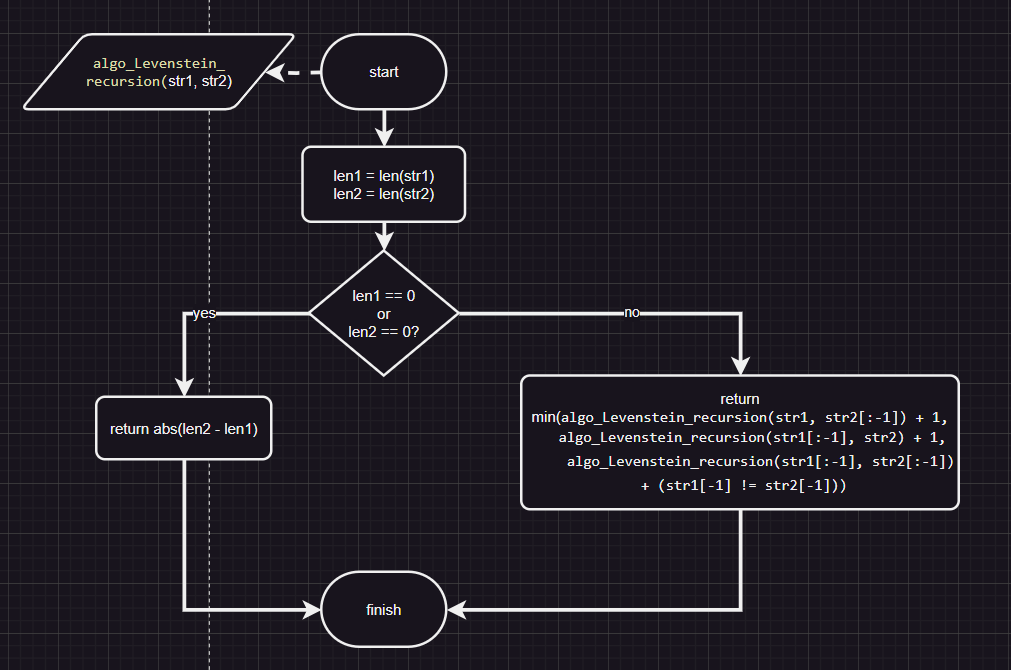


Рисунок 1.2 Блоксхема алгоритма Левенштейна (рекурсивная реализация)

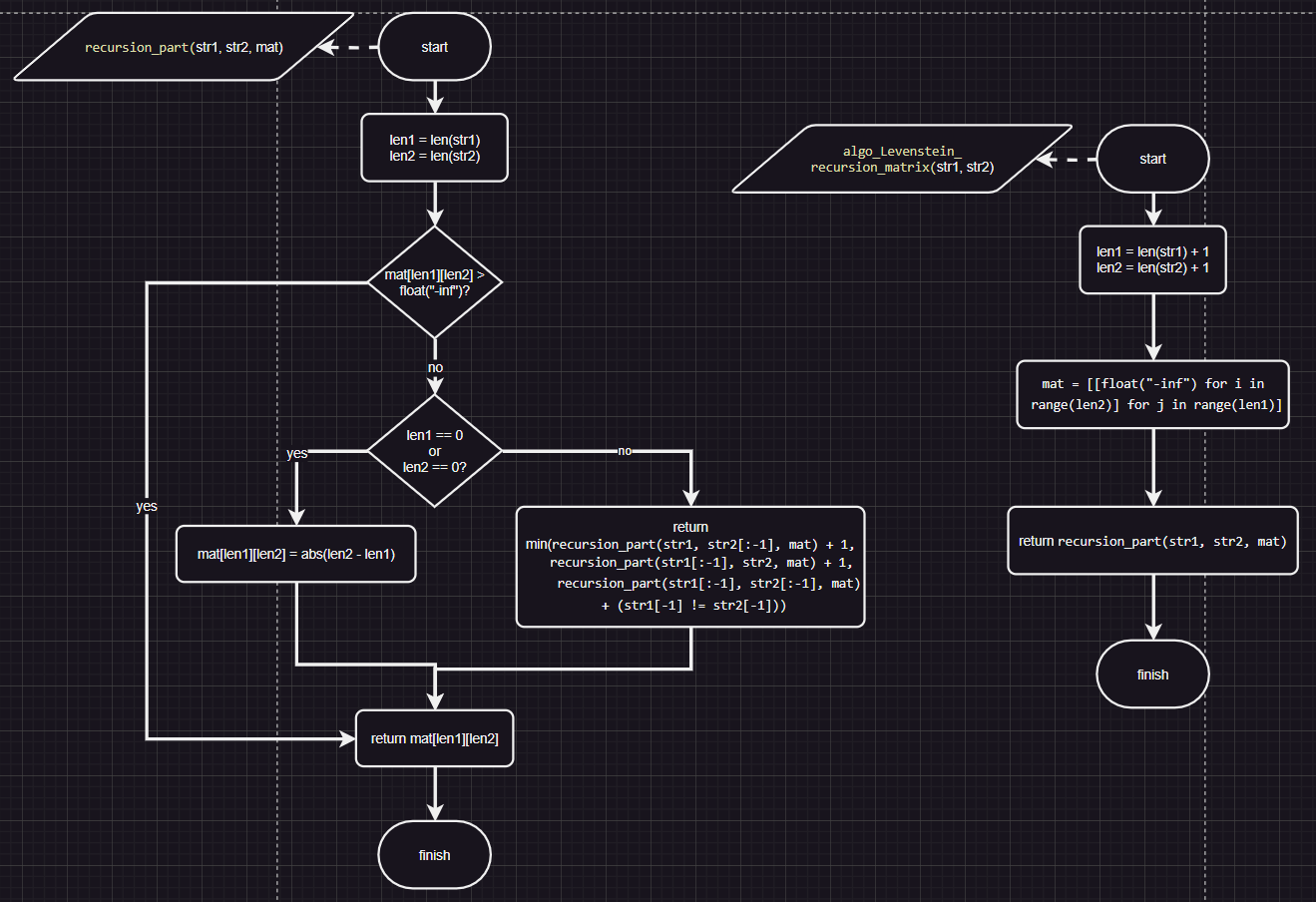


Рисунок 1.3 Блоксхема алгоритма Левенштейна (рекурсивно-матричная реализация)

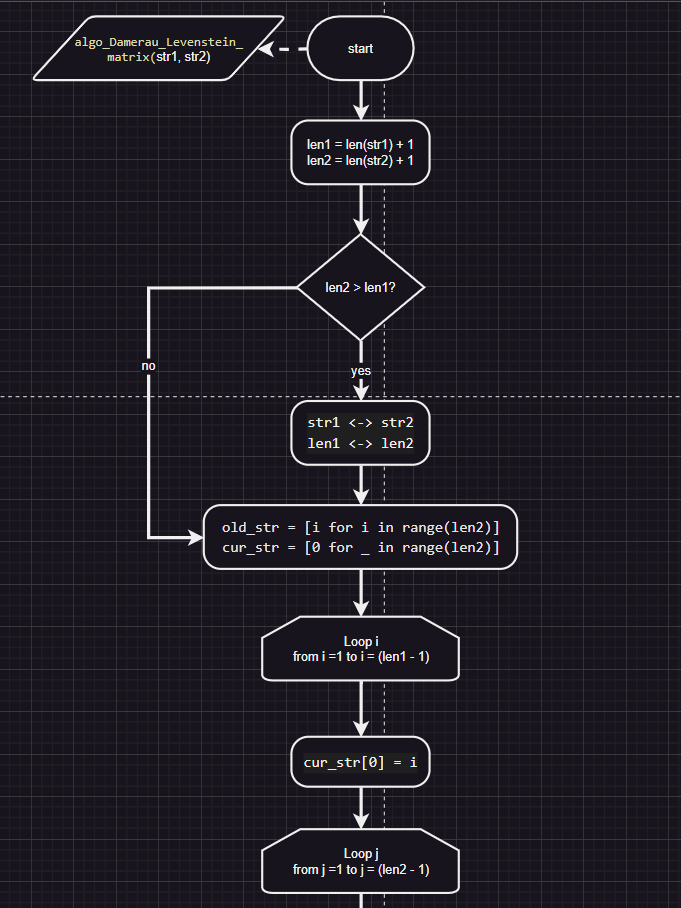


Рисунок 2.1.1 Блоксхема алгоритма Дамерау-Левенштейна (матричная реализация)

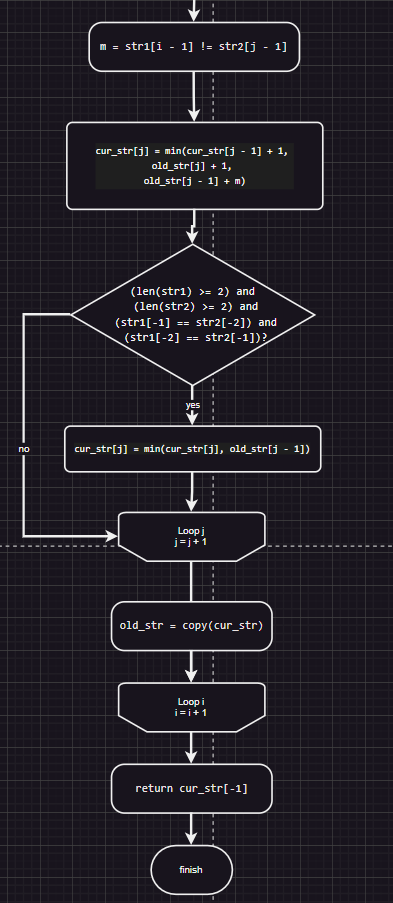


Рисунок 2.1.2 Блоксхема алгоритма Дамерау-Левенштейна (матричная реализация) (продолжение)

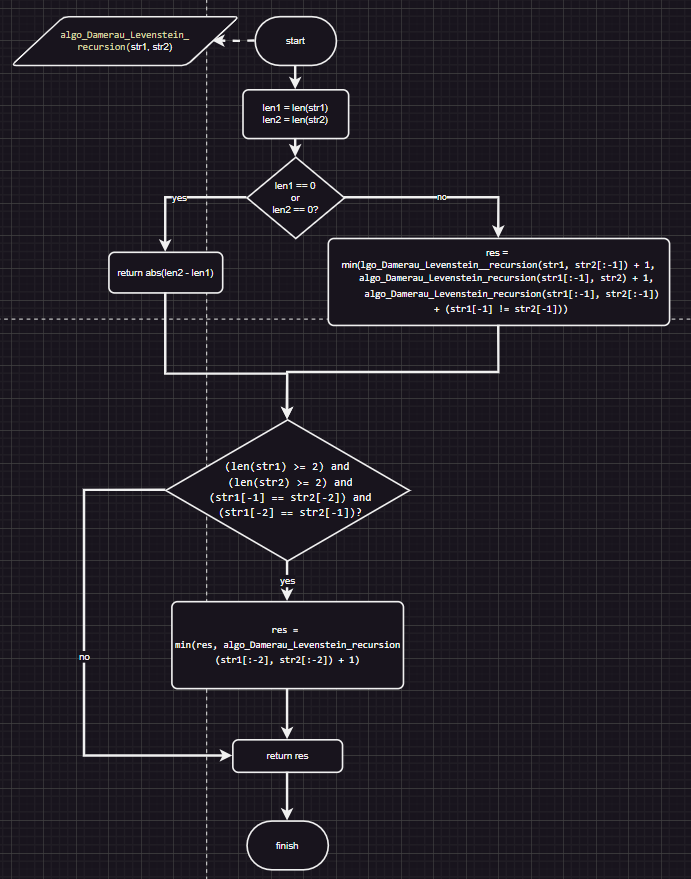


Рисунок 2.2 Блоксхема алгоритма Дамерау-Левенштейна (рекурсивная реализация)

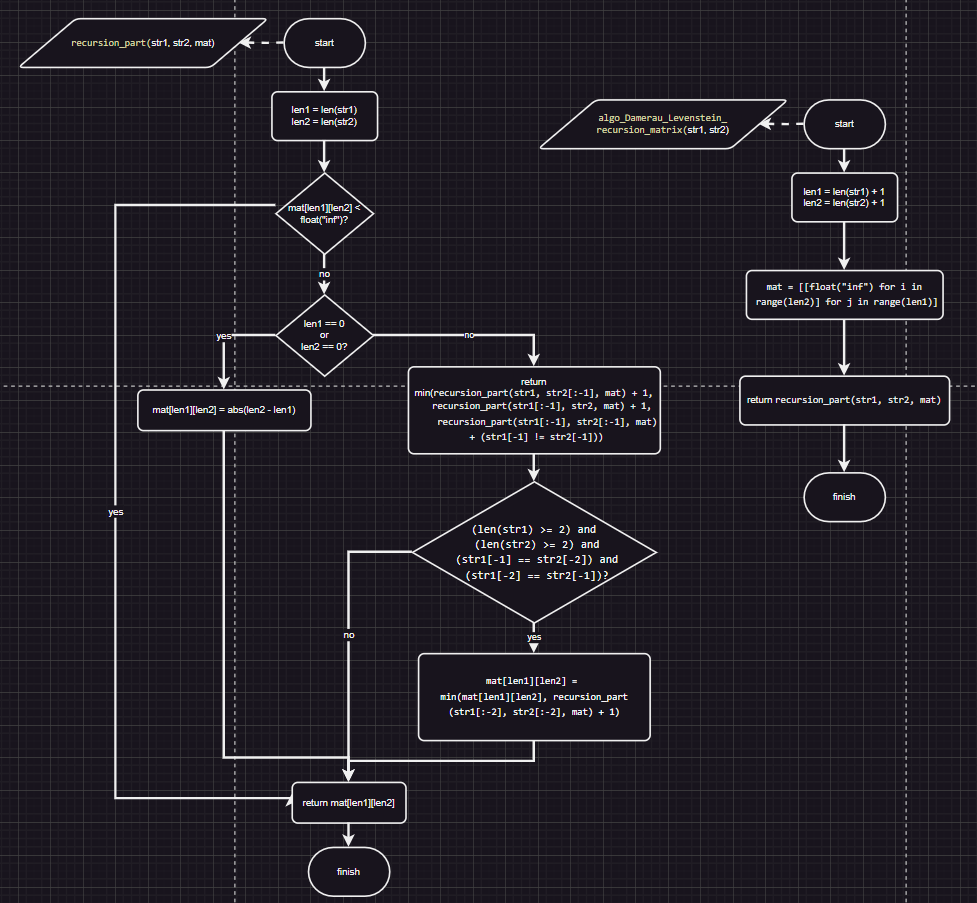


Рисунок 2.3 Блоксхема алгоритма Дамерау-Левенштейна (рекурсивно-матричная реализация)

# Технологическая часть

## Требования к программному обеспечению

На вход программе подаются 2 строки из символов, которые входят в таблицу Юникода (UTF-8).

На выход программа выдаёт число – расстояние между строками, вычисленное алгоритмом Левенштейна или Дамерау-Левенштейна матричной или рекурсивной реализацией. Для матричных реализаций также выводится матрица расстояний. Также в зависимости от выбранного пункта меню программа замеряет время работы алгоритмов и рисует получившиеся графики.

## Средства реализации

Python быстро работает с алгоритмами Левенштейна и Дамерау-Левенштейна благодаря эффективной обработке строк и богатой библиотеке, что делает его идеальным для задач текстового сравнения и редактирования. Поэтому программа была реализована на языке Python, для замеров времени была использована функция process\_time() из библиотеки time, вычисляющая процессорное время[1].

## Реализации алгоритмов

## 

Листинг 1: реализация матричного алгоритма Левенштейна

1 | def algo\_Levenstein\_matrix(str1: str, str2: str) -> int:

2 |     len1, len2 = len(str1) + 1, len(str2) + 1

3 |     if len2 > len1:

4 |         str1, str2 = str2, str1

5 |         len1, len2 = len2, len1

6 |     old\_str = [i for i in range(len2)]

7 |     cur\_str = [0 for \_ in range(len2)]

8 |     for i in range(1, len1):

9 |         cur\_str[0] = i

10|         for j in range(1, len2):

11|             cur\_str[j] = min(cur\_str[j - 1] + 1,

12|                             old\_str[j] + 1,

13|                             old\_str[j - 1] + (str1[i - 1] != str2[j - 1]))

14|         old\_str = cur\_str.copy()

15|     return cur\_str[-1]

Листинг 2: реализация рекурсивного алгоритма Левенштейна

1 | def algo\_Levenstein\_recursion(str1: str, str2: str) -> int:

2 |     len1, len2 = len(str1), len(str2)

3 |     if len1 \* len2 == 0:

4 |         return abs(len2 - len1)

5 |     return min(algo\_Levenstein\_recursion(str1, str2[:-1]) + 1,

6 |                 algo\_Levenstein\_recursion(str1[:-1], str2) + 1,

7 |                 algo\_Levenstein\_recursion(str1[:-1], str2[:-1]) + (str1[-1] != str2[-1]))

Листинг 3: реализация рекурсивно-матричного алгоритма Левенштейна

1 | def algo\_Levenstein\_recursion\_matrix(str1: str, str2: str) -> int:

2 |     len1, len2 = len(str1) + 1, len(str2) + 1

3 |     mat = [[float("-inf") for i in range(len2)] for j in range(len1)]

4 |     *# сама рекурсивная часть*

5 |     def recursion\_part(str1: str, str2: str, mat: List[float] = []) -> int:

6 |         len1, len2 = len(str1), len(str2)

7 |         if mat[len1][len2] > float("-inf"):

8 |             pass

9 |         elif len1 \* len2 == 0:

10|             mat[len1][len2] = abs(len2 - len1)

11|         else:

12|             mat[len1][len2] = min(recursion\_part(str1, str2[:-1], mat) + 1,

13|                     recursion\_part(str1[:-1], str2, mat) + 1,

14|                     recursion\_part(str1[:-1], str2[:-1], mat) + (str1[-1] != str2[-1]))

15|         return mat[len1][len2]

16|     return recursion\_part(str1, str2, mat)

Листинг 4: реализация матричного алгоритма Дамерау-Левенштейна

1 | def algo\_Damerau\_Levenstein\_matrix(str1: str, str2: str) -> int:

2 |     len1, len2 = len(str1) + 1, len(str2) + 1

3 |     if len2 > len1:

4 |         str1, str2 = str2, str1

5 |         len1, len2 = len2, len1

6 |     old\_str = [i for i in range(len2)]

7 |     cur\_str = [0 for \_ in range(len2)]

8 |     for i in range(1, len1):

9 |         cur\_str[0] = i

10|         for j in range(1, len2):

11|             m = str1[i - 1] != str2[j - 1]

12|             cur\_str[j] = min(cur\_str[j - 1] + 1,

13|                             old\_str[j] + 1,

14|                             old\_str[j - 1] + m)

15|             if (i > 1) and (j > 1) and m and (str1[i - 2] == str2[j - 1]) and (str1[i - 1] == str2[j - 2]):

16|                 cur\_str[j] = min(cur\_str[j], old\_str[j - 1])

17|         old\_str = cur\_str.copy()

18|     return cur\_str[-1]

Листинг 5: реализация рекурсивного алгоритма Дамерау-Левенштейна

1 | def algo\_Damerau\_Levenstein\_recursion(str1: str, str2: str) -> int:

2 |     len1, len2 = len(str1), len(str2)

3 |     if len1 \* len2 == 0:

4 |         return abs(len2 - len1)

5 |     res = min(algo\_Damerau\_Levenstein\_recursion(str1, str2[:-1]) + 1,

6 |                 algo\_Damerau\_Levenstein\_recursion(str1[:-1], str2) + 1,

7 |                 algo\_Damerau\_Levenstein\_recursion(str1[:-1], str2[:-1]) + (str1[-1] != str2[-1]))

8 |     if ((len(str1) >= 2) and (len(str2) >= 2) and (str1[-1] == str2[-2]) and (str1[-2] == str2[-1])):

9 |         res = min(res, algo\_Damerau\_Levenstein\_recursion(str1[:-2], str2[:-2]) + 1)

10|     return res

Листинг 6: реализация рекурсивно-матричного алгоритма Дамерау-Левенштейна

1 | def algo\_Damerau\_Levenstein\_recursion\_matrix(str1: str, str2: str) -> int:

2 |     len1, len2 = len(str1) + 1, len(str2) + 1

3 |     mat = [[float("inf") for i in range(len2)] for j in range(len1)]

4 |     *# сама рекурсивная часть*

5 |     def recursion\_part(str1: str, str2: str, mat: List[float] = []) -> int:

6 |         len1, len2 = len(str1), len(str2)

7 |         if mat[len1][len2] < float("inf"):

8 |             pass

9 |         elif len1 \* len2 == 0:

10|             mat[len1][len2] = abs(len2 - len1)

11|         else:

12|             mat[len1][len2] = min(recursion\_part(str1, str2[:-1], mat) + 1,

13|                     recursion\_part(str1[:-1], str2, mat) + 1,

14|                     recursion\_part(str1[:-1], str2[:-1], mat) + (str1[-1] != str2[-1]))

15|             if ((len(str1) >= 2) and (len(str2) >= 2) and (str1[-1] == str2[-2]) and (str1[-2] == str2[-1])):

16|                 mat[len1][len2] = min(mat[len1][len2], recursion\_part(str1[:-2], str2[:-2], mat) + 1)

17|         return mat[len1][len2]

18|     return recursion\_part(str1, str2, mat)

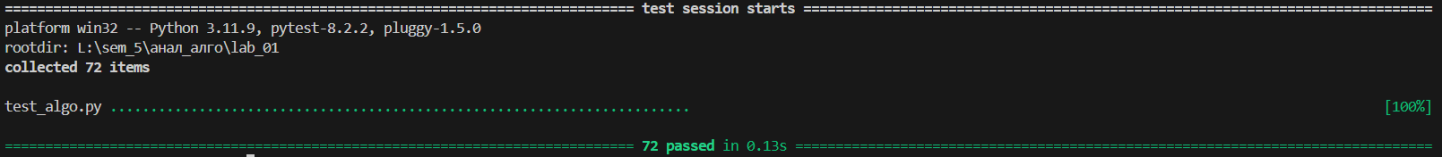
## Тесты

Введём обозначения:

* λ - пустая строка
* Л. – Левенштейн
* Д-Л. - Дамерау – Левенштейн
* мат – матричный
* рек – рекурсивный

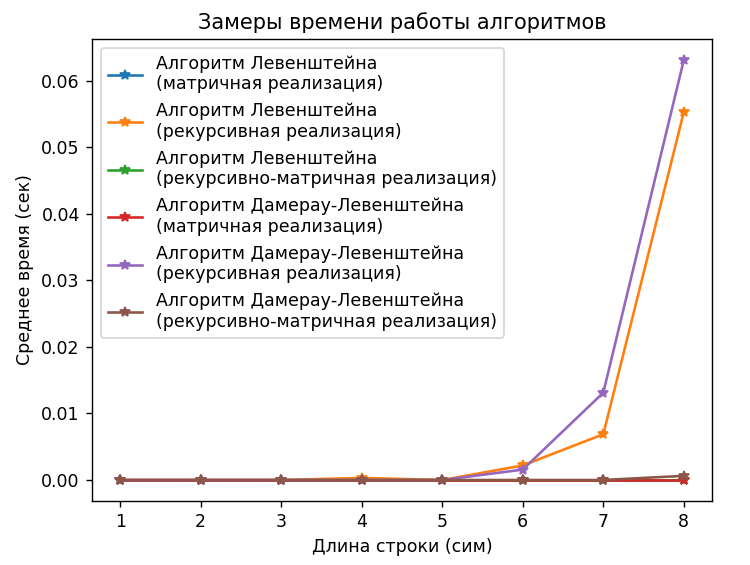
|  |  |  |
| --- | --- | --- |
| Матрица №1 | Матрица №2 |  |
| (()) | (()) |  |
| ((a)) | ((а)) | ((а\*а)) |
| ((a, b, c)) | ((a), (b), (c)) |  |
| λ | a |  |
| a | λ |  |
| a | b |  |
| abc | abs |  |
| odc | abc |  |
| ods | abc |  |
| abcs | abc |  |
| bc | abc |  |
| bac | abc |  |

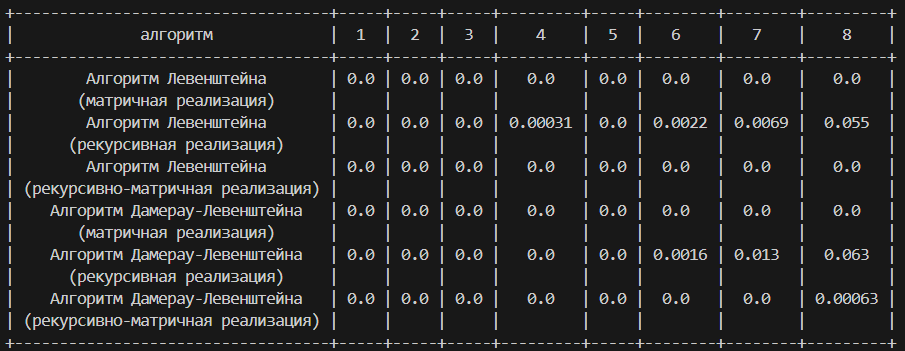
В ходе проведённого тестирования (с помощью pytest) ошибок в алгоритмах не выявлено:



## Исследовательская часть

Для сравнения времени реализаций алгоритмов Левенштейна и Дамерау-Левенштейна в их матричных и рекурсивных реализациях программа была запущена на рандомно сгенерированных строках длинами от 1 до 9 с шагом 2 по 50 замеров каждая строка, среднее значение было вынесено в таблицу и для наглядности изображено на графике:



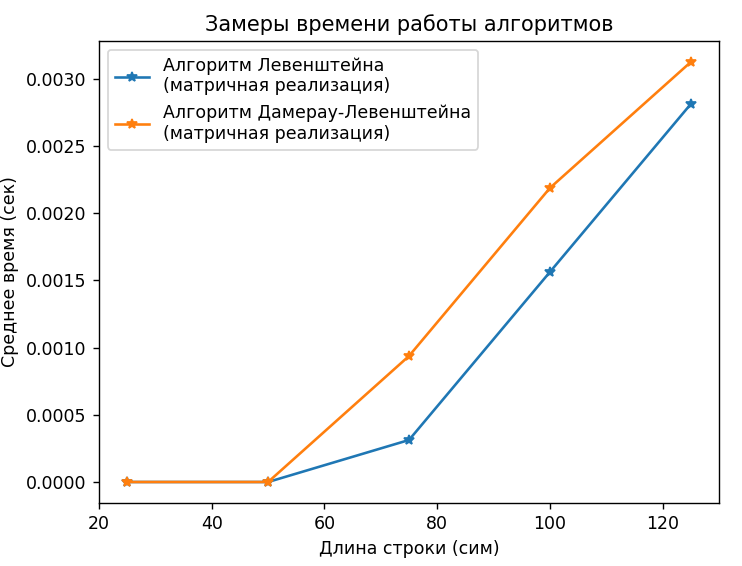


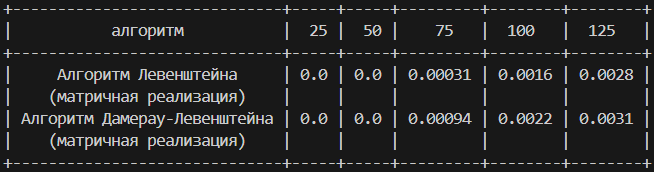
### Сравнение работы матричной, рекурсивной и рекурсивно-матричной реализаций алгоритмов

Из графиков, приведённых выше, очевидно, что матричная реализация обоих алгоритмов быстро становится эффективнее рекурсивной на много порядков. Это происходит из-за того, что при рекурсии даже на небольшой длине строк происходит много рекурсивных вызовов для подстрок, на что тратится большое количество времени и памяти. В то время как для матричной реализации данные, на основе которых вычисляются следующие значения, хранятся в двух массивах длинной в кратчайшую из двух строк, что экономит как время, так и память. При этом рекурсивно-матричная реализация оказалась почти столь же быстрой, как и матричная благодаря исключению повторных вычислений идентичных веток рекурсии, что в разы сократило количество вычислений.

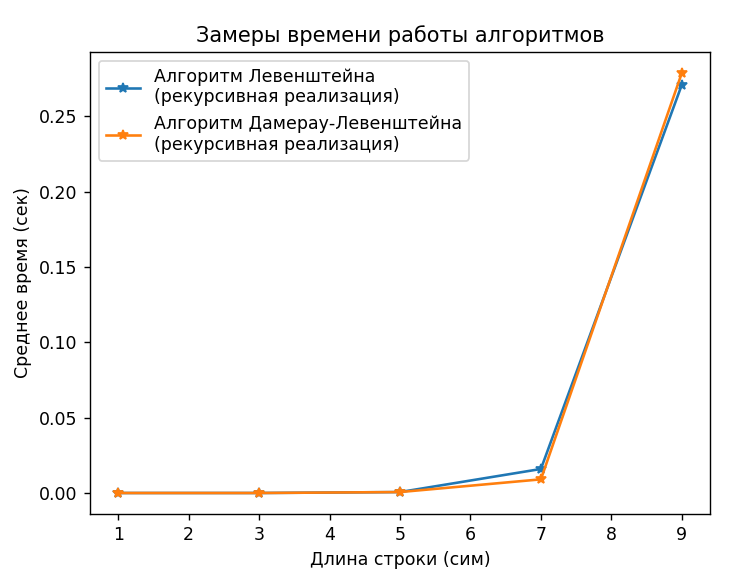
### Сравнение работы алгоритмов Левенштейна и Дамерау-Левенштейна

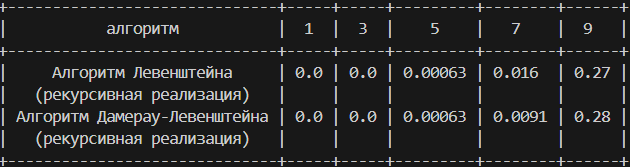
1. Матричная реализация алгоритмов (на 5-и точках с длиной строк от 25 до 125 символов с шагом 25)



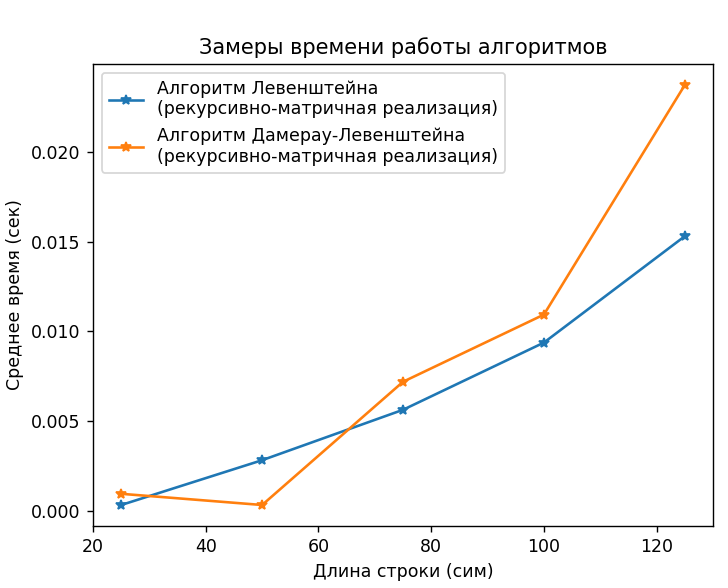


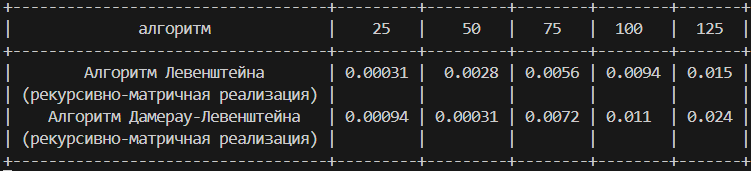
1. Рекурсивная реализация алгоритмов (на 5-и точках с длиной строк от 1 до 9 символов с шагом 2)





1. Рекурсивно-матричная реализация алгоритмов (на 5-и точках с длиной строк от 25 до 125 символов с шагом 25)

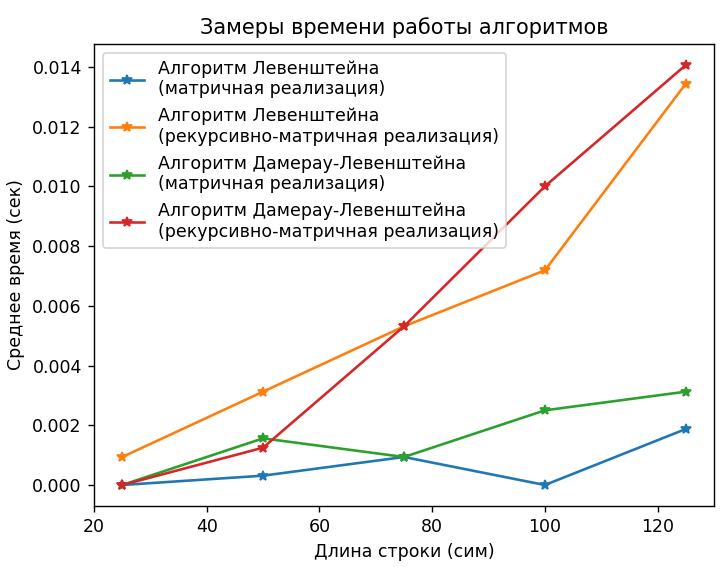


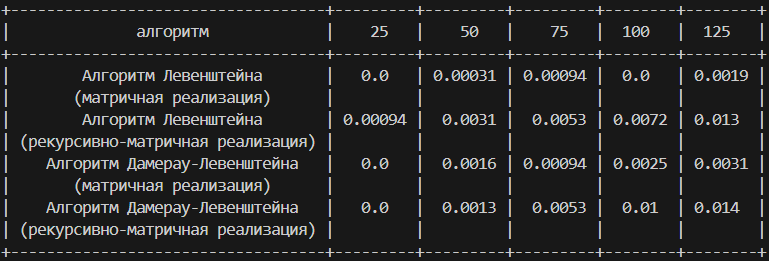


Видно, что алгоритм Левенштейна оказался немного быстрее алгоритма Дамерау-Левенштейна из-за дополнительной проверки во втором, что компенсируется меньшей эффективностью первого при наличии перестановок букв в строках.

### Сравнение работы матричных и рекурсивно-матричных алгоритмов Левенштейна и Дамерау-Левенштейна

Так как на общем графике матричный и рекурсивно-матричный алгоритмы были очень близки по скорости, были проведены отдельные замеры (на 5-и точках с длиной строк от 25 до 125 символов с шагом 25).





По результатам приведённых графиков видно, что и в алгоритме Левенштейна и Дамерау-Левенштейна рекурсивно-матричный метод работает дольше матричного. Это объясняется затратами на вызов функции при рекурсии и на дополнительные проверки является ли искомое значение уже посчитанным. На небольших длинах строк разница в скорости работы алгоритмов отличается несущественно, однако с увеличением данных растёт и разница во времени.

### Вывод

По результатам проведённых исследований была выявлена большая скорость работы алгоритма Левенштейна над алгоритмом Дамерау-Левенштейна за счёт уменьшения числа проверок, что, однако, даёт иной результат при наличии возможности перестановок символов в строках. При этом матричный вариант выигрывает по скорости в обоих алгоритмах, на втором месте оказался рекурсивно-матричный метод, который делает меньше рекурсивных вызовов, чем рекурсивный метод, и исключает повторные вычисления идентичных веток, так как при вызове каждой новой функции в этом методе передаётся в качестве аргумента ссылка на матрицу, которая хранит уже посчитанные значения, но на эту матрицу также необходима память, а проверки на уже вычисленные значения не всегда приносят положительный результат и занимают время.

# Заключение

В результате выполнения лабораторной работы были исследованы алгоритмы вычисления расстояния Левенштейна и Дамерау-Левенштейна в матричной, рекурсивно-матричной и рекурсивной реализациях.

В частности:

* были изучены алгоритмы вычисления расстояния Левенштейна и Дамерау-Левенштейна;
* применён метод динамического программирования для матричных реализаций алгоритмов;
* сравнены матричная, рекурсивно-матричная и рекурсивная реализации алгоритмов;
* сравнены алгоритмы вычисления расстояния Левенштейна и Дамерау-Левенштейна.

# Источники

**Статьи и онлайн-ресурсы:**

1. **Python Documentation**. *time.process\_time()* - Документация по стандартной библиотеке Python. Дата обращения: 03 сентября 2024 г. [URL](https://docs-python.ru/standart-library/modul-time-python/funktsija-process-time-modulja-time/)
2. **Tirinox**. *Алгоритм Левенштейна на Python: реализация и объяснение*. Дата обращения: 02 сентября 2024 г. [URL](https://tirinox.ru/levenstein-python/)

**Книги и научные работы:**

1. **Гасфилд Дэн** «Строки, деревья и последовательности в алгоритмах»: Информатика и вычислительная биология / Пер. с англ. И. В. Романовского. — СПб.: Невский Диалект; БХВ-Петербург, 2003. — 654 с: ил.
2. **Ниёзов Д. Л.** Применение методов нечеткого сравнения строк в прикладных задачах: Выпускная квалификационная работа (Бакалаврская работа). — Тольятти: Тольяттинский государственный университет, 2020. — 45 стр.
3. **В. И. Левенштейн.** Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965. 163.4:845-848.