Министерство науки и высшего образования Российской Федерации
Московский государственный технический университет им. Н. Э. Баумана
Кафедра ИУ7 «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЁТ

по лабораторной работе №1
по курсу «Анализ алгоритмов»
Тема: «Расстояние Левенштейна и Дамерау-Левенштейна»

Выполнила: студентка группы ИУ7-55Б Талышева О.Н.
Преподаватель: Строганов Ю.В.

Москва, 2024

Содержание

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

1. Аналитическая часть

1.1. 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 , M, 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.

1.2. 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]

1.3. Расстояние Левенштейна

Расстояние Левенштейна, или редакционное расстояние, — метрика сходства 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:

- D(a, b) — цена замены символа a на символ b
- D(λ, b) — цена вставки символа b
- D(a, λ) — цена удаления символа a

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

- D(a, a) = 0
- D(a, b) = 1, при a $\neq$ b
- D(λ, b) = 1
- D(a, λ) = 1

Как частный случай, так и задачу для произвольных D, решает алгоритм Вагнера — Фишера, приведённый ниже. Здесь и ниже считается, что все D 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, D('hello', 'hallo') = 1, так как потребуется провести одну замену 'e' на 'a'.
Алгоритм реализуется по следующей формуле:

$$d(S_1,S_2)=D(M,N), \text{где} D(i,j)=\begin{cases} 0, & \text{i = 0, j = 0} \\ i, & \text{i > 0, j = 0} \\ j, & \text{i = 0, j > 0} \\ \min\begin{cases} D(i-1,j)+1 & \text{(удаление)} \\ D(i,j-1)+1 & \text{(вставка)} \\ D(i-1,j-1)+1_{\text{если} a_i \neq b_j} & \text{(замена)} \end{cases} & \text{i > 0, j > 0} \end{cases}$$

Таким образом, требуется вычислить матрицу расстояний размерностью $len(str_1)*len(str_2)$, следовательно, объем требуемой памяти растет как $O(len(str_1)*len(str_2))$. 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.

	h	e	l	l	o		w	o	r	l	d
b	0	1	2	3	4	5	6	7	8	9	10
y	1	1	2	3	4	5	6	7	8	9	10
e	2	2	2	3	4	5	6	7	8	9	10
	3	3	2	3	4	5	6	7	8	9	10
w	4	4	3	3	4	5	5	6	7	8	9
o	5	5	4	4	4	5	6	5	6	7	8
r	6	6	5	5	5	4	5	6	5	6	7
l	7	7	6	6	6	5	5	6	6	5	6
d	8	8	7	6	6	6	6	6	7	6	5
!	9	9	8	7	7	7	7	7	7	7	6

Рис. 1: Пример нахождения расстояния Левенштейна

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.

1.4. 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:

$$\begin{cases} i > 1 \\ j > 1 \\ \mathrm{str}_1[i-1] = \mathrm{str}_2[j] \\ \mathrm{str}_1[i] = \mathrm{str}_2[j-1] \end{cases}$$

2. Конструкторская часть

Схемы алгоритмов

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

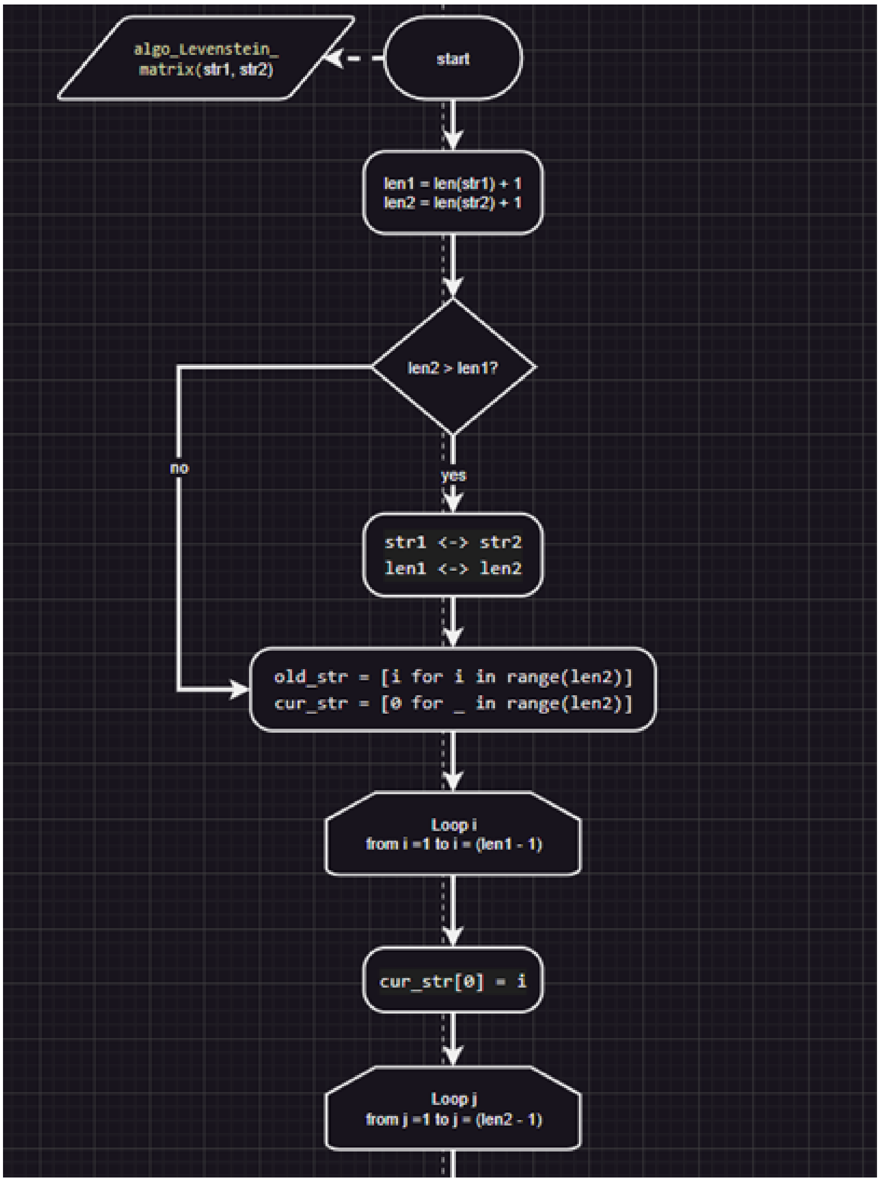

Рис. 2: Блоксхема алгоритма Левенштейна (матричная реализация)

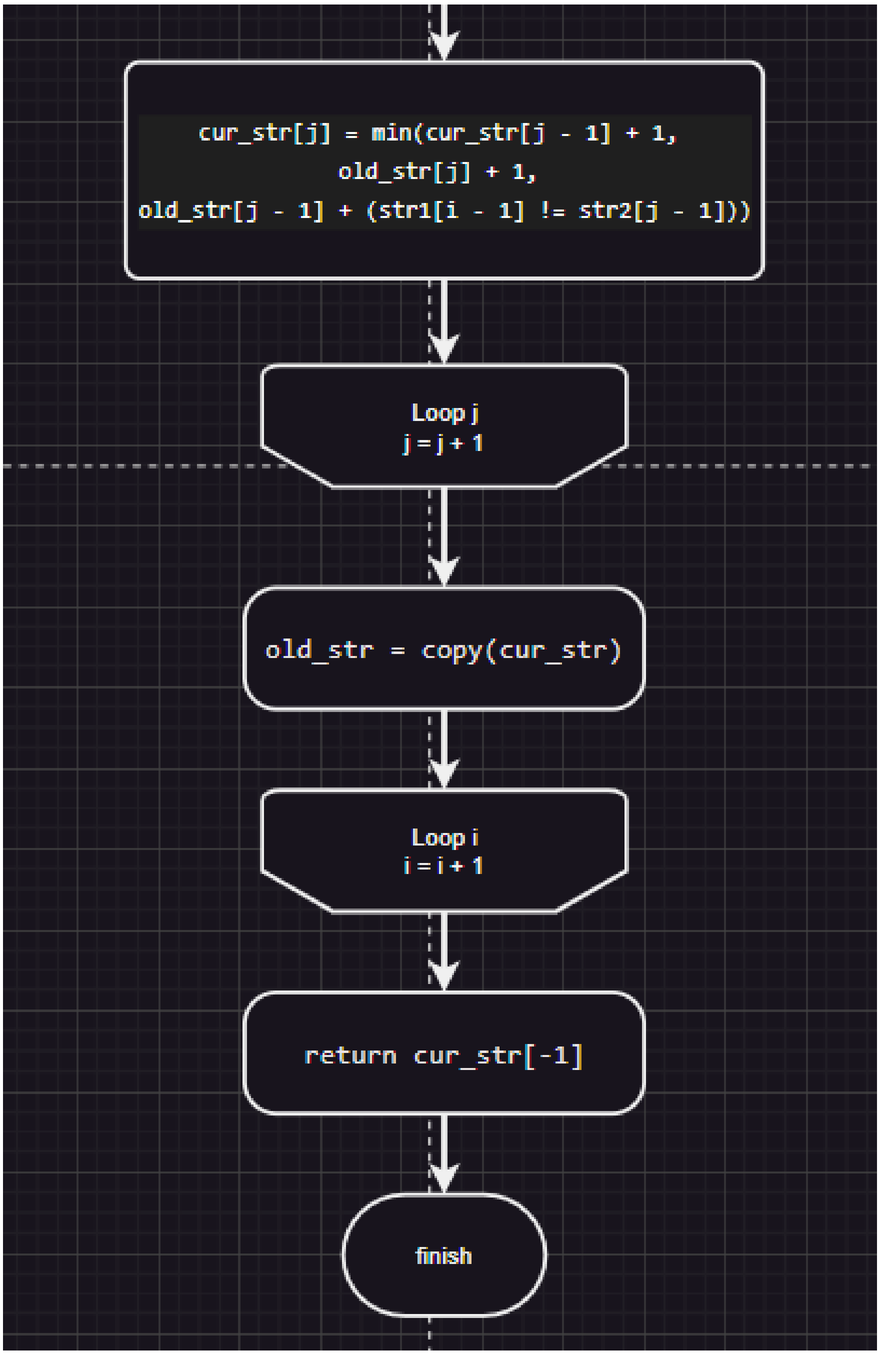

Рис. 3: Блоксхема алгоритма Левенштейна (матричная реализация) (продолжение)

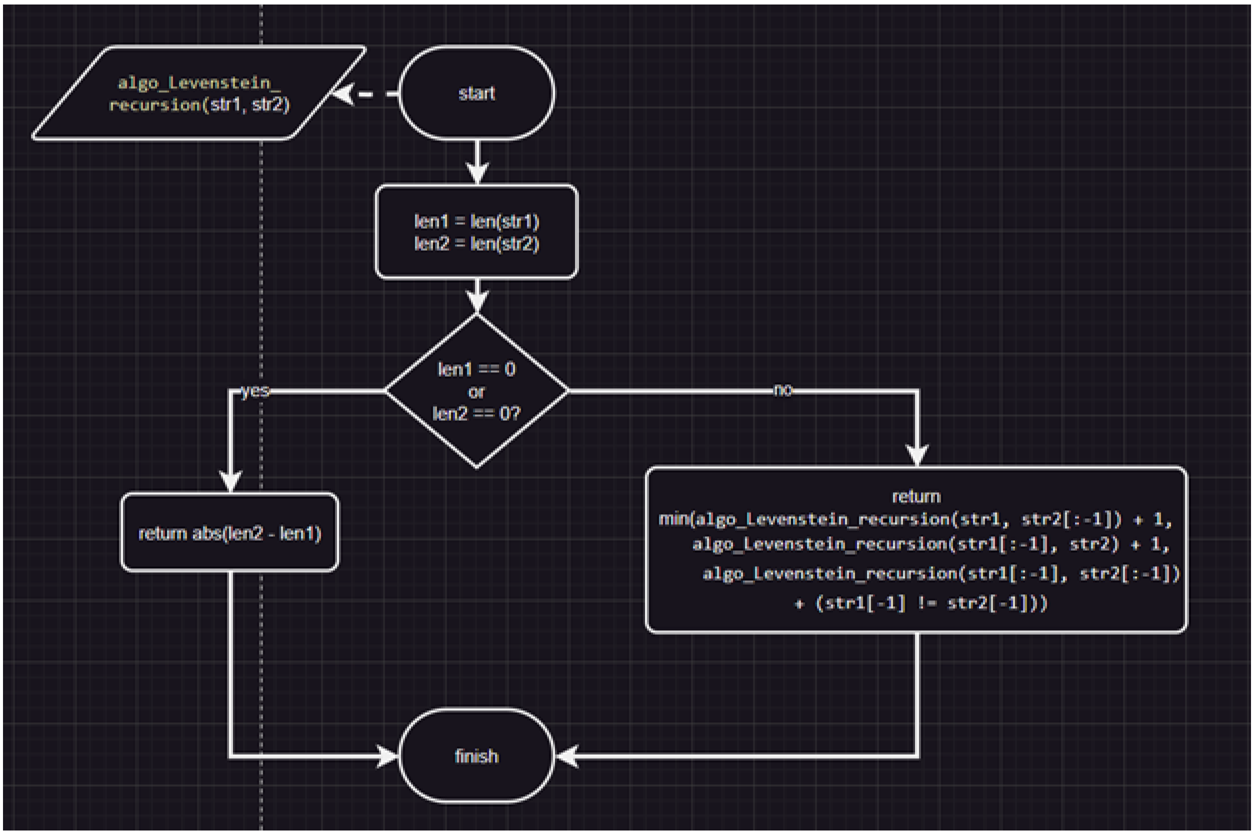

Рис. 4: Блоксхема алгоритма Левенштейна (рекурсивная реализация)

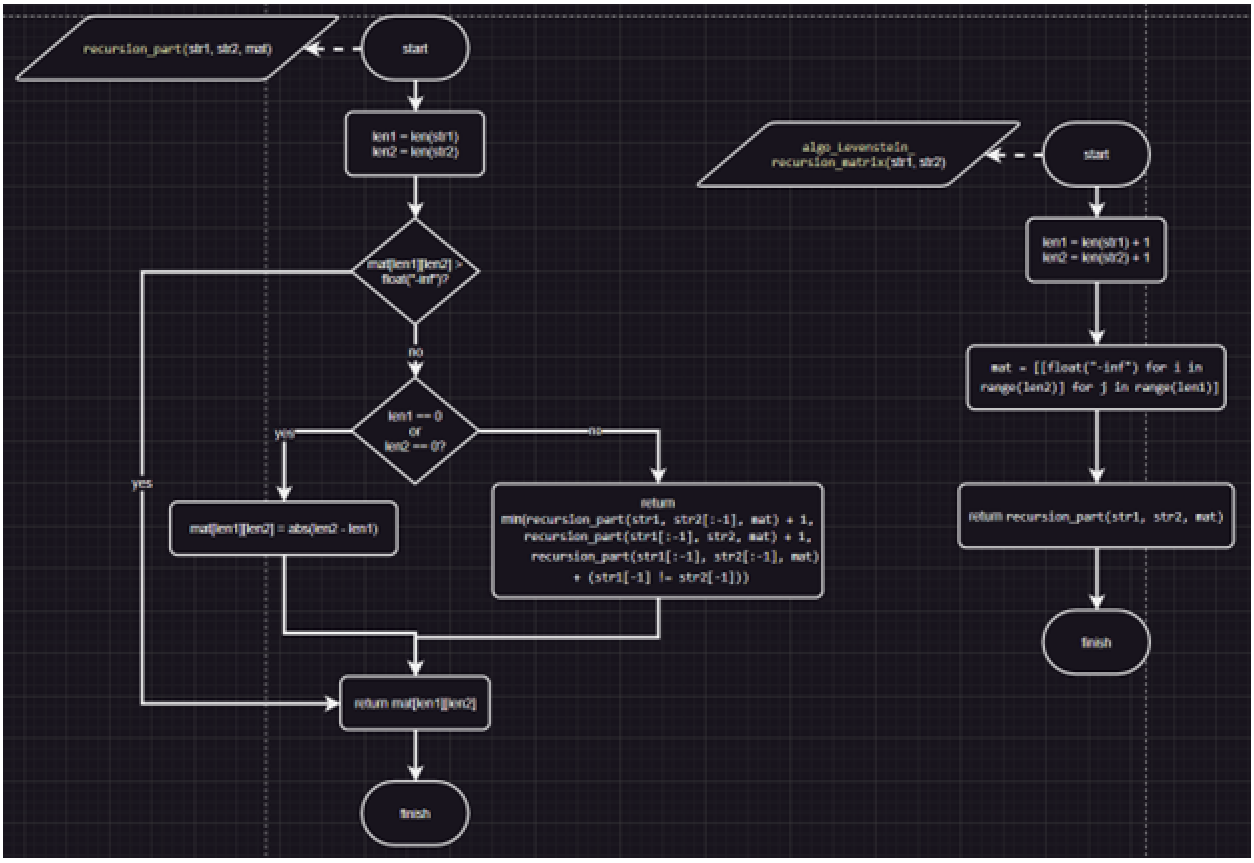

Рис. 5: Блоксхема алгоритма Левенштейна (рекурсивно-матричная реализация)

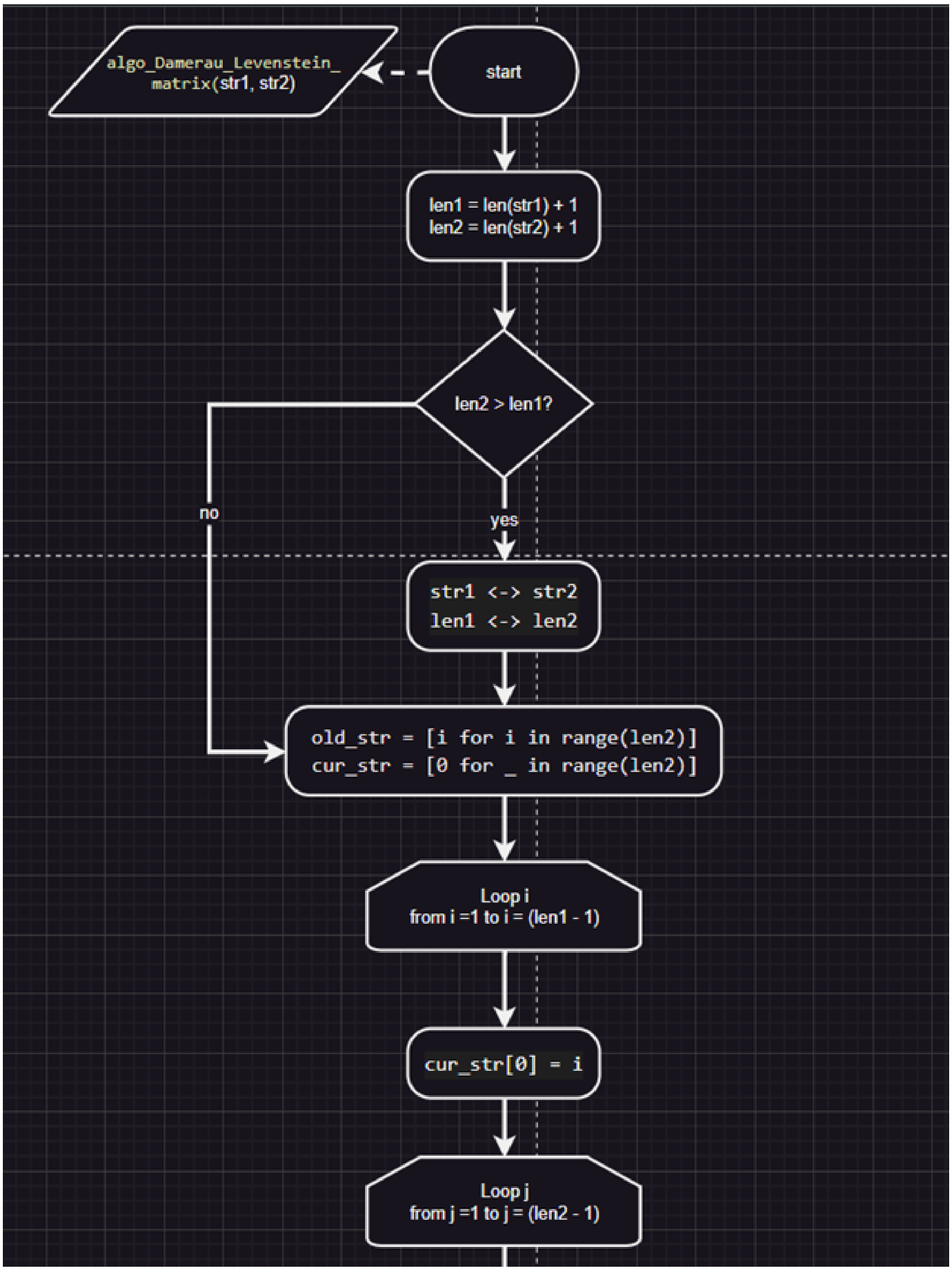

Рис. 6: Блоксхема алгоритма Дамерау-Левенштейна (матричная реализация)

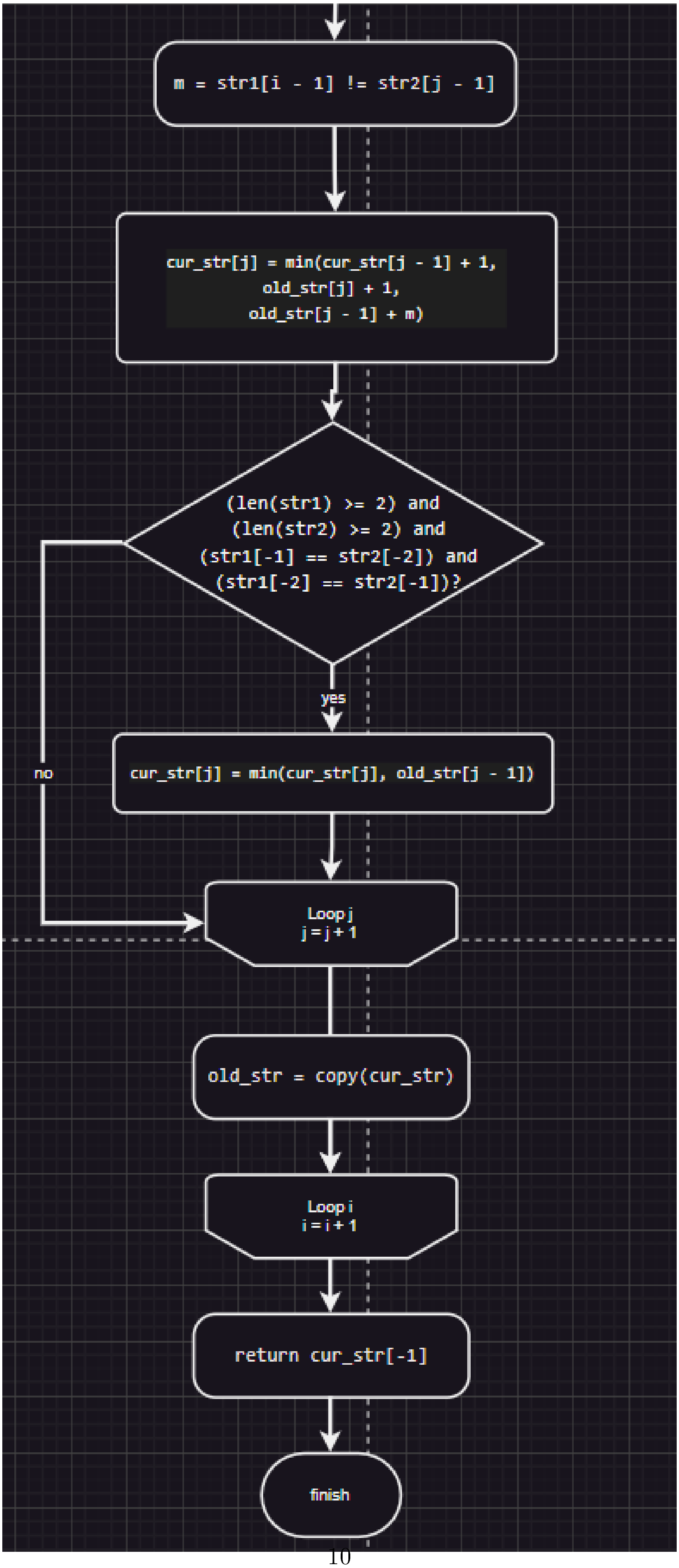

Рис. 7: Блоксхема алгоритма Дамерау-Левенштейна (матричная реализация) (продол-

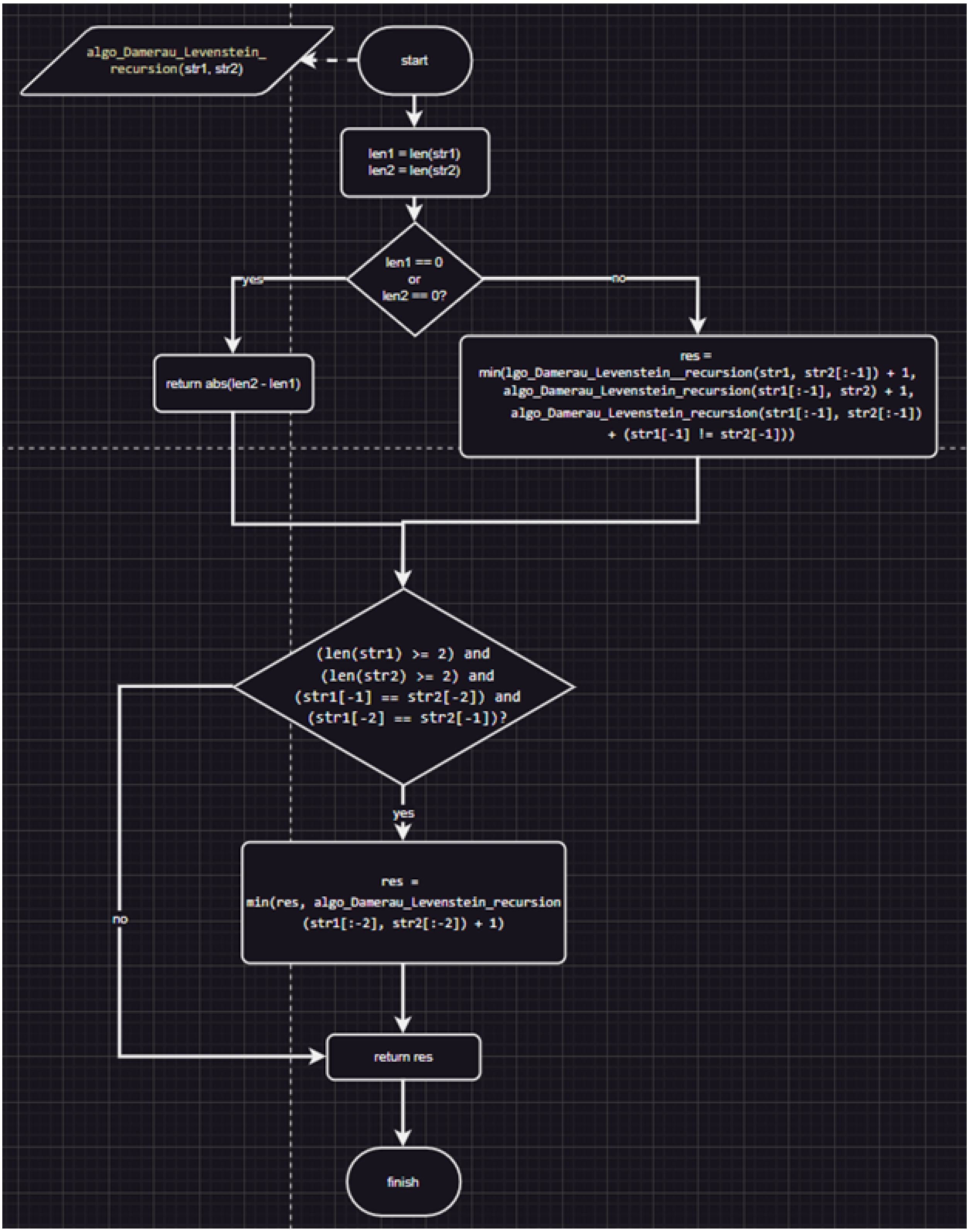

Рис. 8: Блоксхема алгоритма Дамерау-Левенштейна (рекурсивная реализация)

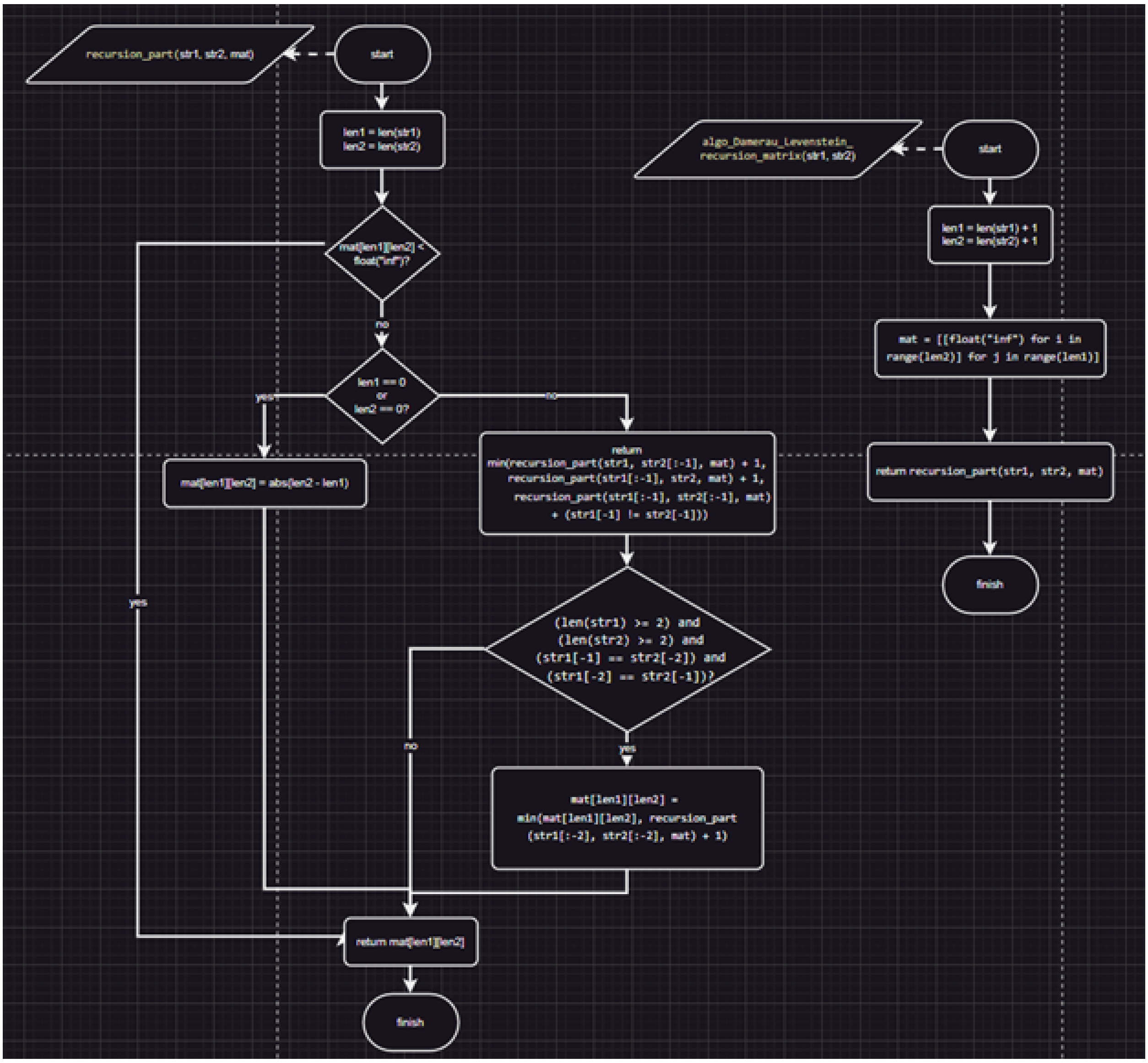

Рис. 9: Блоксхема алгоритма Дамерау-Левенштейна (рекурсивно-матричная реализация)

3. Технологическая часть

3.1. 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.

3.2. 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].

3.3. Реализации алгоритмов

```
def algo_Levenstein_matrix(str1: str, str2: str) -> int:
    len1, len2 = len(str1) + 1, len(str2) + 1
    if len2 > len1:
        str1, str2 = str2, str1
        len1, len2 = len2, len1
    old_str = [i for i in range(len2)]
    cur_str = [0 for _ in range(len2)]
    for i in range(1, len1):
        cur_str[0] = i
        for j in range(1, len2):
            cur_str[j] = min(cur_str[j - 1] + 1,
                             old_str[j] + 1,
                             old_str[j - 1] + (str1[i - 1] != str2[j - 1]))
        old_str = cur_str.copy()
    return cur_str[-1]
```

Листинг 1: реализация матричного алгоритма Левенштейна

```
def algo_Levenstein_recursion(str1: str, str2: str) -> int:
    len1, len2 = len(str1), len(str2)
    if len1 * len2 == 0:
        return abs(len2 - len1)
    return min(algo_Levenstein_recursion(str1, str2[:-1]) + 1,
               algo_Levenstein_recursion(str1[:-1], str2) + 1,
               algo_Levenstein_recursion(str1[:-1], str2[:-1]) + (str1
                   [-1] != str2[-1]))
```

Листинг 2: реализация рекурсивного алгоритма Левенштейна

```
def algo_Levenstein_recursion_matrix(str1: str, str2: str) -> int:
    len1, len2 = len(str1) + 1, len(str2) + 1
    mat = [[float("-inf") for i in range(len2)] for j in range(len1)]
    # the recursive part itself
    def recursion_part(str1: str, str2: str, mat: List[float] = []) -> int
        :
```

```
        len1, len2 = len(str1), len(str2)
        if mat[len1][len2] > float("-inf"):
            pass
        elif len1 * len2 == 0:
            mat[len1][len2] = abs(len2 - len1)
        else:
            mat[len1][len2] = min(recursion_part(str1, str2[:-1], mat) +
                1,
                    recursion_part(str1[:-1], str2, mat) + 1,
                    recursion_part(str1[:-1], str2[:-1], mat) + (str1[-1]
                        != str2[-1]))
        return mat[len1][len2]
    return recursion_part(str1, str2, mat)
```

Листинг 3: реализация рекурсивно-матричного алгоритма Левенштейна

```
def algo_Damerau_Levenstein_matrix(str1: str, str2: str) -> int:
    len1, len2 = len(str1) + 1, len(str2) + 1
    if len2 > len1:
        str1, str2 = str2, str1
        len1, len2 = len2, len1
    old_str = [i for i in range(len2)]
    cur_str = [0 for _ in range(len2)]
    for i in range(1, len1):
        cur_str[0] = i
        for j in range(1, len2):
            m = str1[i - 1] != str2[j - 1]
            cur_str[j] = min(cur_str[j - 1] + 1,
                             old_str[j] + 1,
                             old_str[j - 1] + m)
            if (i > 1) and (j > 1) and m and (str1[i - 2] == str2[j - 1])
                and (str1[i - 1] == str2[j - 2]):
                 cur_str[j] = min(cur_str[j], old_str[j - 1])
        old_str = cur_str.copy()
    return cur_str[-1]
```

Листинг 4: реализация матричного алгоритма Дамерау-Левенштейна

```
def algo_Damerau_Levenstein_recursion(str1: str, str2: str) -> int:
    len1, len2 = len(str1), len(str2)
    if len1 * len2 == 0:
        return abs(len2 - len1)
    res = min(algo_Damerau_Levenstein_recursion(str1, str2[:-1]) + 1,
              algo_Damerau_Levenstein_recursion(str1[:-1], str2) + 1,
              algo_Damerau_Levenstein_recursion(str1[:-1], str2[:-1]) +
                 (str1[-1] != str2[-1]))
    if ((len(str1) >= 2) and (len(str2) >= 2) and (str1[-1] == str2[-2])
       and (str1[-2] == str2[-1])):
        res = min(res, algo_Damerau_Levenstein_recursion(str1[:-2], str2
            [:-2]) + 1)
    return res
```

Листинг 5: реализация рекурсивного алгоритма Дамерау-Левенштейна

```
def algo_Damerau_Levenstein_recursion_matrix(str1: str, str2: str) -> int:
    len1, len2 = len(str1) + 1, len(str2) + 1
    mat = [[float("inf") for i in range(len2)] for j in range(len1)]
    # the recursive part itself
    def recursion_part(str1: str, str2: str, mat: List[float] = []) -> int
        :
        len1, len2 = len(str1), len(str2)
```

```
        if mat[len1][len2] < float("inf"):
            pass
        elif len1 * len2 == 0:
            mat[len1][len2] = abs(len2 - len1)
        else:
            mat[len1][len2] = min(recursion_part(str1, str2[:-1], mat) +
                1,
                    recursion_part(str1[:-1], str2, mat) + 1,
                    recursion_part(str1[:-1], str2[:-1], mat) + (str1[-1]
                        != str2[-1]))
            if ((len(str1) >= 2) and (len(str2) >= 2) and (str1[-1] ==
                str2[-2]) and (str1[-2] == str2[-1])):
                 mat[len1][len2] = min(mat[len1][len2], recursion_part(str1
                    [:-2], str2[:-2], mat) + 1)
        return mat[len1][len2]
    return recursion_part(str1, str2, mat)
```

Листинг 6: реализация рекурсивно-матричного алгоритма Дамерау-Левенштейна

3.4. Тесты

Строка 1	Строка 2	Ожидание	матричный	рекурсивный	рекурсивно-матричный
λ	λ	0	0	0	0
a	a	0	0	0	0
abc	abc	0	0	0	0
λ	a	1	1	1	1
a	λ	1	1	1	1
a	b	1	1	1	1
abc	abs	1	1	1	1
odc	abc	2	2	2	2
ods	abc	3	3	3	3
abcs	abc	1	1	1	1
bc	abc	1	1	1	1
bac	abc	2	2	2	2

Таблица 1: Таблица тестов для алгоритмов Левенштейна

Строка 1	Строка 2	Ожидание	матричный	рекурсивный	рекурсивно-матричный
λ	λ	0	0	0	0
a	a	0	0	0	0
abc	abc	0	0	0	0
λ	a	1	1	1	1
a	λ	1	1	1	1
a	b	1	1	1	1
abc	abs	1	1	1	1
odc	abc	2	2	2	2
ods	abc	3	3	3	3
abcs	abc	1	1	1	1
bc	abc	1	1	1	1
bac	abc	1	1	1	1

Таблица 2: Таблица тестов для алгоритмов Дамерау-Левенштейна

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

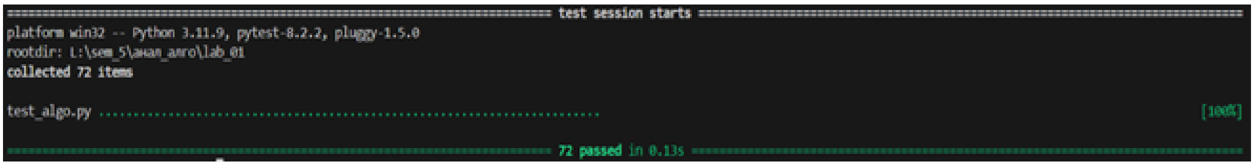

Рис. 10: Результаты тестов с использованием pytest

4. 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:

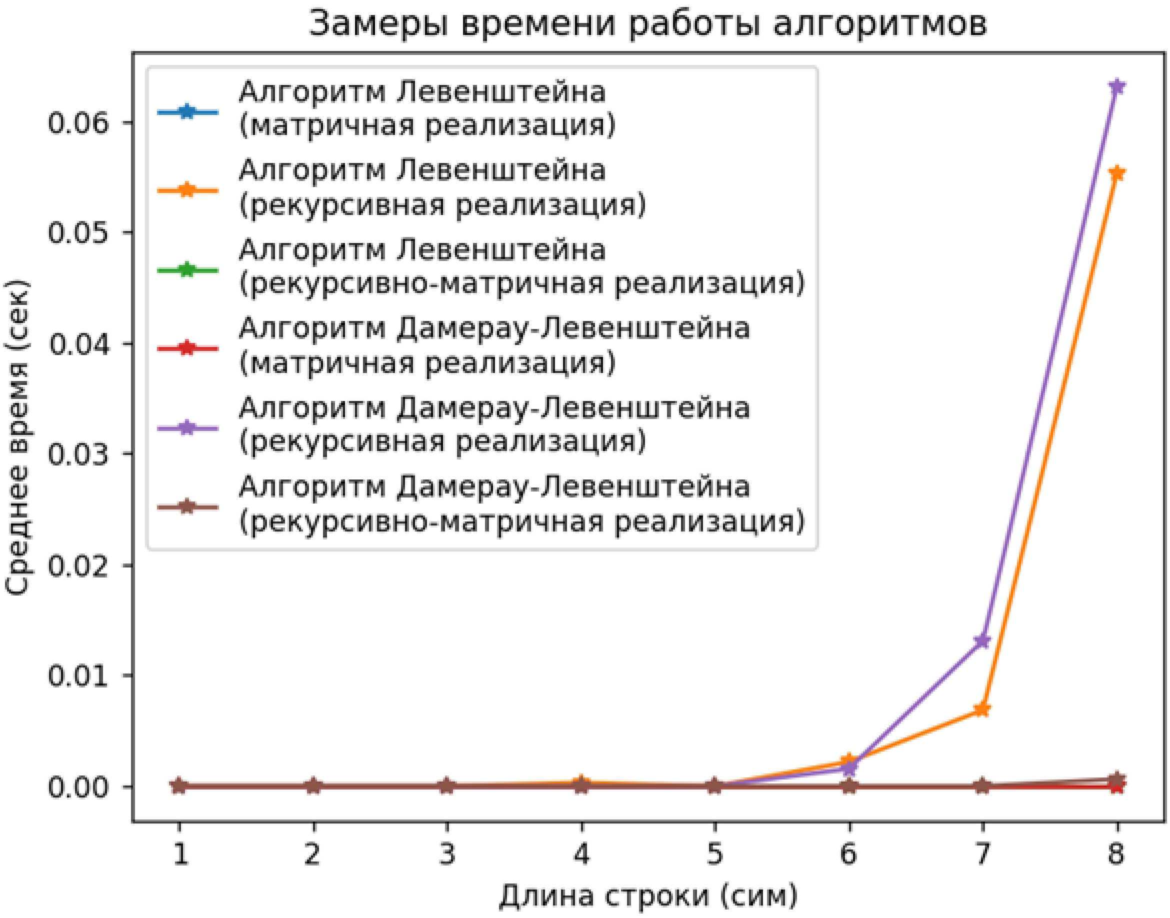

Рис. 11: График времени работы всех алгоритмов в зависимости от длин строк

алгоритм	1	2	3	4	5	6	7	8
Алгоритм Левенштейна (матричная реализация)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Алгоритм Левенштейна (рекурсивная реализация)	0.0	0.0	0.0	0.00031	0.0	0.0022	0.0069	0.055
Алгоритм Левенштейна (рекурсивно-матричная реализация)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Алгоритм Дамерау-Левенштейна (матричная реализация)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Алгоритм Дамерау-Левенштейна (рекурсивная реализация)	0.0	0.0	0.0	0.0	0.0	0.0016	0.013	0.063
Алгоритм Дамерау-Левенштейна (рекурсивно-матричная реализация)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00063

Рис. 12: Таблица времени работы всех алгоритмов в зависимости от длин строк

4.1. 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.

4.2. Сравнение работы алгоритмов Левенштейна и Дамерау-Левенштейна

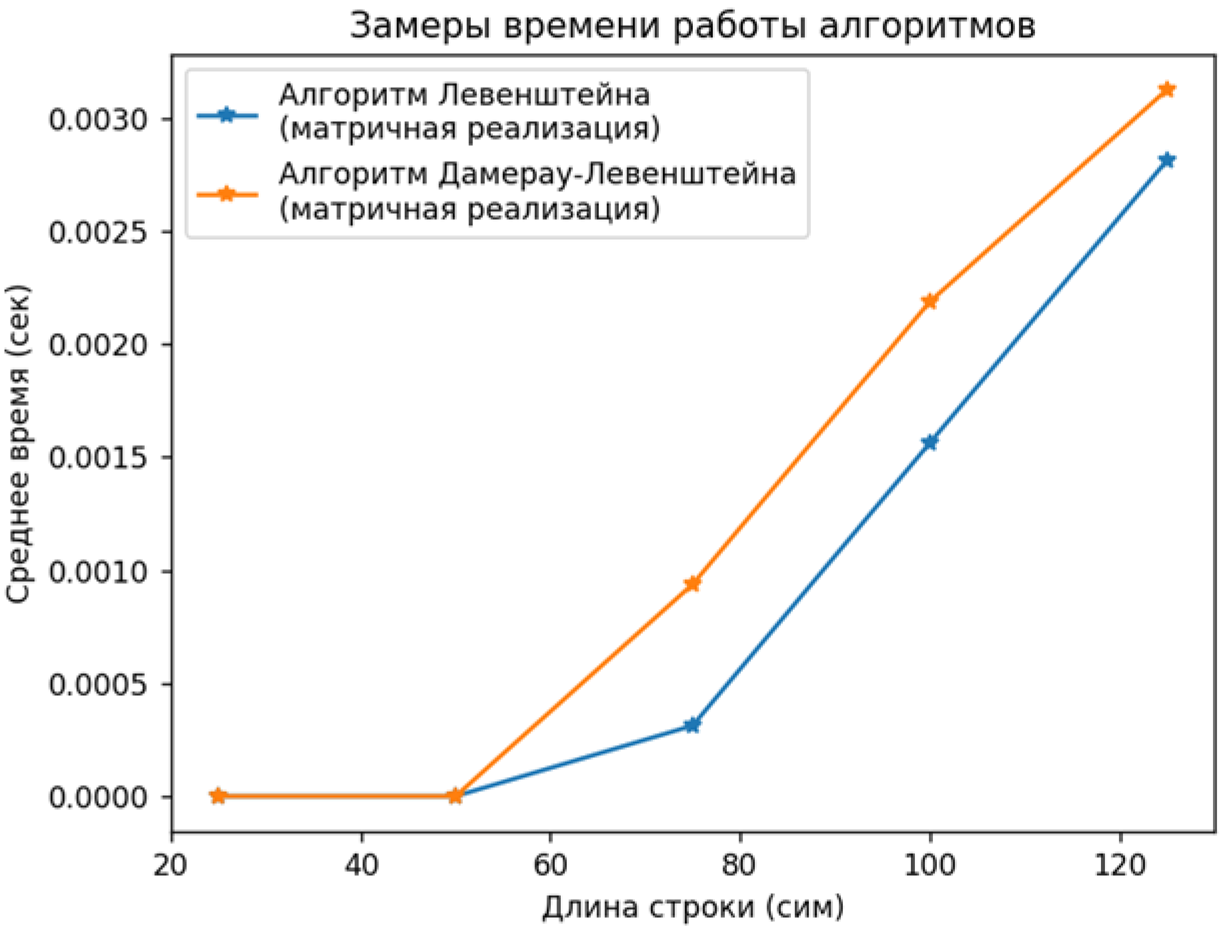

Рис. 13: График времени работы матричных реализаций алгоритмов в зависимости от длин строк

алгоритм	25	50	75	100	125
Алгоритм Левенштейна (матричная реализация)	0.0	0.0	0.00031	0.0016	0.0028
Алгоритм Дамерау-Левенштейна (матричная реализация)	0.0	0.0	0.00094	0.0022	0.0031

Рис. 14: Таблица времени работы матричных реализаций алгоритмов в зависимости от длин строк

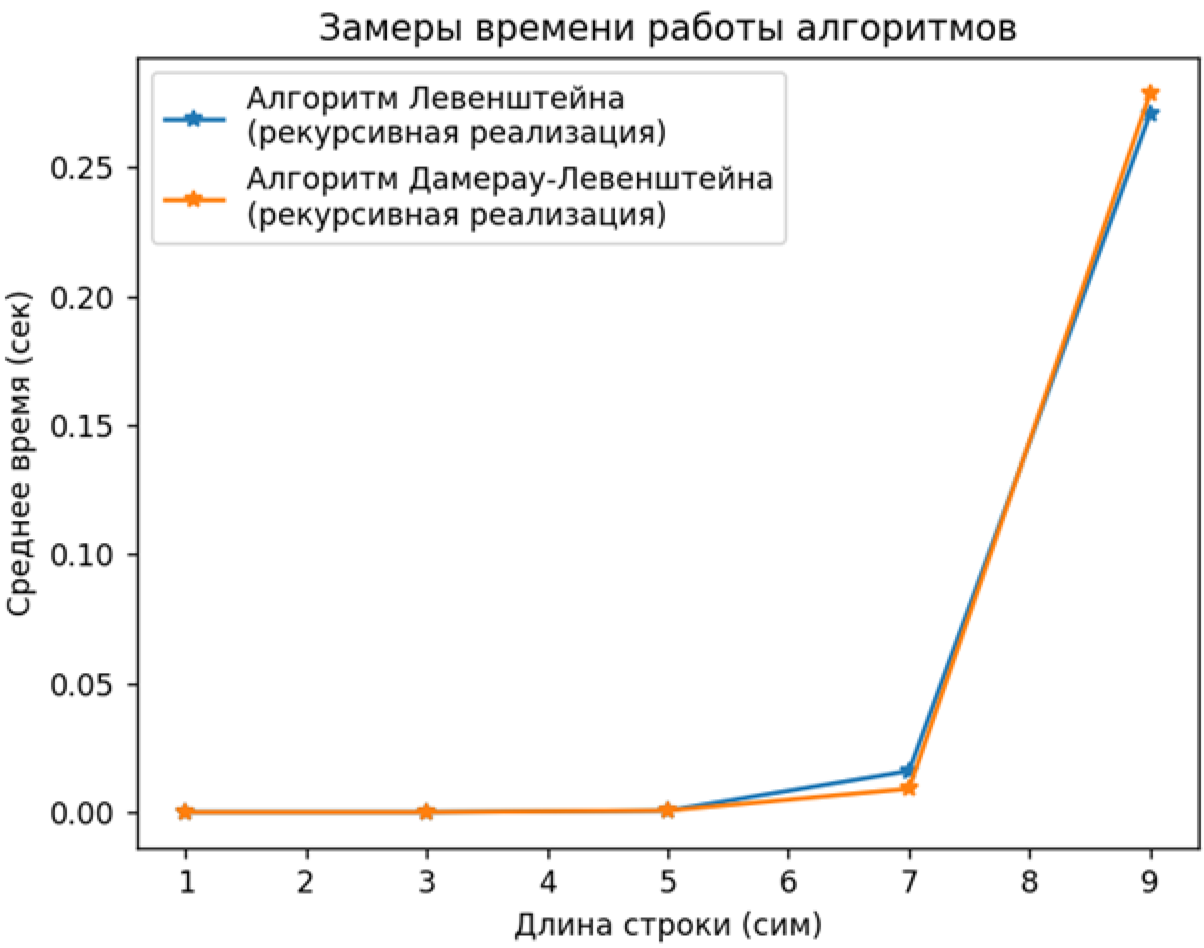

Рис. 15: График времени работы рекурсивных реализаций алгоритмов в зависимости от длин строк

алгоритм	1	3	5	7	9
Алгоритм Левенштейна (рекурсивная реализация)	0.0	0.0	0.00063	0.016	0.27
Алгоритм Дамерау-Левенштейна (рекурсивная реализация)	0.0	0.0	0.00063	0.0091	0.28

Рис. 16: Таблица времени работы рекурсивных реализаций алгоритмов в зависимости от длин строк

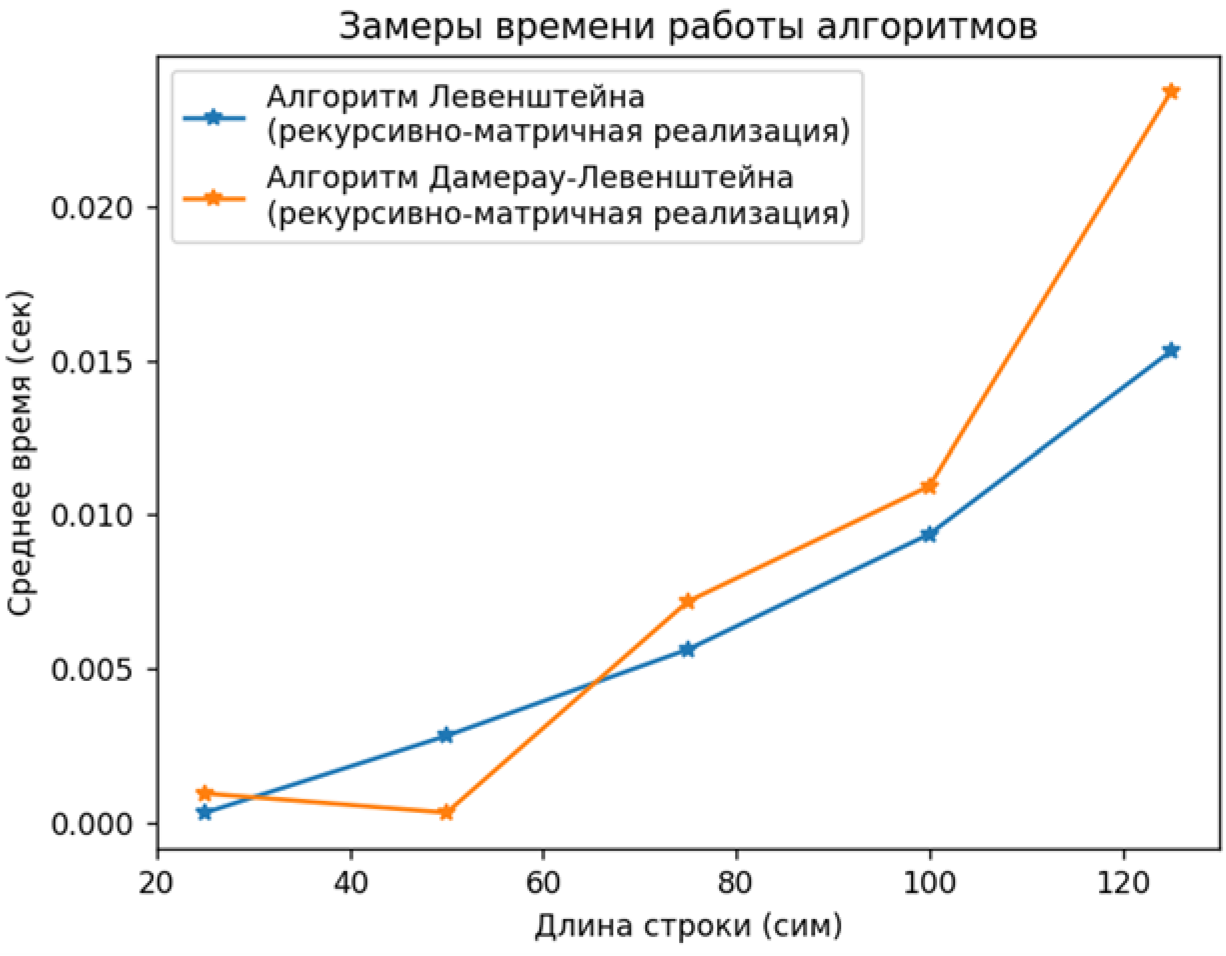

Рис. 17: График времени работы рекурсивно-матричных реализаций алгоритмов в зависимости от длин строк

алгоритм	25	50	75	100	125
Алгоритм Левенштейна (рекурсивно-матричная реализация)	0.00031	0.0028	0.0056	0.0094	0.015
Алгоритм Дамерау-Левенштейна (рекурсивно-матричная реализация)	0.00094	0.00031	0.0072	0.011	0.024

Рис. 18: Таблица времени работы рекурсивно-матричных реализаций алгоритмов в зависимости от длин строк

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.

4.3. 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).

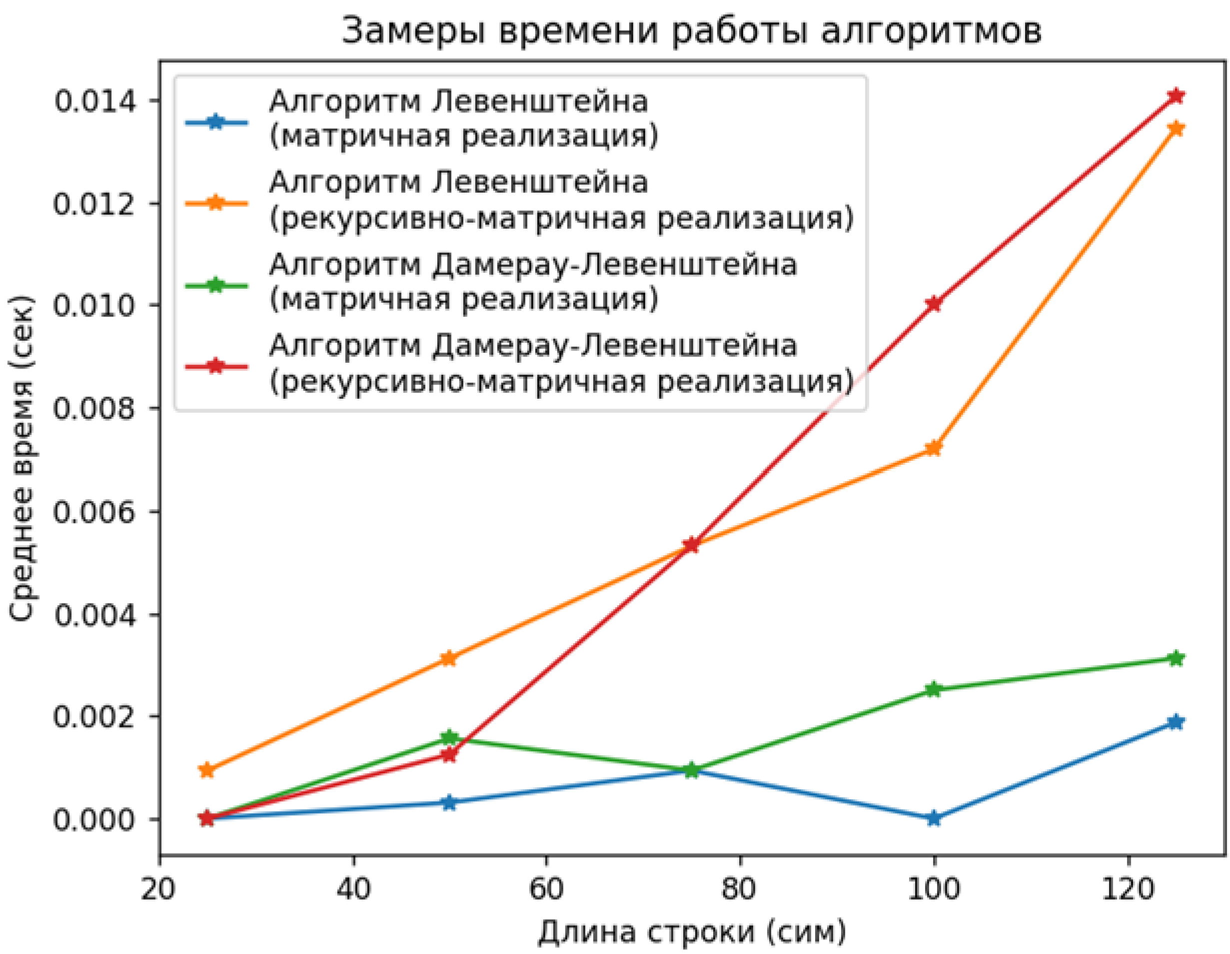

Рис. 19: График времени работы матричных и рекурсивно-матричных реализаций алгоритмов в зависимости от длин строк

алгоритм	25	50	75	100	125
Алгоритм Левенштейна (матричная реализация)	0.0	0.00031	0.00094	0.0	0.0019
Алгоритм Левенштейна (рекурсивно-матричная реализация)	0.00094	0.0031	0.0053	0.0072	0.013
Алгоритм Дамерау-Левенштейна (матричная реализация)	0.0	0.0016	0.00094	0.0025	0.0031
Алгоритм Дамерау-Левенштейна (рекурсивно-матричная реализация)	0.0	0.0013	0.0053	0.01	0.014

Рис. 20: Таблица времени работы матричных и рекурсивно-матричных реализаций алгоритмов в зависимости от длин строк

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.

Список литературы

[1] Python Documentation. time.process_time() - Документация по стандартной библиотеке Python. Дата обращения: 03 сентября 2024 г. [Электронный ресурс]. Доступно по адресу: `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. [Электронный ресурс]. Доступно по адресу: `https://tirinox.ru/levenstein-python/`

[3] 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.

[4] Ниёзов Д. Л. Применение методов нечеткого сравнения строк в прикладных задачах: Выпускная квалификационная работа (Бакалаврская работа). — Тольятти: Тольяттинский государственный университет, 2020. — 45 стр.

[5] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965. 163.4:845-848.