

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Москва, 2021

Оглавление

Введение	3
1 Аналитический раздел	5
1.1 Расстояние Левенштейна	5
1.2 Реккурентный алгоритм	6
1.3 Матрица расстояний	6
1.4 Использование двух строк	7
1.5 Рекурсивный алгоритм с кэшем в форме матрицы	7
1.6 Расстояние Дамерау-Левенштейна	7
2 Конструкторский раздел	9
2.1 Схемы алгоритмов Левенштейна	9
3 Технологический раздел	15
3.1 Требования к программному обеспечению	15
3.2 Выбор средств реализации	15
3.3 Листинги программ	15
3.4 Вспомогательные функции	17
3.5 Тестирование	18
4 Исследовательский раздел	19
4.1 Технические характеристики	19
4.2 Временные характеристики выполнения	19
4.3 Объем потребляемой памяти	21
Заключение	23
Литература	24

Введение

При наборе текста часто возникает проблема, связанная с опечатками. Необходимы средства, которые позволили бы быстро исправлять эти ошибки.

Для решения подобных задач в прикладной лингвистике выделяется такое направление, как компьютерная лингвистика, в которой разрабатываются и используются компьютерные программы, необходимые для исследования языка и моделирования функционирования языка в тех или иных условиях[1].

Одним из первых, кто занялся такой задачей, был советский ученый В.И.Левенштейн[2]. Алгоритм полученного решения связали с его именем. Расстояние Левенштейна - метрика, измеряющая разность двух строк, определяемая в количестве редакторских операций (вставка, удаление, замена), требуемых для преобразования одной последовательности символов в другую. Модификацией данного алгоритма является расстояние Дамерау-Левенштейна, которая добавляет транспозицию, обмен двух соседних символов, к редакторским операциям. Разработанные алгоритмы нашли применение не только в компьютерной лингвистике, но и в биоинформатике для определения схожести разных участков ДНК и РНК.

Целью лабораторной работы является изучение и реализация алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна, а также получения навыка динамического программирования. Для её достижения необходимо выполнить следующие задачи:

- изучение алгоритмов Левенштейна и Дамерау-Левенштейна;
- разработать данные алгоритмы;
- применение методов динамического программирования для реализации алгоритмов;
- выполнить тестирование реализации алгоритмов методом черного ящика;

- провести сравнительный анализ этих алгоритмов по затратам памяти и процессорному выполнению времени на основе экспериментальных данных;

1 Аналитический раздел

1.1 Расстояние Левенштейна

Расстояние Левенштейна (редакторское расстояние) между двумя строками - минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую.

При преобразовании одной строки в другую используются следующие операции:

- a) I (insert) - вставка;
- b) D (delete) - удаление;
- c) R (replace) - замена;

Будем считать, что стоимость каждой из этих операции (штраф) равна единице.

Введем еще одну операцию M (match) - совпадение. Её стоимость будет равна нулю.

Необходимо найти последовательность замен с минимальным суммарным штрафом.

1.2 Реккурентный алгоритм

Расстояние между двумя строками $s1$ и $s2$ рассчитывается по реккурентной формуле 1.2:

$$D(s1[1..i], s2[1..j]) = \begin{cases} 0, & i=0, j=0 \\ j, & i=0, j>0 \\ i, & i>0, j=0 \\ \min \begin{cases} D(s1[1..i], s2[1..j-1]) + 1 \\ D(s1[1..i-1], s2[1..j]) + 1 \\ D(s1[1..i-1], s2[1..j-1]) + f(s1, s2) \end{cases} & \end{cases} \quad (1.2.1)$$

Функция $f(s1, s2)$ определяется следующим образом:

$$f(s1, s2) = \begin{cases} 0, & s1=s2 \\ 1, & \text{иначе} \end{cases} \quad (1.2.2)$$

1.3 Матрица расстояний

Реализация алгоритма по формуле (1.2.1) при больших значениях i, j , оказывается менее эффективной по времени ввиду того, что приходится вычислять промежуточные результаты неоднократно. Для оптимизации нахождения расстояния Левенштейна необходимо использовать матрицу стоимостей для хранения этих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы значениями $D(i, j)$

1.4 Использование двух строк

Модификацией использования матрицы является использование только двух строк этой матрицы, в которых хранятся промежуточные значения. После выполнения вычислений выполняется обмен значений этих двух строк. Алгоритм продолжает работать и перезаписывать значения только второй строки.

1.5 Рекурсивный алгоритм с кэшем в форме матрицы

При помощи матрицы можно выполнить оптимизацию рекурсивного алгоритма заполнения. Основная идея такого подхода заключается в том, что при каждом рекурсивном вызове алгоритма выполняется заполнение матрицы стоимостей. Главное отличие данного метода от того, что был описан в разделе (1.3) - начальная инициализация матрицы значением ∞ . Если рекурсивный алгоритм выполняет вычисления для данных, которые не были обработаны, значение результата минимального расстояния для данного вызова заносится в матрицу. Если рекурсивный вызов уже обрабатывался (ячейка матрицы была заполнена), то алгоритм не выполняет вычислений, а сразу переходит к следующему шагу.

1.6 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна является модификацией расстояния Левенштейна, которая задействует еще одну редакторскую операцию - транспозицию T (transposition). Она выполняет обмен соседних символов в слове.

Дамерау показал, что 80% человеческих ошибок при наборе текстов является перестановка соседних символов, пропуск символа, добавление нового символа или ошибочный символ[3]. Таким образом, расстояние Дамерау-Левенштейна часто используется в редакторских программах

для проверки правописания. Это расстояние может быть вычислено по следующей рекуррентной формуле:

$$D(s1[1..i], s2[1..j]) = \begin{cases} 0, & i=0, j=0 \\ j, & i=0, j>0 \\ i, & i>0, j=0 \\ \min \begin{cases} D(s1[1..i], s2[1..j-1]) + 1 \\ D(s1[1..i-1], s2[1..j]) + 1 \\ D(s1[1..i-1], s2[1..j-1]) + f(s1, s2) \\ D(s1[1..i-1], s2[1..j-1]) + 1, \\ i, j > 1, a_i = b_{j-1}, a_{i-1} = b_j \\ \infty, \text{ иначе} \end{cases} \end{cases} \quad (1.6.1)$$

Вывод

В данном разделе были рассмотрены основные способы нахождения редакторского расстояния между двумя строками. Формулы для нахождения расстояния Левенштейна и Дамерау-Левенштейна задаются рекуррентно, следовательно, алгоритмы могут быть реализованы как рекурсивно, так и итерационно.

2 Конструкторский раздел

2.1 Схемы алгоритмов Левенштейна

Ниже представлены следующие схемы алгоритмов:

- рис. 2.1 - схема алгоритма итеративного Левенштейна с использованием двух строк;
- рис. 2.2 - схема алгоритма рекурсивного Левенштейна без кэша;
- рис. 2.3 и рис. 2.4 - схема алгоритма рекурсивного Левенштейна с использованием матрицы;
- рис. 2.5 - схема алгоритма рекурсивного Дамерау-Левенштейна.

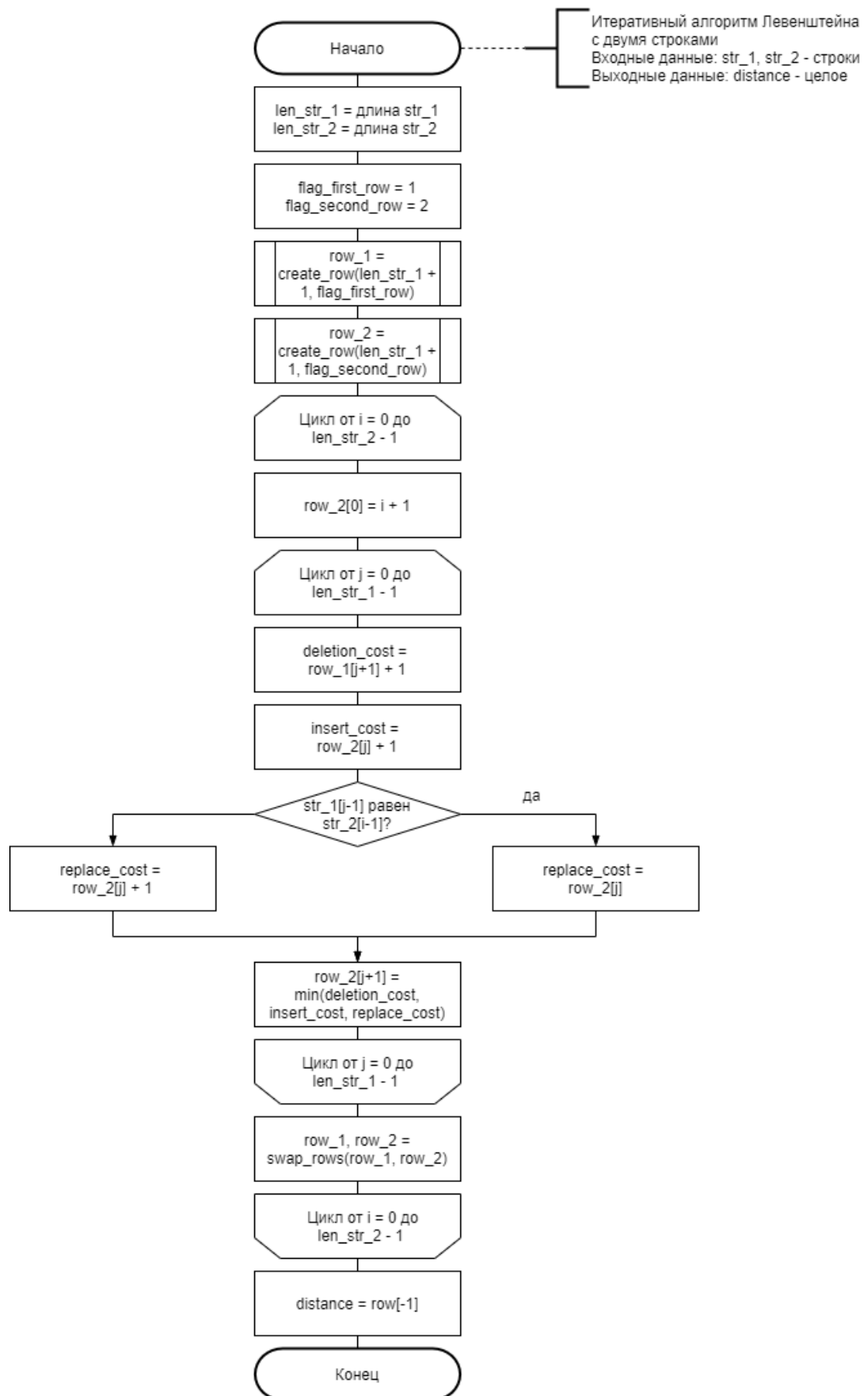


Рисунок 2.1 – Ссхема алгоритма итеративного Левенштейна с использо-
ванием двух строк.

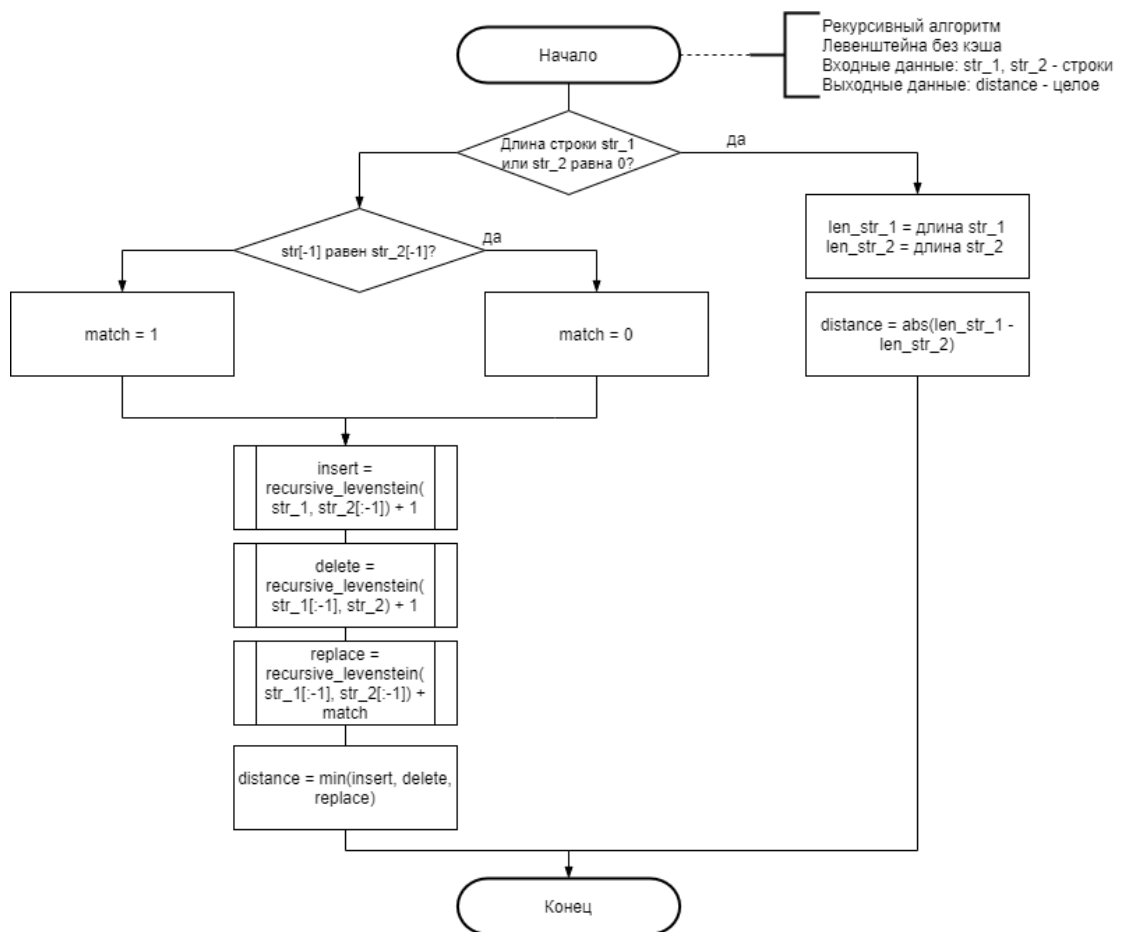


Рисунок 2.2 – Схема алгоритма рекурсивного Левенштейна без кэша.

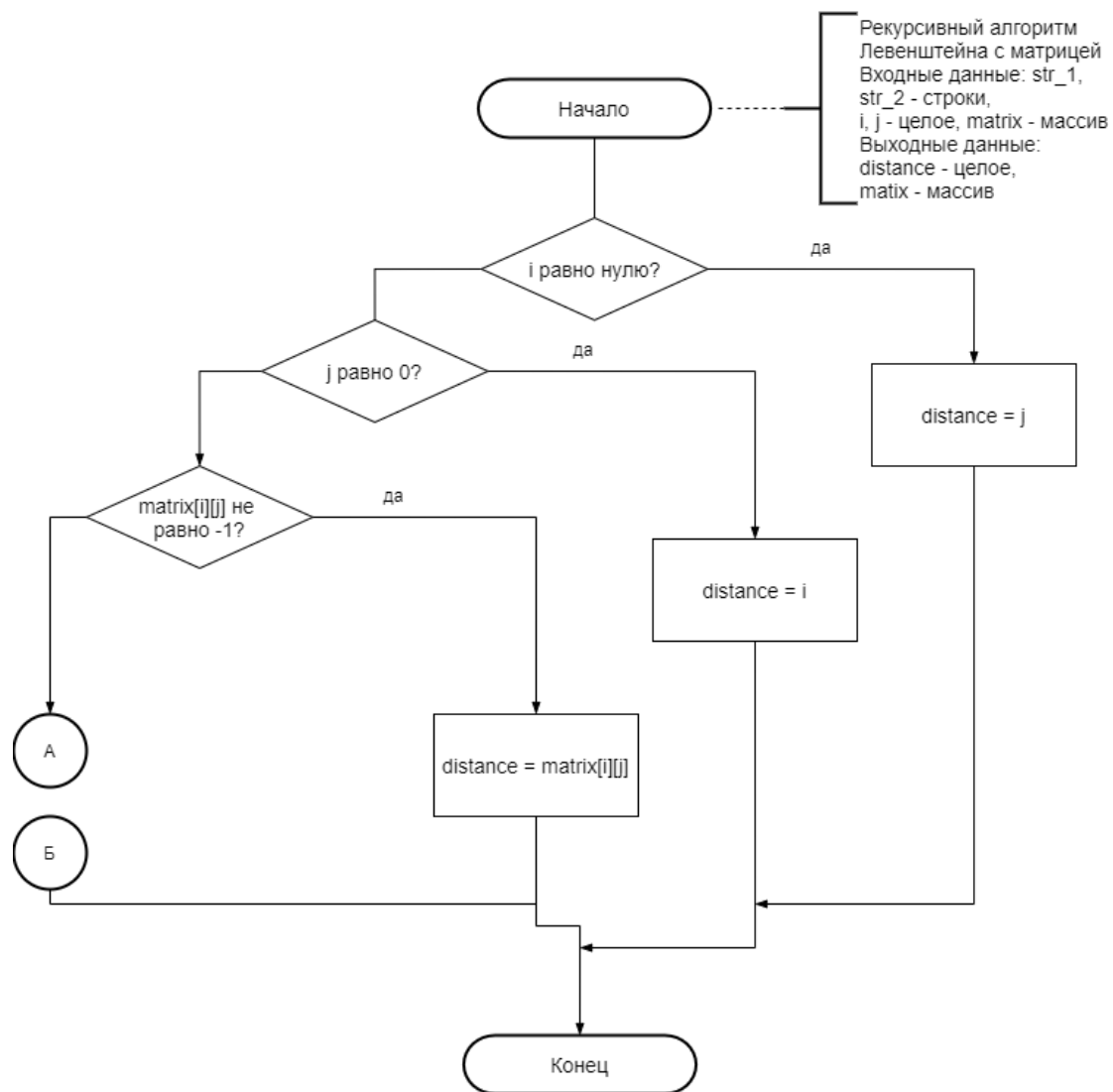


Рисунок 2.3 – Схема алгоритма рекурсивного Левенштейна с использованием матрицы.

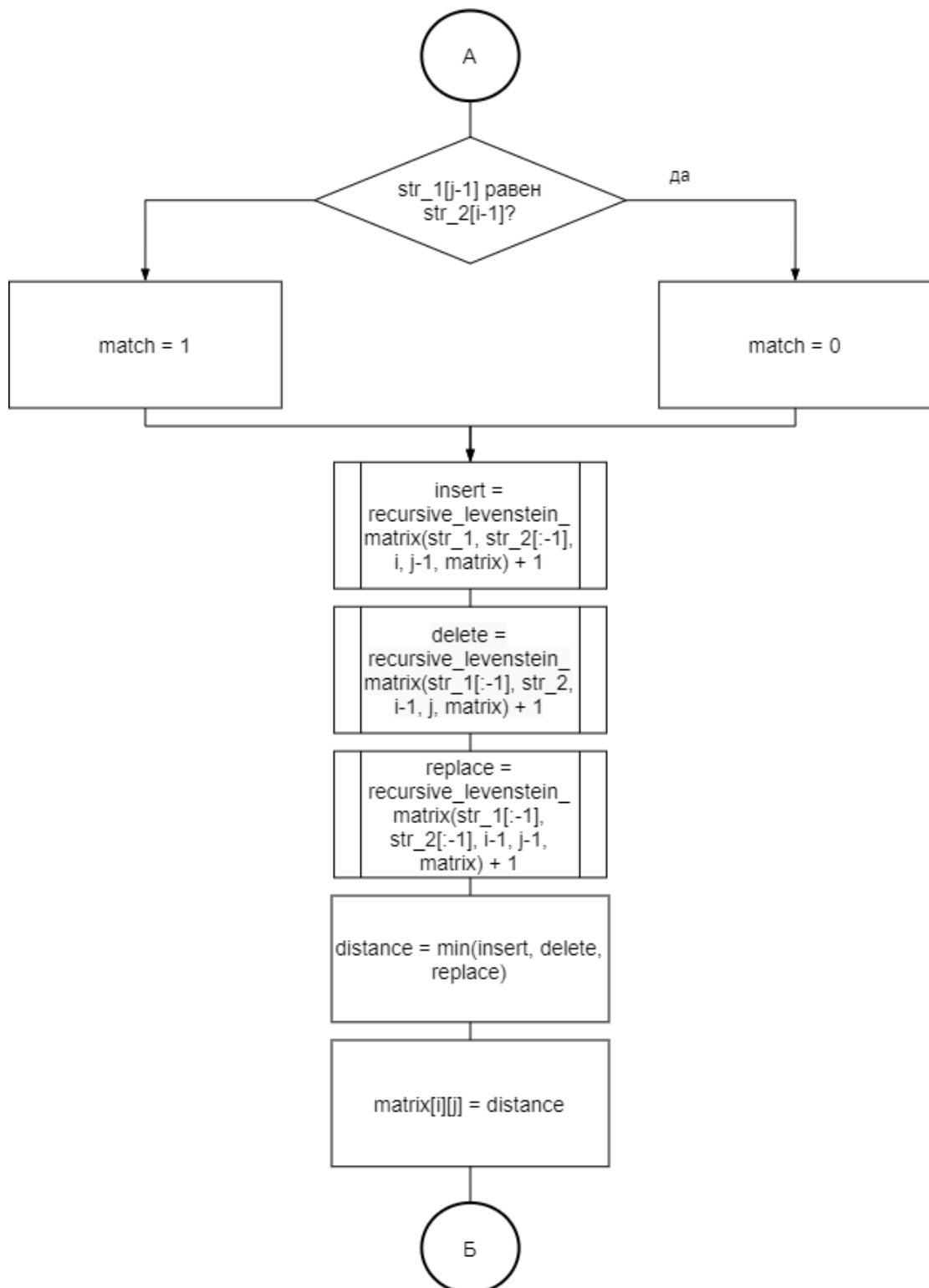


Рисунок 2.4 – Схема алгоритма рекурсивного Левенштейна с использованием матрицы.

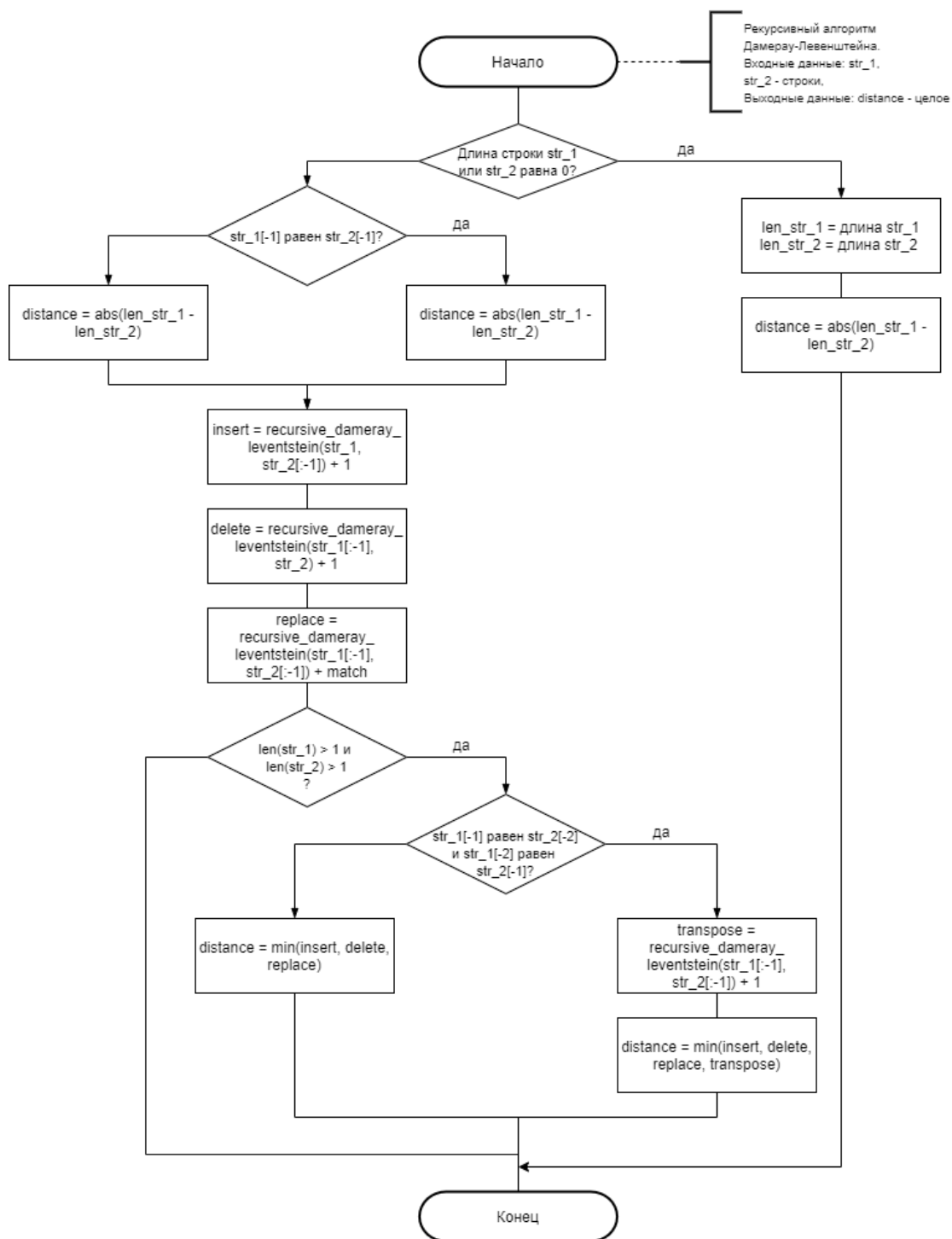


Рисунок 2.5 – Схема алгоритма рекурсивного Дамерау-Левенштейна.

Вывод

На основе теоретических данных, полученных в аналитическом разделе, были построены схемы нужных алгоритмов.

3 Технологический раздел

3.1 Требования к программному обеспечению

Программа должна отвечать следующим требованиям:

- на вход программе подаются две строки на русском или английском языке в любом регистре;
- осуществляется выбор алгоритма нахождения расстояния из меню;
- на выходе программа выдает результат - найденное расстояние между двумя строками выбранным пользователем алгоритмом.

3.2 Выбор средств реализации

Для реализации алгоритмов в данной лабораторной работе был выбран язык программирования Python 3.9.7[4]. Он является кроссплатформенным. Имеется опыт разработки на этом языке. В качестве среды разработки был использован Visual Studio Code[5], так как в нем можно работать как на операционной системе Windows, так и на дистрибутивах Linux. При замере процессорного времени был использован модуль time[6]. Замеры используемой памяти и число вызовов рекурсии проводились при помощи модуля cProfiler[7].

3.3 Листинги программ

Ниже представлены листинги разработанных алгоритмов Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Программный код нахождения расстояния Левенштейна итеративно с использованием двух строк

```
1 def iterative_levenstein_two_rows(str_1: str, str_2: str) -> int:  
2     len_str_1 = len(str_1); len_str_2 = len(str_2)
```

```

3     flag_first_row = 1; flag_second_row = 2
4
5     row_1 = create_row(len_str_1 + 1, flag_first_row)
6     row_2 = create_row(len_str_1 + 1, flag_second_row)
7
8     for i in range(len_str_2):
9         row_2[0] = i + 1
10        for j in range(len_str_1):
11            deletion_cost = row_1[j + 1] + 1
12            insert_cost = row_2[j] + 1
13            replace_cost = row_1[j] if str_1[j - 1] == str_2[i - 1] \
14                           else row_1[j] + 1
15
16            row_2[j + 1] = min(deletion_cost, insert_cost, replace_cost)
17        row_1, row_2 = swap_rows(row_1, row_2)
18    distance = row_1[-1]
19    return distance

```

Листинг 3.2 – Программный код нахождения расстояния Левенштейна рекурсивно без использования кэша

```

1 def recursive_levenstein(str_1: str, str_2: str) -> int:
2     if str_1 == '' or str_2 == '':
3         return abs(len(str_1) - len(str_2))
4     match = 0 if str_1[-1] == str_2[-1] else 1
5     distance = min(recursive_levenstein(str_1, str_2[:-1]) + 1,
6                   recursive_levenstein(str_1[:-1], str_2) + 1,
7                   recursive_levenstein(str_1[:-1], str_2[:-1]) + match)
8     return distance

```

Листинг 3.3 – Программный код нахождения расстояния Левенштейна рекурсивно с использованием матрицы

```

1     def recursive_levenstein_matrix(str_1: str, str_2: str, i: int,
2                                     j: int, matrix: list) -> Tuple[int, list[list[int]]]:
3         if i == 0:
4             return j, matrix
5         if j == 0:
6             return i, matrix
7         if matrix[i][j] != -1:
8             return matrix[i][j], matrix
9         match = 0 if str_1[-1] == str_2[-1] else 1
10
11        insert, matrix = recursive_levenstein_matrix(str_1, str_2[:-1], i,
12                                                       j-1, matrix)
13        delete, matrix = recursive_levenstein_matrix(str_1[:-1], str_2, i-1,
14                                                       j, matrix)
15        replace, matrix = recursive_levenstein_matrix(str_1[:-1], str_2[:-1],

```



```

16                                     -1, j-1, matrix)
17     insert += 1; delete += 1; replace += match
18
19     distance = min(insert, delete, replace)
20     matrix[i][j] = distance
21     return distance, matrix

```

Листинг 3.4 – Программный код нахождения расстояния Дameraу-Левенштейна рекурсивно

```

1 def recursive_damery_levenstein(str_1: str, str_2: str) -> int:
2     if str_1 == '' or str_2 == '':
3         return abs(len(str_1) - len(str_2))
4     match = 0 if str_1[-1] == str_2[-1] else 1
5     insert = recursive_damery_levenstein(str_1, str_2[:-1]) + 1
6     delete = recursive_damery_levenstein(str_1[:-1], str_2) + 1
7     replace = recursive_damery_levenstein(str_1[:-1], str_2[:-1]) + \
8                 match
9
10    if len(str_1) > 1 and len(str_2) > 1 and str_1[-1] == str_2[-2] \
11        and str_2[-1] == str_1[-2]:
12        distance = min(insert, delete, replace,
13                        recursive_damery_levenstein(str_1[:-2], str_2[:-2]) + 1)
14    else:
15        distance = min(insert, delete, replace)
16    return distance

```

3.4 Вспомогательные функции

На листингах представлены программные модули, которые используются в данных функциях:

Листинг 3.5 – Программный код создания для кэша в виде строки

```

1 def create_row(len_row: int, flag_row: int) -> list[int]:
2     row = list()
3     if flag_row == 1:
4         for i in range(len_row):
5             row.append(i)
6     else:
7         for i in range(len_row):
8             row.append(0)
9     return row

```

Листинг 3.6 – Программный код обмена двух строк

```

1 def swap_rows(row_1: list[int], row_2: list[int]) \
2     -> Tuple[list[int], list[int]]:
3     temp_row = list()
4     temp_row = deepcopy(row_1)
5     row_1 = deepcopy(row_2)
6     row_2 = deepcopy(temp_row)
7     return row_1, row_2

```

3.5 Тестирование

Для тестирования используется метод черного ящика. В данном разделе приведена таблица 3.1, в которой указаны классы эквивалентностей тестов:

Таблица 3.1 – Таблица тестов

№	Описание теста	Слово 1	Слово 2	Алгоритм	
				Левенштейн	Дамерау-Левенштейн
1	Пустые строки	”	”	0	0
2	Нет повторяющихся символов	deepcopy	раздел	8	8
3	Инверсия строк	insert	tresni	6	6
4	Два соседних символа	heart	heatr	2	1
5	Одинаковые строки	таблица	таблица	0	0
6	Одна строка меньше другой	город	горо	1	1

Вывод

В данном разделе был выбран язык программирования, среда разработки. Реализованы функции, описанные в аналитическом разделе, и проведено их тестирование методом черного ящика по таблице 3.1.

4 Исследовательский раздел

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система: Windows 10 Pro;
- память: 8 GiB;
- процессор: Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz.

Тестирование проводилось на ноутбуке, который был подключен к сети питания. Во время проведения тестирования ноутбук был нагружен только встроенными приложениями окружения, самим окружением и системой тестирования.

4.2 Временные характеристики выполнения

Ниже был проведен анализ времени работы алгоритмов. Исходными данными будут случайно сгенерированные строки длиной $\{3, 4, 5, 6, 7, 8\}$. Единичные замеры выдадут крайне маленький результат, поэтому проведем работу каждого алгоритма $n = 1000$ раз и поделим на число n . Получим среднее значение работы каждого из алгоритмов. Результат приведен на рис 4.1:

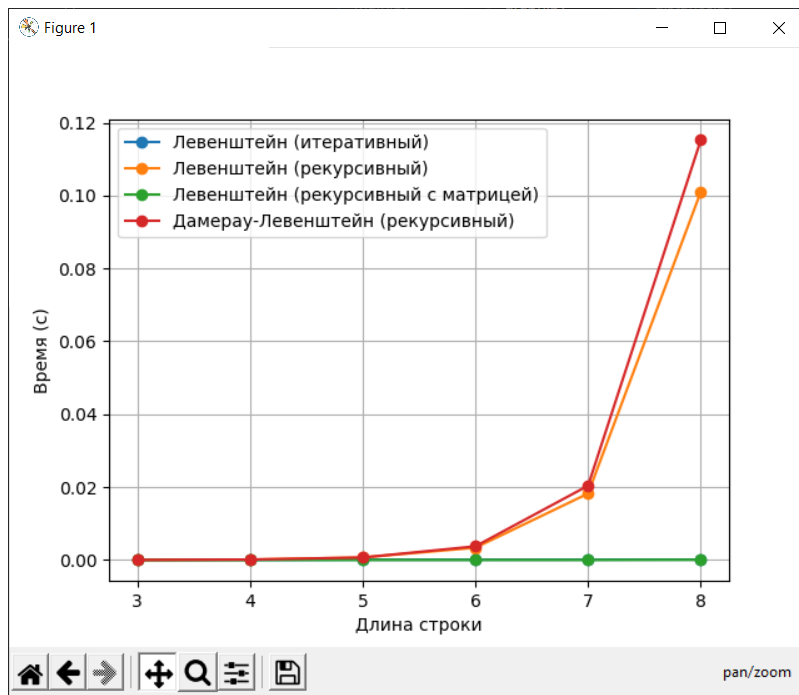


Рисунок 4.1 – Сравнение времени работы алгоритмов Левенштейна.

Как видно из результатов, рекурсивный алгоритм Левенштейна без кэша и алгоритм Дамерау-Левенштейна имеют большой рост, начиная уже со строки длиной 7. Последний имеет наибольший рост. Это объясняется тем, что этот алгоритм задействует дополнительную операцию - транспозицию, которая тоже приводит к вызову рекурсии.

Выполнив анализ двух остальных алгоритмов на значения входных строк длиной {25, 50, 75, 100, 125, 150}, получим следующий результат, представленный на рис 4.2:

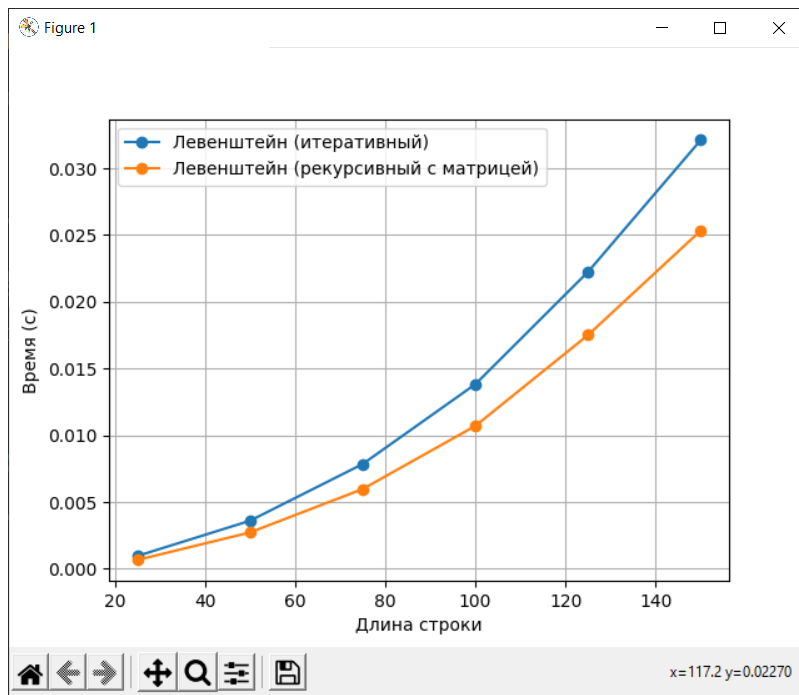


Рисунок 4.2 – Сравнение времени работы рекурсивного и некурсивного алгоритмов Левенштейна.

Рекурсивный алгоритм Левенштейна с использованием матрицы выполняется быстрее, чем итеративный с использованием двух строк. Это объясняется тем, что в итеративном случае выполняется дополнительная операция по обмену значений двух строк. На это требуется дополнительное время.

4.3 Объем потребляемой памяти

При исходных строках, длиной 3, требуется 52,8 Мб памяти. Результаты вызовов и объем потребляемой памяти приведены в таблице 4.1:

Таблица 4.1 – Число вызовов каждого алгоритма

Левенштейн			Дамерау-Левенштейн
Итеративный с двумя строками	Рекурсивный без кэша	Рекурсивный с матрицей	Рекурсивный
1	94	28	94

Общее значение потребляемой памяти складывается по формуле 4.3:

$$S = n_calls * V \quad (4.3.1)$$

где:

- n_calls - число вызовов функций;
- V - объем памяти, занимаемый одним вызовом функции.

По результатам исследования памяти алгоритмы Левенштейна и Дамерау-Левенштейна потребляют наибольшую память при выполнении по сравнению с другими. Итеративный алгоритм Левенштейна с двумя строками занимает меньше памяти в сравнении с остальными.

Вывод

Рекурсивный вызов Левенштейна без кэша и Дамерау-Левенштейна проигрывают как по скорости, так и по памяти итеративному. Причем рекурсивный алгоритм Левенштейна с матрицей работает быстрее, чем итеративный с двумя строками, но также проигрывает ему по памяти.

Сравнивая между собой рекурсивные вызовы алгоритмов Левенштейна и Дамерау-Левенштейна, можно сделать вывод о том, что рекуррентный алгоритм поиска расстояния Левенштейна с матрицей выигрывает как по времени, так и по памяти у других реализаций этих алгоритмов, а рекуррентный Дамерау-Левенштейн проигрывает им по обоим параметрам. Однако, стоит отметить, что в системах автоматического исправления текста, где чаще всего встречаются ошибки, связанные с транспозицией двух символов, выполнение исправления ошибок выбором алгоритма Дамерау-Левенштейна будет наиболее оптимальным решением.

Заключение

В ходе выполнения лабораторной работы были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна. Были выполнены описание каждого из этих алгоритмов, приведены соответствующие математические расчёты. Были получены навыки динамического программирования, а также реализованы данные алгоритмы. При тестировании каждого из них и анализе временных характеристик и объема потребляемой памяти можно сделать следующие выводы: выбор алгоритма Дамерау-Левенштейна является наиболее оптимальным решением ввиду того, что чаще всего необходимо исправлять ошибки, связанные с обменом двух соседних символов. В ином случае этот алгоритм является проигрышным как по времени, так и по памяти в сравнении с различными реализациями алгоритма Левенштейна. Самым быстрым среди данной группы алгоритмов по времени является рекурсивный алгоритм Левенштейна с кэшем в виде матрицы; но он достаточно много использует памяти за счет большого числа вызовов. Поэтому в иных ситуациях, не связанных с транспозицией, следует использовать итеративный алгоритм.

Литература

- [1] КОМПЬЮТЕРНАЯ ЛИНГВИСТИКА Большая российская энциклопедия - электронная версия [Электронный ресурс]. Режим доступа: <https://bigenc.ru/linguistics/text/2087783> (дата обращения: 20.09.2021).
- [2] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. – М.: Доклады АН СССР, 1965. Т. 163. С. 845–848.
- [3] Fred Damerau J. A technique for computer detection and correction of spelling errors. 1964. Т. 7. С. 171—176.
- [4] Python. [Электронный ресурс]. Режим доступа: <https://docs.python.org/3/> (дата обращения: 21.09.2021).
- [5] Documentation for Visual Studio Code. [Электронный ресурс]. Режим доступа: <https://code.visualstudio.com/docs> (дата обращения: 21.09.2021).
- [6] time — Time access and conversions — Python 3.9.7 documentation. [Электронный ресурс]. Режим доступа: <https://docs.python.org/3/library/time.html> (дата обращения: 22.09.2021).
- [7] The Python Profilers — Python 3.9.7 documentation. [Электронный ресурс]. Режим доступа: <https://docs.python.org/3/library/profile.html> (дата обращения: 22.09.2021).