



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 Дamerau-Левенштейна

Студент Зайцева А.А.

Группа ИУ7-52Б

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2021 г.

# Оглавление

<b>Введение</b>	<b>2</b>
<b>1 Аналитическая часть</b>	<b>3</b>
1.1 Расстояние Левенштейна . . . . .	3
1.2 Расстояние Дамерау-Левенштейна . . . . .	4
<b>2 Конструкторская часть</b>	<b>6</b>
2.1 Алгоритмы поиска расстояния Левенштейна . . . . .	6
2.2 Алгоритм поиска расстояния Дамерау - Левенштейна . . . . .	6
<b>3 Технологическая часть</b>	<b>11</b>
3.1 Требования к ПО . . . . .	11
3.2 Выбор средств реализации . . . . .	11
3.3 Листинги кода . . . . .	11
3.4 Тестирование . . . . .	14
<b>4 Исследовательская часть</b>	<b>16</b>
4.1 Пример работы . . . . .	16
4.2 Технические характеристики . . . . .	17
4.3 Время выполнения алгоритмов . . . . .	17
4.4 Пиковая требуемая алгоритмами память . . . . .	18
<b>Заключение</b>	<b>22</b>
<b>Список использованной литературы</b>	<b>23</b>

# Введение

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

Определение расстояния Левенштейна (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 одного слова в другое.

Если текст был набран с клавиатуры, то вместо расстояния Левенштейна чаще используют расстояние Дameraу – Левенштейна, в котором добавляется еще одно возможное действие - перестановка двух соседних символов. [1]

Расстояния Левенштейна и Дameraу – Левенштейна применяются в таких сферах, как:

- компьютерная лингвистика (автозамена в поисковых запросах, текстовая редакция);
- 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 Дameraу-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 Дameraу–Левенштейна – это минимальное количество действий, необходимых для преобразования одной строки в другую. Различие между этими расстояниями - в наборе допустимых операций.

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

Данным операциям можно назначить цену (штраф). Часто используется следующий набор штрафов: для операции M он равен нулю, а для остальных (I, D, R, X) - единице.

Тогда задача нахождения расстояний Левенштейна и Дameraу–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):

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

Первые три уравнения 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  $S_1$  (первое уравнение в группе  $\min$ ), удаления символа из  $S_1$ , (второе уравнение в группе  $\min$ ), а также совпадения или замены, в зависимости от равенства рассматриваемых на данном этапе символов строк (третье уравнение 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 уравнения (1.2) в группу  $\min$ :

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

Это уравнение подразумевает перестановку соседних символов в  $S_1$ , ес-

ли 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 данное уравнение не учитывается при поиске минимума.

Замечание: если в дополнение к условиям в формуле 1.2 также верно, что  $a_i = b_j$ , то прибавлять единицу к  $D(S_1[1...i-2], S_2[1...j-2])$  нет необходимости, так как это будет означать, что оба соседних рассматриваемых символа в строках совпадают, однако эти два идущих подряд совпадения будут учтены в третьем уравнении в группе  $\min$  формулы 1.1

## 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.

[illegible]

```
str1, str2
```

<code>n = len(str1) + 1</code>
--------------------------------

$m = \text{len}(\text{str2}) + 1$
-----------------------------------

```
matrix = int(0)[n][m]
```

Цикл по  $i$  от 1 до  $n$

```
matrix[i][0]=i
```

Конец цикла по  $i$  от 1 до  $n$

Цикл по  $j$  от 1 до  $m$

```
matrix[0][j]=j
```

Конец цикла по  $j$  от 1 до  $m$

Цикл по  $i$  от 1 до  $n$

Цикл по  $j$  от 1 до  $m$

```
insertion=
matrix[i][j-1]+1
```

```
deletion=
matrix[i-1][j]+1
```

```
replacement=
matrix[i-1][j-1]
```

```
graph TD; A[ ] --> B{str1[i-1] != str2[j-1]};
```

Да

```
replacement=
replacement + 1
```

matrix[i][j]= min(insertion, deletion, replacement)
--

Конец цикла по  $j$  от 1 до  $m$

Конец цикла по  $i$  от 1 до  $n$

```
matrix[n-1][m-1]
```

Конец программы  
lowen\_matrix

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



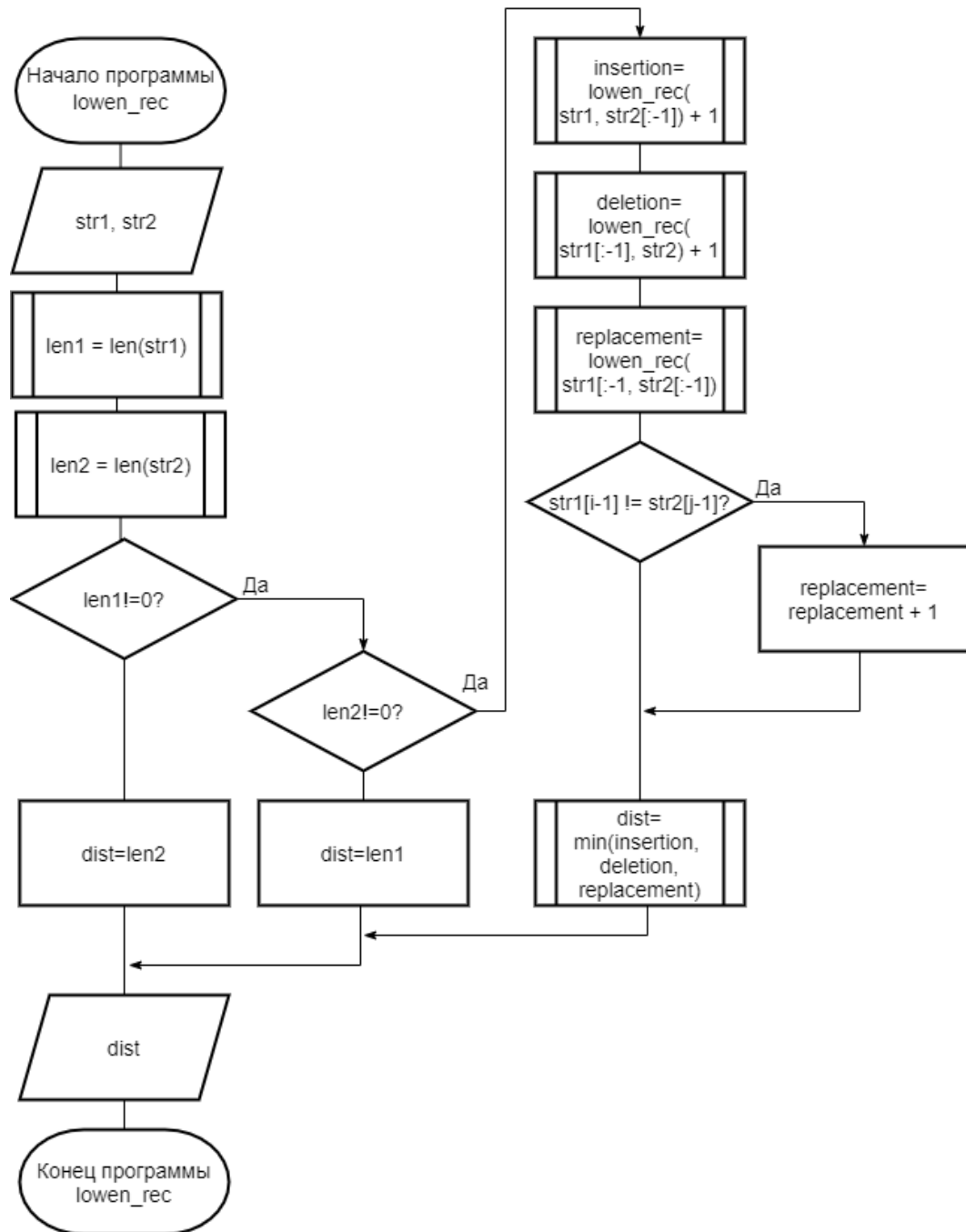


Рисунок 2.2 – Схема рекурсивного алгоритма поиска расстояния Левенштейна без кеширования



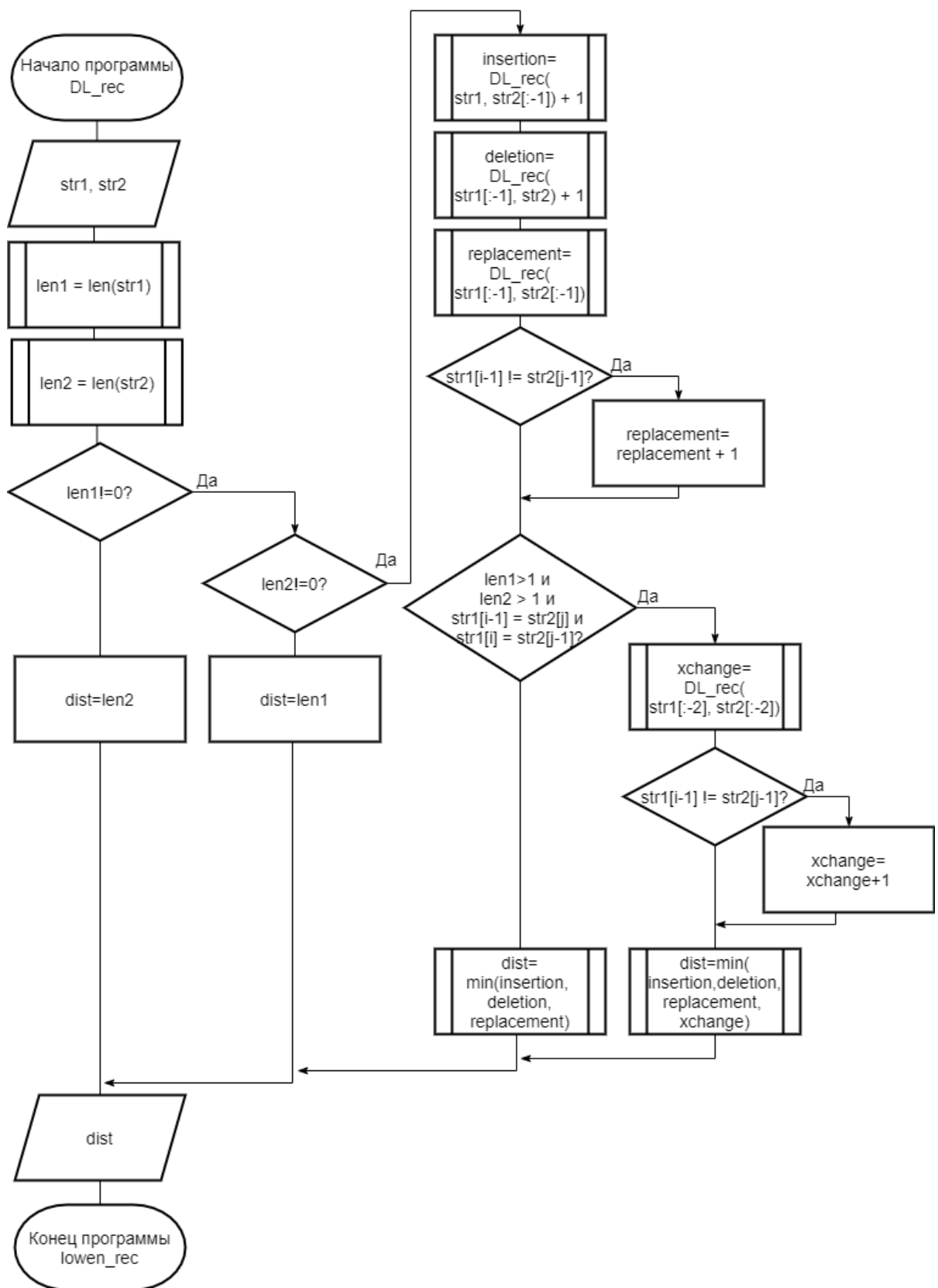


Рисунок 2.4 – Схема рекурсивного алгоритма поиска расстояния Дameraу-Левенштейна без кеширования

## 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 Дameraу-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 структуры 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 кросс-платформенным, а также предоставляет удобный и функциональный 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

```
1 def lowenstein_dist_matrix_classic(str1, str2):
2     # +1 because of an empty string
3     n = len(str1) + 1
4     m = len(str2) + 1
5     matrix = [[0 for i in range(m)] for j in range(n)] # MATCH
6
7     # fill with trivial rules
8     for i in range(1, n):
9         matrix[i][0] = i # DELETION
10    for j in range(1, m):
11        matrix[0][j] = j # INSERTION
12
13    # fill the rest of the matrix
14    for i in range(1, n):
15        for j in range(1, m):
16            insertion = matrix[i][j - 1] + 1
17            deletion = matrix[i - 1][j] + 1
18            replacement = matrix[i - 1][j - 1] + int(str1[i - 1] !=
19                str2[j - 1])
20
21            matrix[i][j] = min(insertion, deletion, replacement)
22
23    return matrix[n - 1][m - 1]
```

Листинг 3.2 – Функция рекурсивного алгоритма поиска расстояния Левенштейна без кеширования

```
1 def lowenstein_dist_recursion_classic(str1, str2):
2     # trivial rules
3     if not str1:
4         return len(str2)
5     elif not str2:
6         return len(str1)
7
8     insertion = lowenstein_dist_recursion_classic(str1, str2[:-1])
9         + 1
10    deletion = lowenstein_dist_recursion_classic(str1[:-1], str2)
11        + 1
12    replacement = lowenstein_dist_recursion_classic(str1[:-1],
```

```

11         str2[: -1]) + int(str1[-1] != str2[-1])
12     return min(insertion, deletion, replacement)

```

Листинг 3.3 – Функция рекурсивного алгоритма поиска расстояния Левенштейна с кешированием

```

1  def lowenstein_dist_recursion_optimized(str1, str2):
2      def _lowenstein_dist_recursion_optimized(str1, str2, matrix):
3          len1 = len(str1)
4          len2 = len(str2)
5
6          # trivial rules
7          if not len1:
8              matrix[len1][len2] = len2
9          elif not len2:
10             matrix[len1][len2] = len1
11          else:
12             # insertion
13             if matrix[len1][len2 - 1] == -1:
14                 _lowenstein_dist_recursion_optimized(str1, str2[: -1],
15                                                         matrix)
16             # deletion
17             if matrix[len1 - 1][len2] == -1:
18                 _lowenstein_dist_recursion_optimized(str1[: -1], str2,
19                                                         matrix)
20             # replacement
21             if matrix[len1 - 1][len2 - 1] == -1:
22                 _lowenstein_dist_recursion_optimized(str1[: -1], str2
23                                                         [: -1], matrix)
24
25             matrix[len1][len2] = min(matrix[len1][len2 - 1] + 1,
26                                     matrix[len1 - 1][len2] + 1,
27                                     matrix[len1 - 1][len2 - 1] + int(str1[-1] != str2[-1]))
28
29             # +1 because of an empty string
30             n = len(str1) + 1
31             m = len(str2) + 1
32             matrix = [[-1 for i in range(m)] for j in range(n)]
33             _lowenstein_dist_recursion_optimized(str1, str2, matrix)
34
35     return matrix[n - 1][m - 1]

```

Листинг 3.4 – Функция рекурсивного алгоритма поиска расстояния  
Дамерау-Левенштейна

```
1 def damerau_lowenstein_dist_recursion(str1, str2):
2     # trivial rules
3     if not str1:
4         return len(str2)
5     elif not str2:
6         return len(str1)
7
8     insertion = lowenstein_dist_recursion_classic(str1, str2[:-1])
9         + 1
10    deletion = lowenstein_dist_recursion_classic(str1[:-1], str2)
11        + 1
12    replacement = lowenstein_dist_recursion_classic(str1[:-1],
13        str2[:-1]) + int(str1[-1] != str2[-1])
14
15    if len(str1) > 1 and len(str2) > 1 and str1[-1] == str2[-2]
16        and str1[-2] == str2[-1]:
17        xchange = lowenstein_dist_recursion_classic(str1[:-2], str2
18           [:-2]) + int(str1[-1] != str2[-1])
19        return min(insertion, deletion, replacement, xchange)
20    else:
21        return min(insertion, deletion, replacement)
```

## 3.4 Тестирование

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна и Дамерау — Левенштейна. Тесты пройдены успешно.

## Вывод

Был произведен выбор средств реализации, реализованы и протестированы алгоритмы поиска расстояний: Левенштейна - итерационный и рекурсивный (с кешем и без), Дамерау-Левенштейна - рекурсивный без кеша

Таблица 3.1 – Тесты

Строка 1	Строка 2	Ожидаемый результат	
		Алг. Левенштейна	Алг. Дамерау-Левенштейна
		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



## 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 алгоритмов

Все реализованные алгоритмы 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. Замеры времени для каждой длины строк и каждого алгоритма производились 10 раз, а затем вычислялось среднее время работы алгоритма со строкой.

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

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

На рисунке 4.3 приведено сравнение времени выполнения реализаций итерационного и рекурсивного (с кешированием) алгоритмов поиска расстояния Левенштейна. Как видно, время их работы растет соизмеримо, что обусловлено их схожестью в отсутствии вызова функций для вычисления значений, которые уже были подсчитаны ранее. Однако второй алгоритм

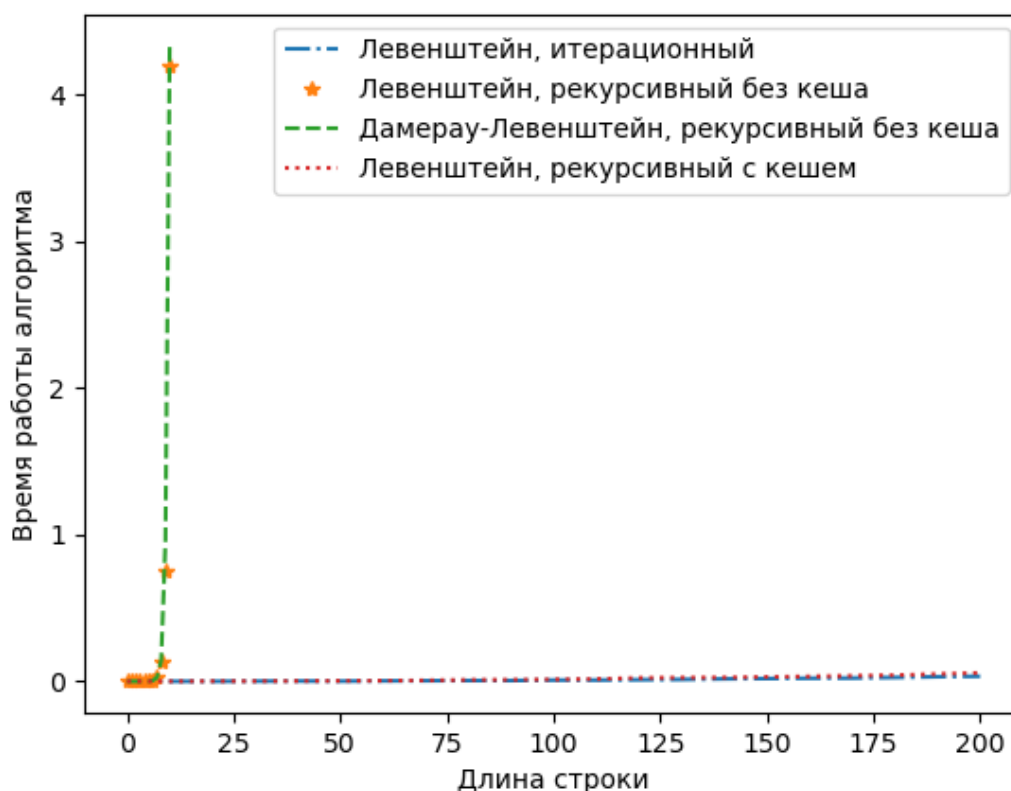


Рисунок 4.2 – Сравнение времени работы алгоритмов

все же работает дольше, так как в нем тратится время на рекурсивный вызов самого себя.

На рисунке 4.4 приведено сравнение времени выполнения реализаций рекурсивных алгоритмов (без кеширования) поиска расстояний Левенштейна и Дамерау-Левенштейна. Эти алгоритмы работали примерно одинаковое количество времени, однако алгоритм 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

Алгоритмы нахождения расстояний Левенштейна и Дамерау—Левенштейна имеют одинаковый подход к использованию памяти. Следовательно, достаточно сравнить, как с увеличением длин строк изменяется пиковая затрачиваемая память в рекурсивных и итерационных реализациях этих алго-

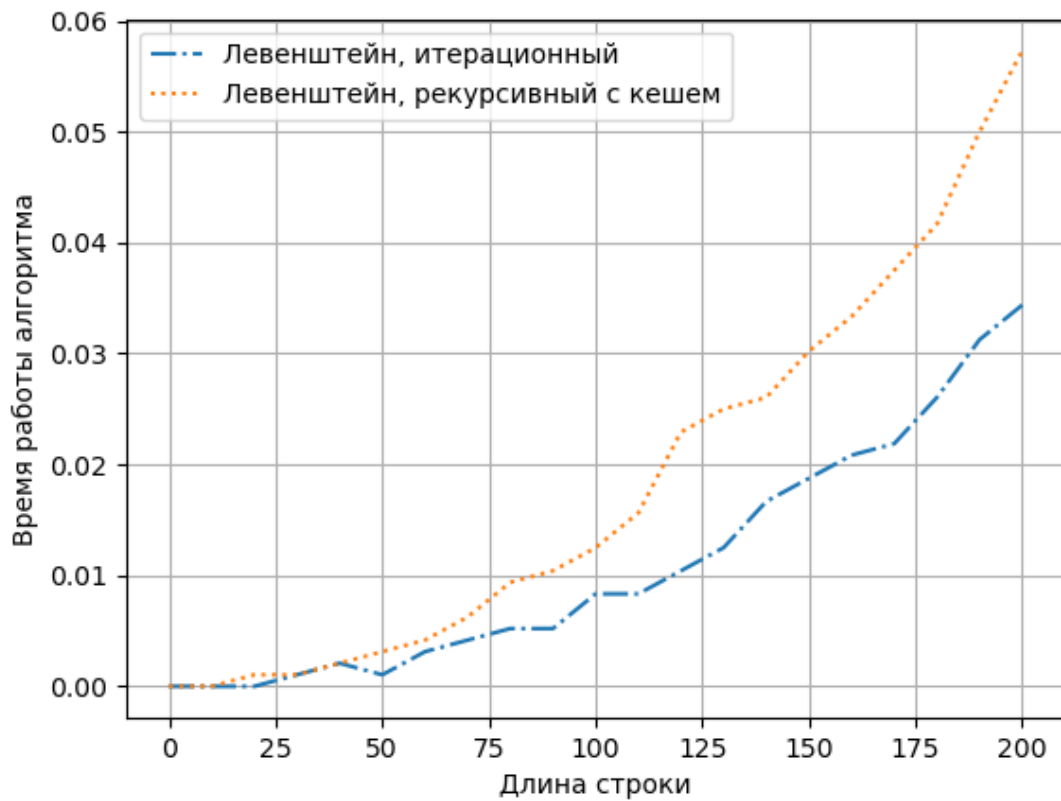


Рисунок 4.3 – Сравнение времени работы рекурсивных алгоритмов, использующих матрицы расстояний

ритмов.

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 равна сумме длин исходных строк. Следовательно, пиковая затрачиваемая память растет пропорционально сумме длин строк.

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

Однако итерационный алгоритм можно оптимизировать по памяти, храня лишь последние две строки матрицы расстояний, и тогда пиковая затрачиваемая память будет расти пропорционально изменению длины одной из строк, что меньше, чем при рекурсивной реализации.

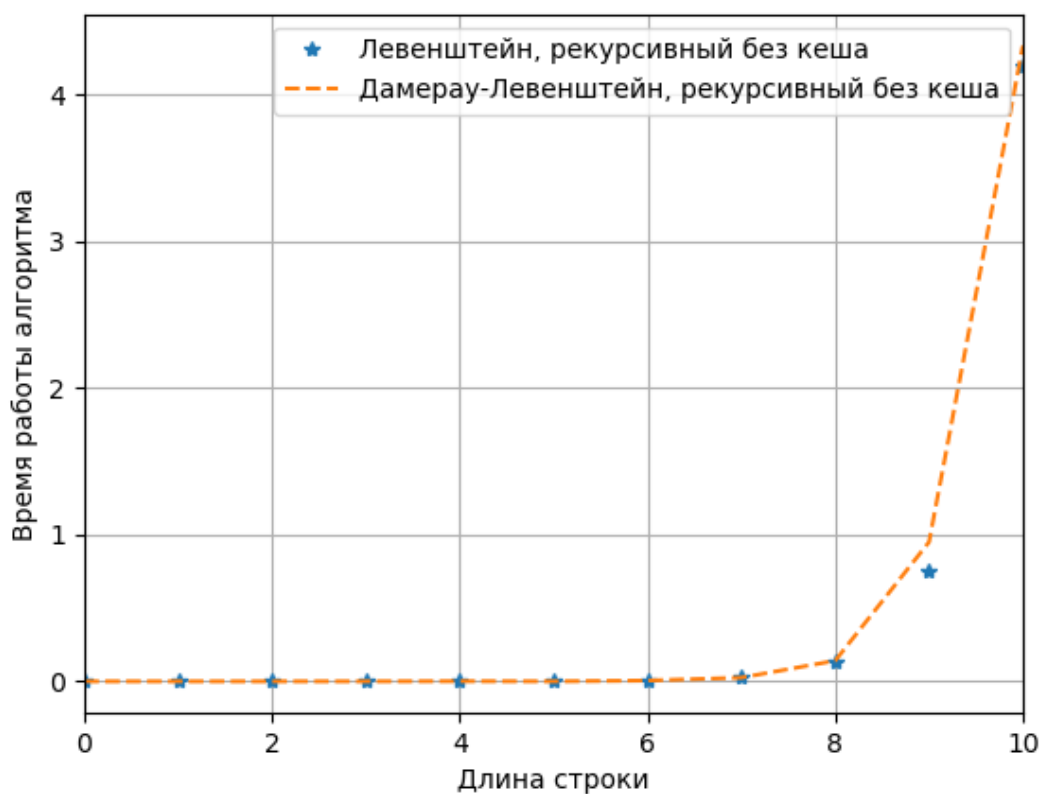


Рисунок 4.4 – Сравнение времени работы рекурсивных алгоритмов, не использующих кеш

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

Алгоритм нахождения расстояния Дамерау — Левенштейна по времени выполнения сопоставим с алгоритмом нахождения расстояния Левенштейна, хотя и немного уступает второму в связи с дополнительной проверкой, позволяющей находить ошибки пользователя, связанные с неверным порядком букв. Однако эта операция зачастую позволяет найти более корот-

кое расстояние между строками.

Рекурсивные алгоритмы нахождения расстояний Левенштейна и Дamerau-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 алгоритмы проигрывают рекурсивным: максимальный размер используемой памяти в них пропорционален произведению длин строк, в то время как в рекурсивных алгоритмах — сумме длин строк.

Если же применить к итеративным алгоритмам 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. Черненко 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).