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»

Лабораторная работа №1
по курсу "Анализ алгоритмов"

Тема Расстояние Левенштейна

Студент Мишина Е.В.

Группа ИУ7-54Б

Оценка (баллы) _____

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

Москва — 2020 г.

Оглавление

Введение

Целью данной лабораторной работы является изучение метода динамического программирования на примере алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

Задачами лабораторной работы являются:

- предоставление математического описания расстояний Левенштейна и Дамерау-Левенштейна;
- разработка алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна;
- реализация динамической и рекурсивной вариации указанных алгоритмов;
- тестирование реализованных алгоритмов;
- проведение сравнительного анализа алгоритмов по временной и емкостной эффективности.

1 | Аналитическая часть

Расстояние Левенштейна[1](редакционное расстояние, дистанция редактирования) – минимальное количество операций удаления, вставки и замены символа, необходимое для преобразования одной строки в другую.

Расстояние Левенштейна[2] и его обобщения активно применяется:

- для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);
- для сравнения текстовых файлов утилитой diff и ей подобными. Здесь роль «символов» играют строки, а роль «строк» — файлы;
- в биоинформатике для сравнения генов, хромосом и белков.

Задача по нахождению расстояния Левенштейна заключается в поиске минимального количества редакторских операций (вставка/удаление/замена) необходимых для преобразования одной строки в другую.

При нахождении расстояния Дамерау — Левенштейна добавляется операция перестановки двух соседних символов или транспозиция. Полное определение рассмотрено в [1].

Действия обозначаются так:

- I (англ. insert) — вставить;
- D (англ. delete) — удалить;
- R (replace) — заменить;
- M(match) - совпадение.

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по рекуррентной формуле (1.1), см [2]:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min(& \\ D(i,j-1)+1, & \\ D(i-1,j)+1, & j > 0, i > 0 \\ D(i-1,j-1)+m(S_1[i],S_2[j]) & \\ ), & \end{cases} \tag{1.1}$$

где $m(a,b)$ равна нулю, если $a = b$ и единице в противном случае; $min\{\, a,b,c\}$ возвращает наименьший из аргументов.

Расстояние Дамерау-Левенштейна вычисляется по рекуррентной формуле (1.2), см [2]:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & i > 0, j = 0 \\ j, & i = 0, j > 0 \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[i]), \\ D(i-2,j-2)+m(S_1[i],S_2[i]), \end{cases} & \text{, если } i,j > 0 \\ & \text{и } S_1[i] = S_2[j-1] \\ & \text{и } S_1[i-1] = S_2[j] \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[i]), \end{cases} & \text{, иначе} \end{cases} \tag{1.2}$$

Вывод

В данном разделе было дано математическое описание расстояний Левенштейна и Дамерау-Левенштейна. Расстояние Дамерау-Левенштейна

отличается от классического расстояния Левенштейна тем, что включает транспонирование в число его допустимых операций в дополнение к трем классическим операциям редактирования.

2 | Конструкторская часть

Ввод:

- на вход подаются две строки;
- строки могут быть пустыми, содержать пробелы, а также любые печатные символы UTF-8;
- буквы верхнего и нижнего регистра считаются разными.

Вывод:

- программа выводит посчитанные каждым из алгоритмов расстояния;
- для динамических реализаций алгоритмов выводятся заполненные матрицы;
- в режиме замера ресурсов программа выводит средние время, затраченное каждым алгоритмом.

2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы матричного алгоритма нахождения расстояния Левенштейна, рекурсивного алгоритма нахождения расстояния Левенштейна, матричного алгоритма нахождения расстояния Дамерау-Левенштейн, рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна и показаны на рисунках 2.1, 2.2, 2.3 и 2.4, соответственно.

2.2 Вывод

В данном разделе были рассмотрены схемы матричного алгоритма нахождения расстояния Левенштейна, рекурсивного алгоритма нахождения расстояния Левенштейна, матричного алгоритма нахождения расстояния Дамерау-Левенштейн, рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна.

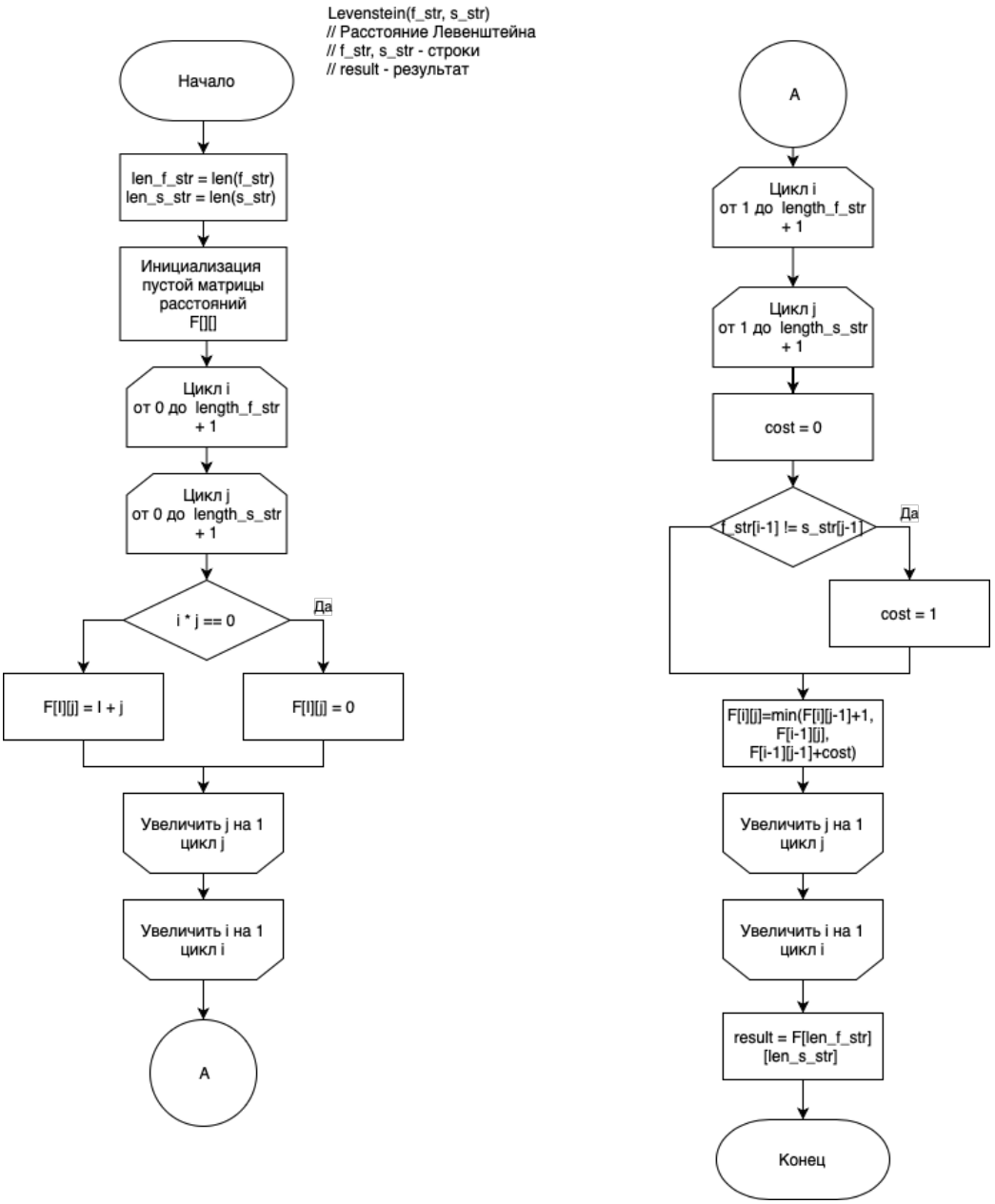

Рис. 2.1: Схема матричного алгоритма нахождения расстояния Левенштейна

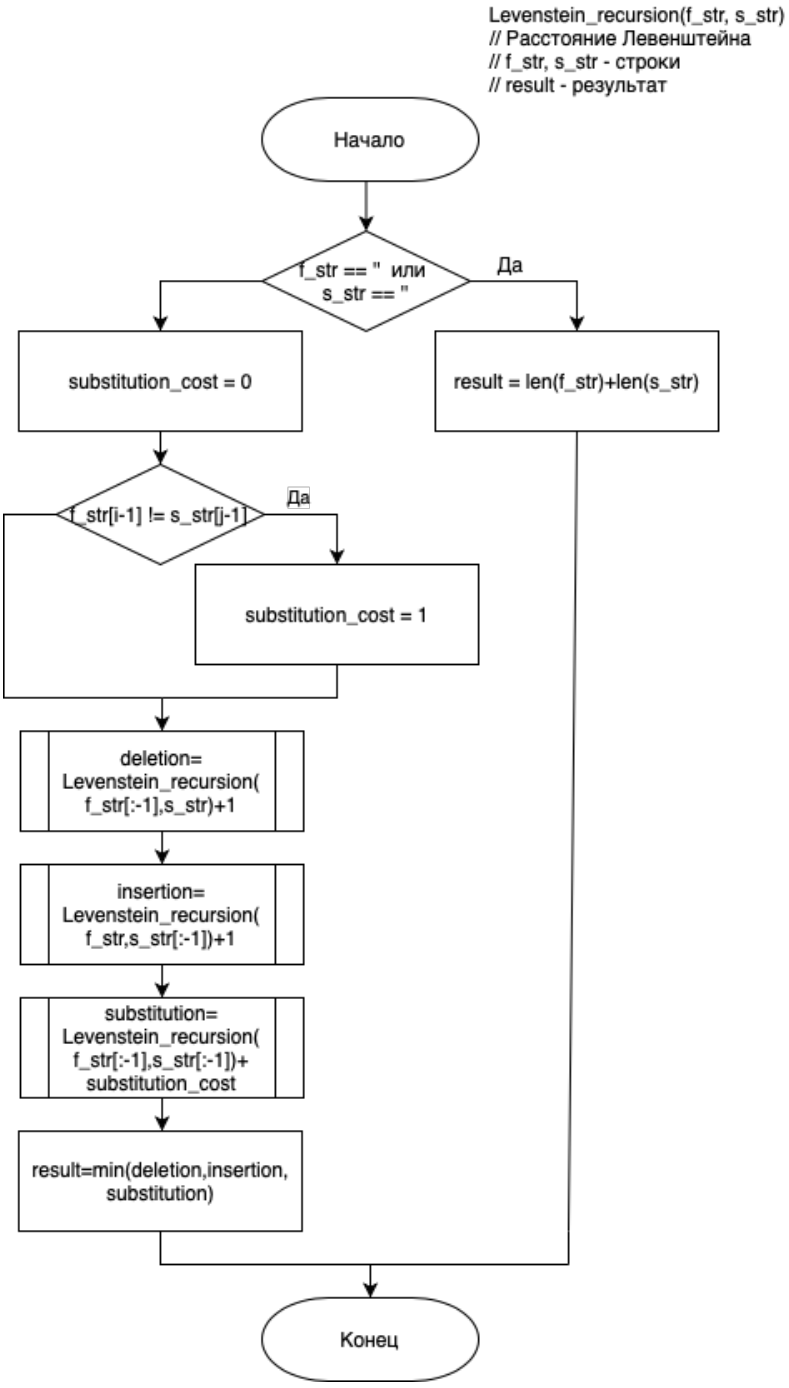

Рис. 2.2: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

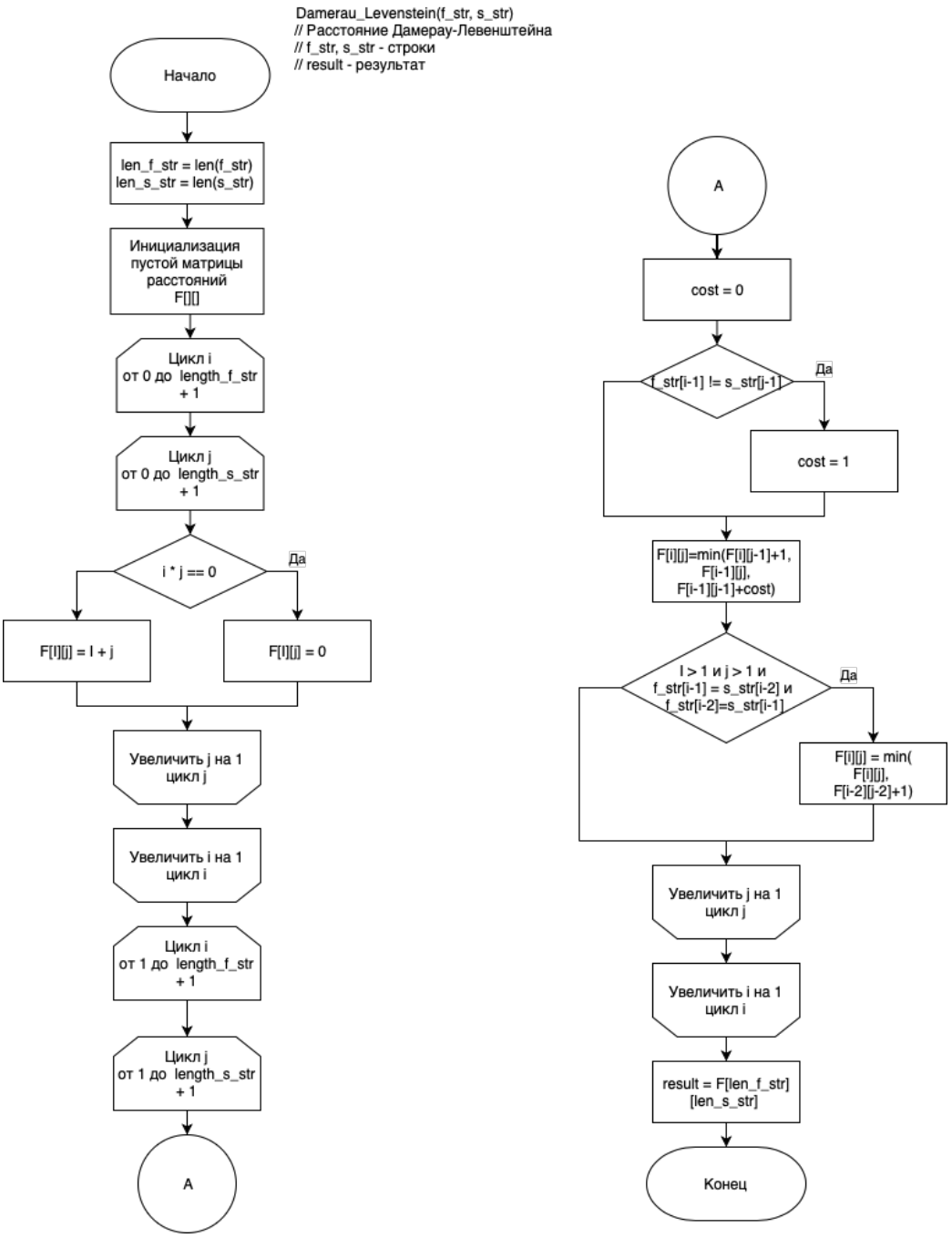

Рис. 2.3: Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

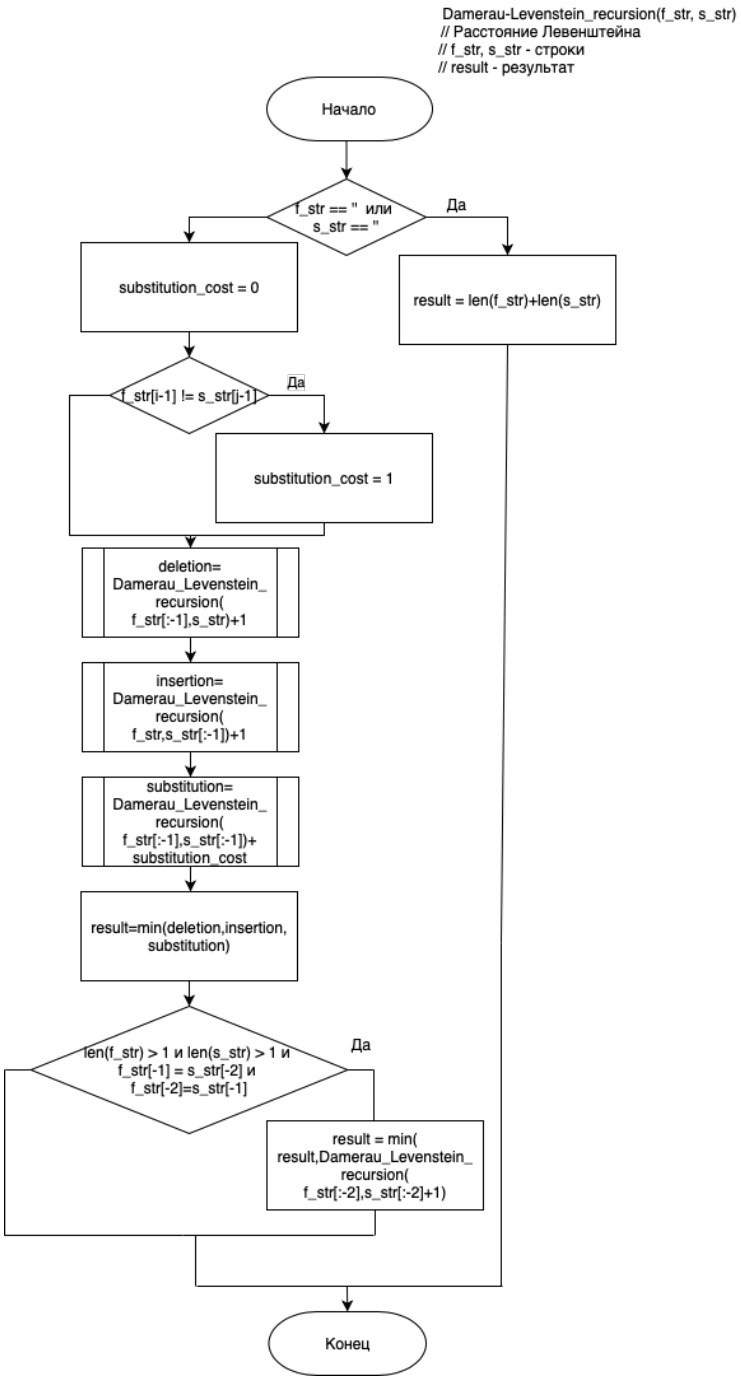

Рис. 2.4: Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

3 | Технологическая часть

3.1 Выбор ЯП

Для реализации программы был выбран Python[3] из-за наличия опыта разработки на данном языке программирования. Для тестирования были выбрана операционная система - Windows 10. Процессор - 2,3 GHz Quad-Core Intel Core i5. Память - 8 GB 2133 MHz LPDDR3. Графическое ядро - Intel Iris Plus Graphics 655 1536 MB.

3.2 Реализация алгоритма

Листинг 3.1: Функция нахождения расстояния Левенштейна матрично

```
1     def Levenstein(f_str, s_str):
2         len_f_str = len(f_str)
3         len_s_str = len(s_str)
4         F = [[(i + j) if i * j == 0 else 0 for j in range(
              len_s_str + 1)]
5         for i in range(len_f_str + 1)]
6
7         for i in range(1, len_f_str + 1):
8             for j in range(1, len_s_str + 1):
9                 cost = 0
10                if f_str[i - 1] != s_str[j - 1]:
11                    cost = 1
12                F[i][j] = min(F[i - 1][j] + 1, F[i][j - 1]
                      + 1, F[i - 1][j - 1] + cost)
13        print_matrix(F, f_str, s_str)
14
```

```
14
15          result = F[len_f_str][len_s_str]
16          return result
```

Листинг 3.2: Функция нахождения расстояния Левенштейна рекурсивно

```
1       def Levenstein_recursion(f_str, s_str):
2           if f_str == '' or s_str == '':
3               result = len(f_str) + len(s_str)
4               return result
5
6           substitution_cost = 0
7
8           if f_str[-1] != s_str[-1]:
9               substitution_cost = 1
10
11          deletion = Levenstein_recursion(f_str[:-1], s_str)
               + 1
12          insertion = Levenstein_recursion(f_str, s_str[:-1])
               + 1
13          substitution = Levenstein_recursion(f_str[:-1],
               s_str[:-1]) + substitution_cost
14
15          result = min(deletion, insertion, substitution)
16          return result
```

Листинг 3.3: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
1       def Damerau_Levenshtein(f_str, s_str):
2           len_f_str = len(f_str)
3           len_s_str = len(s_str)
4
5           F = [[(i + j) if i * j == 0 else 0 for j in range(
               len_s_str + 1)]
6               for i in range(len_f_str + 1)]
7
8           for i in range(1, len_f_str + 1):
9               for j in range(1, len_s_str + 1):
10              cost = 0
11              if f_str[i - 1] != s_str[j - 1]:
```

```
                cost = 1

            F[i][j] = min(F[i - 1][j] + 1, F[i][j - 1] + 1,
                F[i - 1][j - 1] + cost)

            if i > 1 and j > 1 and f_str[i - 1] == s_str[j
               - 2] and f_str[i - 2] == s_str[j - 1]:
                F[i][j] = min(F[i][j], F[i - 2][j - 2] + 1)

        print_matrix(F, f_str, s_str)
        result = F[len_f_str][len_s_str]
        return result
```

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
    def Damerau_Levenshtein_recursion(f_str, s_str):
        if f_str == '' or s_str == '':
            result = len(f_str) + len(s_str)
            return result

            substitution_cost = 0

            if f_str[-1] != s_str[-1]:
                substitution_cost = 1

        deletion = Damerau_Levenshtein_recursion(f_str
           [:-1], s_str) + 1
        insertion = Damerau_Levenshtein_recursion(f_str,
           s_str[:-1]) + 1
        substitution = Damerau_Levenshtein_recursion(f_str
           [:-1], s_str[:-1]) +          substitution_cost

        result = min(deletion, insertion, substitution)


        if len(f_str) > 1 and len(s_str) > 1 and f_str[-1]
           == s_str[-2] and f_str[-2] ==    s_str[-1]:
            result = min(result,
               Damerau_Levenshtein_recursion(f_str[:-2],
               s_str[:-2]) + 1)
```

```
20
21
22          return result
```

Листинг 3.5: Функция вывода матрицы

```
1       def print_matrix(matr, f_str, s_str):
2           print("\n   ", end=" ")
3           for i in s_str:
4               print(i, end=' ')
5           for i in range(len(matr)):
6               if i != 0:
7                   print("\n" + f_str[i - 1], end=" ")
8               else:
9                   print("\n ", end=" ")
10              for j in range(len(matr[i])):
11                  print(matr[i][j], end=' ')
12          print("\n")
```

3.3 Вывод

В данном разделе были рассмотрены алгоритмы матричного алгоритма нахождения расстояния Левенштейна, рекурсивного алгоритма нахождения расстояния Левенштейна, матричного алгоритма нахождения расстояния Дамерау-Левенштейн, рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна.

4 Исследовательская часть

4.1 Сравнение на основе замеров времени работы алгоритмов

Был проведен замер времени работы каждого из алгоритмов. Результат замера показан в таблице 4.1.

Таблица 4.1: Время работы алгоритмов (в секундах)

len	Lev	Lev-rec	D-Lev	D-Lev-rec
1	0.0000141	0.0000120	0.0000108	0.0000062
2	0.0000166	0.0000149	0.0000170	0.0000146
3	0.0000198	0.0000516	0.0000202	0.0000497
4	0.0000232	0.0001950	0.0000258	0.0002202
5	0.0000317	0.0009891	0.0000312	0.0011373
6	0.0000406	0.0053555	0.0000427	0.0057266
7	0.0000505	0.0280688	0.0000546	0.0299394

Полученная зависимость времени работы алгоритмов от длины строк показана на рисунке 4.1.

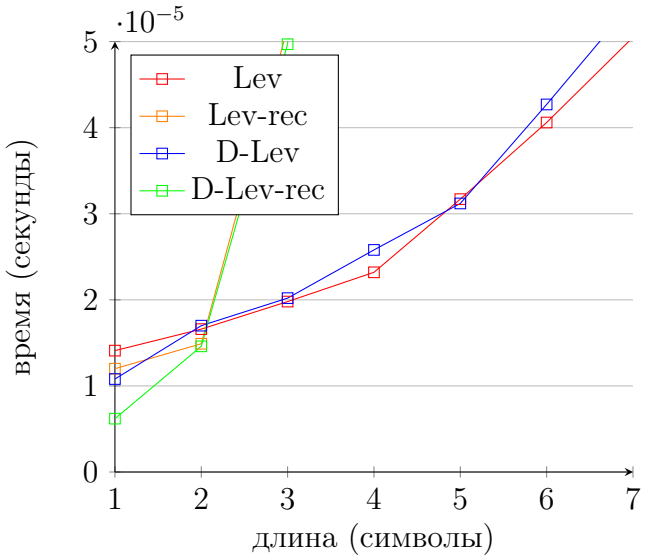

Рис. 4.1: Зависимость времени работы от длины строк

Рекурсивные алгоритмы более быстрые на коротких строках, но при увеличении длины, динамические алгоритмы стали более медленные из-за того, что каждая ячейка матрицы рассчитывается единожды, в отличие от повторных расчетов в рекурсивных реализациях. Алгоритм Дамерау-Левенштейна в среднем работает несколько дольше алгоритма Левенштейна, что объясняется наличием дополнительных проверок для операции транспозиции.

4.2 Сравнительный анализ алгоритмов на основе замеров затрачиваемой памяти

Был проведен замер памяти, затрачиваемой алгоритмами. Результат замера показан в таблице 4.2.

Таблица 4.2: Количество памяти, затрачиваемой алгоритмом (в блоках)

len	Lev	Lev-rec	D-Lev	D-Lev-rec
1	1976	1008	1976	1040
2	592	1008	592	1040
3	624	1110	624	1638
4	800	1214	800	750
5	896	1587	896	1793
6	960	1428	960	1956
7	1024	1538	1024	1570

Полученная зависимость памяти затрачиваемой алгоритмами от длины строк показана на рисунке 4.2.

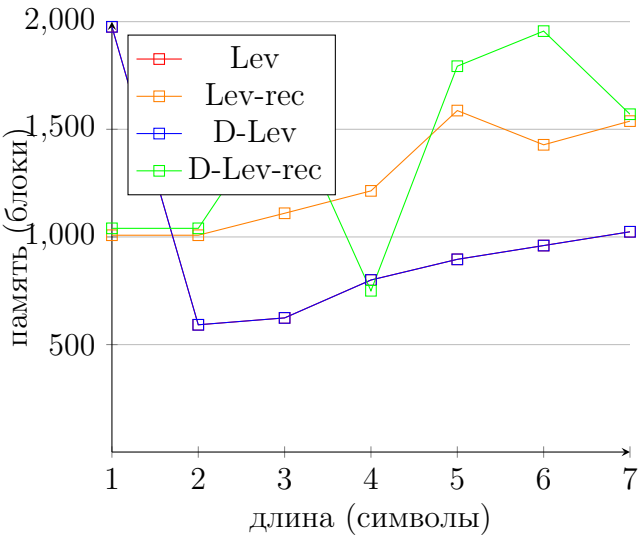

Рис. 4.2: Зависимость затрачиваемой памяти от длины строк

Таким образом, рекурсивные алгоритмы при небольших длинах строк выигрывают по количеству затрачиваемой памяти с динамическими, но на длинных строках количество памяти, затрачиваемой рекурсивными алгоритмами возрастает из-за локальных переменных, создаваемых при каждом вызове алгоритма. Память динамических алгоритмов изменяется только из-за увеличения матрицы.

4.3 Тестовые данные

Проведем тестирование программы. В столбцах "Ожидаемый результат"и "Полученный результат"4 числа соответсвуют матричному, рекурсивному алгоритму нахождения расстояния Левенштейна, матричному и рекурсивному алгоритму расстояния Дамерау-Левенштейна.

Таблица 4.3: Таблица тестовых данных

№	Первое слово	Второе слово	Ожидаемый результат	Полученный результат
1			0 0 0 0	0 0 0 0
2	bmstu	bmstu	0 0 0 0	0 0 0 0
3	стол	тело	3 3 3 3	3 3 3 3
4	qwerty	qweryt	2 2 1 1	2 2 1 1
5		ra	2 2 2 2	2 2 2 2
6	qw		2 2 2 2	2 2 2 2
7	app	all	2 2 2 2	2 2 2 2
8	latex	tex	2 2 2 2	2 2 2 2
9	telephone	telpeohne	3 3 2 2	3 3 2 2

Заключение

Было дано математическое описание алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна, были рассмотрены их схемы и реализации алгоритмов на языке программирования Python.

Рекурсивные алгоритмы более быстрые на коротких строках (в 1.5 раза), но при увеличении длины, динамические алгоритмы стали более быстрыми (в 3 раза) из-за того, что каждая ячейка матрицы рассчитывается единожды, в отличие от повторных расчетов в рекурсивных реализациях. Алгоритм Дамерау-Левенштейна в среднем работает несколько дольше алгоритма Левенштейна, что объясняется наличием дополнительных проверок для операции транспозиции.

Рекурсивные алгоритмы при небольших длинах строк выигрывают по количеству затрачиваемой памяти (почти в 2 раза) с динамическими, но на длинных строках количество памяти, затрачиваемой рекурсивными алгоритмами возрастает (в 2 раза) из-за локальных переменных, создаваемых при каждом вызове алгоритма. Память динамических алгоритмов изменяется только из-за увеличения матрицы.

Таким образом, матричная реализация данных алгоритмов выигрывает по времени и по количеству выделяемой памяти при росте длины строк, следовательно матричная реализация более применима в реальных проектах.

Список использованных источников

1. Расстояние Левенштейна [Электронный ресурс]. – Режим доступа: https://programm.top/c-sharp/algorithm/levenshtein-distance. – Дата доступа: 5.10.2020.

2. Расстояние Левенштейна [Электронный ресурс]. – Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=Задача-о-редакционном-расстоянии,-алгоритм-Вагнера-Фишера. – Дата доступа: 9.10.2020.

3. Python [Электронный ресурс]. – Режим доступа: https://www.python.org. – Дата доступа: 9.10.2020.

4. PyCharm [Электронный ресурс]. – Режим доступа: https://www.jetbrains.com/ru-ru/pycharm. – Дата доступа: 9.10.2020.