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» (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.01 Информатика и вычислительная техника**

О Т Ч Е Т

по лабораторной работе № 1

Название: Расстояние Левенштейна и Дамерау-Левенштейна

Дисциплина: Анализ алгоритмов

Студент ИУ7-54Б ________________ Елгин И.Ю.

(Группа) (Подпись, дата) (И.О. Фамилия)

Преподаватель ________________ Волкова Л.Л.

(Подпись, дата) (И.О. Фамилия)

Москва, 2021

Оглавление

Введение

Расстояние Левенштейна - минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую [2].

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для:

- исправления ошибок в слове;
- сравнения текстовых файлов;
- в биоинформатике для сравнения генов, хромосом и белков.

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

Задачами данной лабораторной являются:

1. Изучение рекурсивных и нерекурсивных алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
2. Применение метода динамического программирования для нерекурсивной реализации алгоритмов Левенштейна и Дамерау-Левенштейна;

3. Получение практических навыков реализации следующих алгоритмов: алгоритм Левенштейна с кешом в две строки,алгоритм Дамерау-Левенштейна с матрицей, рекурсивный алгоритм Левенштейна, рекурсивный алгоритм Дамерау-Левенштейна;

4. Сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);

5. Экспериментальное подтверждение различий во временно́й эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма;

6. Описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

Задача по нахождению расстояния Левенштейна заключается в поиске минимального количества операций вставки/удаления/замены для превращения одной строки в другую.

При нахождении расстояния Дамерау — Левенштейна добавляется операция транспозиции (перестановки соседних символов).

Действия обозначаются так:

1. D (англ. delete) — удалить;
2. I (англ. insert) — вставить;
3. R (replace) — заменить;
4. M (match) - совпадение;
5. T (transposition) - транспозиция в алгоритме Дамерау - Левенштейна.

1.1 Анализ алгоритмов

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

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

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 следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min(\\ D(i, j-1) + 1, \\ D(i-1, j) + 1, & j > 0, i > 0 \\ D(i-1, j-1) + m(S_1[i], S_2[j]) \\ D(i-2, j-2) + 1, & \text{if } i, j > 1 \text{ and } a_i = b_{j-1}, a_{i-1} = b_j \\ ). \end{cases}$$

1.2 Способ измерения времени работы алгоритма

Существует два подхода для замера времени измерение реалного времени и измерение процессорного времени.

Измерение реального времени данный способ прост так как использует только разность системного времени начала и конца выполнения алгоритма. Однако процессы происходящие паралельно с выполнением алгоритма могут повлиять на скорость выполнения алгоритма в связи с чем реалное время работы алгоритма могут меняться.

Измерение процессорного времени замеряет время выполнения только процесса замеряемого алгоритма что даёт наиболее точный результат с меньшим количеством погрешностей вызваных другими процессами выполняющимися паралельно с данным. Второй метод наиболее точен, поэтому выберем его.

2 Конструкторская часть

В данной главе рассматриваются требования к программе и приводятся схемы алгоритмов.

2.1 Требования

Требования к вводу:

1. На вход подаются две строки;
2. Буквы верхнего и нижнего регистров считаются разными.

Требования к выводу: На выходе необходимо получить число означающее минимальное количество операций, необходимых для получения из одной строки другую.

Требования к программе: Программа должна коректно работать при вводе любых двух строк.

2.2 Схемы алгоритмов

На рисунках 2.1, 2.2, 2.3, 2.4, 2.5, 2.6 приведены схемы рекурсивных и нерекурсивных алгоритмов Левенштейна и Дамерау ливенштейна.

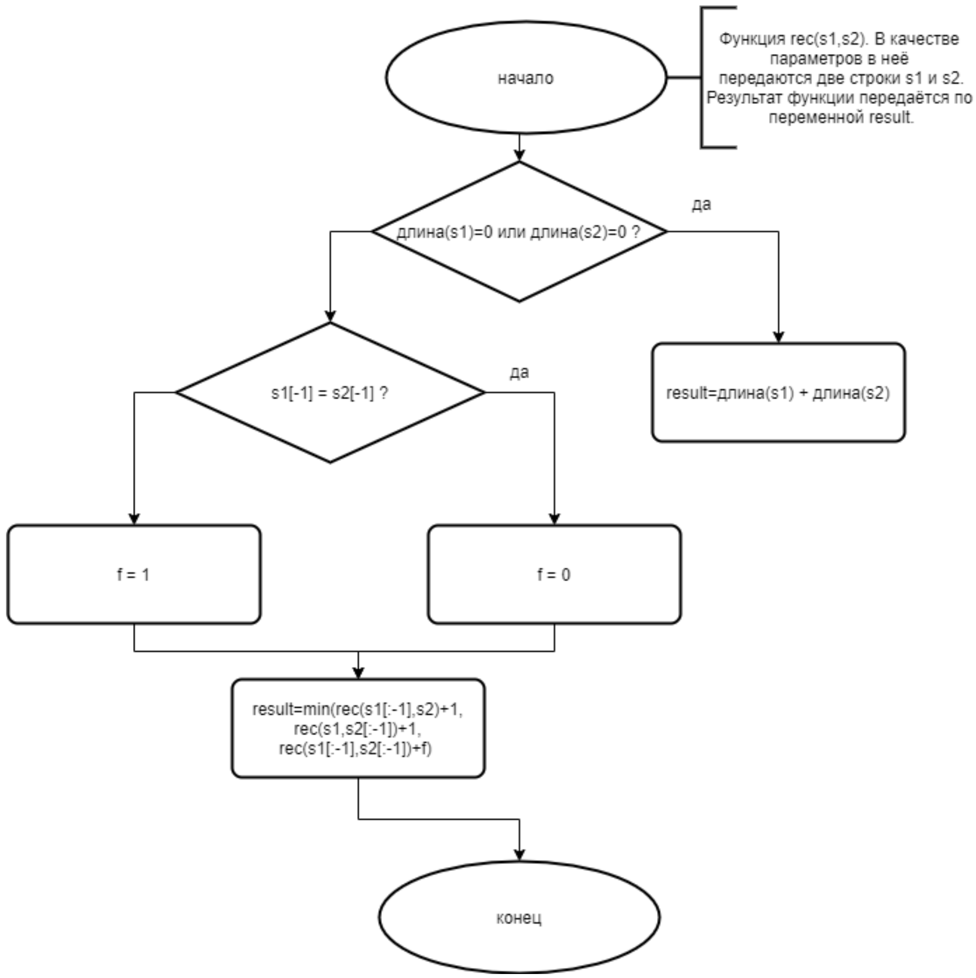

Рис. 2.1: Рекурсивный алгоритм Левенштейна

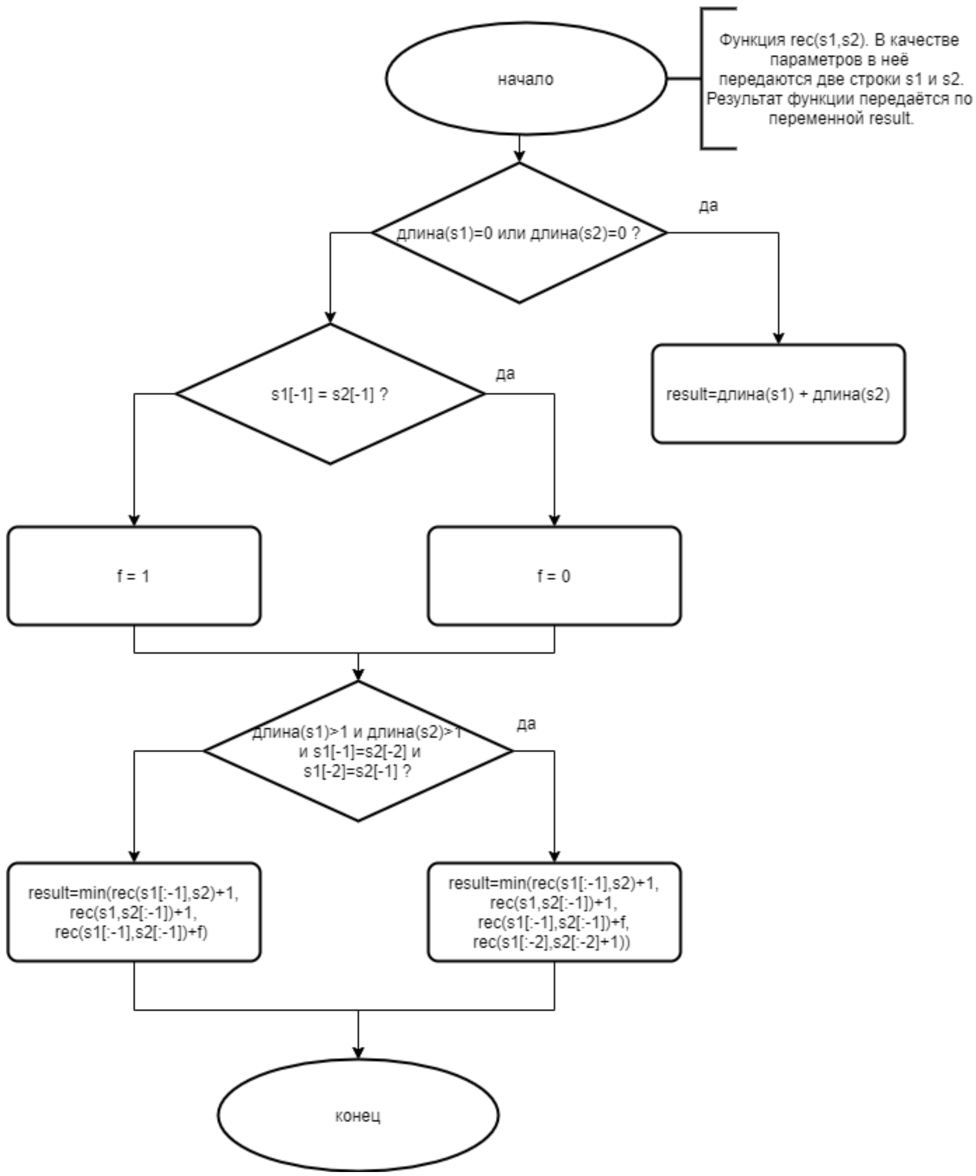

Рис. 2.2: Рекурсивный алгоритм Дамерау-Левенштейна

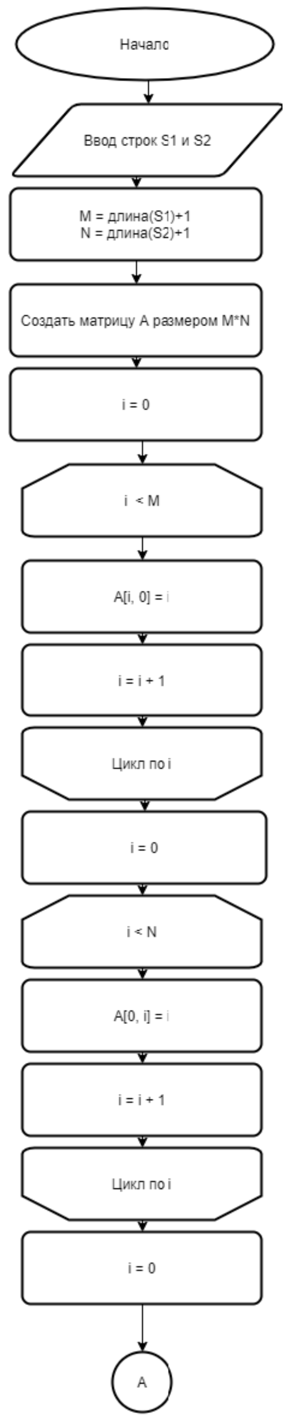

Рис. 2.3: Алгоритм Левенштейна с кэшем в две строки часть 1

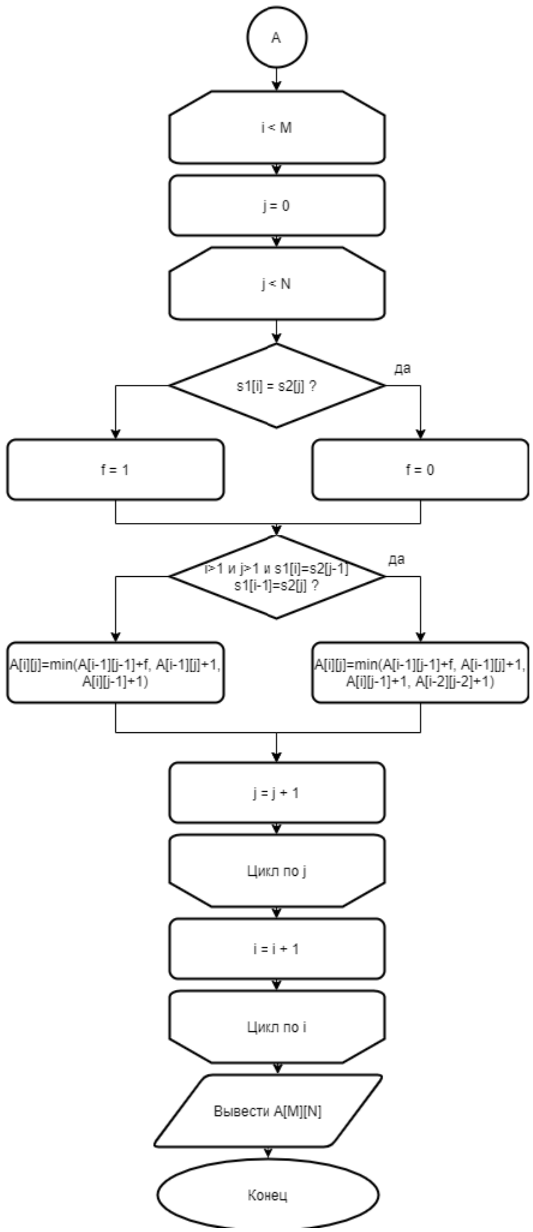

Рис. 2.4: Алгоритм Левенштейна с кэшем в две строки часть 2

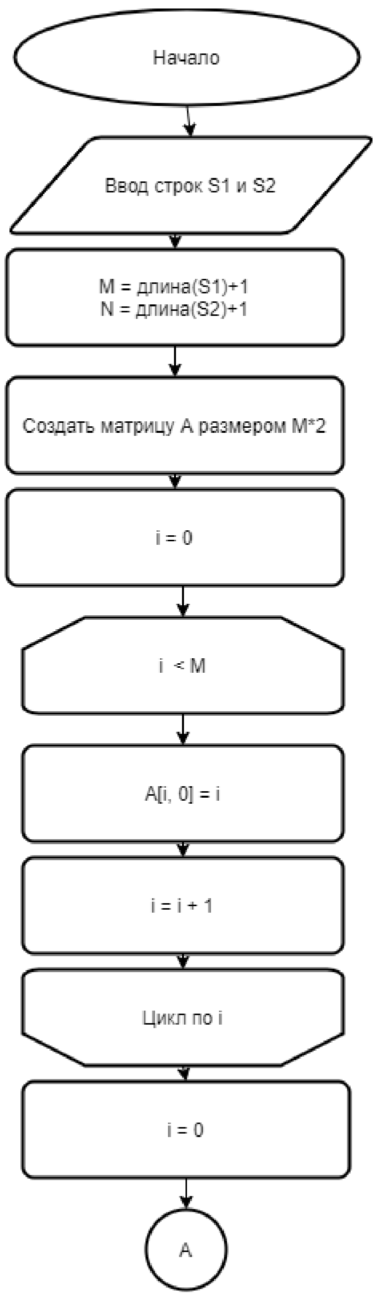

Рис. 2.5: Алгоритм Дамерау-Левенштейна с кэшем в виде матрицы часть 1

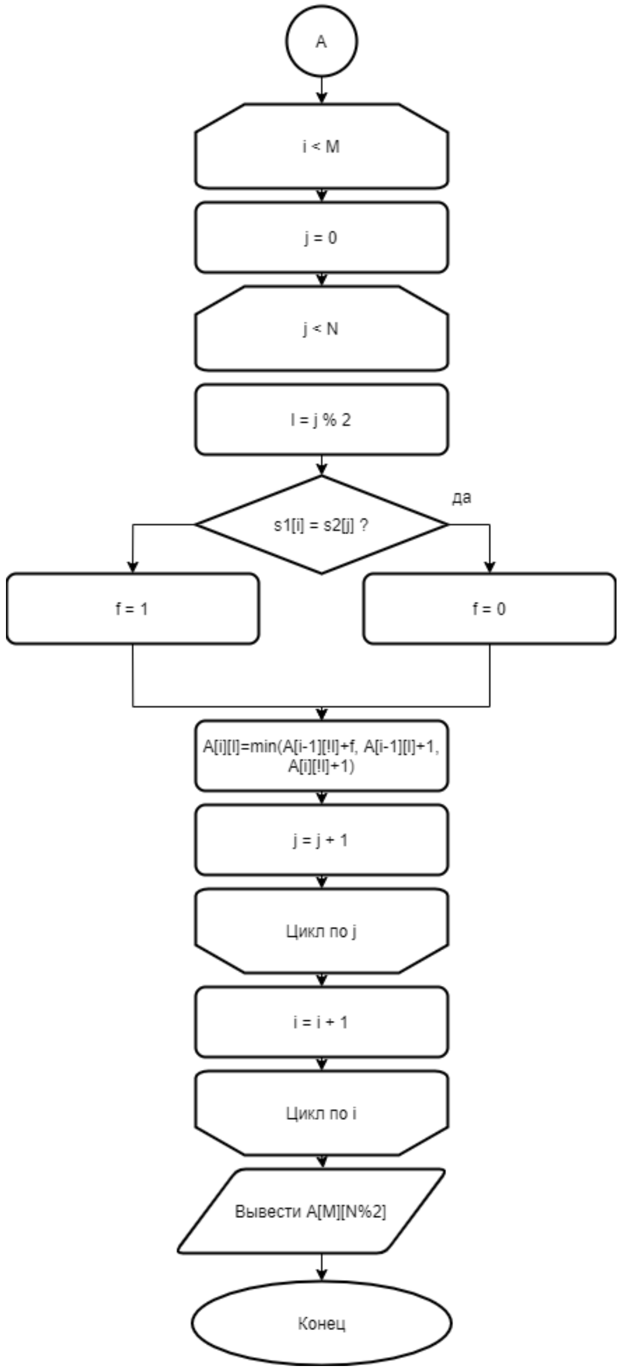

Рис. 2.6: Алгоритм Дамерау-Левенштейна с кэшем в виде матрицы часть 2

3 Технологическая часть

В данной главе выбирается язык программирования для реализации рекурсивных и нерекурсивные алгоритмы Левенштейна и Дамерау-Левенштейна, приводятся листинги функций реализующих данные алгоритмы. Приводится тестирование данных алгоритмов и сравнение памяти используемой алгоритмами.

3.1 Выбор ЯП

В качестве языком программирования мною выбран язык Python, так как я имею опыт работы с данным языком программирования, Pyton позволяет удобно работать со строками и матрицами.

Время работы алгоритмов было замерено с помощью функции process_time() из библиотеки time [4].

3.2 Сведения о модулях программы

Программа состоит из:

- lab1.py - главный файл программы, в котором располагаются алгоритмы, меню и тесты

Листинг 3.1: Функция нахождения расстояния Левенштейна рекурсивно

```
def levensteinRecursion(s1, s2):
    if (s1 == "" or s2 == ""):
        return len(s1) + len(s2)

    if (s1[-1] == s2[-1]):
        f = 0
    else:
        f = 1

    return min(levensteinRecursion(s1[:-1], s2) + 1,
               levensteinRecursion(s1, s2[:-1]) + 1,
               levensteinRecursion(s1[:-1], s2[:-1]) + f)
```

Листинг 3.2: Функция нахождения расстояния Левенштейна с кешом в 2 строки

```
def levensteinTable(s1, s2, isPrint):
    lenI = len(s1) + 1
    lenJ = len(s2) + 1

    table = [[j for j in range(lenJ)] for i in range(2)]
    for i in range(1, lenI):
        for j in range(1, lenJ):
            l = i % 2
            table[l][0] = i
            if (s1[i - 1] == s2[j - 1]):
                f = 0
            else:
                f = 1
            table[l][j] = min(table[not l][j] + 1,
                              table[l][j-1] + 1,
                              table[not l][j - 1] + f)

    return table[-1][-1]
```

Листинг 3.3: Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
def DameraulevensteinRecursion(s1, s2):
    if (s1 == "" or s2 == ""):
```

```
        return len(s1) + len(s2)

    if (s1[-1] == s2[-1]):
        f = 0
    else:
        f = 1
    if (len(s1) > 1 and len(s2) > 1 and s1[-2] == s2[-1]
       and s1[-1] == s2[-2]):
        return min(DameraulevensteinRecursion(s1[:-1], s2)
            + 1,
                   DameraulevensteinRecursion(s1, s2[:-1]) + 1,
                   DameraulevensteinRecursion(s1[:-1], s2[:-1])
                       + f,
                    DameraulevensteinRecursion(s1[:-2], s2
                       [:-2]) + 1)
    else:
        return min(DameraulevensteinRecursion(s1[:-1], s2)
            + 1,
                   DameraulevensteinRecursion(s1, s2[:-1]) + 1,
                   DameraulevensteinRecursion(s1[:-1], s2[:-1])
                       + f)
```

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
def damerauLevenstein(s1, s2, isPrint):
    lenI = len(s1) + 1
    lenJ = len(s2) + 1

    table = [[i + j for j in range(lenJ)] for i in range(
       lenI)]

    for i in range(1, lenI):
        for j in range(1, lenJ):
            if (s1[i - 1] == s2[j - 1]):
                f = 0
            else:
                f = 1

            table[i][j] = min(table[i - 1][j] + 1,
                              table[i][j - 1] + 1,
```

```
                            table[i - 1][j - 1] + f)

            if (i > 1 and j > 1 and s1[i - 1] == s2[j - 2]
                and s1[i - 2] == s2[j - 1]):
                table[i][j] = min(table[i][j], table[i -
                    2][j - 2] + 1)

    if isPrint:
        tablePrint(table)

    return table[-1][-1]
```

3.3 Тесты

Было организовано функциональное тестирование по принципу чёрного ящика.

Тестирование проводилось на подготовленных данных, наборы тестовых случаев полностью покрывают функциональную область. Данные тестов приведены в таблице 3.1.

test	str1	str2	Lev Rec	Dam Rec	Lev 2 str	Dam Tab
пустой			0	0	0	0
пустой		"f"	1	1	1	1
пустой	"f"		1	1	1	1
совпадающий	"asd"	"asd"	0	0	0	0
совпадающий	"f"	"f"	0	0	0	0
совпадающий	"f"	"F"	1	1	1	1
случайный	"a"	"s"	1	1	1	1
случайный	"asd"	"bsf"	2	2	2	2
случайный	"asd"	"as"	1	1	1	1
случайный	"a"	"adws"	3	3	3	3
случайный	"as"	"sa"	2	1	2	1

Таблица 3.1: Тестовые случаи с данными и результатами

Для тестирования функций по времени создаётся случайная строка, заданной длины.

Листинг 3.5: Функция генерации случайной строки

```
def takeRandomString(size):
    return ''.join(random.choice(string.ascii_letters) for
        _ in range(size))
```

3.4 Сравнительный анализ алгоритмов по памяти

Пусть на вход подаются строки длинами m и n. Учитывая специфику реализации, можно получить формулу вычисления памяти в байтах;

$$X_{matr} = ((m+1)*(n+1))*Sizeof(int) \tag{3.1}$$

Также в алгоритме используется 2 переменных под размеры n и m, 2 под циклы, сами строки m и n и 1 переменная под флаг.
Итоговый размер в байтах

$$X_{matr} = 4((m+1)*(n+1)) + m + n + 20 \tag{3.2}$$

$$2*S_{string} = 2*m*Sizeof(int) \tag{3.3}$$

Также в алгоритме используется 2 переменных под размеры n и m, 2 под циклы, сами строки m и n и 2 переменных под флаг.
Итоговый размер в байтах

$$2*S_{string} = 8*(m+1) + m + n + 24 \tag{3.4}$$

В рекурсивных алгоритмах количество занимаемой памяти зависит от глубины рекурсии. Глубина рекурсии равна m+n. Количество дополнительных переменных в рекурсивных алгоритмах Левенштейна и Дамерау-Левенштейна совпадают, поэтому памяти на них будет выделено одинаково.

$$X_{recur} = \sum_{i=0}^{m+n}(2 * S_{string} + (m + n + 2 - i) * S_{char} + c * S_i) \qquad (3.5)$$

Результаты подсчёта памяти в байтах для приведённых выше алгоритмов для различных размеров строк приведены в таблице 3.2.

str len	Levenshtain Rec	Damamerau Rec	Levenshtain 2 str	Damerau Tab
10	1270	1270	132	524
50	10350	10350	532	10524
100	30700	30700	1032	41024
500	553500	553500	5032	1005024

Таблица 3.2: Сравнение памяти, потребляемой алгоритмами

4 Исследовательская часть

В данной главе исследуются временные показатели для рекурсивных и нерекурсивных алгортитмов Левенштейна и Дамерау-Левенштейна.

4.1 Результаты временных тестов

Был проведен замер времени работы каждого из алгоритмов. Результаты замеров в секундах приведены в таблице 4.1.

str len	Levenshtain Rec	Damamerau Rec	Levenshtain 2 str	Damerau Tab
7	0.05953125	0.06006258	0.00011573	0.0001696
8	0.31359387	0.34478142	0.00015625	0.00029921
9	1.70062500	1.88374617	0.00031250	0.00033949
10	9.67343751	11.82792422	0.00043863	0.00483723
11	53.6781259	56.38492565	0.00051859	0.00054846

Таблица 4.1: Сравнение времени работы алгоритмов

4.2 График зависимости времени от длины строки

Данные из таблицы 4.1 для наглядности представим в виде графика рисунок 4.1.

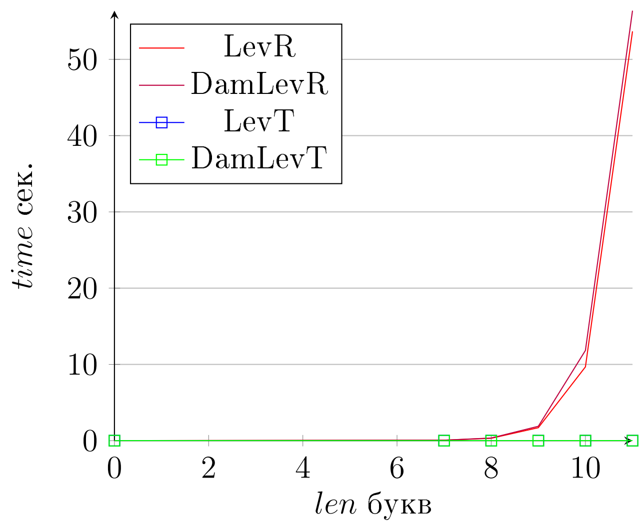

Рис. 4.1: Сравнение времени работы алгоритмов

4.3 Вывод по полученым данным

Рекурсивные реализации сравнимы по времени между собой. При увеличении длины строк становится очевидна выигрышность по времени матричного варианта. Уже при длине в 7 символов матричная реализация в 600 раз быстрее.

Заключение

Был изучен метод динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна. Также изучены алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками, получены практические навыки раелизации указанных алгоритмов в матричной и рекурсивных версиях.

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработаного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк.

В результате исследований я пришел к выводу, что матричная реализация данных алгоритмов выигрывает по времени при росте длины строк.

Список литературы

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

2. Гасфилд. Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология. Невский Диалект БВХ-Петербург, 2003.

3. R. A. Wagner, M. J. Fischer. The string-to-string correction problem. J. ACM 21 1 (1974). P. 168—173

4. Функция process_time() модуля time в Python [Электронный ресурс]. – Режим доступа:https://docs-python.ru/standart-library/modul-time-python/funktsija-process-time-modulja-time/(дата обращения 25.09.21)