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»

ДИСЦИПЛИНА «Анализ алгоритмов»

Лабораторная работа № 1

Тема Расстояние Левенштейна

Студент Воякин А. Я.

Группа ИУ7-54Б

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

Москва.
2020 г.

Оглавление

Введение

Расстояние Левенштейна - согласно [4] - минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для:

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

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

Задачами лабораторной работы являются:

- изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
- реализация рекурсивной и динамической вариации указанных алгоритмов;
- тестирование реализованных алгоритмов;
- проведение сравнительного анализа алгоритмов по затрачиваемым ресурсам (времени и памяти).

1 Аналитическая часть

1.1 Описание алгоритмов

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). Полное определение рассмотрено в [1].

Действия обозначаются так:

- D (англ. delete) — удалить;
- I (англ. insert) — вставить;
- 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 [3]:

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

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]:

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

Недостатком использования рекуррентных формул для измерения редакционного расстояния являются повторные вчисления. Решением данного недостатка является использование матричного алгоритма. Для хранения используется матрица размером (len(S1) + 1 x len(S2) + 1).

1.2 Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который является модификаций первого, учитывающего возможность перестановки соседних символов.

2 Конструкторская часть

2.1 Техническое задание

Ввод:

- на вход подаются две строки;
- строки могут быть пустыми, содержать пробелы, а также любые печатные символы UTF-8;
- uppercase и lowercase буквы 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 и память, затраченные каждым алгоритмом.

2.2 Разработка алгоритмов

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

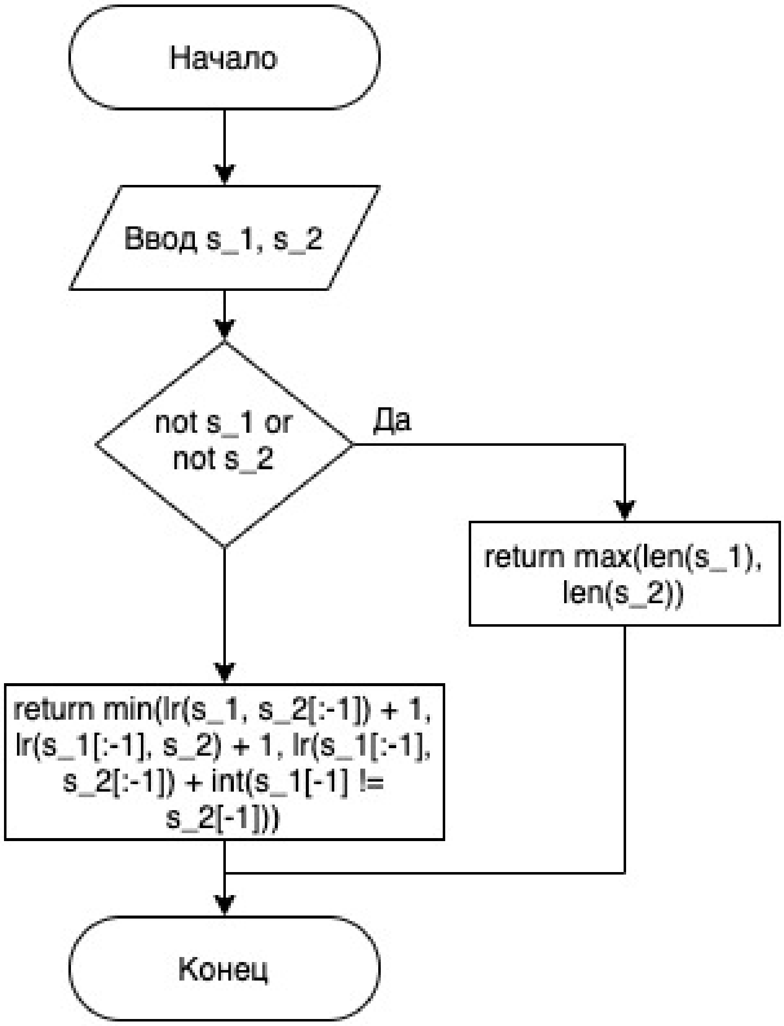

Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

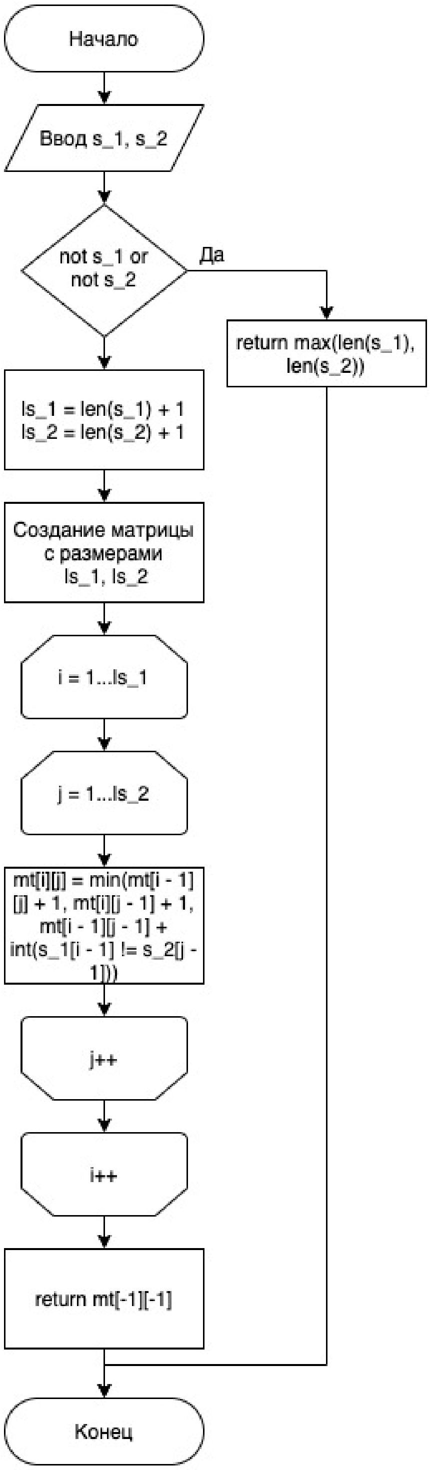

Рис. 2.2: Схема матричного алгоритма нахождения расстояния Левенштейна

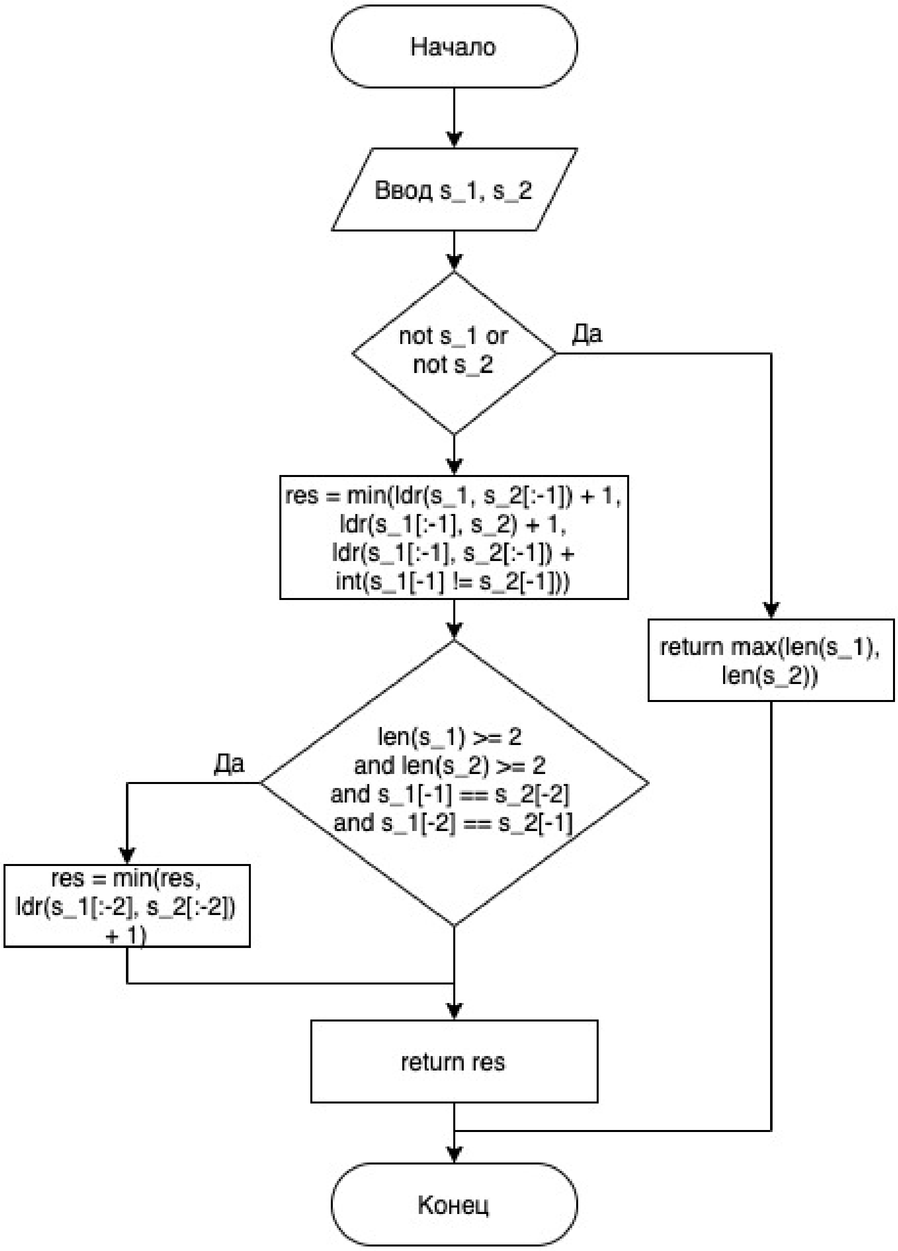

Рис. 2.3: Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

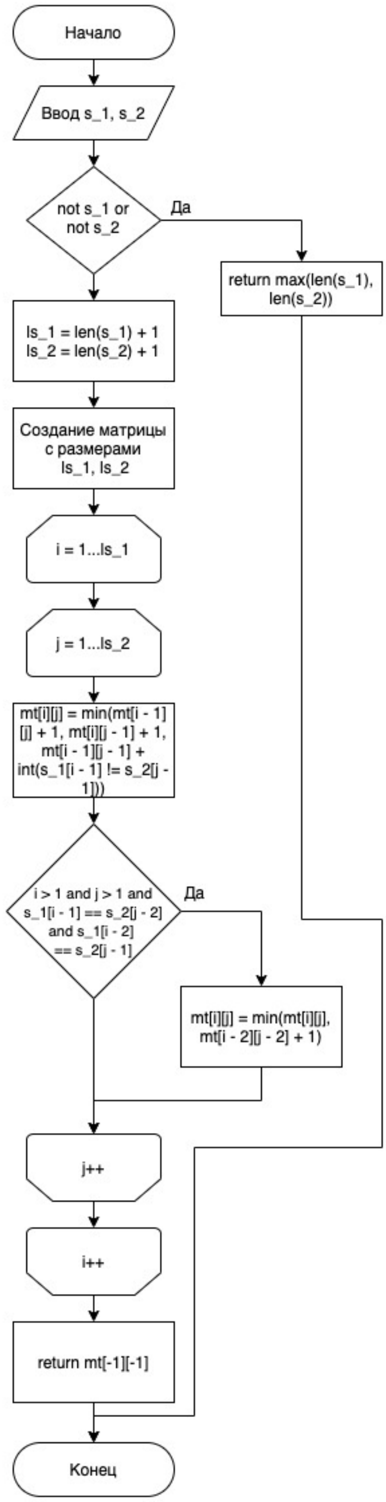

Рис. 2.4: Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

3 Технологическая часть

3.1 Выбор ЯП

Для реализации программы был выбран Python из-за наличия опыта разработки на данном языке программирования. Среда разработки - PyCharm.

3.2 Реализация алгоритма

```
def lr(s_1, s_2):
    if not s_1 or not s_2:
        return max(len(s_1), len(s_2))
    return min(lr(s_1, s_2[:-1]) + 1,
                       lr(s_1[:-1], s_2) + 1,
                       lr(s_1[:-1], s_2[:-1]) + int(s_1[-1]
                          != s_2[-1]))
```

Листинг 3.1: Функция нахождения расстояния Левенштейна рекурсивно

```
def lm(s_1, s_2, return_matrix=False):
    if not s_1 or not s_2:
        return max(len(s_1), len(s_2))
    ls_1 = len(s_1) + 1
    ls_2 = len(s_2) + 1
    mt = [[i + j for j in range(ls_2)] for i in range(ls_1)
        ]
    for i in range(1, ls_1):
        for j in range(1, ls_2):
            mt[i][j] = min(mt[i - 1][j] + 1, mt[i][j - 1] +
                 1, mt[i - 1][j - 1] + int(s_1[i - 1] != s_2
                [j - 1]))
    if return_matrix:
        return mt
```

```
    return mt[-1][-1]
```

Листинг 3.2: Функция нахождения расстояния Левенштейна матрично

```
def ldr(s_1, s_2):
    if not s_1 or not s_2:
        return max(len(s_1), len(s_2))
    res = min(ldr(s_1, s_2[:-1]) + 1, ldr(s_1[:-1], s_2) +
       1, ldr(s_1[:-1], s_2[:-1]) + int(s_1[-1] != s_2[-1])
       )
    if len(s_1) >= 2 and len(s_2) >= 2 and s_1[-1] == s_2
       [-2] and s_1[-2] == s_2[-1]:
        res = min(res, ldr(s_1[:-2], s_2[:-2]) + 1)
    return res
```

Листинг 3.3: Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
def ldm(s_1, s_2, return_matrix=False):
    if not s_1 or not s_2:
        return max(len(s_1), len(s_2))
    ls_1 = len(s_1) + 1
    ls_2 = len(s_2) + 1
    mt = [[i + j for j in range(ls_2)] for i in range(ls_1)
       ]
    for i in range(1, ls_1):
        for j in range(1, ls_2):
            mt[i][j] = min(mt[i - 1][j] + 1, mt[i][j - 1] +
                 1, mt[i - 1][j - 1] + int(s_1[i - 1] != s_2
                [j - 1]))
            if i > 1 and j > 1 and s_1[i - 1] == s_2[j - 2]
                 and s_1[i - 2] == s_2[j - 1]:
                mt[i][j] = min(mt[i][j], mt[i - 2][j - 2] +
                     1)
    if return_matrix:
        return mt
    return mt[-1][-1]
```

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
def mt_print(mt):
    if type(mt) == list:
        for row in mt:
```

```
            print(' '.join(map(str, row)))
    else:
        print("Matrix not used.")
```

Листинг 3.5: Функция вывода матрицы на экран

```
print("| len |    LevRec    |    LevMat    |    LevDamRec    |
   LevDamMat    |")
for i in range(1, 8):
    s_1 = ''.join(sample(ascii_letters, i))
    s_2 = ''.join(sample(ascii_letters, i))
    lr_time_arr = []
    lm_time_arr = []
    ldr_time_arr = []
    ldm_time_arr = []
    for _ in range(1000):
        lr_time_arr.append(cpu_time(alg.lr, s_1, s_2))
        lm_time_arr.append(cpu_time(alg.lm, s_1, s_2))
        ldr_time_arr.append(cpu_time(alg.ldr, s_1, s_2))
        ldm_time_arr.append(cpu_time(alg.ldm, s_1, s_2))
    print("%5d" % i, "%12d" % int(sum(lr_time_arr) / len(
       lr_time_arr)),
            "%12d" % int(sum(lm_time_arr) / len(lm_time_arr
               )),
            "%15d" % int(sum(ldr_time_arr) / len(
               ldr_time_arr)),
            "%15d" % int(sum(ldm_time_arr) / len(
               ldm_time_arr)))
```

Листинг 3.6: Измерение процессорного времени выполнения алгоритмов

```
def cpu_time(func, s_1, s_2):
    start = process_time_ns()
    func(s_1, s_2)
    end = process_time_ns()
    return end - start
```

Листинг 3.7: Функция замера процессорного времени

4 Исследовательская часть

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведен замер времени работы каждого из алгоритмов. Для замера времени генерировались две различные строки необходимой длины. Результаты представленные в таблице 4.1 получены усреднением значений каждого алгорима, выполненного 1000 раз при одинаковых входных данных.

Таблица 4.1: Время работы алгоритмов в нано секундах.

len	LevRec	LevMat	LevDamRec	LevDamMat
1	3222	5316	3477	5292
2	12329	9482	13058	9867
3	51963	13505	55535	14453
4	285784	22975	305489	25249
5	1503606	33575	1601602	37265
6	8650819	51708	9243470	58086
7	56342756	83139	59932134	93704

Полученная зависимость времени работы алгоритмов от длины строк показана на рисунках 4.1, 4.2.

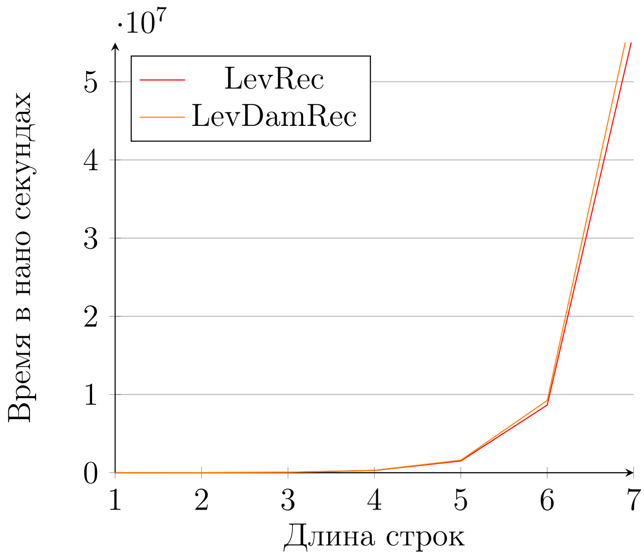

Рис. 4.1: Зависимость времени работы рекурсивных реализаций алгоритмов от длины строк

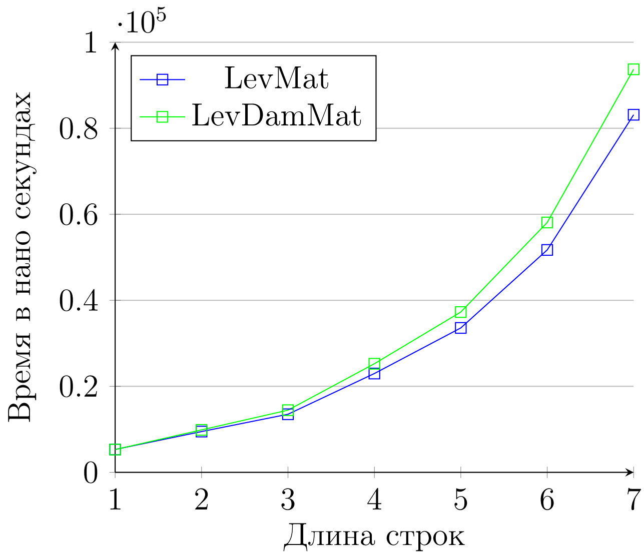

Рис. 4.2: Зависимость времени работы матричных реализаций алгоритмов от длины строк

На основе проведённых измерений можно сделать вывод, что рекурсивные алгоритмы эффективней для коротких строк. Однако при увелечении длины, динамические алгоритмы выступают более эффективными, что обусловлено большим количеством повторных рассчетов в рекурсивных реализациях, в то время как в динамических реализациях ячейка матрицы расчитывается единожды. Также установлено, что алгоритм Дамерау Левенштейна в среднем работает несколько дольше алгоритма Левенштейна, что объясняется наличием дополнительных проверок, однако алгоритмы сравнимы по временной эффективности.

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: Затрачиваемая алгоритмами память в байтах

len	LevRec	LevMat	LevDamRec	LevDamMat
1	504	244	532	244
2	2410	246	2578	246
3	11950	248	12818	248
4	61210	282	65690	282
5	321372	284	344920	284
6	1717362	286	1843194	286
7	9295242	288	9976174	288

Полученная зависимость памяти, затрачиваемой алгоритмами, от длины строк показана на рисунках 4.3, 4.4.

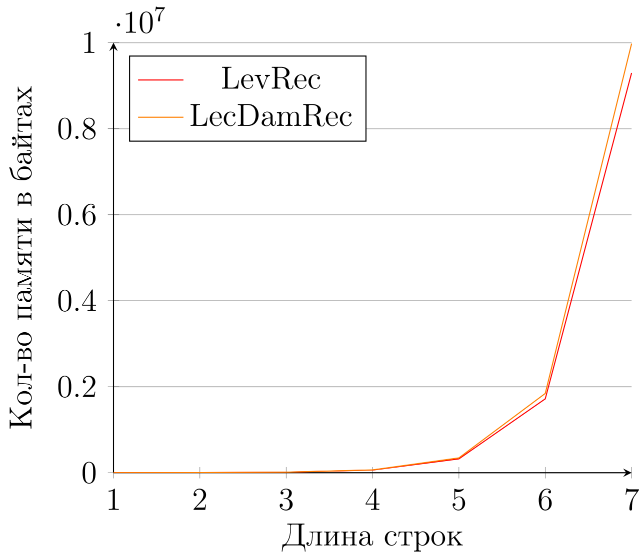

Рис. 4.3: Зависимость затрачиваемой памяти рекурсивными реализациями алгоритмов от длины строк

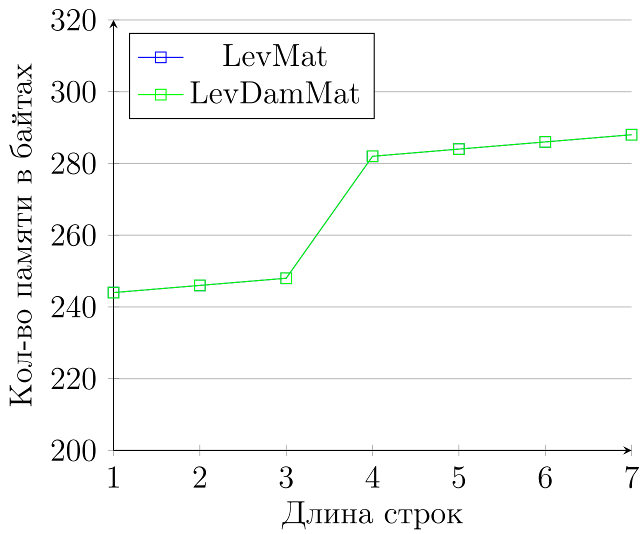

Рис. 4.4: Зависимость затрачиваемой памяти матричными реализациями алгоритмов от длины строк

На основе проведённых измерений можно сделать вывод, что рекурсивные алгоритмы сравнимы по количеству затрачиваемой памяти с динамическими при малых длинах входных строк. Однако при росте длины строк количество памяти, затрачиваемой рекурсивными алгоритмами резко возрастает из-за локальных переменных, создаваемых при каждом вызове алгоритма, в то время как память динамических алгоритмов изменяется слабо - только из-за увеличения хранимой матрицы.

4.3 Тестовые данные

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

Таблица 4.3: Таблица тестовых данных

№	Строка № 1	Строка № 1	Ожидаемый рез.	Полученный рез.
1			0 0 0 0	0 0 0 0
2	help	me	3 3 3 3	3 3 3 3
3	Im	tired	5 5 5 5	5 5 5 5
4	pen	pne	2 2 1 1	2 2 1 1
5		empty	5 5 5 5	5 5 5 5
6	123		3 3 3 3	3 3 3 3
7	Я	русский	7 7 7 7	7 7 7 7
8	еен	ене	2 2 1 1	2 2 1 1

Заключение

Были изучены методы динамического и рекурсивного программирования на примере алгоритмов Левенштейна и Дамерау-Левенштейна. Получены практические навыки реализации указанных алгоритмов в матричной и динамической реализации.

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 можно прийти к вводу, что матричная реализация данных алгоритмов заметно выигрывает по времени при росте длины строк, следовательно более применима в реальных проектах.

Список использованных источников

1. Задача о расстоянии Дамерау-Левенштейна [Электронный ресурс]. – Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=Задача-о-расстоянии-Дамерау-Левенштейна. – Дата доступа: 27.10.2020.

2. Вычисление редакционного расстояния [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/117063/. – Дата доступа: 27.10.2020.

3. Вычисление расстояния Левенштейна [Электронный ресурс]. – Режим доступа: https://foxford.ru/wiki/informatika/vychislenie-rasstoyaniya-levenshteyna. – Дата доступа: 27.10.2020.

4. Расстояние Левенштейна [Электронный ресурс]. – Режим доступа: https://vc.ru/newtechaudit/129654-rasstoyanie-levenshteyna-dlya-poiska-opechatok-v-dannyh-klienta. – Дата доступа: 27.10.2020.