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 дисциплине «Анализ алгоритмов»

Тема Расстояние Левенштейна
Студент Звягин Д.О.
Группа ИУ7-53Б
Преподаватель Волкова Л.Л., Строганов Д.В.

Москва, 2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Расстояние Левенштейна – минимальное количество редакторских операций для превращения одной строки в другую.

Под редакторской операцией понимается одна из операций следующего списка:

— вставка символа;

— удаление символа;

— замена символа;

— другие операции в вариациях алгоритма. Например, Транспозиция для алгоритма Дамерау-Левенштейна.

Несколько сфер применения алгоритма Расстояния Левенштейна:

— компьютерная лингвистика (например, исправление ошибок пользовательского ввода текста);

— биоинформатика (например, Анализ иммунитета).

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

Задачами данной лабораторной являются:

1) рассмотрение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;

2) разработка рекурсивного алгоритма расстояния Левенштейна, а также применение методов динамического программирования для разработки алгоритмов расстояний Левенштейна и Дамерау-Левенштейна с использованием кеширования;

3) экспериментальное подтверждение различий во временной эффективности рекурсивной и итерациной реализаций алгоритма определения расстояния Левенштейна с помощью замеров процессорного времени и используемой памяти при выполнении алгоритмов на варьирующихся длинах строк.

1 Аналитическая часть

1.1 Расстояние Левенштейна

Расстояние Левенштейна – между двумя строками в теории информации и компьютерной лингвистике – это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую. [6]

1.1.1 Рекурсивная формула расстояния Левенштейна

Пусть существует две последовательности символов S_1 и S_2 длины N и M соответсвенно. Ввёдем функцию $D(i, j)$, равную расстоянию левенштейна между подстроками $S_1[1...i]$ и $S_2[1...j]$.

Пусть

— совпадение символа обозначено как M;

— вставка символа обозначена как A;

— удаление символа обозначено как D;

— замена символа обозначена как R.

Тогда расстояние Левенштейна для двух строк может быть выражено следующей функцией:

$$D_1(s1[1..i], s2[1..j]) = \begin{cases} 0, & i = 0,\ j = 0, \quad M \\ j, & i = 0,\ j > 0, \quad I \\ i, & i > 0,\ j = 0, \quad D \\ min \begin{pmatrix} D_1(s1[1..i], s2[1..j-1]) + 1, & I \\ D_1(s1[1..i-1], s2[1..j]) + 1, & D \\ D_1(s1[1..i-1], s2[1..j-1]) + \begin{cases} 0, & s1_i = s2_i \quad M \\ 1, & else \qquad R \end{cases} \end{pmatrix} & \text{иначе} \end{cases} \tag{1.1}$$

Результат функции - это минимальное расстояние Левенштейна от строки 1 до строки s2, представленных в виде последовательности символов. Базой рекурсии являются ситуации, когда хотя бы одна из строк пустая. В остальных случаях выбирается наименьшее значение из трёх случаев:

— вставка символа в s1 со штрафом 1;

— удаление символа из s1 со штрафом 1;

— либо совпадение символа со штрафом 0, либо замена со штрафом 1.

Под понятием штраф имеется в виду “цена” редакторской операции, учитываемая в итоговом значении расстояния Левенштейна

1.1.2 Расстояние Левенштейна с кешированием

Рекурсивная форма может оказаться малоэффективной при больших значениях N и M, так как в ней могут постоянно пересчитываться значения $D(i, j)$ для одних и тех же аргументов. Для устранения потенциального недостатка используют итерационный алгоритм, который сохраняет промежуточные значения в виде матрицы размерности $(N + 1)x(M + 1)$.

Значения в ячейке [i, j] матрицы содержат значение $D(i, j)$, то есть расстояние левенштейна между подстроками $S_1[1...i]$ и $S_2[1...j]$. При этом нулевая строка и нулевой столбец матрицы заполняются слева-направо и сверху-вниз от 0 до соотвествующей размерности, так как приведение пустой строки к любой строке длины, скажем, i требует i операций вставки.

Далее алгоритм построчно заполняет матрицу по формуле 1.1, при этом вместо просчитывания предыдущих значений D используются значения из матрицы в соотвествующих ячейках.

Данный алгоритм может работать эффективнее по времени, однако за счёт хранения дополнительной матрицы может тратить больше памяти. Чтобы это оптимизировать, на каждой итерации хранят только текущую строку матрицы и предыдущую, так как остальные не нужны для просчёта, однако я не делал этого в своих реализациях алгоритмов.

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна – между двумя строками, состоящими из конечного числа символов – это минимальное число операций вставки, удаления, замены одного символа и транспозиции двух соседних символов, необходимых для перевода одной строки в другую. [7]

1.2.1 Рекурсивная формула расстояния Дамерау-Левенштейна

Введём также как и для расстояния Левенштейна функцию $D(s1, s2)$.

Пусть операция транспозиции обозначена как T

Тогда, с учётом транспозиции формула 1.1 преобразутся в формулу 1.2

$$D_1(s1[1..i], s2[1..j]) = \begin{cases} 0, & i = 0,\ j = 0,\ M \\ j, & i = 0,\ j > 0,\ I \\ i, & i > 0,\ j = 0,\ D \\ min \begin{pmatrix} D_1(s1[1..i], s2[1..j-1]) + 1, & I \\ D_1(s1[1..i-1], s2[1..j]) + 1, & D \\ D_1(s1[1..i-1], s2[1..j-1]) + \\ \quad \begin{cases} 0, & s1_i = s2_i \quad M \\ 1, & else \qquad R \end{cases} \end{pmatrix} & i, j > 1, \\ & S_1[i] = S_2[j-1], \\ & S_1[i-1] = S_2[j] \\ min \begin{pmatrix} D_1(s1[1..i], s2[1..j-1]) + 1, & I \\ D_1(s1[1..i-1], s2[1..j]) + 1, & D \\ D_1(s1[1..i-1], s2[1..j-1]) + \\ \quad \begin{cases} 0, & s1_i = s2_i \quad M \\ 1, & else \qquad R \end{cases} \end{pmatrix} & \text{иначе} \end{cases} \tag{1.2}$$

Фактическое отличие от алгоритма расстояния Левенштейна в том, что при j > 1 и i > 1, а также при условии что соседние символы в строках равны “накрест”, может быть проведена операция транспозиции. Транспозиция считается редакторской операцией со сложностью 1, когда в обычном алгоритме поиска расстояния Левенштейна, такая операция имела бы сложность 2

1.2.2 Расстояние Дамерау-Левенштейна с кешированием

Также как и в случае расстояния Левенштейна, рекурсивный алгоритм может быть малоэффективным, поэтому имеет место быть итерационный алгоритм с кешированием, используя матрицу, либо текущую и последние две строки матрицы. Алгоритм аналогичен алгоритму расстояния, за исключением того, что при равенстве накрест двух соседних символов строк и значению i>1 и j>1 при расчёте очередной ячейки нужно проверить ещё и вариант с транспозицией.

Вывод

В результате аналитического раздела были рассмотрены расстояния Левенштейна и Дамерау-Левенштейна, а также их рекурсивные формулы, итерационные алгоритмы с кешированием матрицами и отдельными строками матриц.

2 Конструкторская часть

2.1 Требования к программному обеспечению

К разрабатываемой программе предъявлен ряд требований:

— на вход подаются две строки – s1 и s2;

— на выход подаётся целое число;

— заглавные и строчные символы считаются разными;

— возможность обработки строк как на кириллице так и на латинице;

— должна быть возможность замера процессорного времени программы;

— должна быть возможность вывода графиков и таблиц замеров процессорного времени и памяти.

Был введён вспомогательный алгоритм инициализации таблицы (кэша) для алгоритмов с кэшем.

Для остальных алгоритов всегда подразумевается, что на вход подаётся две строки, являющиеся последовательностями символов с именами s1 и s2, а на выходе ожидается целое число

2.2 Разработка алгоритмов

Алгоритм инициализации таблицы (кэша) изображённый на рисунке 2.1

Рекурсивный алгоритм нахождения расстояния Левенштейна изображён на рисунке 2.2

Алгоритм нахождения расстояния Левенштейна с кэшем изображён на рисунке 2.3

Алгоритм нахождения расстояния Дамерау-Левенштейна с кэшем изображён на рисунке 2.4

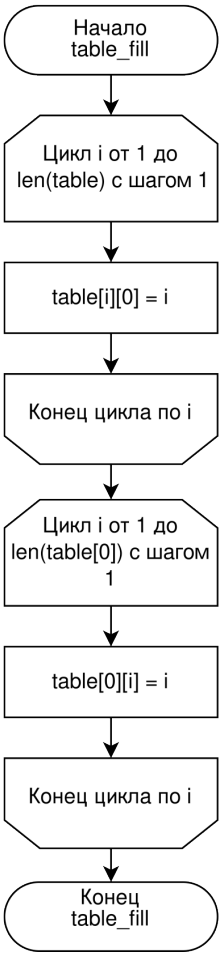

Рисунок 2.1 — Алгоритм инициализации кэш-таблицы

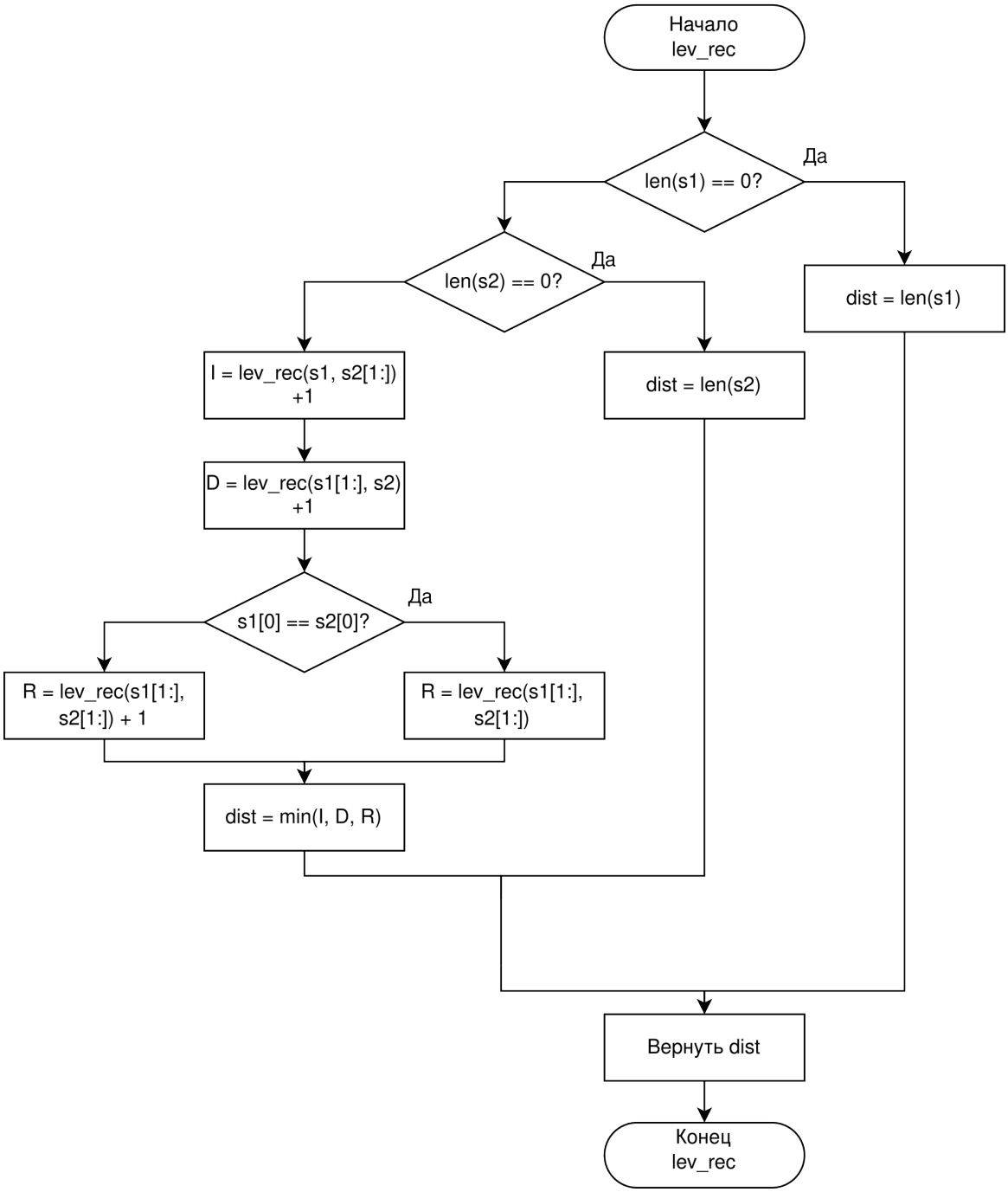

Рисунок 2.2 — Рекурсивный алгоритм нахождения расстояния Левенштейна

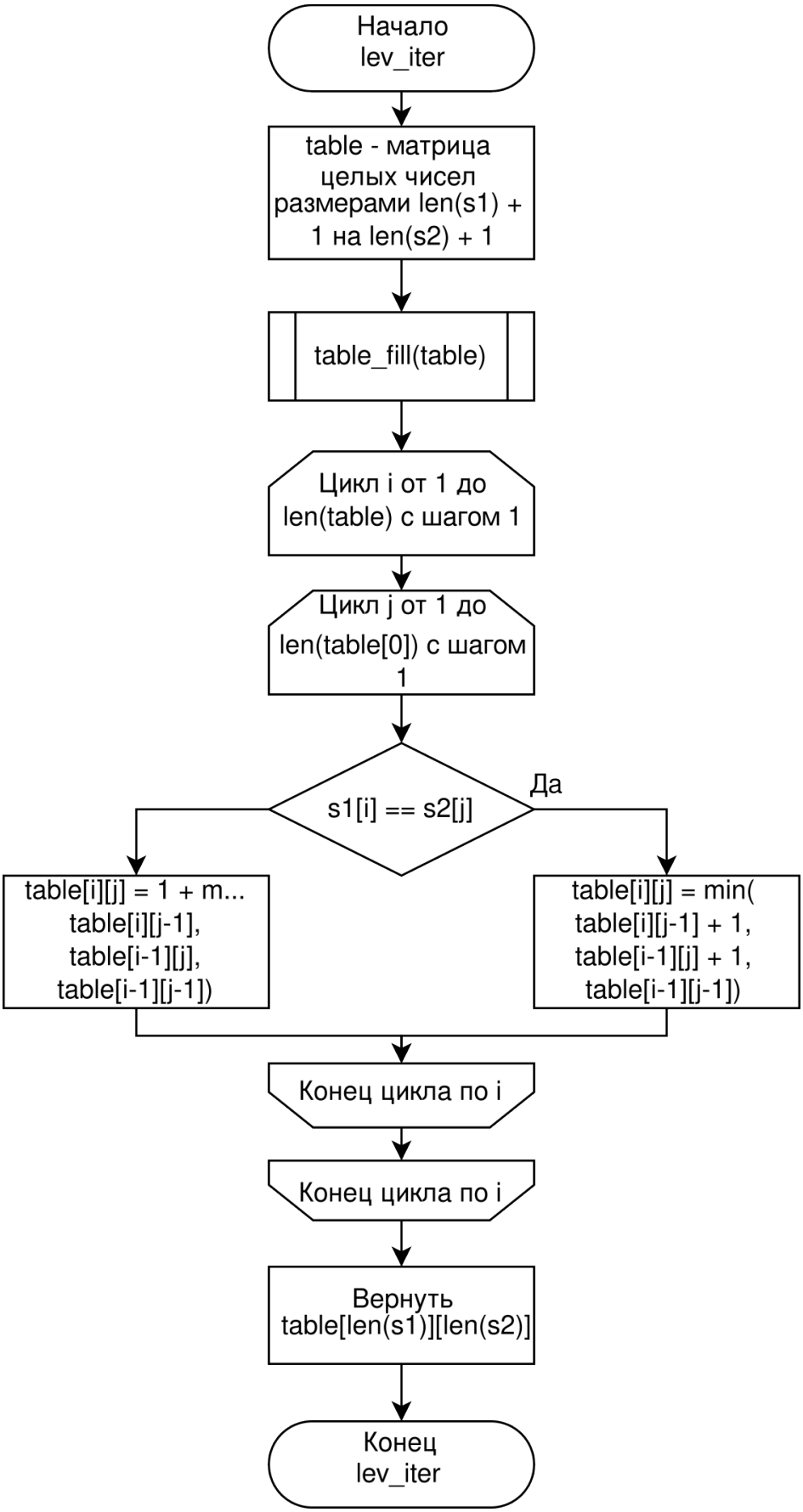

Рисунок 2.3 — Алгоритм нахождения расстояния Левенштейна с кешэм

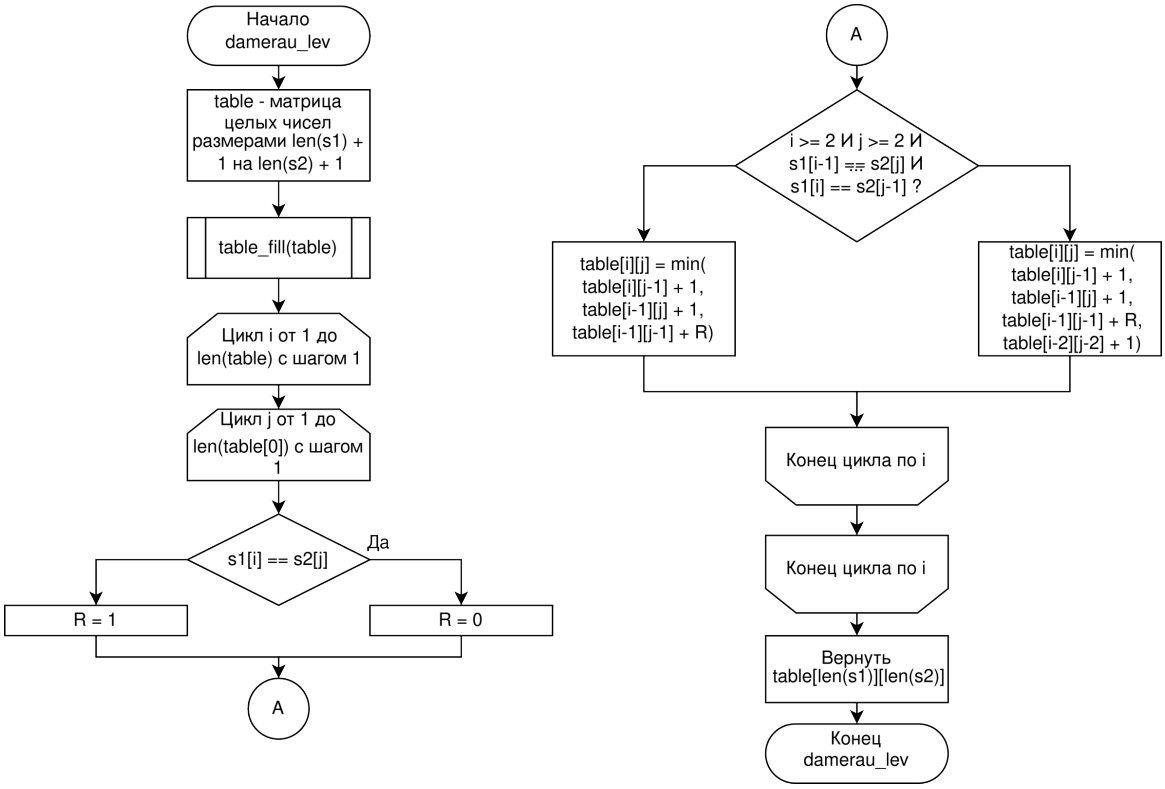

Рисунок 2.4 — Алгоритм нахождения расстояния Дамерау-Левенштейна с кешэм

2.3 Вывод

В результате конструкторской части были определены требования к ПО, а также построены схемы алгоритмов рекурсивного поиска расстояния Левенштейна, итерационного поиска с кешэм и итерационного алгоритма поиска расстояния Дамерау-Левенштейна с кешэм.

3 Технологическая часть

3.1 Средства разработки

В качестве языка программирования был выбран python3 [1], так как в его стандартной библиотеке присутсвуют функции замера процессорного времени, которые требуются в условиях, а также данный язык обладает множеством инструментов для визуализации и работы с данными и таблицами.

В качестве основного файла был выбран инструмент jupyter notebook [2], так как он позволяет организовать код в виде блоков, а также выводить данные и графики прямо в нём, что позволяет наглядно продемонстрировать все замеры.

Для построения графиков использовалась библиотека matplotlib [4].

Для замера времени использовалась функция process_time_ns из стандартного модуля time [3].

Для замеров памяти использовалась функция get_traced_memory стандартного модуля tracemalloc [5], которая получает текущую и пиковую выделенную память относительно начала замера.

3.2 Реализация алгоритмов

Листинг 3.1 — алгоритм заполнения кэш-таблицы

```
def fill_table(s1: str, s2: str) -> list[list[int]]:
    table = [
        [0] * (len(s2) + 1)
        for _ in range(len(s1) + 1)
    ]

    for i in range(len(table)):
        table[i][0] = i

    for i in range(len(table[0])):
        table[0][i] = i

    return table
```

Листинг 3.2 — Рекурсивный алгоритм нахождения расстояния Левенштейна

```
def levenstein_rec(s1: str, s2: str) -> int:
    if len(s1) == 0:
        return len(s2)

    if len(s2) == 0:
```

```
        return len(s1)

    if s1[0] == s2[0]:
        return levenstein_rec(s1[1:], s2[1:])

    return 1 + min(
        levenstein_rec(s1[1:], s2),
        levenstein_rec(s1, s2[1:]),
        levenstein_rec(s1[1:], s2[1:])
    )
```

Листинг 3.3 — Итерационный алгоритм нахождения расстояния Левенштейна с кешэм

```
from .fill_table import fill_table


def levenstein_iter(s1: str, s2: str) -> int:
    table = fill_table(s1, s2)

    for i in range(1, len(table)):
        for j in range(1, len(table[0])):
            table[i][j] = min(
                table[i-1][j] + 1,
                table[i][j-1] + 1,
                table[i-1][j-1] + (int(s1[i-1] != s2[j-1]))
            )

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

Листинг 3.4 — Итерационный алгоритм нахождения расстояния Дамерау-Левенштейна с кешэм

```
from .fill_table import fill_table


def damerau_levenstein(s1: str, s2: str) -> int:
    table = fill_table(s1, s2)

    for i in range(1, len(table)):
        for j in range(1, len(table[0])):
            # check if swap is possible
            if i >= 2 and j >= 2 and s1[i-1] == s2[j-2] and s1[i-2] == s2[
               j-1]:
                table[i][j] = min(
                    table[i-1][j] + 1,
```

```
                    table[i][j-1] + 1,
                    table[i-1][j-1] + (int(s1[i-1] != s2[j-1])),
                    table[i-2][j-2] + 1  # swap
                )
            else:
                table[i][j] = min(
                    table[i-1][j] + 1,
                    table[i][j-1] + 1,
                    table[i-1][j-1] + (int(s1[i-1] != s2[j-1]))
                )

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

3.3 Функциональные тесты

В таблице 3.1 приведены тесты для алгоритмов нахождения расстояния Левенштейна и Дамерау Левенштейна.

Таблица 3.1 — Функциональные тесты

Входные данные		**Алгоритм и расстояния**				
		Левенштейна			Дамерау-Левенштейна	
Строка 1	Строка 2	Рекурсивный	С кешем	Ожидаемое	С кешем	Ожидаемое
lorem ipsum	dolor sit amet	10	10	10	10	10
lorem ipsum		11	11	11	11	11
	consectetur	11	11	11	11	11
LaTeX	latex	3	3	3	3	3
latex	altex	2	2	2	1	1
latex	laTxe	3	3	3	2	2
арбуз	автобус	4	4	4	4	4

Все тесты пройдены успешно

Вывод

В ходе работы были разработаны алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна, а также проведено их тестирование.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором проводились замеры:

— операционная система: EndeavourOS x86_64;

— процессор: 13th Gen Intel(R) Core(TM) i53500H (16) С частотой 4.70 ГГц;

— оперативная память: 16 ГБ с частотой 5200 МГц.

При проведении замеров ноутбук был подключен к сети и переведён в режим максимальной производительности. Запущена была только система и Jupyter Notebook

Для сравнения характеристик алгоритмов, на вход алгоритмам подаются две случайные строки одинаковой длины. Длины строк приведены в соответствующих таблицах.

4.2 Временные характеристики

В таблице 4.1 приведены временные характеристики, полученные в результате замеров времени всех трёх алгоритмов.

Для каждой длины замер проводился 50 раз, при этом для каждой итерации создавались 2 случайные строки заданной длины и использовались как аргументы к алгоритмам. Итоговое время взято как среднее арифметическое всех полученных замеров.

Рекурсивный алгоритм нахождения расстояния Левенштейна работает очень долго уже при длине строки равной 12, поэтому замеры времени для строк б*о*льшей длины проводились только на алгоритмах с кэшированием. знак — в таблице значит отсутствие данных.

Таблица 4.1 — Таблица зависимости времени обработки строк от длины строк

Длина	Рек. алгоритм (нс)	Итер. алгоритм (нс)	Дамерау-Левенштейна (нс)
1	1 545	1 893	1 414
2	2 981	2 282	10 693
3	7 267	3 279	3 333
4	40 702	4 412	4 807
5	211 776	5 936	6 818
6	1 142 547	8 848	9 103
7	7 755 973	16 490	14 253
8	28 786 587	19 901	15 577
9	132 354 547	20 192	18 618
10	868 123 744	23 266	22 291
11	3 564 314 047	29 834	26 889
12	27 435 792 490	33 648	31 348
13	—	33 332	35 821

14	—	35 886	40 974
15	—	39 146	45 935
16	—	45 502	52 078
17	—	49 880	59 384
18	—	55 522	66 703
19	—	61 686	73 413
20	—	68 260	78 292
21	—	74 541	124 237
22	—	81 820	95 959
23	—	87 478	102 833
24	—	94 959	111 450
25	—	102 708	122 344
26	—	110 758	132 019
27	—	119 178	140 822
28	—	128 560	151 071
29	—	136 739	160 726
30	—	147 645	175 105
31	—	155 039	182 966
32	—	167 214	197 397
33	—	176 132	210 144
34	—	188 196	221 512
35	—	203 662	240 423
36	—	211 826	248 477
37	—	222 059	263 062
38	—	232 068	276 508
39	—	246 967	293 653
40	—	259 247	305 639
41	—	274 170	324 889
42	—	286 795	340 290
43	—	297 447	351 554
44	—	312 274	372 037
45	—	332 470	393 392
46	—	342 904	406 540
47	—	359 710	426 905
48	—	372 541	442 847
49	—	385 227	457 590
50	—	402 756	477 728

Полученные замеры также визуализированы на графике 4.1:

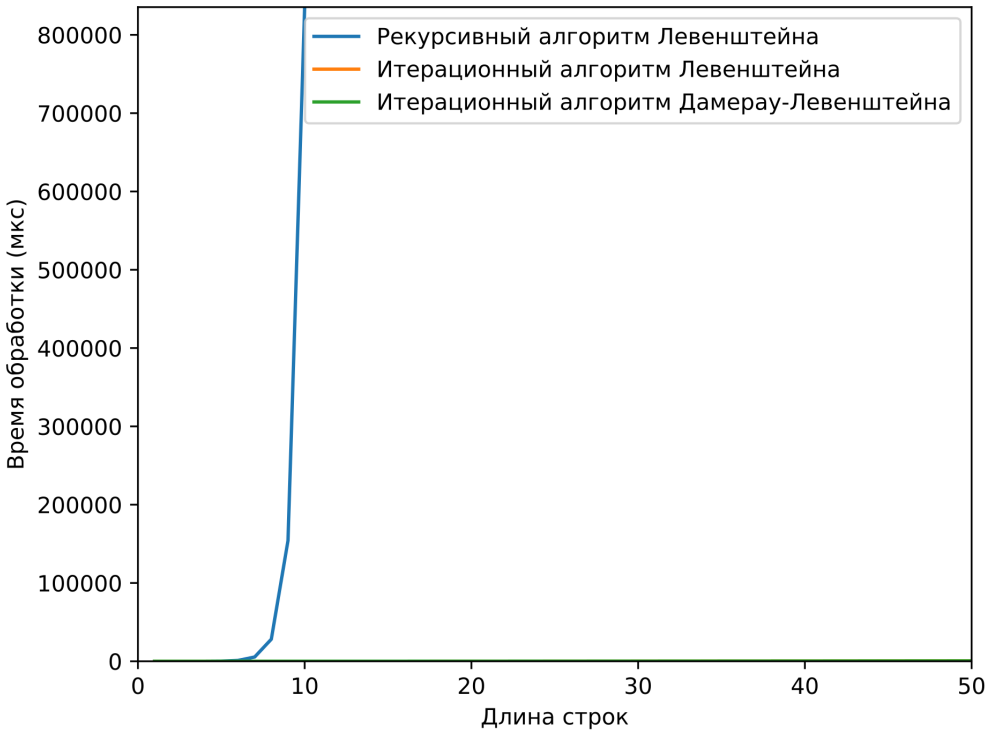

Рисунок 4.1 — График зависимости времени работы алгоритмов от длины строк

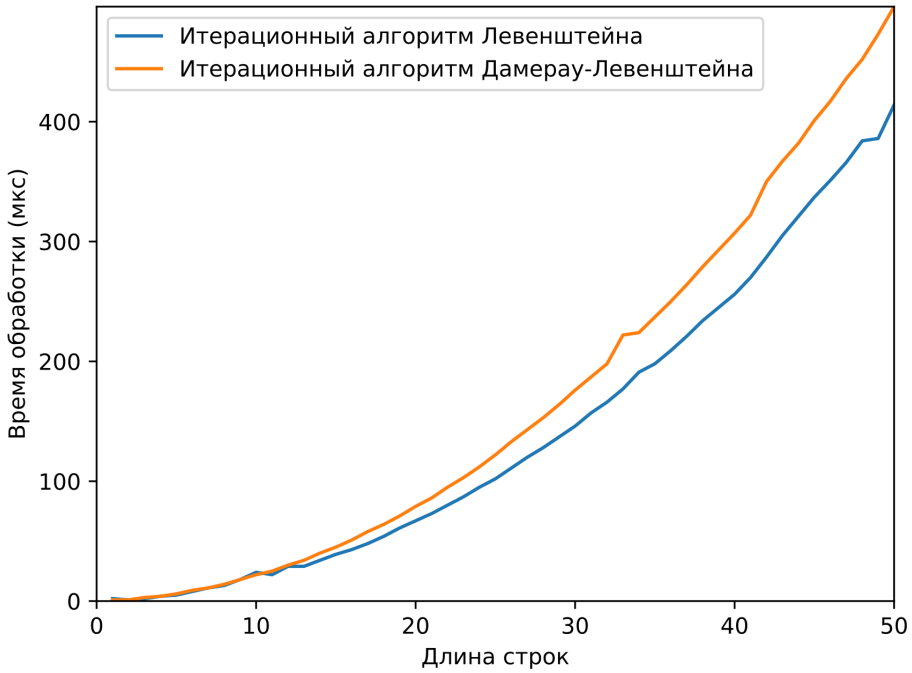

Рисунок 4.2 — График зависимости времени работы алгоритмов от длины строк без рекурсивного алгоритма Левенштейна

Как видно из таблицы 4.1 и графика 4.1, рекурсивный алгоритм существенно уступает

итерационному по временным характеристикам.

Чтобы наглядно сравнить временные характеристики итерационных алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна, я также построил график 4.2.

Из таблицы 4.1 и графика 4.2 также видно, что итерационный алгоритм Дамерау-Левенштейна стабильно хуже по временным характеристикам, чем итерациный алгоритм Левенштейна.

4.2.1 Замеры с платы

Замеры проводились на плате со следующими техническими характеристиками:

— модель – STM32F767

— процессор – Arm-Cortex-M7 с частотой 216 МГц

— операцивная память – 512 КБ

Результаты замеров можно увидеть в таблице 4.2

Таблица 4.2 — Таблица зависимости времени обработки строк от длины строк на плате

Длина	Рек. алгоритм (мс)	Итер. алгоритм (мс)	Дамерау-Левенштейна (мс)
1	0	0	0
2	0	1	0
3	2	0	1
4	11	1	0
5	61	1	1
6	184	1	2
7	1 527	2	2
8	8 359	2	2

4.3 Емкостные характеристики

В таблице 4.3 приведены емкостные характеристики, полученные в результате замеров памяти рекурсивного и итерационного алгоритмов с помощью модуля tracemalloc. Так как для рекурсивной реализации размер выделяемой памяти существенно зависит от глубины рекурсии, то есть от входных строк, то для каждой длины строки для рекурсивного алгоритма алгоритм был проведен 50 раз на случайных строках. Для итерационного алгоритма выделяемая память не зависит от входных строк(при одинаковой длине), так как размер матрицы кеша зависит только от размеров строк. За характеристику памяти взято среднее арифметическое значение использованной памяти.

Таблица 4.3 — Таблица зависимости объёма использованной памяти при обработке строк от их длины

Длина строки	Рек. Алгоритм (Б)	Итер. Алгоритм (Б)
1	47	192
2	48	232
3	253	288
4	251	392
5	398	480
6	477	584
7	568	704
8	664	904
9	832	1056
10	1180	1224

Также зависимость из таблицы отражена на графике 4.3.

Как видно из графика 4.3 и таблицы 4.3, в среднем, рекурсивная реализация выигрывает итерационную по ёмкостным характеристикам.

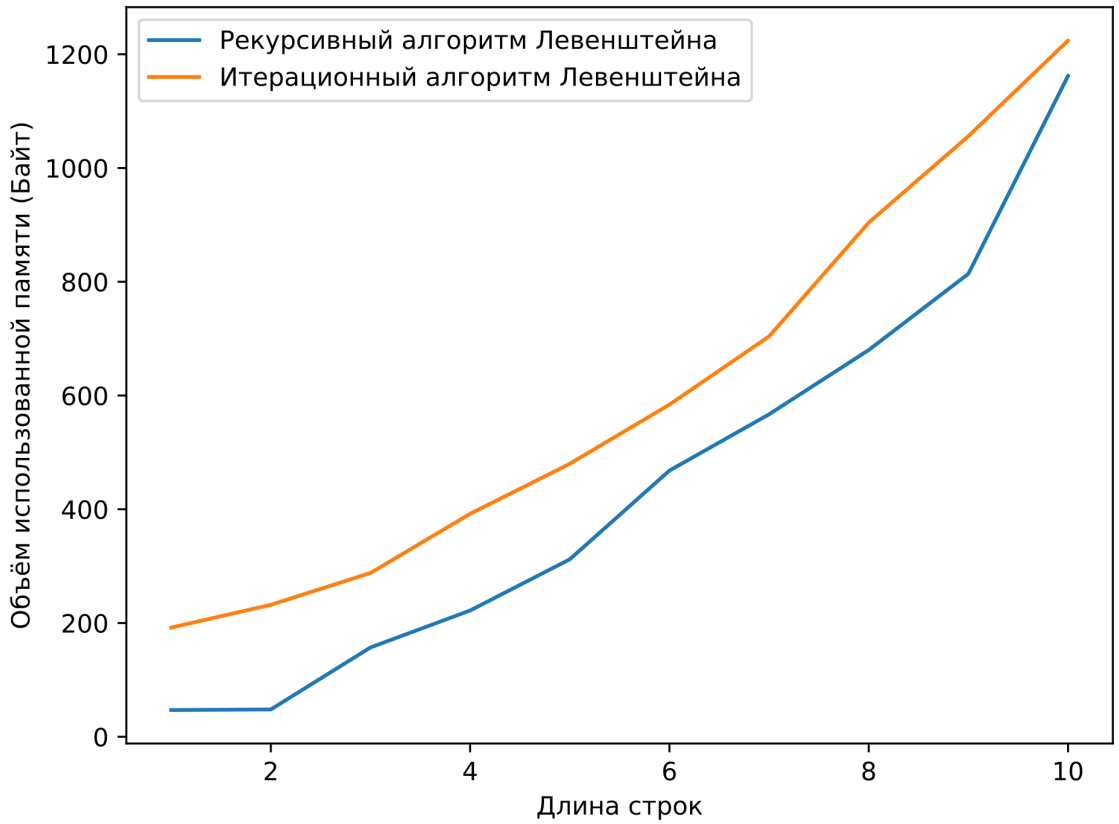

Рисунок 4.3 — График зависимости требуемой памяти для работы алгоритмов от длины входных строк

4.4 Вывод

В данном разделе было проведено исследование временных и емкостных характеристик алгоритмов рекурсивного поиска расстояния Левенштейна и итерационного поиска расстояние Левенштейна с кешем.

По результатам исследования оказалось, что рекурсивная реализация существенно проигрывает итерационной по времени, однако в отношении ёмкостных характеристик ситуация противоположная - в большинстве случаев, рекурсивная реализация эффективнее итерационной.

ЗАКЛЮЧЕНИЕ

По результатам исследования оказалось, что рекурсивная реализация алгоритма нахождения расстояния Левенштейна существенно проигрывает итерационной по временным характеристикам, однако в отношении ёмкостных характеристик ситуация противоположная - в большинстве случаев, рекурсивная реализация эффективнее итерационной.

В ходе данной лабораторной работы были выполнены следующие задачи:

1) были рассмотрены алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;

2) были разработаны:

— рекурсивный алгоритм нахождения расстояния Левенштейна;

— алгоритм нахождения расстояния Левенштейна с кэшированием;

— алгоритм нахождения расстояния Дамерау-Левенштейна с кэшированием.

3) были экспериментально подтверждены различия во временной эффективности рекурсивной и итерациной реализаций алгоритма определения расстояния Левенштейна с помощью замеров процессорного времени и используемой памяти при выполнении алгоритмов на варьирующихся длинах строк.

Для разработки алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна с кэшированием, были изучены и применены методы динамического программирования.

Цель и задачи лабораторной работы были выполнены.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Документация Языка программирования Python / [Электронный ресурс] // Python : [сайт]. — URL: https://www.python.org/ (дата обращения: 25.09.2024).

2. Документация Jupyter Notebook: The Classic Notebook Interface / [Электронный ресурс] // Jupyter : [сайт]. — URL: https://jupyter.org/ (дата обращения: 25.09.2024).

3. Документация библиотеки time — Time access and conversions / [Электронный ресурс] // Python 3.12.6 documentation : [сайт]. — URL: https://docs.python.org/3/library/time.html#time.process_time_ns (дата обращения: 25.09.2024).

4. Документация библиотеки Matplotlib: Visualization with Python / [Электронный ресурс] // Matplotlib : [сайт]. — URL: https://matplotlib.org/ (дата обращения: 25.09.2024).

5. Документация библиотеки tracemalloc — Trace memory allocations / [Электронный ресурс] // Python 3.12.6 documentation : [сайт]. — URL: https://docs.python.org/3/library/tracemalloc.html#tracemalloc.get_traced_memory (дата обращения: 25.09.2024).

6. Романовский И.В. Дискретный анализ: Учебное пособие для студентов, специализирующихся по прикладной маткматике и информатике. // 4-е издание, исправленное и дополненное. // СПб.: Невский диалект; БХВ-Петербург, 2008. // 336 страниц

7. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн Алгоритмы: построение и анализ // 3-е изд. // М.: «Вильямс», 2013. // 440 страниц