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 и Дамерау–Левенштейна

Студент Куликов Е. А.

Группа ИУ7-56Б

Преподаватель Волкова Л. Л.

Москва — 2024 г.

Содержание

Введение

В данной рабораторной работе изучается расстояние Левенштейна. Расстояние Левенштейна — минимальное количество операций (вставки, удаления, замены), которое необходимо для перевода одной строки в другую. Это расстояние помогает определить, насколько «близки» две строки.

Расстояние Левенштейна применяется в теории информации, компьютерной лингвистике и биоинформатике для:

— автозамены(исправления опечаток) и автозаполнения;

— сравнения текстовых файлов утилитой diff, например для определения различий между двумя файлами в Linux или разницы между двумя Git-деревьями;

— сравнения цепочек ДНК.

Целью данной лабораторной работы является изучение и реализация алгоритмов определения расстояния Левенштейна (итерационного, рекурсивного и рекурсивного с кэшированием) и Дамерау–Левенштейна(только итерационного). Для достижения поставленной цели необходимо решить следующие задачи:

— изучить итеративный и рекурсивный методы вычисления расстояния Левенштейна и Дамерау–Левенштейна;

— реализовать указанные алгоритмы поиска расстояния Левенштейна и Дамерау–Левенштейна;

— провести сравнение эффективности указанных алгоритмов по затраченному процессорному времени и по затраченной памяти;

— описать и обосновать полученные результаты в отчете о выполненной лабораторной работе.

1 Аналитическая часть

В данном разделе будут рассмотрены алгоритмы нахождения расстояний Левенштейна и Дамерау–Левенштейна.

1.1 Расстояние Левенштейна

Расстояние Левенштейна [1] — это минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую.

Указанные операции имеют цену (штраф)[4], при этом в общем случае цены операций могут не совпадать. В этой работе будут рассматриваться алгоритмы, в которых цены редакторских операций одинаковы и равны 1. В общем случае (λ означает пустую строку).

— $w(a, b)$ — цена замены символа a на b, R (от англ. replace);

— $w(\lambda, b)$ — цена вставки символа b, I (от англ. insert);

— $w(a, \lambda)$ — цена удаления символа a, D (от англ. delete).

1.2 Вычисление Расстояния Левенштейна

Для решения задачи о расстоянии Левенштейна находится последовательность операций, минимизирующая суммарную цену. В данной лабораторной работе рассматривается частный случай поиска этого расстояния при:

— $w(a, a) = 0$;

— $w(a, b) = 1$, $a \neq b$;

— $w(\lambda, b) = 1$;

— $w(a, \lambda) = 1$.

Имеется две строки S_1 и S_2, длинной M и N соответственно. Расстояние Левенштейна рассчитывается по рекуррентной формуле (1.1):

$D(S_1, S_2) = D(M, N)$, где

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

где функция $m(a, b)$ определена как:

$$m(a,b) = \begin{cases} 0 & \text{если a} = \text{b}, \\ 1 & \text{иначе} \end{cases}. \tag{1.2}$$

Рекурсивный алгоритм реализует формулу (1.1). Функция D составлена таким образом, что для перевода из строки a в строку b требуется выполнить последовательно несколько операций (удаление, вставка, замена) в некоторой последовательности. Полагая, что a', b' — строки a и b без последнего символа соответственно, цена преобразования из строки a в строку b может быть выражена как:

— сумма цены преобразования строки a' в b' и цены проведения операции удаления, которая необходима для преобразования a' в a;

— сумма цены преобразования строки a' в b' и цены проведения операции вставки, которая необходима для преобразования b' в b;

— сумма цены преобразования из a' в b' и операции замены, предполагая, что a и b оканчиваются на разные символы;

— цена преобразования из a' в b', предполагая, что a и b оканчиваются на один и тот же символ.

Ценой преобразования будет минимальное значение приведенных вариантов.

1.3 Расстояние Дамерау–Левенштейна

Расстояние Дамерау–Левенштейна — минимальное число операций вставки, удаления, замены и транспозиции соседних символов. То есть, по сравнению с расстоянием Левенштейна добавляется еще одна редакторская операция — транспозиция T (от англ. transposition).

1.4 Вычисление расстояния Дамерау–Левенштейна

Расстояние Дамерау–Левенштейна может быть вычисленно по рекуррентной формуле (1.3):

$$D(i,j) = \begin{cases} 0 & ,j = 0, i = 0 \\ i & ,j = 0, i > 0 \\ j & ,j > 0, i = 0 \\ min \begin{cases} D(i,j-1)+1 \\ D(i-1,j)+1 \\ D(i-1,j-1)+ \\ \qquad m(S_1[i],S_2[i]) \\ D(i-2,j-2)+ \\ \qquad m(S_1[i],S_2[i]) \end{cases} & \begin{array}{l}\text{, если i} > 1\text{, j} > 1 \\ ,S_1[i] = S_2[j-1] \\ ,S_1[j] = S_2[i-1]\end{array} \\ min \begin{cases} D(i,j-1)+1 \\ D(i-1,j)+1 \\ D(i-1,j-1)+ \\ \qquad m(S_1[i],S_2[i]) \end{cases} & \text{,иначе} \end{cases} \tag{1.3}$$

Вывод

В данном разделе были рассмотрены расстояния Левенштейна и Дамерау–Левенштейна. Формулы для нахождения этих расстояний, а следовательно, алгоритмы могут быть реализованы рекурсивно и итерационно.

2 Конструкторская часть

В данном разделе будут разработаны алгоритмы нахождения расстояний Левенштейна и Дамерау–Левенштейна.

2.1 Разработка алгоритмов

На вход алгоритмов падаются строки s1 и s2, которые могут содержать как русские, так и английские буквы, на выходе получаем едиственное число – искомое расстояние.

На рис. 2.1 — 2.4 приведены схемы рекурсивных и итерационных алгоритмов Левенштейна и Дамерау–Левенштейна.

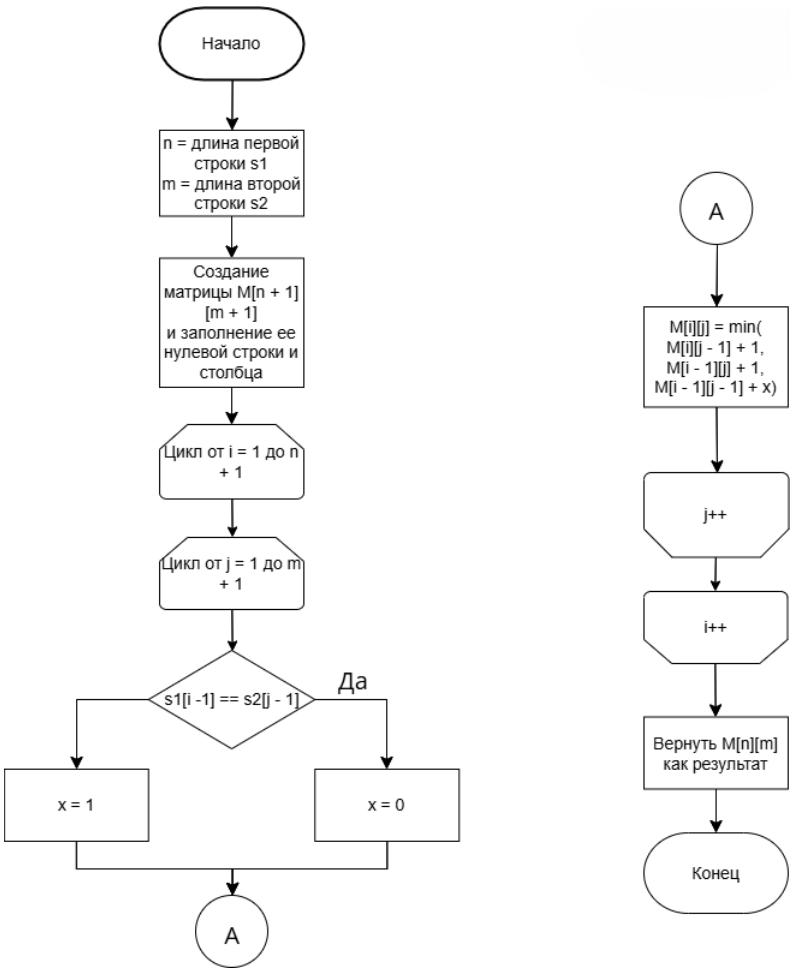

Рисунок 2.1 – Итеративный алгоритма нахождения расстояния Левенштейна

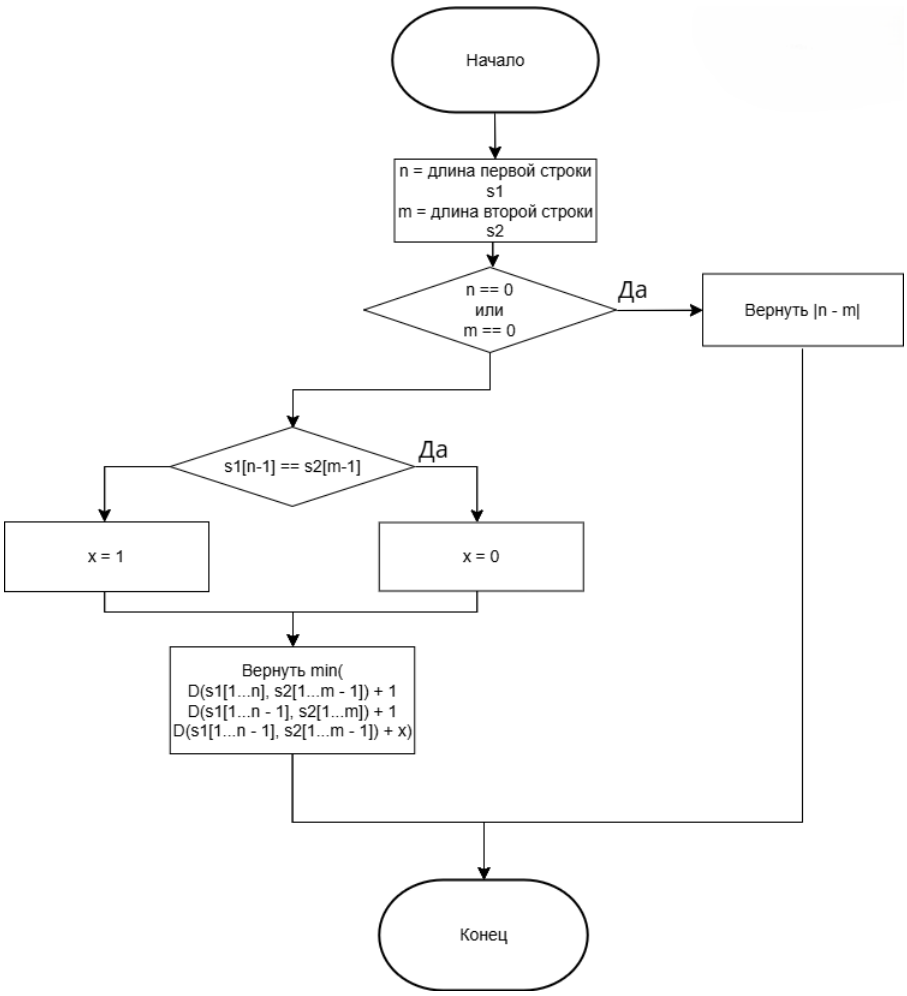

Рисунок 2.2 – Рекурсивный алгоритм нахождения расстояния Левенштейна

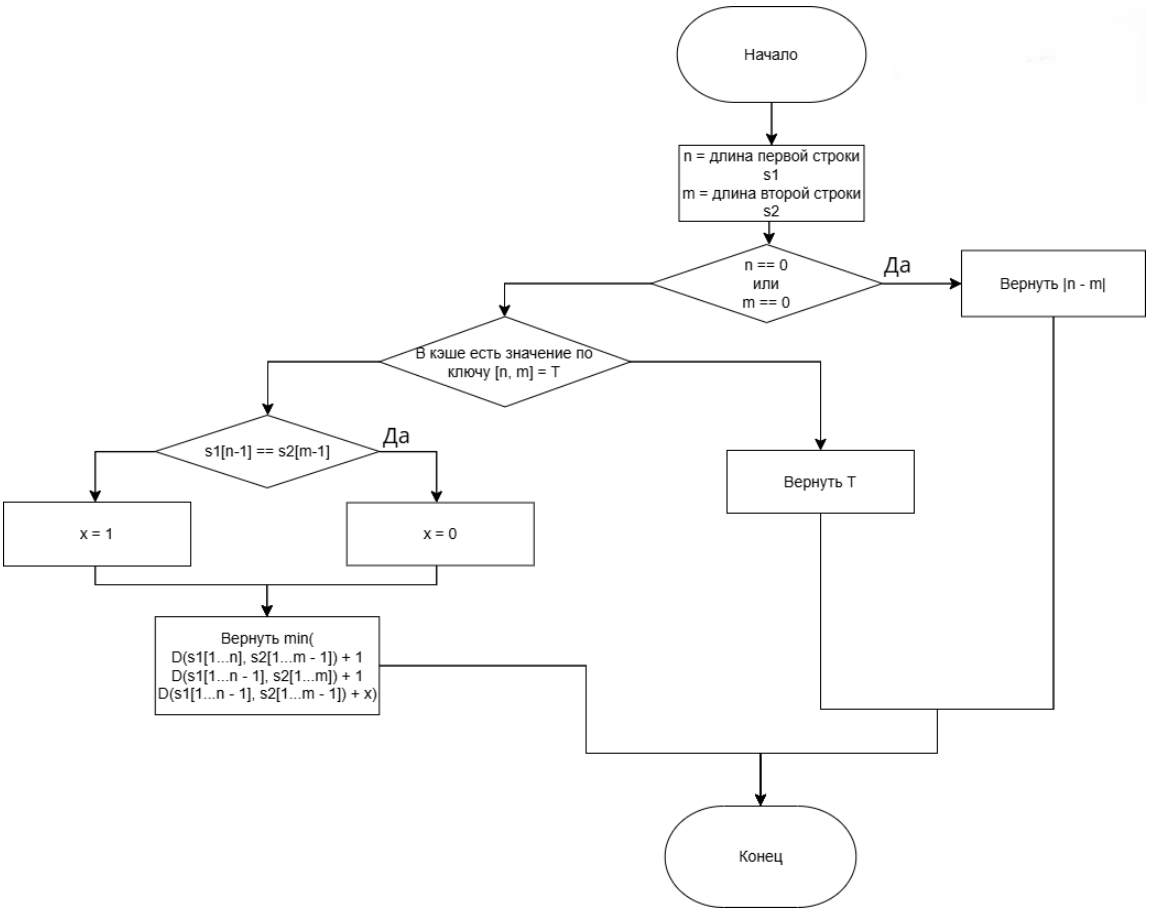

Рисунок 2.3 – Рекурсивный алгоритм нахождения расстояния Левенштейна с кэшированием

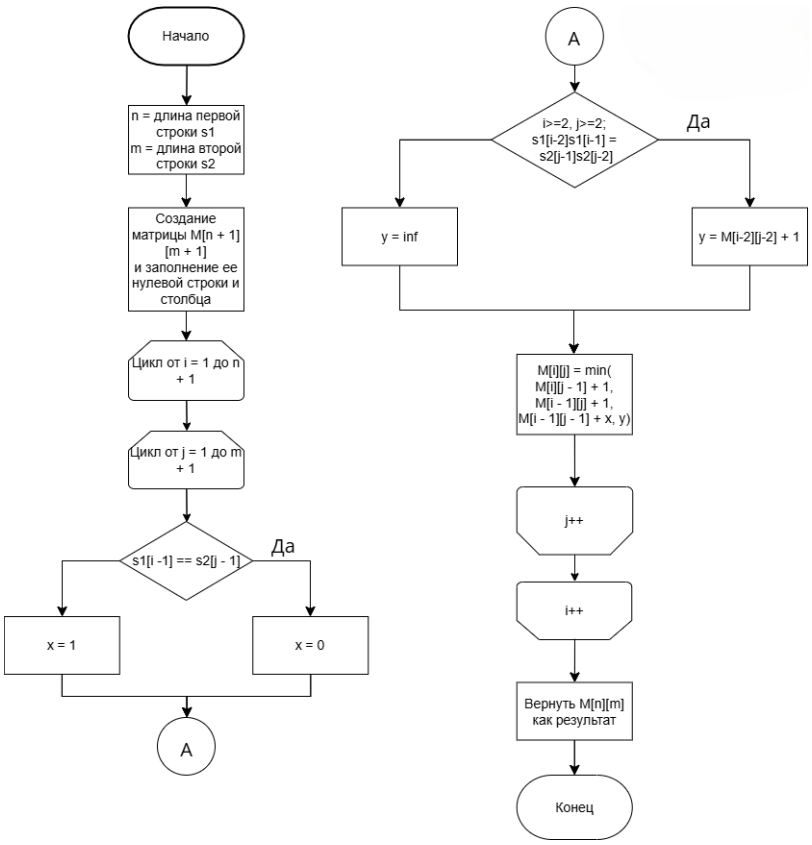

Рисунок 2.4 – Итеративный алгоритм нахождения расстояния Дамерау–Левенштейна

Вывод

В данном разделе были разработаны алгоритмы для итерационного, рекурсивного, рекурсивного с мемоизацией алгоритмов поиска расстояния Левенштейна, а также итерационного алгоритма поиска расстояния Дамерау-Левенштейна.

3 Технологическая часть

В данном разделе будут приведены данные о выбранном языке программирования, коды алгоритмов и тесты для каждого алгоритма.

3.1 Требования к ПО

Реализуемое ПО будет давать возможность выбрать алгоритм, ввести две сравниваемые строки и вывести результат вычислений, а также опционально вывести матрицу расстояний для итерационных алгоритмов. Должна быть реализована возможность сравнения эффективности алгоритмов по времени выполнения.

3.2 Язык программирования

В данной работе для реализации алгоритмов был выбран язык программирования *Python* [2]. Язык предоставляет возможности для оценки затрат времени и памяти алгоритма. Время работы было замерено с помощью функции *process_time_ns* из библиотеки *time* [3].

3.3 Реализация алгоритмов

В листингах 3.1 — 3.4 представлены реализации алгоритмов нахождения расстояния Левенштейна и Дамерау–Левенштейна.

Листинг 3.1 – Реализация итерационного нахождения расстояния Левенштейна

```
def levenstein_iterative(fir_word: str, sec_word: str) -> (int,
    list):
    fir_len = len(fir_word)
    sec_len = len(sec_word)

    matrice = [[0] * (sec_len + 1) for _ in range(fir_len + 1)]

    for i in range(fir_len + 1):
        matrice[i][0] = i
    for i in range(sec_len + 1):
        matrice[0][i] = i

    for i in range(1, fir_len + 1):
        for j in range(1, sec_len + 1):
            insert = matrice[i][j - 1] + insert_cost
            delete = matrice[i - 1][j] + delete_cost
            change = matrice[i - 1][j - 1]
            if fir_word[i - 1] != sec_word[j - 1]:
                change += change_cost

            matrice[i][j] = min(insert, delete, change)

    return matrice[fir_len][sec_len], matrice
```

Листинг 3.2 – Реализация нахождения расстояния Левенштейна рекурсивно

```
1  def levenstein_recursive(fir_word: str, sec_word: str) -> int:
2      fir_len = len(fir_word)
3      sec_len = len(sec_word)
4
5      def recursion(fir_word: str, fir_len: int, sec_word: str,
           sec_len: int) -> int:
6          if fir_len == 0 or sec_len == 0:
7              return max(fir_len, sec_len)
8          change = recursion(fir_word, fir_len - 1, sec_word, sec_len
               - 1)
9          if fir_word[fir_len - 1] != sec_word[sec_len - 1]:
10             change += change_cost
11
12         return min(insert, delete, change)
13
14     return recursion(fir_word, fir_len, sec_word, sec_len)
```

Листинг 3.3 – Реализация нахождения расстояния Левенштейна рекурсивно с кэшированием

```
def levenstein_recursive_memoization(fir_word: str, sec_word: str)
    -> int:
    fir_len = len(fir_word)
    sec_len = len(sec_word)
    answers = {}

    def recursion_memoization(fir_word: str, fir_len: int,
        sec_word: str, sec_len: int) -> int:
        if fir_len == 0 or sec_len == 0:
            return max(fir_len, sec_len)

        key = f"{fir_len} {sec_len}"
        if key in answers:
            return answers[key]
        change = recursion_memoization(fir_word, fir_len - 1,
            sec_word, sec_len - 1)
        if fir_word[fir_len - 1] != sec_word[sec_len - 1]:
            change += change_cost

        result = min(insert, delete, change)
        answers[key] = result
        return result

    return recursion_memoization(fir_word, fir_len, sec_word,
        sec_len)
```

Листинг 3.4 – Реализация итерационного нахождения расстояния Дамерау–Левенштейна

```
def damerau_levenstein_iterative(fir_word: str, sec_word: str) ->
    (int, list):
    fir_len = len(fir_word)
    sec_len = len(sec_word)

    matrice = [[0] * (sec_len + 1) for _ in range(fir_len + 1)]

    for i in range(fir_len + 1):
        matrice[i][0] = i
    for i in range(sec_len + 1):
        matrice[0][i] = i

    for i in range(1, fir_len + 1):
        for j in range(1, sec_len + 1):
            change = matrice[i - 1][j - 1]
            if fir_word[i - 1] != sec_word[j - 1]:
                change += change_cost
            swap = insert + delete + change
            if i >= 2 and j >= 2:
                if fir_word[i - 1] == sec_word[j - 2] and
                   fir_word[i - 2] == sec_word[j - 1]:
                    swap = matrice[i - 2][j - 2] + swap_cost

            matrice[i][j] = min(insert, delete, change, swap)

    return matrice[fir_len][sec_len], matrice
```

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна и Дамерау—Левенштейна. Числа в скобках i(j) обозначают несовпадение ожидаемых результатов вычисления расстояния Левенштейна и Дамерау–Левенштейна. Все тесты программа прошла успешно.

Таблица 3.1 – Функциональные тесты

1-я строка	2-я строка	Результаты				
		Эталон	И. Лев.	Р. Лев.	Р. х. Лев.	И. Д-Л
[]	[]	0	0	0	0	0
[]	Moscow	6	6	6	6	6
Moscow	cow	3	3	3	3	3
Moscow	Mocsow	2(1)	2	2	2	1
Moscow	Mosco	1	1	1	1	1
Moscow	Moscoe	1	1	1	1	1
Moscow	wocsoM	4(3)	4	4	4	3
Moscow	Russia	5	5	5	5	5

Вывод

В данном разделе были приведены коды реализаций итерационных алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна, рекурсивных алгоритмов поиска расстояния Левенштейна, а также функциональные тесты.

4 Исследовательская часть

В данном разделе будут представлены результаты исследования эффективности работы алгоритмов по времени работы.

4.1 Технические характеристики

В данном разделе представлены технические характеристики устройства, на котором проводилось исследование.

— Операционная система: Windows11 Домашняя, версия 23H2, сборка ОС 22631.4037;

— Оперативная память: 16 ГБ;

— Процессор: 13th Gen Intel(R) Core(TM) i5-13500H 2.60 ГГц.

При тестировании ноутбук был включен в сеть электропитания и нагружен только встроенными приложениями окружения, а также системой тестирования.

4.2 Время выполнения алгоритмов

Результаты замеров времени работы алгоритмов нахождения расстояний Левенштейна и Дамерау–Левенштейна приведены в таблице 4.1. Замеры времени проводились на строках одинаковой длины, состоящих из строчных латинских символов, их результаты усреднялись для 1000 одинаковых измерений.

Таблица 4.1 – Время работы алгоритмов (в микросекундах)

Длина строк	Лев итер.	Лев рек.	Лев рек. кэш	Дам-Лев итер.
0	0.568	0.435	1.04	0.703
1	0.952	0.837	1.55	0.954
2	1.68	2.5	3.56	2.14
3	2.81	11.7	7.42	3.82
4	4.11	59.2	11.6	5.03
5	5.74	$2.97 \cdot 10^2$	18.4	7.35
6	7.94	$1.59 \cdot 10^3$	31.1	11.0
7	11.7	$8.64 \cdot 10^3$	39.8	13.1
8	13.1	$4.7 \cdot 10^4$	48.4	16.9
9	15.9	$2.64 \cdot 10^5$	61.2	21.1
10	19.6	$1.44 \cdot 10^6$	80.1	26.3

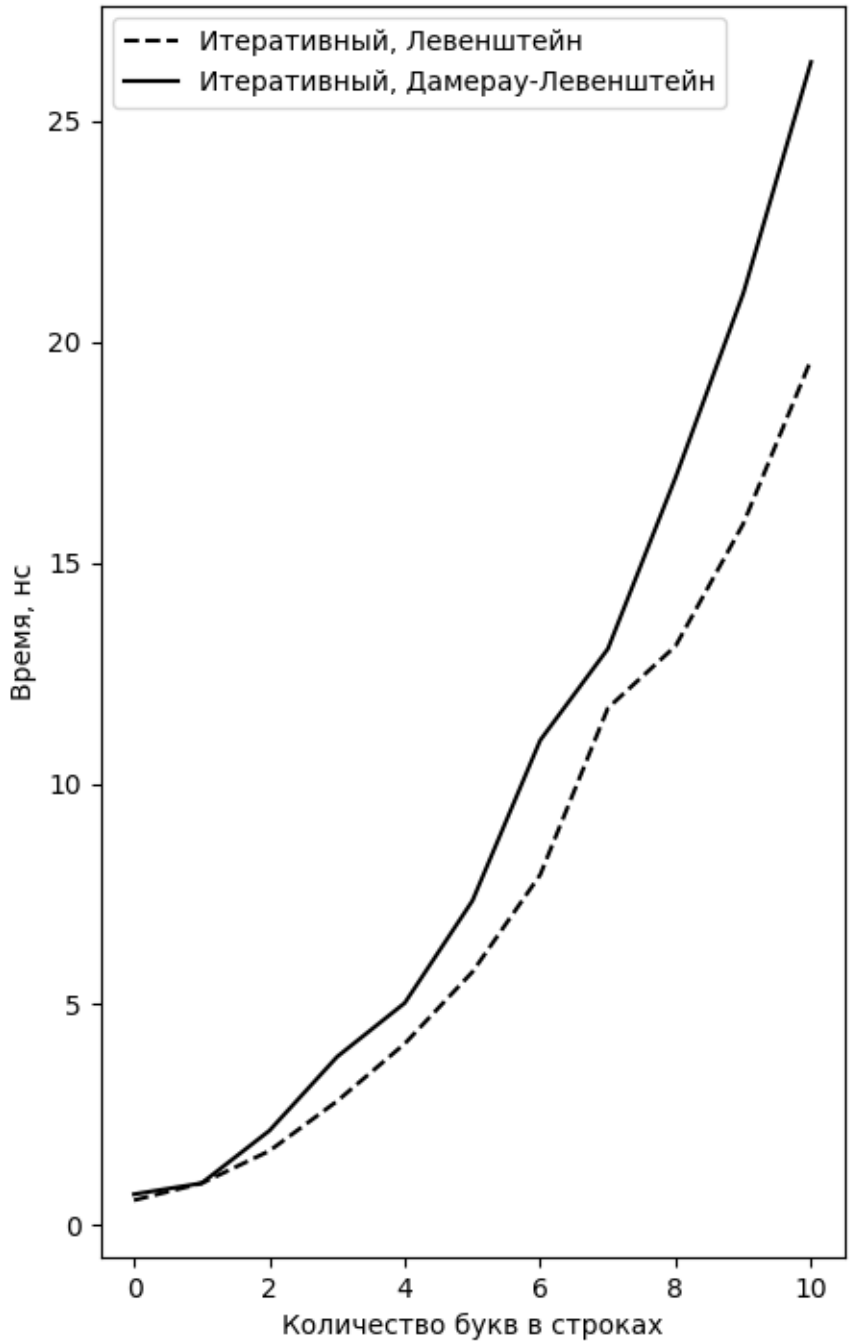

Рисунок 4.1 – Сравнение итеративных алгоритмов вычисления расстояний Левенштейна и Дамерау–Левенштейна

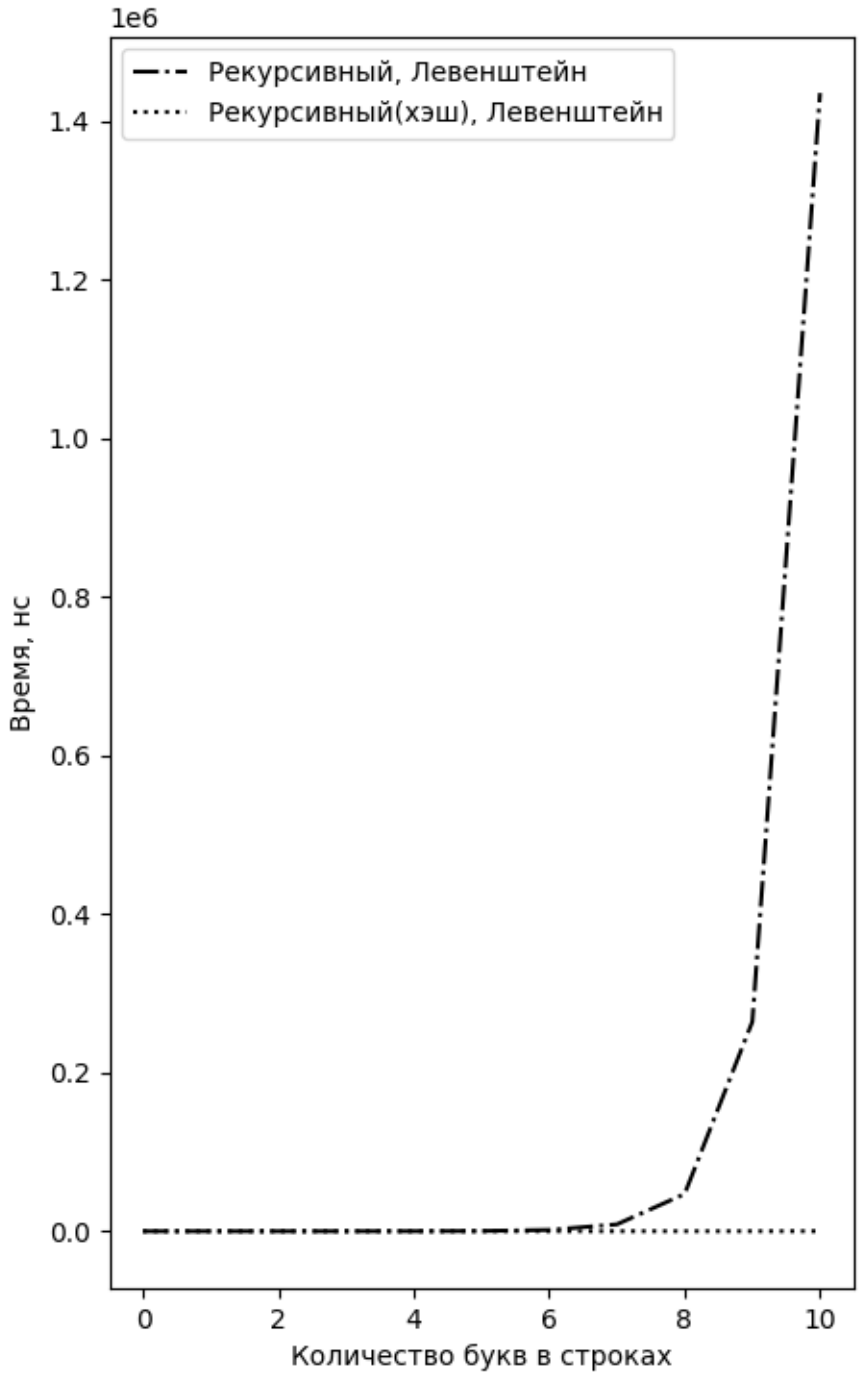

Рисунок 4.2 – Сравнение рекурсивных алгоритмов вычисления расстояния Левенштейна

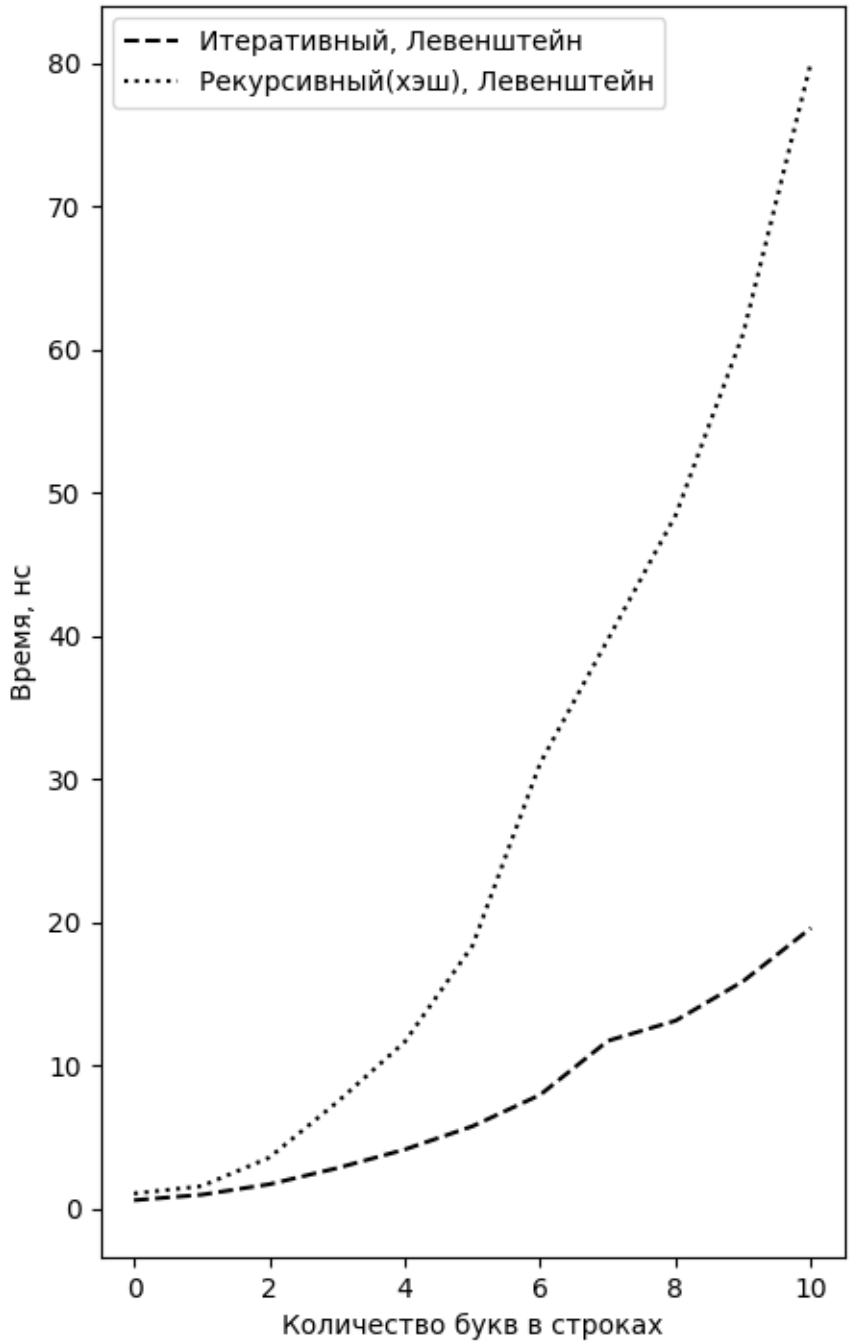

Рисунок 4.3 – Сравнение итеративного и рекурсивного(с кэшем) алгоритмов вычисления расстояния Левенштейна

4.3 Эффективность алгоритмов по памяти

Алгоритмы Левенштейна и Дамерау–Левенштейна не отличаются по использованию памяти, так как используют матрицы одинаковых размеров. Можно сравнить рекурсивную и итерационную реализацию алгоритмов для поиска расстояния Левенштейна.

Пусть S_1, S_2 — строки, size — функция, возвращающая размер аргумента; len — функция, возвращающая длину строки, char — символьный тип, int — целочисленный.

Для итеративной реализации расход памяти рассчитывается следующим образом: память затрачивается на хранение матрицы из $(len(S_1)+1)\cdot(len(S_2)+1)$ чисел типа int — расстояний Левенштейна для всех возможных подстрок S_1, S_2, на хранение самих строк — двух массивов типа $char$ с длинами $len(S_1), len(S_2)$ и на хранение длин строк и ответа — еще три int-а. Итого:

$$(len(S_1)+1)\cdot(len(S_2)+1)\cdot size(int)+(len(S_1)+len(S_2))\cdot(size(char)+3\cdot size(int) \tag{4.1}$$

При рекурсивной реализации память затрачивается на сами рекурсивные вызовы и на хранение двух входных строк. Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, а на каждый вызов функции требуется еще 3 дополнительных переменных типа int (длины строк и ответ), соответственно, максимальный расход памяти

$$(len(S_1) + len(S_2)) \cdot ((len(S_1) + len(S_2)) \cdot size(char) + 3 \cdot Size(\text{int})) \tag{4.2}$$

По затратам памяти итеративные алгоритмы проигрывают рекурсивным, так как они хранят одновременно все расстояния Левенштейна для всех подстрок, максимальный размер используемой памяти в них растёт как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк.

Вывод

Итеративные алгоритмы для поиска расстояний Левенштейна и Дамерау–Левенштейна не различаются по количеству используемой памяти, однако алгоритм для поиска расстояния Дамерау–Левенштейна работает немного дольше (см. рисунок 4.1), так как он является дополненной версией предыдущего алгоритма (на один if больше, а также проверки двух последних символов строк на возможность перестановки). Итеративные алгоритмы работают существенно быстрее чем рекурсивные (см. рисунок 4.3). При этом рекурсивный алгоритм с кэшированием работает гораздо быстрее простого рекурсивного алгоритма (см. рисунок 4.2), так как в алгоритме с кэшированием большое количество рекурсивных вызовов, вычисляющих одни и те же данные заменяются на обращение к кэшу с уже вывчисленными значениями. Уже на длине строк в 10 символов алгоритм с кэшированием оказывается быстрее простого рекурсивного алгоритма в 18406 раз. Однако алгоритм с кэшированием требует дополнительной памяти в виде словаря-кэша.

Заключение

В ходе выполнения данной лабораторной работы были изучены итеративные и рекурсивные алгоритмы поиска расстояний Левенштейна и Дамерау–Левенштейна, а также решены следующие задачи:

— изучены итеративный и рекурсивный методы вычисления расстояния Левенштейна и Дамерау–Левенштейна;

— реализованы указанные алгоритмы поиска расстояния Левенштейна и Дамерау–Левенштейна;

— проведено сравнение эффективности указанных алгоритмов по затраченному процессорному времени и по затраченной памяти;

— описаны и обоснованы полученные результаты в отчете о выполненной лабораторной работе.

Итеративные алгоритмы для поиска расстояний Левенштейна и Дамерау–Левенштейна не различаются по количеству используемой памяти, однако алгоритм для поиска расстояния Дамерау–Левенштейна работает немного дольше (см. рисунок 4.1, график 1), так как он является дополненной версией предыдущего алгоритма (на один условный оператор больше, а также проверки двух последних символов строк на возможность перестановки). Итеративные алгоритмы работают существенно быстрее чем рекурсивные (см. рисунок 4.1, график 3). При этом рекурсивный алгоритм с кэшированием работает гораздо быстрее простого рекурсивного алгоритма (см. рисунок 4.1, график 2), так как в алгоритме с кэшированием большое количество рекурсивных вызовов, вычисляющих одни и те же данные заменяются на обращение к хэшу с уже вывчисленными значениями. Уже на длине строк в 10 символов алгоритм с кэшированием оказывается быстрее простого рекурсивного алгоритма в 18406 раз. Однако алгоритм с кэшированием требует дополнительной памяти в виде словаря-кэша. Поставленная цель достигнута, все задачи решены.

Список использованных источников

[1] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. — М.: Доклады АН СССР, 1965. Т. 163. С. 845– 848.

[2] Документация языка Welcome to Python [Электронный ресурс]. Режим доступа: `https://www.python.org` (дата обращения: 12.09.2024).

[3] Документация модуля time —- Time access and conversions [Электронный ресурс]. Режим доступа: `https://docs.python.org/3/library/time.html#time.process_time_ns` (дата обращения: 12.09.2024).

[4] Кормановский, М.В. Граф на основе расстояния Левенштейна и его визуализация / М. В. Кормановский, Н. П. Артюхин, А. А. Косарев [и др.] // Проблемы лингвистики и лингводидактики в неязыковом вузе : 5-я Международная научно-практическая конференция: сборник материалов конференции. В 2-х томах, Москва, 15 декабря 2022 года. Том 1. – Москва: Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет), 2023. – С. 310-319.