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»

Отчёт

по лабораторной работе №1

Название «Расстояния Левенштейна и Дамерау-Левенштейна»

Дисциплина «Анализ алгоритмов»

Студент ИУ7-55Б _____ Бугаенко А.П.
(подпись, дата) (Фамилия И.О.)

Преподователь _____ Волкова Л.Л.
(подпись, дата) (Фамилия И.О.)

Москва, 2021

Содержание

Введение

Расстояние Левенштейна (редакционное расстояние) - минимальное количество односимвольных операций (вставки, удаления, замены), необходимых для преобразования одной последовательности символов в другую. Расстояние Левенштейна используется в решении таких задач, как:

— исправление ошибок в слове;
— сравнение генов, хромосом и белков в биоинформатике;
— сравнение текстовых файлов утилитой diff и ей подобными.

Целью данной работы является изучение, реализация и сравнительный анализ различных алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна, а также получение практических навыков при реализации указанных алгоритмов. Для достижения поставленной цели необходимо выполнить следующие задачи:

1) изучить алгоритмы:
    а) поиска расстояния Левенштейна с помощью табличного способа;
    б) поиска расстояния Левенштейна с помощью рекурсивного способа;
    в) поиска расстояния Левенштейна с помощью рекурсивного способа с заполнением и дополнительными проверками матрицы;
    г) рпоиска расстояния Дамерау-Левенштейна с помощью табличного способа.
2) построить схемы алгоритмов и провести функциональный анализ ПО;
3) сделать сравнительный анализ данных алгоритмов по затрачиваемым в процессе работы алгоритмов ресурсам (памяти и времени);
4) получить экспериментальное подтверждение различий эффективности различных версий реализации данных алгоритмов (рекурсивной и нерекурсивной) при помощи программного обеспечения на основе замеров процессорного времени выполнения реализации данных алгоритмов для строк различной длины;
5) сформировать описание и обоснование полученных в процессе работы результатов в отчёте о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитический раздел

В данном разделе будет рассмотрено описание работы алгоритмов нахождения межстрочного расстояния Левенштейна и Дамерау-Левенштейна.

1.1 Описание алгоритмов

Расстояние Левенштейна между двумя строками - это минимальное количество операций вставок (I), удалений (D) и замен (R) одного символа на другой, необходимых для преобразования одной строки в другую. Также в алгоритмах будет использоваться операция совпадения (M). Задачей алгоритмов, которые рассматриваются в этом разделе, является нахождение данного расстояния. В случае расстояния Дамерау-Левенштейна добавляется ещё одна операция - транспозиция (перестановка соседних символов, обозначается, как T).

1.2 Расстояние Левенштейна

Пускай у нас есть две строки S_1 и S_2, имеющие длину M и N соответственно. В таком случае расстояние Левенштейна может быть подсчитано по следующей рекуррентной формуле:

$D(S_1[1...i],S_2[1...j]) =$

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

При рассчёте разрешены следующие операции:

1) удаление (D, штраф - 1);
2) удаление (I, штраф - 1);
3) удаление (R, штраф - 1);
4) удаление (M, штраф - 0).

1.3 Расстояние Дамерау-Левенштейна

Алгоритм нахождения расстояния Дамерау-Левенштейна является модификацией расстояния Левенштейна. В частности ко множеству операций алгоритма расстояния Левенштейна была добавлена операция транспозиции (T) или перестановки символов. Данная модификация

стала очевидно необходимой вследствие того, что значительная часть ошибок пользователей при наборе текста по сути является перестановкой символов.

Пускай у нас есть две строки S_1 и S_2, имеющие длину M и N соответственно. В таком случае расстояние Дамерау-Левенштейна может быть подсчитано по следующей рекуррентной формуле:

$D(S_1[1...i],S_2[1...j]) =$

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

1.4 Вывод

В данном разделе выбли рассмотрены расстояния Левенштейна и Дамерау-Левенштейна. Были приведены формулы, позволяющие вычислить оба расстояния. Была указана разница между расстоянием Левенштейна и расстоянием Дамерау-Левенштейна, заключающаяся в добавлении операции транспозиции, а также причины, приведшие к появлению этой разницы. В результате добавления этой операции области применения расстояния Левенштейна и расстояния Дамерау-Левенштейна могут различаться. В качестве входных данных оба алгоритма получают на вход две строки, причём длина обеих строк может быть как равна нулю, так и достигать максимально допустимой для реализации алгоритма. На выходе данные алгоритмы возвращают целое число - минимальное редакторское расстояние между строками.

2 Конструкторский раздел

В данном разделе будут рассмотрены схемы, структуры данных, способы тестирования, описания памяти для следующих алгоритмов:

1) нерекурсивный алгоритм поиска расстояния Левенштейна (табличный способ);
2) рекурсивный алгоритм поиска расстояния Левенштейна без заполнения матрицы;
3) рекурсивный алгоритм поиска расстояния Левенштейна с заполнением и дополнительными проверками матрицы;
4) нерекурсивный алгоритм поиска расстояния Дамерау-Левенштейна (табличный способ).

Общие замечания для всех алгоритмов:

1) сложные структуры данных не будут использоваться, так как в них нету необходимости;
2) все описанные выше алгоритмы в качестве результата возращают число - посчитаное минимальное редакторское расстояние;

2.1 Тестирование алгоритмов

Выделение классов эквивалентности для тестирования алгоритмов:

1) проверка на пустых строках;
2) проверка на одну пустую строку;
3) проверка на одинаковых строках;
4) проверка на общем случае.

Описание тестов:

1) обе строки, подающиеся на вход, пустые;
2) певая строка, подающаяся на вход, пустая;
3) вторая строка, подающаяся на вход, пустая;
4) на вход подаются две разные непустые строки;
5) на вход подаются две одинаковые непустые строки.

2.2 Нерекурсивный алгоритм поиска расстояния Левенштейна (табличный способ)

Описание типов и структур данных, используемых в алгоритме:

1) строковые переменные, представляющая из себя массив символов;
2) целочисленные переменные, использующаяся для хранения индексов и результата
3) массивы целых числел, использующиеся как кэш-строки

Алгоритм поиска расстояния Левенштейна подразумевает использование таблицы для вычисления редакторского расстояния. Однако для экономии памяти будут использоваться

только два кэш-массива. Данные массивы будут иметь длину равную длине первой строки, и будут содержать целочисленные значения. Также помимо этих массивов в программе будут использоваться две строки a и b, подающиеся на вход функции, локальные целочисленные переменные n, m, хранящие длину поданных на вход строк и локальные целочисленные переменные I, D, R - хранящие промежуточные результаты операций при итерации вычисления редакторского расстояния.

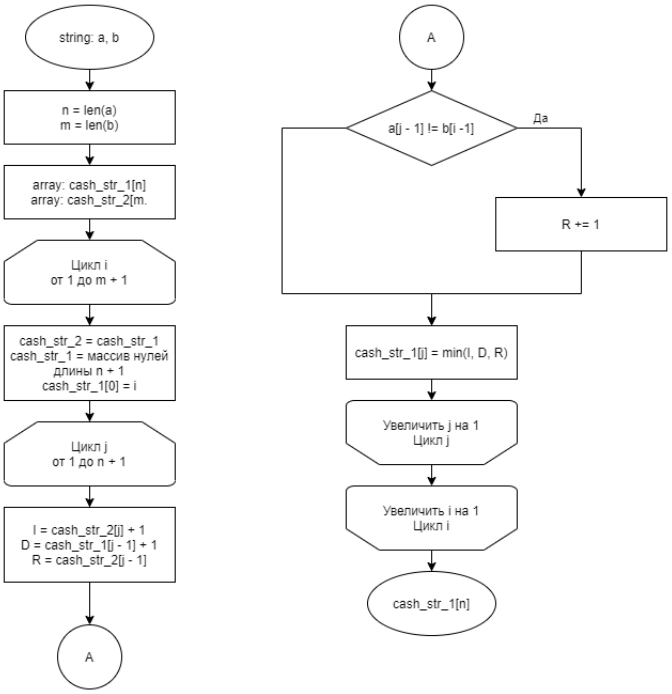

Рисунок 2.1 — Схема нерекурсивного алгоритма поиска расстояния Левенштейна табличным способом.

2.3 Рекурсивный алгоритм поиска расстояния Левенштейна без заполнения матрицы

Описание типов и структур данных, используемых в алгоритме:

1) строковые переменные, представляющая из себя массив символов;
2) целочисленные переменные, использующаяся для хранения индексов и результата

Рекурсивный алгоритм Левенштейна использует одну локальную целочисленную переменную - result, хранящую результат выполнения очередной итерации вычисления редакторского расстояния. Также используются две строковые переменные a и b, хранящие поданные на вход функции строки.

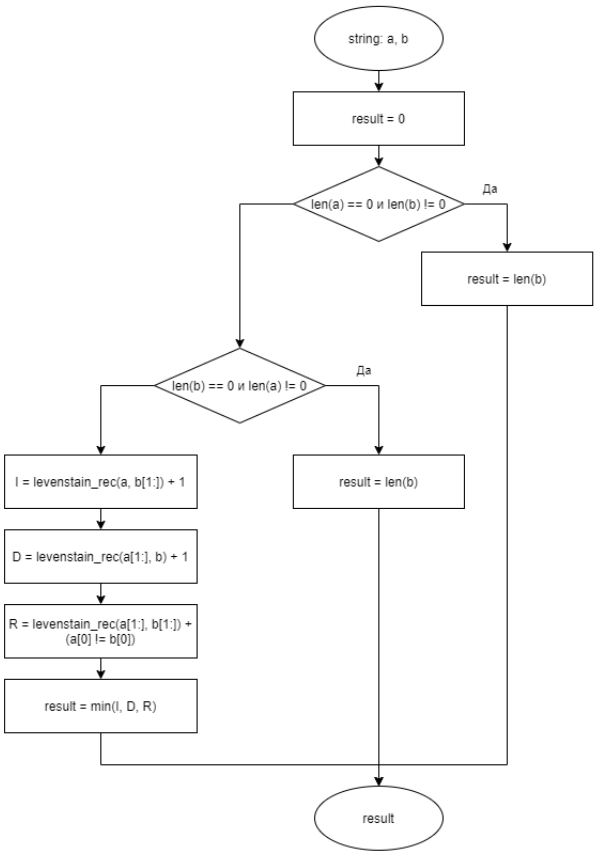

Рисунок 2.2 — Схема рекурсивного алгоритма поиска расстояния Левенштейна без заполнения матрицы.

2.4 Рекурсивный алгоритм поиска расстояния Левенштейна с заполнением и дополнительными проверками матрицы

Описание типов и структур данных, используемых в алгоритме:

1) строковые переменные, представляющая из себя массив символов;
2) целочисленные переменные, использующаяся для хранения индексов и результата
3) массив массивов целых чисел, использующийся как таблица при вычислении результата

Для работы данного алгоритма при каждом рекурсивном вызове функции необходимо помимо строк передавать индексы i, j и само тело матрицы. Мы не будем делать проверку внутри функции, поскольку это приведёт к излишним временным затратам, к тому же мы можем быть уверены, что после первой итерации алгоритма в функцию будут передаваться верные значения. В качестве входных параметров фукцния получает две строковые переменные a, b, два целочисленных числа - i, j, и целочисленную матрицу mat, имеющую размер [len(b) + 1, len(a) + 1]. Также внутри функции есть локальные целочисленные переменные result и R. result хранит результат работы функции. R - хранит число, добавляемое при вычислении стоимости операции.

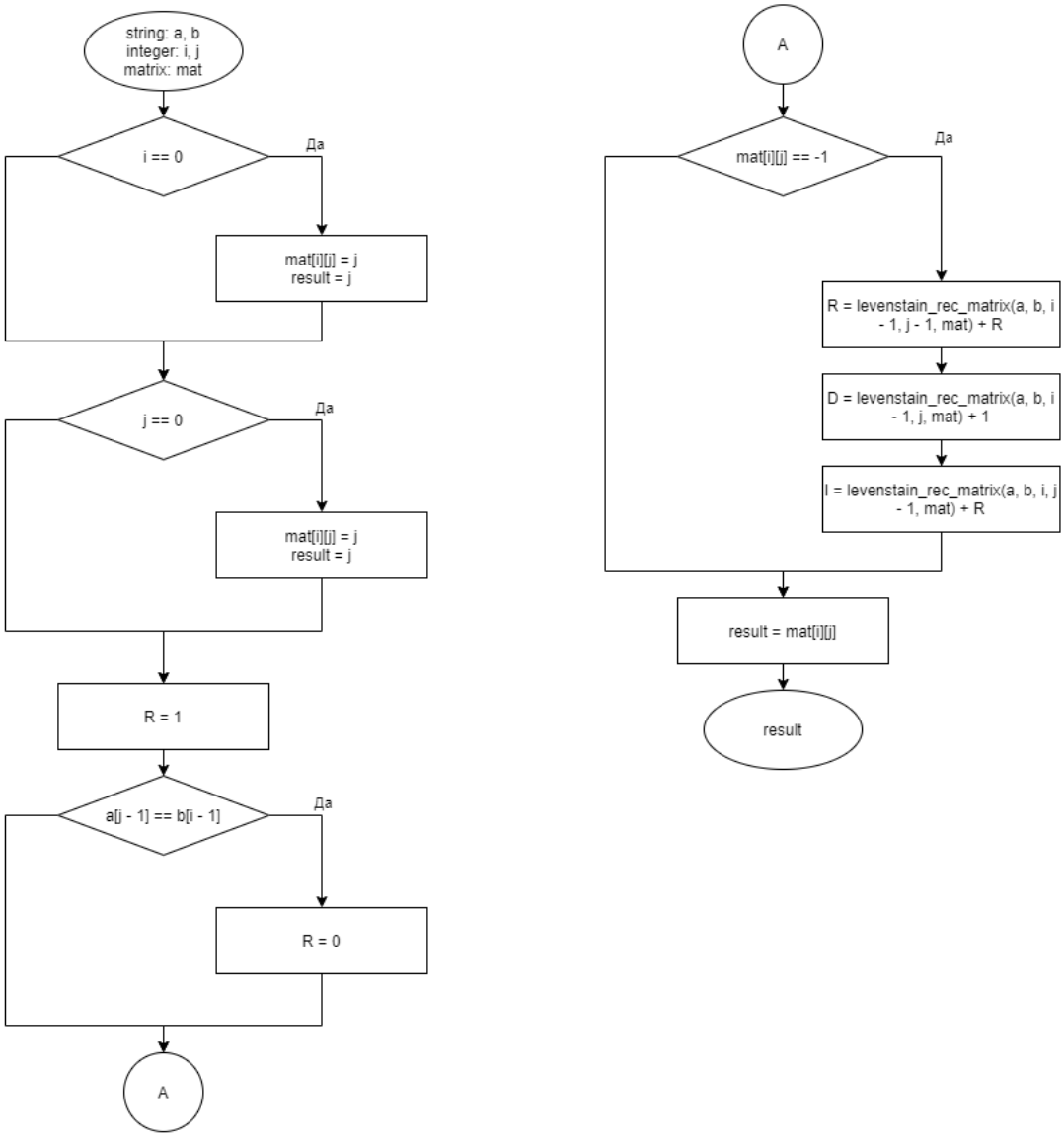

Рисунок 2.3 — Схема рекурсивного алгоритма поиска расстояния Левенштейна с заполнением и дополнительными проверками матрицы.

2.5 Нерекурсивный алгоритм поиска расстояния Дамерау-Левенштейна (табличный способ)

Описание типов и структур данных, используемых в алгоритме:

1) строковые переменные, представляющая из себя массив символов;
2) целочисленные переменные, использующаяся для хранения индексов и результата
3) массив массивов целых чисел, использующийся как таблица при вычислении результата

Алгоритм поиска расстояния Дамерау-Левенштейна подразумевает использование таблицы для вычисления редакторского расстояния. В данном случае будет использоваться таблица целых чисел mat, имеющая размер n на m, где n и m - локальные целочисленные переменные, равные длине подаваемых на вход функции строк a и b соответственно. Помимо этого в программе используется локальная целочисленная переменная R, использующаяся для хранения значения, использующегося при вычислении стоимости операций замены и транспозиции.

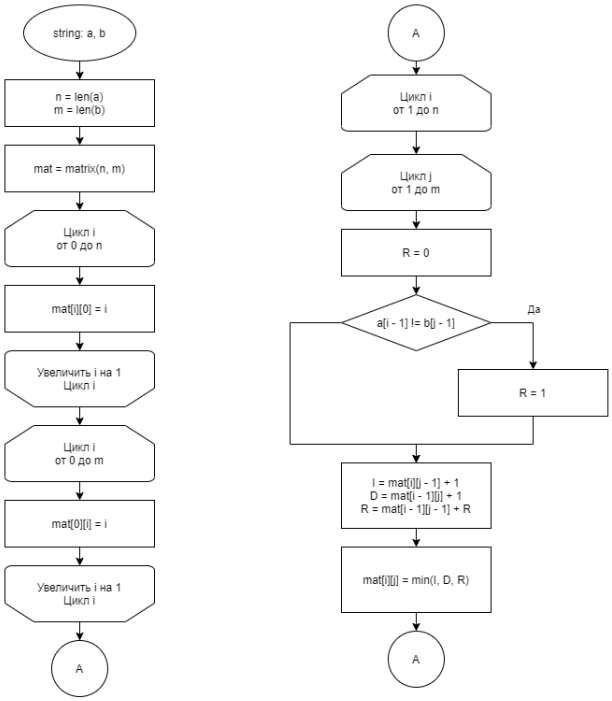

Рисунок 2.4 — Схема нерекурсивного алгоритма поиска расстояния Дамерау-Левенштейна табличным способом. Часть 1.

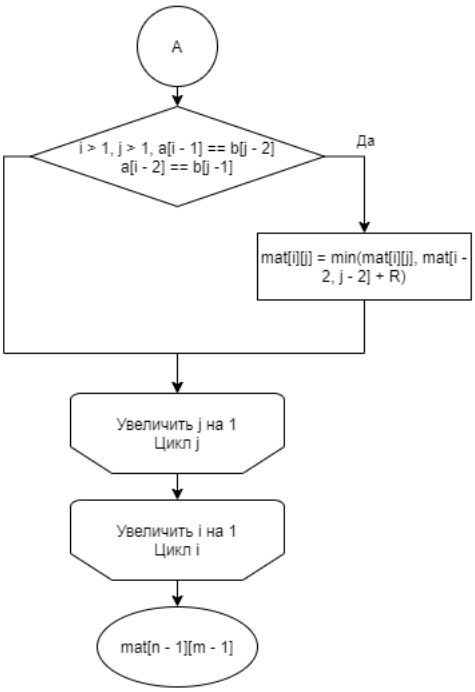

Рисунок 2.5 — Схема нерекурсивного алгоритма поиска расстояния Дамерау-Левенштейна табличным способом. Часть 2.

2.6 Функциональная схема ПО

Рисунок 2.6 — Схема нерекурсивного алгоритма поиска расстояния Дамерау-Левенштейна табличным способом. Часть 2.

2.7 Вывод

В данном разделе были рассмотрены схемы алгоритмов для каждого из способов вычисления редакторского расстояния, и были определены общие тесты для каждого алгоритма, юыли описаны типы и вид входных данных. Помимо этого были выделены и описаны особенности реализации алгоритмов, в которых используются дополнительные структуры данных для хранения и обработки информации при вычислении редакторского расстояния (кэш-строки в табличном способе вычисления расстояния Левенштейна, матрица в рекурсивном способе вычисления алгоритма Левенштейне, таблица табличном способе вычисления алгоритма Дамерау-Левенштейне).

3 Технологический раздел

В данном разделе будут рассмотрены подробности реализации описаных выше алгоритмов. Также будут обоснованы выбор языка программирования для реализации, выбор библиотек для проведения экспериментов и представлены важные фрагменты кода написанной в рамках работы программы.

3.1 Выбор языка программирования

В качестве языка программирования для реализации данной лабораторной работы использовался язык программирования python (3.9.7) [1] в целях упрощения работы со структурами данных и визцализацией данных сравнительных анализов и наличием опыта работы с данным языком программирования. В качестве среды разработки использовалась Visual Studio Code [2]. Для замеров процедурного времени использовалась функция process_time() из библиотеки time [3].

3.2 Сведения о модулях программы

Реализованное ПО состоит из трёх модулей:

1) algos.py - в данном модуле реализованы алгоритмы рассчёта редакторского расстояния;
2) time.py- в данном модуле реализованы замеры временных характеристик алгоритмов;
3) test.py - в данном модуле реализованы тесты программ.

3.3 Реализация алгоритмов поиска редакторского расстояния

Листинг 3.1 — Реализация нерекурсивного алгоритма поиска расстояния Левенштейна табличным способом.

```
1  def levenstain_table(a, b):
2      n, m = len(a), len(b)
3      cash_str_1 = [*range(n + 1)]
4      cash_str_2 = [*range(n + 1)]
5
6      for i in range(1, m + 1):
7          cash_str_2 = cash_str_1
8          cash_str_1 = [i] + [0] * n
9          for j in range(1, n + 1):
10             I = cash_str_2[j] + 1
11             D = cash_str_1[j - 1] + 1
12             R = cash_str_2[j - 1]
13             if a[j - 1] != b[i - 1]:
14                 R += 1
15             cash_str_1[j] = min(I, D, R)
16     return cash_str_1[n]
```

Листинг 3.2 — Реализация рекурсивного алгоритма поиска расстояния Левенштейна без заполнения матрицы.

```
def levenstain_rec(a, b):
    result = 0
    if len(a) == 0 and len(b) == 0:
        result = 0
    elif len(a) == 0 and len(b) != 0:
        result = len(b)
    elif len(b) == 0 and len(a) != 0:
        result = len(a)
    else:
        result = min(
            levenstain_rec(a, b[1:]) + 1,
            levenstain_rec(a[1:], b) + 1,
            levenstain_rec(a[1:], b[1:]) + (a[0] != b[0])
        )
    return result
```

Листинг 3.3 — Реализация рекурсивного алгоритма поиска расстояния Левенштейна с заполнением и дополнительными проверками матрицы.

```
def levenstain_rec_matrix(a, b, i, j, mat):
    result = 0
    if len(a) == 0 and len(b) == 0:
        result = 0
    elif i == 0:
        mat[i][j] = j
        result = j
    elif j == 0:
        mat[i][j] = i
        result = i
    else:
        R = 1
        if a[j - 1] == b[i - 1]:
            R = 0

        if mat[i][j] == -1:
            mat[i][j] = min(
                levenstain_rec_matrix(a, b, i - 1, j - 1, mat) + R, # R
                levenstain_rec_matrix(a, b, i - 1, j, mat) + 1, # D
                levenstain_rec_matrix(a, b, i, j - 1, mat) + 1 # I
                )
        result = int(mat[i][j])
    return result
```

Листинг 3.4 — Реализация нерекурсивного алгоритма поиска расстояния Дамерау-Левенштейна табличным способом.

```
def damerau_levenstain_table(a, b):
    n = len(a) + 1
    m = len(b) + 1
    mat = np.zeros([n, m]).astype(int)

    for i in range(0, n):
        mat[i, 0] = i
    for i in range(0, m):
        mat[0, i] = i

    for i in range(1, n):
        for j in range(1, m):
            R = 0
            if a[i - 1] != b[j - 1]:
                R = 1
            mat[i][j] = min(
                mat[i][j - 1] + 1, # I
                mat[i - 1][j] + 1, # D
                mat[i - 1][j - 1] + R # R
            )
            if (i > 1 and j > 1) and a[i - 1] == b[j - 2] and a[i - 2] == b[j - 1]:
                mat[i][j] = min(mat[i][j], mat[i - 2][j - 2] + R) # T
    return mat[n - 1][m - 1]
```

3.4 Реализация тестирования алгоритмов

Для тестирования алгоритмов было реализовано несколько тесто для каждого класса эквивалентности:

1) обе строки, подающиеся на вход, пустые;
2) первая из строк, подающихся на вход, пустая, вторая имеет ненулевую длину;
3) вторая из строк, подающихся на вход, пустая, первая имеет ненулевую длину;
4) обе строки, подающиеся на вход, имеют одинаковую ненулевую длину;
5) обе строки, подающиеся на вход, имеют разную ненулевую длину.

Листинг 3.5 — Реализация функции рандомной генерации строк.

```
def str_gen(str_len):
    src_str = string.ascii_lowercase
    return ''.join(random.choice(src_str) for i in range(str_len))
```

Листинг 3.6 — Реализация тестирования на пустых строках.

```
def test_empty():
```

```
    a = ''
    b = ''
    mat_lrm = np.zeros([len(b) + 1, len(a) + 1]).astype(int)
    mat_lrm.fill(-1)
    return levenstain_table(a, b) == 0 and levenstain_rec(a, b) == 0\
         and levenstain_rec_matrix(a, b, len(b), len(a), mat_lrm) == 0 and\
              damerau_levenstain_table(a, b) == 0
```

Листинг 3.7 — Реализация тестирования первой пустой строки.

```
def test_first(test_len):
    a = ''
    b = str_gen(test_len)
    mat_lrm = np.zeros([len(b) + 1, len(a) + 1]).astype(int)
    mat_lrm.fill(-1)
    return levenstain_table(a, b) == len(b) and levenstain_rec(a, b) == len(b)\
         and levenstain_rec_matrix(a, b, len(b), len(a), mat_lrm) == len(b) and\
              damerau_levenstain_table(a, b) == len(b)
```

Листинг 3.8 — Реализация тестирования второй пустой строки.

```
def test_second(test_len):
    a = str_gen(test_len)
    b = ''
    mat_lrm = np.zeros([len(b) + 1, len(a) + 1]).astype(int)
    mat_lrm.fill(-1)
    return levenstain_table(a, b) == len(a) and levenstain_rec(a, b) == len(a)\
         and levenstain_rec_matrix(a, b, len(b), len(a), mat_lrm) == len(a) and\
              damerau_levenstain_table(a, b) == len(a)
```

Листинг 3.9 — Реализация тестирования строк равной ненулевой длины.

```
def test_equal(test_len):
    a = str_gen(test_len)
    b = a
    mat_lrm = np.zeros([len(b) + 1, len(a) + 1]).astype(int)
    mat_lrm.fill(-1)
    return levenstain_table(a, b) == 0 and levenstain_rec(a, b) == 0\
         and levenstain_rec_matrix(a, b, len(b), len(a), mat_lrm) == 0 and\
              damerau_levenstain_table(a, b) == 0
```

Листинг 3.10 — Реализация тестирования строк разной ненулевой длины.

```
def test_diff(test_len, diff_len):
    a = str_gen(test_len)
    b = str_gen(test_len + diff_len)
    mat_lrm = np.zeros([len(b) + 1, len(a) + 1]).astype(int)
    mat_lrm.fill(-1)
    return levenstain_table(a, b) == levenstain_rec(a, b) == \
```

```
7         levenstain_rec_matrix(a, b, len(b), len(a), mat_lrm) ==
              damerau_levenstain_table(a, b)
```

3.5 Сравнительный анализ потребляемой памяти

Для того, чтобы провести анализ, нам необходимо знать количество потребляемой памяти для различных классов величин.

Таблица 3.1 — Количество потребляемой памяти для различных классов величин.

Классы элементов	Представление в коде	Занимаемая память в байт
Пустой numpy массив	[]	48
Numpy массив длины 5, заполненный нулями	[0, 0, 0, 0, 0]	88
Numpy массив с массивом	[[]]	56
Целое число	Int(10)	14
Пустая строка	Str()	25

Как мы можем увидеть, хранение элементов массива реализовано с помощью указателей, в связи с чем для вложенных массивов требуется отдельный подсчёт занимаемой памяти.

Посчитаем минимальный объём потребляемой памяти для каждого алгоритма:

Алгоритм Левенштейна нерекурсивный табличный:

— два кэш-массива - 48 * 2 + 8 * (len(str_1) + 1) + 8 * (len(str_2 + 1)) байт;
— пять вспомогательных целочисленных переменных - 70 байт;
— два целочисленных счётчика - 28 байт;
— передача двух строк - 50 байт.

Алгоритм Левенштейна рекурсивный:

— целочисленная переменная для хранения результата - 14 байт;
— передача двух строк - 50 байт.

Алгоритм Левенштейна рекурсивный с проверками матрицы:

— передача входных параметров - 25 + 25 + 14 * 2 + 48 = 122 байта;
— две вспомогательные целочисленные переменные - 14 * 2 = 28 байт;
— целочисленная матрица - 48 + 8 * (len(str_2) + 1) * (len(str_1) + 1).

Алгоритм Дамерау-Левенштейна:

— передача входных параметров - 25 + 25 = 50 байт;
— вспомогательные целочисленные переменные - 14 * 3 = 42 байта;
— два целочисленных счётчика - 14 * 2 = 28 байт;
— целочисленная матрица - 48 + 8 * (len(str_2) + 1) * (len(str_1) + 1).

Оценим потребляемую память на словах длиной в 5 и 100 символов. Для слов, длиной в 5 символов:

Алгоритм Левенштейна нерекурсивный табличный:

— два кэш-массива - 48 * 2 + 8 * 6 + 8 * 6 = 182 байта;
— пять вспомогательных целочисленных переменных - 70 байт;
— два целочисленных счётчика - 28 байт;
— передача двух строк - 50 байт.

Алгоритм Левенштейна рекурсивный:

— целочисленная переменная для хранения результата - 14 * 10 байт;
— передача двух строк - 50 * 10 байт.

Алгоритм Левенштейна рекурсивный с проверками матрицы:

— передача входных параметров - 25 + 25 + 14 * 2 + 48 = 122 * 25 = 3050 байта;
— две вспомогательные целочисленные переменные - 14 * 2 = 28 * 25 = 700 байт;
— целочисленная матрица - 48 + 8 * 6 * 6 = 336 байт.

Алгоритм Дамерау-Левенштейна:

— передача входных параметров - 25 + 25 = 50 байт;
— вспомогательные целочисленные переменные - 14 * 3 = 42 байта;
— два целочисленных счётчика - 14 * 2 = 28 байт;
— целочисленная матрица - 48 + 8 * 6 * 6 = 336 байт.

Для слов, длиной в 500 символов:

Алгоритм Левенштейна нерекурсивный табличный:

— два кэш-массива - 48 * 2 + 8 * 501 + 8 * 501 = 4104 байт;
— пять вспомогательных целочисленных переменных - 70 байт;
— два целочисленных счётчика - 28 байт;
— передача двух строк - 50 байт.

Алгоритм Левенштейна рекурсивный:

— целочисленная переменная для хранения результата - 14 * 500 байт;
— передача двух строк - 50 * 500 байт.

Алгоритм Левенштейна рекурсивный с проверками матрицы:

— передача входных параметров - 25 + 25 + 14 * 2 + 48 = 122 байта;
— две вспомогательные целочисленные переменные - 14 * 2 = 28 байт;
— целочисленная матрица - 48 + 8 * 501 * 501 = 2008056 байт.

Алгоритм Дамерау-Левенштейна:

— передача входных параметров - 25 + 25 = 50 байт;
— вспомогательные целочисленные переменные - 14 * 3 = 42 байта;
— два целочисленных счётчика - 14 * 2 = 28 байт;
— целочисленная матрица - 48 + 8 * 501 * 501.

Таблица 3.2 — Результаты сравнения количества потребляемой памяти для 5 символов.

Алгоритм	Занимаемая память в байтах
Левенштейна нерекурсивный табличный	330
Левенштейна рекурсивный	640
Левенштейна рекурсивный с проверками матрицы	4086
Дамерау-Левенштейна	456

Таблица 3.3 — Результаты сравнения количества потребляемой памяти для 500 символов.

Алгоритм	Занимаемая память в байтах
Левенштейна нерекурсивный табличный	4252
Левенштейна рекурсивный	32000
Левенштейна рекурсивный с проверками матрицы	2008206
Дамерау-Левенштейна	2008176

3.6 Вывод

В данном разделе была представлена реализация алгоритмов нахождения расстояния Левенштейна (нерекурсивный способ, рекурсивный способ, рекурсивный способ с заполнением и дополнительные проверками матрицы), а также Дамерау-Левенштейна (табличный способ). Были разработаны алгоритмы тестирования разработанных методов по методу чёрного ящика. Был проведён анализ потребляемой памяти, в ходе которого был сделан вывод о том, что классический алгоритм, реализованный с использованием двух кэш-массивов, эффективнее по памяти, чем остальные алгоритмы.

4 Экспериментальный раздел

В данном разделе будет изучено влияние длины подаваемых на вход строк на время выполнения программы. Также будет проведено сравнение полученных результатов и сделаны выводы о временной эффективности данных алгоритмов.

4.1 Измерение временных характеристик для рекурсивного алгоритма Левенштейна с проверками и заполнением матрицы, нерекурсивного Левенштейна, Дамерау-Левенштейна

Для измерения времени данных алгоритмов будет использовать рандомно генерирующиеся строки длиной от 100 до 500 с шагом 100. Для более точных измерений, для каждой длины замеры будут проводиться 50 раз и в результат будет заноситься средняя величина.

Таблица 4.1 — Результаты временных тестов.

Длина строки	Левенштейн нерекурсивный табличный	Левенштейн рекурсивный с проверками
100	0.0040625	0.03375
200	0.01625	0.114375
300	0.0353125	0.2653125
400	0.06875	0.4675
500	0.115625	0.7459375

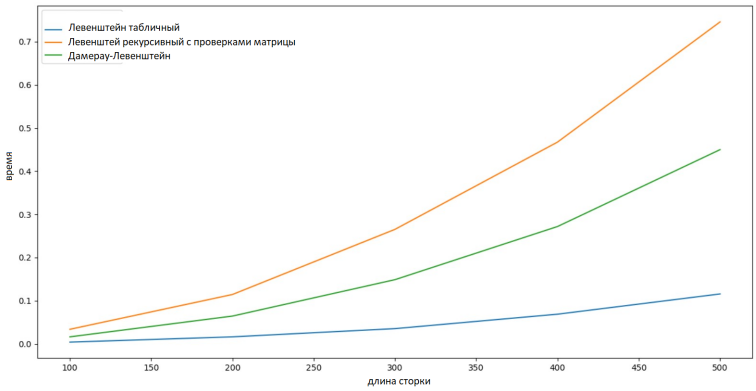

Рисунок 4.1 — Графики времени выполнения алгоритмов при различных длинах строки.

4.2 Измерение временных характеристик для рекурсивного алгоритма Левенштейна

Для измерения времени данных алгоритмов будет использовать рандомно генерирующиеся строки длиной от 1 до 9 с шагом 1. Для более точных измерений, для каждой длины замеры будут проводиться 50 раз и в результат будет заноситься средняя величина.

Таблица 4.2 — Результаты временных тестов.

Длина строки	рекурсивного алгоритма Левенштейна
1	0
2	0
3	0
4	0.0003125
5	0.0009375
6	0.005625
7	0.0303125
8	0.16625
9	0.89375

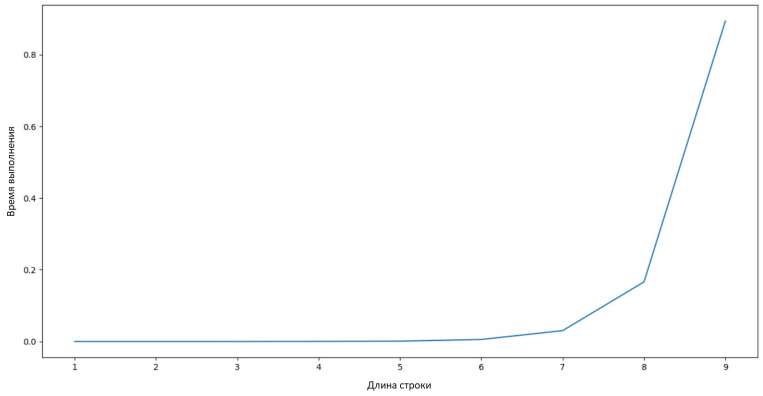

Рисунок 4.2 — График времени выполнения рекурсивного алгоритма Левенштейна.

4.3 Вывод

В результате эксперимента было получено, что на рандомно генерирующейся строке длиной от 1 до 9 с шагом 1, наименее времяёмким из всех рассмотренных алгоритмов является табличный алгоритм Левенштейна. В результате можно сделать вывод, что для строк длинной от 1 до 9 символов данных предпочтительно использовать табличный алгоритм Левенштейна.

Заключение

В процессе выполнения данной лабораторной работы были изучены алгоритмы нахождения расстояния Левенштейна (нерекурсивный, рекурсивный и рекурсивный с заполнением и дополнительными проверками матрицы) и алгоритм нахождения Дамерау-Левенштейна. Было выполнено сравнение по памяти выше перечисленных алгоритмов, в ходе которого был сделан вывод, что нерекурсивный алгоритм нахождения расстояния Левенштейна использует наименьшее количество памяти. В результате эксперимента было получено, что на рандомно генерирующейся строке длиной от 1 до 9 с шагом 1, наименее времяёмким из всех рассмотренных алгоритмов является табличный алгоритм Левенштейна. В результате можно сделать вывод, что для строк длинной от 1 до 9 символов данных предпочтительно использовать табличный алгоритм Левенштейна. Также были изучены зависимости времени выполнения алгоритмов от длины строк и были выполнены все поставленные задачи. Целью данной лабораторной работы являлось изучения алгоритмов нахождения минимального редакторского расстояния, что было успешно достигнуто.

Список литературы

[1] Майкл, Доусон. Python Programming for the Absolute Beginner, 3rd Edition / Доусон Майкл. - Прогресс книга, 2019 - P. 416.

[2] Lutz, M. The IDLE User Interface / M Lutz. - O'Reilly Media, 2013.

[3] Time. https://docs.python.org/3/library/time.html.

[4] В. И. Левенштейн. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965.

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