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 «Расстояния Левенштейна и Дамерау-Левенштейна»

Студент Чалый А.А.

Группа ИУ7 – 52Б

Оценка (баллы) _____

Преподаватели Волкова Л.Л.
Строганов Ю.В.

Москва.
2020 г.

Оглавление

Введение

Целью данной работы является изучение и применение различных реализации алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна, а также получение практических навыков реализации указанных алгоритмов.

Требуется реализовать описанные ниже алгоритмы:

1. расстояние Левенштейна (табличный способ);
2. расстояние Левенштейна (рекурсивный способ);
3. расстояние Левенштейна (рекурсивный способ с заполнением и дополнительными проверками матрицы);
4. расстояние Дамерау-Левенштейна (табличный способ).

Помимо реализации требуется изучить данные алгоритмы и понять какой из них наиболее эффективен. Для этого будут произведены замеры времени и памяти вышеперечисленных алгоритмов. По окончании данной работы нам будут известны плюсы и минусы каждого из алгоритмов, а также точные данные о использованной ими памяти и времени на их выполнение.

1. Аналитическая часть

В данном разделе будет рассмотрено описание алгоритмов нахождения 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 превращения одной строки в другую.

Эти алгоритмы активно применяются для решения следующих задач:

- исправление ошибок в слове;
- использование в поисковых системах;
- использование в биоинформатике для сравнения генов, хромосом и белков;
- использование для сравнения текстовых файлов.

1.2 Расстояние Левенштейна

Для двух строк S1 и S2 расстояние Левенштейна можно посчитать по рекуррентной формуле (1) через расстояния между подстроками.

$$d\,(S1[1..i],\ S2[1..j]) = D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & i > 0, j = 0 \\ j, & i = 0, j > 0 \\ min\begin{cases} D(i, j-1) + 1, \\ D(i-1, j) + 1, \\ D(i-1, j-1) + m(S1[i], S2[j]) \end{cases} & j > 0, i > 0 \end{cases} \quad (1)$$

Разрешены операции:

1) удаления (D, штраф – 1);
2) добавления (I, Штраф – 1);
3) замены (R, Штраф – 1);
4) совпадения (M, Штраф – 0).

В таблице 1 минимальное расстояние между словом “столб” и словом “тело” – 3.

Таблица 1. Пример преобразования слова “столб” в слово “тело”

	Ø	с	т	о	л	б
Ø	0	1	2	3	4	5

т	1	1	1	2	3	4
е	2	2	2	2	3	4
л	3	3	3	3	2	3
о	4	4	4	3	3	3

1.3 Расстояние Дамерау-Левенштейна

Расстояние нахождения расстояния Дамерау-Левенштейна является модификацией расстояния Левенштейна: ко всем прочим операциям, определенным в расстоянии Левенштейна, добавлена операция транспозиции (перестановки) символов. Модификация была введена, потому что значительная часть ошибок пользователей при наборе текста как раз и есть транспозиция.

$$D_{S1S2}(i,j)\begin{cases} max(i,j), & \text{i = 0 || j = 0} \\ \min\begin{cases} d_{S1S2}(i-1,j)+1, \\ d_{S1S2}(i,j-1)+1, \\ d_{S1S2}(i-1,j-1)+\begin{bmatrix}1_{(s1_i\ !=\ s2_j)}\\0\end{bmatrix} \\ d_{S1S2}(i-2,j-2)+1 \end{cases} & \begin{matrix}\text{i, j > 1 and} \\ S1_i = S2_{j-1} \\ \text{and } S1_{i-1} = \\ S2_j\end{matrix} \\ \min\begin{cases} d_{S1S2}(i-1,j)+1, \\ d_{S1S2}(i,j-1)+1, \\ d_{S1S2}(i-1,j-1)+\begin{bmatrix}1_{(s1_i\ !=\ s2_j)}\\0\end{bmatrix} \\ . \end{cases} & \begin{matrix}\text{В ином} \\ \text{случае}\end{matrix} \end{cases} \quad (2)$$

1.4 Вывод

Были рассмотрены расстояния Левенштейна и Дамерау-Левенштейна, принципиальная разница которого - наличие транспозиции, области применения данных расстояний могут разниться.

2. Конструкторская часть

В данном разделе будут рассмотрены схемы алгоритмов:

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

2.1 Разработка алгоритмов

На рис. 1-4 приведены схемы указанных алгоритмов. В нерекурсивном алгоритме Левенштейна в целях экономии памяти используется не вся матрица, а 2 строки, текущая и предыдущая (на рис. 1 они обозначены как массивы с названиями prev_r и cur_r).

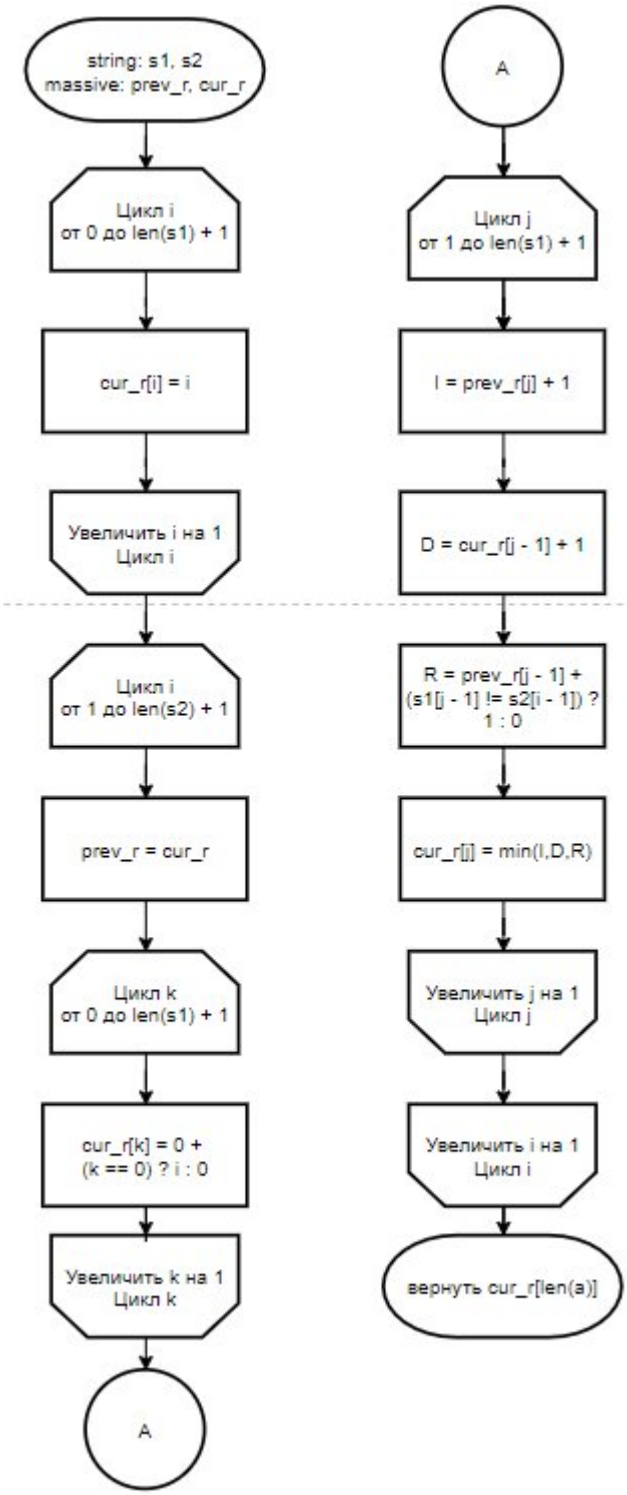

Рисунок 1. Схема алгоритма поиска расстояния Левенштейна

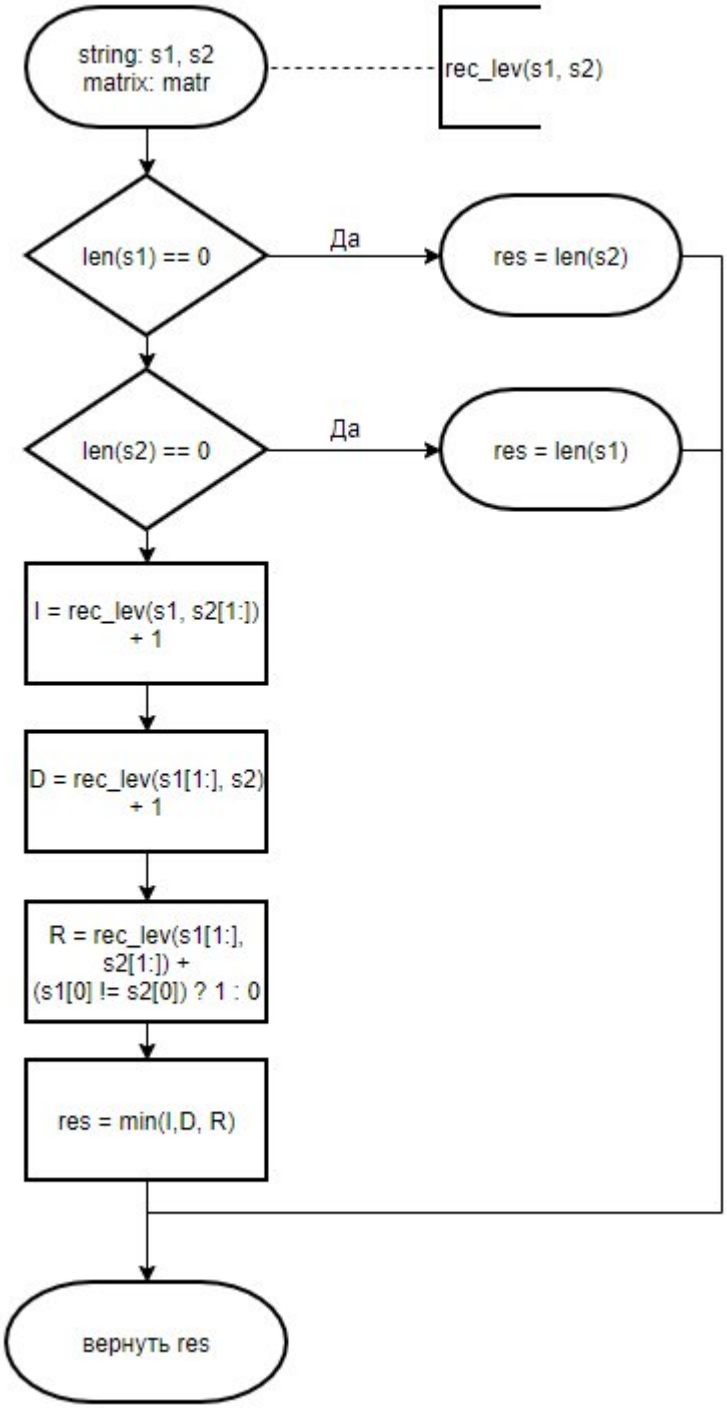

Рисунок 2. Схема рекурсивного алгоритма поиска расстояния Левенштейна

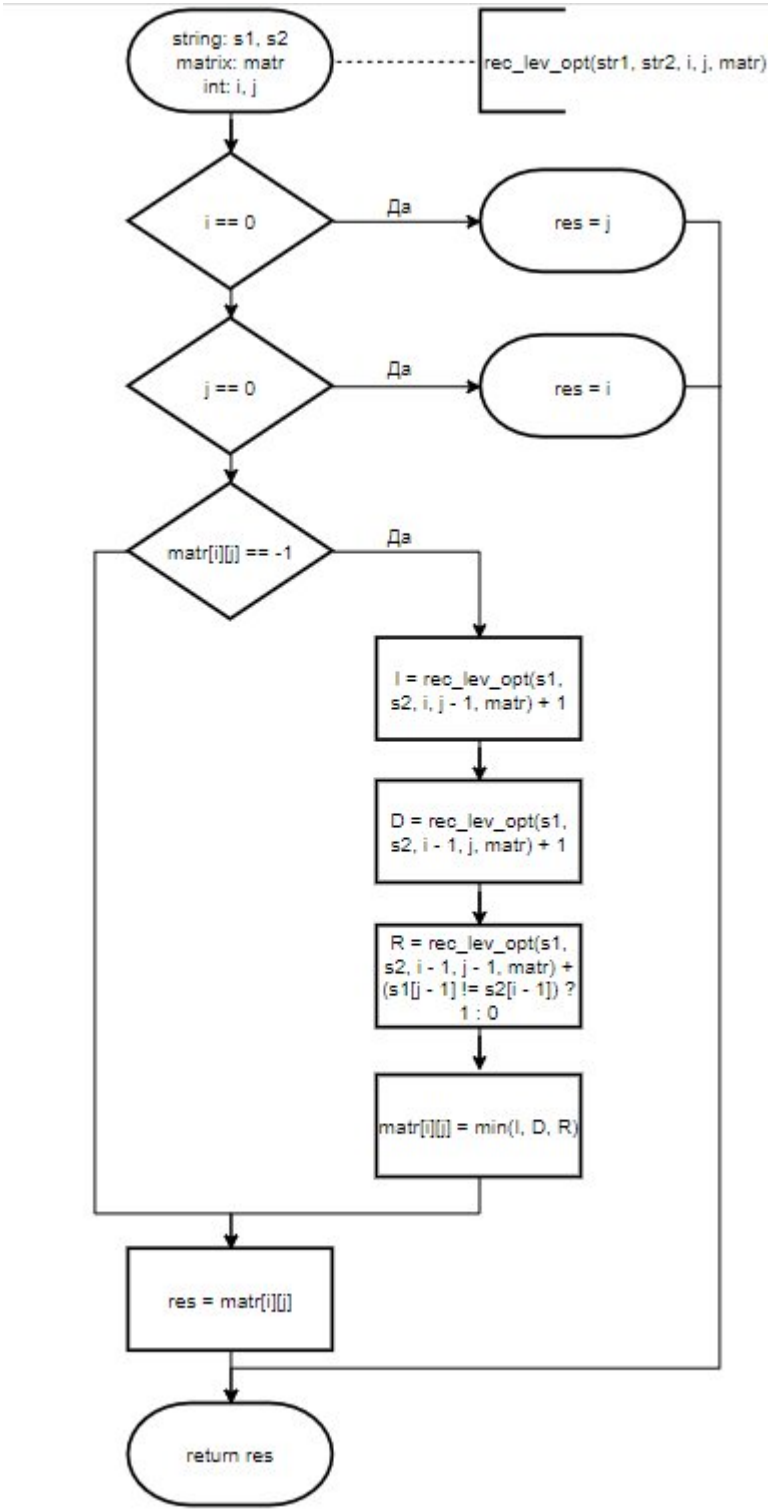

Рисунок 3. Схема рекурсивного алгоритма поиска расстояния Левенштейна с заполнением и дополнительными проверками матрицы

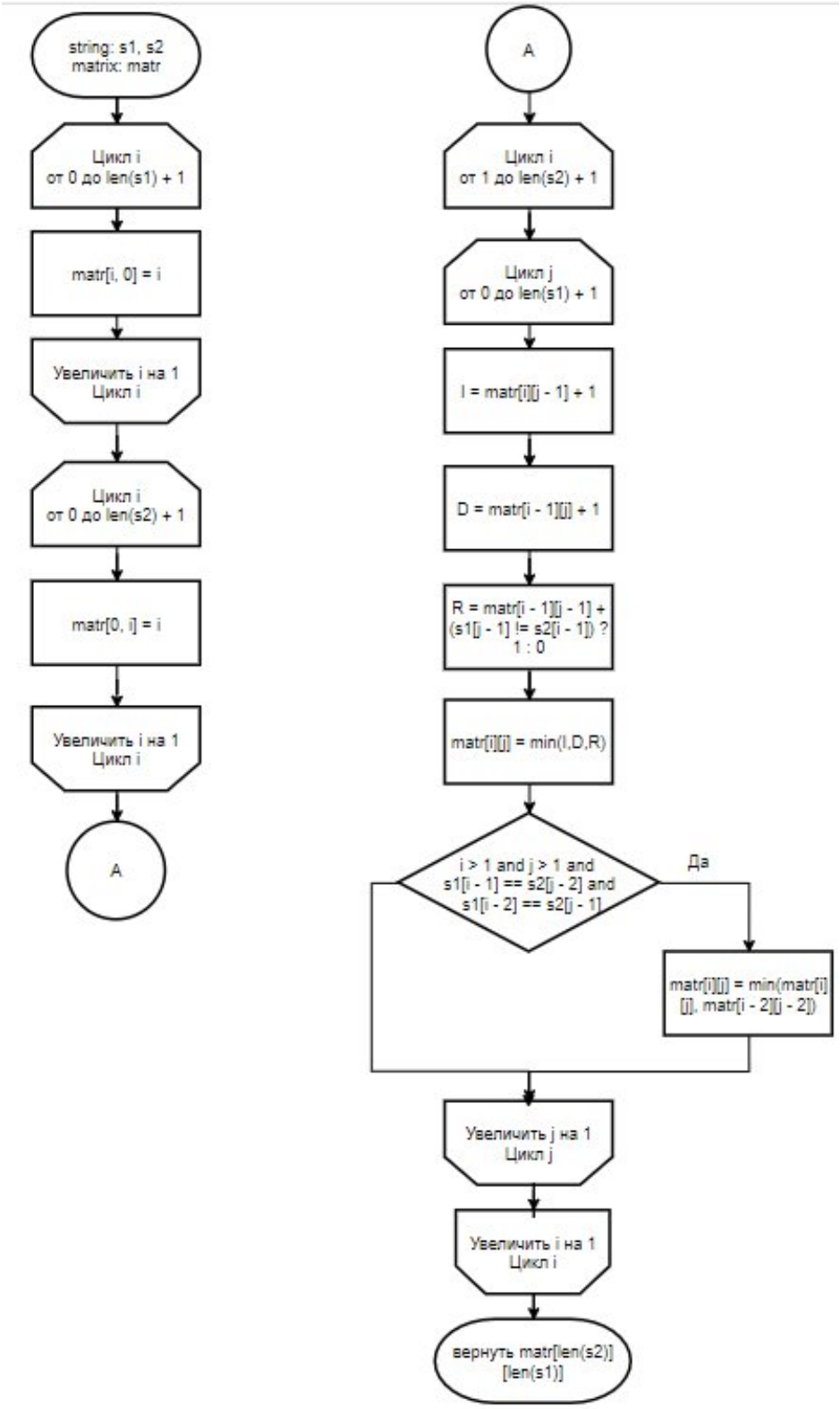

Рисунок 4. Схема алгоритма поиска расстояния Дамерау-Левенштейна

2.2 Вывод

В данном разделе были рассмотрены схемы алгоритмов поиска расстояния Левенштейна (нерекурсивного и рекурсивных без заполнения матрицы и с таковым и доп. проверками матрицы) и алгоритма поиска расстояния Дамерау-Левенштейна.

3.Технологическая часть

В данном разделе будут рассмотрены требования к программному обеспечению, средства реализации и представлен листинг кода.

3.1 Требования к программному обеспечению

Входные данные: s1 - первое слово, s2 - второе слово.

Выходные данные: значение расстояния между двумя словами

Функциональная схема процесса поиска редакционного расстояния между строками в нотации IDEF0 представлена на рис. 5.

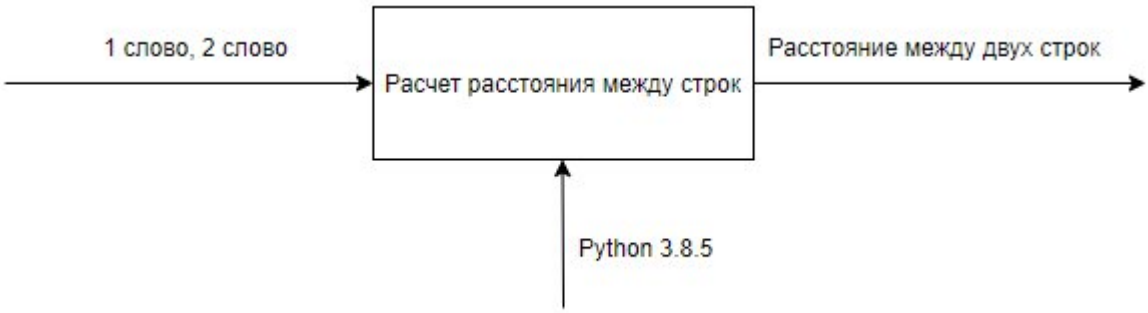

Рисунок 5. Функциональная схема процесса поиска редакционного расстояния между строками

3.2 Средства реализации

В данной работе был использован язык программирования Python. Проект выполнен в IDE PyCharm с использованием сторонней библиотеки для создания массивов numpy.

Для замера процессорного времени была использована python библиотека time.

3.3 Листинг кода

В данном пункте представлен листинг кода функций:

1) поиск расстояния Левенштейна;
2) поиск расстояния Левенштейна (рекурсивный способ);
3) поиск расстояния Левенштейна (рекурсивный способ с заполнением и дополнительными проверками матрицы);
4) поиск расстояния Дамерау-Левенштейна (табличный способ).

Листинг 1. Функция поиска расстояния Левенштейна

```
    def levenshtein(a, b):
        n, m = len(a), len(b)
        current_row = range(n + 1)
        for i in range(1, m + 1):
            previous_row, current_row = current_row, [i] + [0] * n
            for j in range(1, n + 1):
                insert, delete, replace = previous_row[j] + 1,\
                                          current_row[j - 1] + 1,\
                                          previous_row[j - 1]
                if a[j - 1] != b[i - 1]:
                    replace += 1
                current_row[j] = min(insert, delete, replace)

        return current_row[n]
```

Листинг 2. Функция поиска расстояния Левенштейна (рекурсивный способ)

```
def recursion_levenshtein(a, b):
    if not a:
        return len(b)
    if not b:
        return len(a)
    return min(recursion_levenshtein(a[1:], b[1:]) + (a[0] != b[0]),
               recursion_levenshtein(a[1:], b) + 1,
               recursion_levenshtein(a, b[1:]) + 1)
```

Листинг 3. Функция поиска расстояния Левенштейна (рекурсивный способ с заполнением и дополнительными проверками матрицы)

```
def recursion_levenshtein_opt(a, b, i, j, matr):
    if i == 0:
        return j
    if j == 0:
        return i

    if a[j - 1] == b[i - 1]:
        replace = 0
    else:
        replace = 1
    if matr[i][j] == -1:
        matr[i][j] = min(recursion_levenshtein_opt(a, b, i - 1, j - 1, matr)
+ replace,
                         recursion_levenshtein_opt(a, b, i, j - 1, matr) + 1,
                         recursion_levenshtein_opt(a, b, i - 1, j, matr) + 1)
    return matr[i][j]
```

Листинг 4. Функция поиска расстояния Дамерау-Левенштейна (табличный способ)

```
def damerau_levenshtein(matr, a, b):
    n = len(a) + 1
    m = len(b) + 1
    for i in range(1, m):
        for j in range(1, n):
            if a[i - 1] != b[j - 1]:
                replace = 1
            else:
                replace = 0
            matr[i][j] = min(
                matr[i - 1][j] + 1,  # delete
                matr[i][j - 1] + 1,  # insert
                matr[i - 1][j - 1] + replace  # replace
                )
```

```
            if i and j and a[i - 1] == b[j - 2] and a[i - 2] == b[j - 1]:
                matr[i][j] = min(matr[i][j], matr[i - 2][j - 2] + replace)  # transpose

    return matr[m - 1][n - 1]
```

3.4 Сравнительный анализ потребляемой памяти

Для проведения анализа замерим потребляемую память у разных классов, см. табл. 2.

Таблица 2. Потребляемая различными величинами память

Классы элементов	Представление в коде	Занимаемая память (байты)
Пустой numpy массив	[]	48
Numpy массив с одним элементом	[0]	56
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

Полученные данные позволяют понять, что хранение элементов массива, реализовано с помощью указателей. В связи с чем требуется просматривать занимаемую память у каждого массива, находящегося внутри массива по отдельности.

В алгоритме Левенштейна используется память, согласно табл. 3.

Таблица 3. Потребляемая память структурами данных в алгоритме нахождения расстояния Левенштейна.

Структура данных	Занимаемая память (байты)
Два массива	48 * 2 + 8 * (len(str1) + 1) + 8 * (len(str2) + 1)
Две вспомогательный переменные (int)	28
Два счетчика (int)	28
Передача параметров	25 + 25

В алгоритме Левенштейна (рекурсивный способ) используется память, согласно табл. 4.

Таблица 4. Потребляемая память структурами данных в алгоритме нахождения расстояния Левенштейна рекурсивным способом.

Структура данных	Занимаемая память (байты)
Передача параметров	25 + 25
4 переменных для подсчета IDR (I – insert, D – delete, R - replace)	56

Максимальная глубина рекурсивного вызова функции равна сумме двух строк. Исходя из этого, оценка максимальной потребляемой реализацией рекурсивного алгоритма памяти составит кол-во памяти на один вызов, умноженное на сумму длин обеих строк.

В алгоритме Левенштейна (рекурсивный способ с заполнением и дополнительными проверками матрицы) используется память, согласно табл. 5.

Таблица 5. Потребляемая память структурами данных в алгоритме нахождения расстояния Левенштейна рекурсивным способом с заполнением и дополнительными проверками матрицы.

Структура данных	Занимаемая память (байты)
Передача параметров	25 + 25 + 14 * 2 + 48
4 переменных для подсчета IDR (I – insert, D – delete, R - replace)	56
Матрица	48 + 8 * (len(str2) + 1) * (len(str1) + 1)

Примечание: Кол-во выполнений функции равно размеру матрицы MXN, где m - длина первой строки, а n - длина второй строки

В алгоритме Дамерау-Левенштейна используется память, согласно табл. 6.

Таблица 6. Потребляемая память структурами данных в алгоритме нахождения расстояния Дамерау-Левенштейна

Структура данных	Занимаемая память (байты)
Передача параметров	48 + 25 + 25
Три вспомогательный переменные (int)	42
Два счетчика (int)	28
Матрица	48 + 8 * (len(str2) + 1) * (len(str1) + 1)

3.4.1 Оценка потребляемой памяти на словах длиной 5 и 500 символов

Оценим алгоритмы на словах длиной 5 символов:

Таблица 7. Потребляемая память структурами данных в алгоритме нахождения расстояния Левенштейна.

Структура данных	Занимаемая память (байты)
Два массива	182
Две вспомогательный переменные (int)	28
Два счетчика (int)	28
Передача параметров	60
Сумма Данных	298

Таблица 8. Потребляемая память структурами данных в алгоритме нахождения расстояния Левенштейна рекурсивным способом.

Структура данных	Занимаемая память (байты)
Передача параметров	50 * 10 = 500
4 переменных для подсчета IDR (I – insert, D – delete, R - replace)	56 * 10 = 560
Сумма Данных	1060

Таблица 9. Потребляемая память структурами данных в алгоритме нахождения расстояния Левенштейна рекурсивным способом с заполнением и дополнительными проверками матрицы.

Структура данных	Занимаемая память (байты)
Передача параметров	126 * 25 = 3150
4 переменных для подсчета IDR (I – insert, D – delete, R - replace)	56 * 25 = 1400
Матрица	336
Сумма Данных	4886

Таблица 10. Потребляемая память структурами данных в алгоритме нахождения расстояния Дамерау-Левенштейна

Структура данных	Занимаемая память (байты)
Передача параметров	98
Три вспомогательный переменные (int)	42
Два счетчика (int)	28
Матрица	336
Сумма Данных	504

Оценим алгоритмы на словах длиной 500 символов:

Таблица 11. Потребляемая память структурами данных в алгоритме нахождения расстояния Левенштейна.

Структура данных	Занимаемая память (байты)

Два массива	8112
Две вспомогательный переменные (int)	28
Два счетчика (int)	28
Передача параметров	50
Сумма Данных	8218

Таблица 12. Потребляемая память структурами данных в алгоритме нахождения расстояния Левенштейна рекурсивным способом.

Структура данных	Занимаемая память (байты)
Передача параметров	50 * 500 =25000
4 переменных для подсчета IDR (I – insert, D – delete, R - replace)	56 * 500 = 28000
Сумма Данных	53000

Таблица 13. Потребляемая память структурами данных в алгоритме нахождения расстояния Левенштейна рекурсивным способом с заполнением и дополнительными проверками матрицы.

Структура данных	Занимаемая память (байты)
Передача параметров	31500000
4 переменных для подсчета IDR (I – insert, D – delete, R - replace)	14000000
Матрица	2008056
Сумма Данных	47 508 056

Таблица 14. Потребляемая память структурами данных в алгоритме нахождения расстояния Дамерау-Левенштейна

Структура данных	Занимаемая память (байты)
Передача параметров	96
Три вспомогательный переменные (int)	42
Два счетчика (int)	28
Матрица	2008056
Сумма Данных	2008222

Таким образом классический алгоритм, реализованный с помощью двух массивов эффективнее всех прочих по памяти. Также рекурсивный алгоритм поиска расстояния Левенштейна лучше Дамерау-Левенштейна при обработке длинных строк

3.5 Тестирование

В тестах будет рассмотрена работы операций:

- удаление;
- замена;

- добавление;
- перестановка.

Также будут рассмотрены следующие тесты:

- на пустую строку;
- на одинаковые строки.

Тестирование производится методом черного ящика.

На рисунках ниже будут приведены тесты с целью демонстрации корректности работы программы. Для теста 1 (рис. 5) ожидаемые результаты для расстояния Левенштейна – 1, для расстояния Дамерау-Левенштейна – 1.

```
S1 =  telo S2 =  tlo
Левенштейн D =  1
Левенштейн рек. D =  1
Левенштейн рек. с доп. проверками D =  1
Дамерау-Левенштейн D =  1
```

Рисунок 5. Проверка работы алгоритмов при удалении символа

Для теста 2 (рис. 6) ожидаемые результаты для расстояния Левенштейна – 1, для расстояния Дамерау-Левенштейна – 1.

```
S1 =  telo S2 =  talo
Левенштейн D =  1
Левенштейн рек. D =  1
Левенштейн рек. с доп. проверками D =  1
```

Рисунок 6. Проверка работы алгоритмов при замене символа

Для теста 3 (рис. 7) ожидаемые результаты для расстояния Левенштейна – 5, для расстояния Дамерау-Левенштейна – 5.

```
S1 =  blank S2 =
Левенштейн D =  5
Левенштейн рек. D =  5
Левенштейн рек. с доп. проверками D =  5
Дамерау-Левенштейн D =  5
```

Рисунок 7. Проверка работы алгоритмов при добавлении символов

Для теста 4 (рис. 8) ожидаемые результаты для расстояния Левенштейна – 2, для расстояния Дамерау-Левенштейна – 1.

```
S1 =  telo S2 =  tleo
Левенштейн D =  2
Левенштейн рек. D =  2
Левенштейн рек. с доп. проверками D =  2
Дамерау-Левенштейн D =  1
```

Рисунок 8. Проверка работы алгоритмов при транспозиции символов

Для теста 5 (рис. 9) ожидаемые результаты для расстояния Левенштейна – 0, для расстояния Дамерау-Левенштейна – 0.

```
S1 = "" S2 = ""
Левенштейн D =  0
Левенштейн рек. D =  0
Левенштейн рек. с доп. проверками D =  0
Дамерау-Левенштейн D =  0
```

Рисунок 9. Проверка работы алгоритмов при пустых строках

Для теста 6 (рис. 10) ожидаемые результаты для расстояния Левенштейна – 0, для расстояния Дамерау-Левенштейна – 0.

```
S1 = telo S2 = telo
Левенштейн D =  0
Левенштейн рек. D =  0
Левенштейн рек. с доп. проверками D =  0
Дамерау-Левенштейн D =  0
```

Рисунок 10. Проверка работы алгоритмов при одинаковых строках

Все тесты пройдены успешно.

3.6 Вывод

В данном разделе была представлена реализация алгоритмов нахождения расстояния Левенштейна (нерекурсивный способ, рекурсивный способ, рекурсивный способ с заполнением и дополнительными проверками матрицы), а также расстояния Дамерау-Левенштейна (табличный способ). Произведен анализ по потребляемой памяти, в ходе которого был сделан вывод о том, что классический алгоритм, реализованный с помощью двух массивов, эффективнее всех прочих по памяти. Также проведено тестирование разработанных методов по методу чёрного ящика.

4. Экспериментальная часть

В данном разделе будет проведен эксперимент и сравнительный анализ полученных данных.

4.1 Постановка эксперимента

В рамках данной части были проведены эксперименты, описанные ниже.

1. Сравнение времени работы нерекурсивных алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна при длине слов 5 и 10. Один эксперимент ставился 10000 раз;
2. Сравнение времени работы рекурсивных алгоритмов поиска расстояния Левенштейна без и с заполнением матрицы при длине слов 5 и 10. Один эксперимент ставился 10 раз

4.2 Сравнительный анализ на материале экспериментальных данных

Сравнение времени работы нерекурсивных алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна при длине слов 5 и 10 и количестве итераций 10000 изображены на рис. 11.

```
Number of iterations are 10000
words len is 5
dam-lev =  0.0002859375
lev =  6.25e-05
Number of iterations are 10000
words len is 10
dam-lev =  0.001125
lev =  0.0002109375
```

Рисунок 11. Время работы нерекурсивных алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна

Сравнение времени работы алгоритмов нахождения расстояния Левенштейна (рекурсивных без заполнения матрицы и с таковым и доп. проверками матрицы) при длине слов 5 и 10 и количестве итераций 10 изображены на рис. 12.

```
Number of iterations are 15
words len is 5
rec_matr =  0.0010416666666666667
rec =  0.005208333333333333
Number of iterations are 15
words len is 10
rec_matr =  0.005208333333333333
rec =  14.00625
```

Рисунок 12. Время работы алгоритмов нахождения расстояния Левенштейна (рекурсивных без заполнения матрицы и с таковым и доп. проверками матрицы)

4.3 Вывод

В данном разделе был поставлен эксперимент по замеру времени выполнения каждого алгоритма. По итогам замеров можно сделать вывод о том, что нерекурсивный алгоритм нахождения расстояния Левенштейна является самым быстрым. Нерекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна медленнее предыдущего по той причине, что там производится дополнительная проверка на транспозицию. Среди рекурсивных алгоритмов, алгоритм нахождения расстояния Левенштейна с заполнением и доп. проверками матрицы работает значительно быстрее рекурсивного алгоритма нахождения расстояния Левенштейна, по той причине, что с заполнением матрицы каждый из случаев обрабатывается один раз, когда как без данных проверок некоторые из случаев обрабатываются до 3 раз.

Заключение

В ходе работы были изучены алгоритмы нахождения расстояния Левенштейна (нерекурсивный, рекурсивный и рекурсивный с заполнением и доп. проверками матрицы) и Дамерау-Левенштейна. Выполнено сравнение по памяти всех вышеперечисленных алгоритмов в ходе которого был сделан вывод, что нерекурсивный алгоритм нахождения расстояния Левенштейна использует наименьшее кол-во памяти. Также были произведены замеры времени выполнения алгоритмов, в котором алгоритм нахождения расстояния Левенштейна также показал наилучший результат, когда как рекурсивный оказался самым медленным. Изучены зависимости времени выполнения алгоритмов от длины строк. Все поставленные задачи были выполнены. Целью лабораторной работы являлось изучение алгоритмов нахождения расстояния, что также было достигнуто.

Список использованных источников

1. time – Time access and conversions: сайт. – URL: https://docs.python.org/3/library/time.html (дата обращения 16.09.2020). – Текст: электронный.
2. Дж. Макконнелл. Анализ алгоритмов. Активный обучающий подход. – М.: Техносфера, 2017. – 267 с.