|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **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 **ПРОГРАМНОЕ ОБЕСПЕЧЕНИЕ ЭВМ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.04.03 ПРОГРАММНАЯ ИНЖЕНЕРИЯ**

**Отчет**

|  |  |
| --- | --- |
| **По лабораторной работе №** | 1 |

**Название:**

Расстояние Левенштейна и Дамерау-Левенштейна

**Дисциплина:** Анализ Алгоритмов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент | ИУ7-52Б |  |  | Н.А. Гарасев |
|  | (Группа) |  | (Подпись, дата) | (И.О. Фамилия) |
|  |  |  |  |  |
| Преподаватель |  |  |  | Л.Л. Волкова |
|  |  |  | (Подпись, дата) | (И.О. Фамилия) |

Москва, 2020

Оглавление

[Введение 3](#_Toc50566282)

[Цель 3](#_Toc50566283)

[Задачи: 3](#_Toc50566284)

[Аналитическая часть 4](#_Toc50566285)

[Конструкторская часть 5](#_Toc50566286)

[Блок-схемы 5](#_Toc50566287)

[Технологическая часть 8](#_Toc50566288)

[Реализация алгоритмов 8](#_Toc50566289)

[Тестирование функций 9](#_Toc50566290)

[Экспериментальная часть 10](#_Toc50566291)

[Интерфейс 10](#_Toc50566292)

[Тесты 10](#_Toc50566293)

[Сравнение алгоритмов по времени 12](#_Toc50566294)

[Сравнение алгоритмов по затраченной памяти 12](#_Toc50566295)

[Заключение 14](#_Toc50566296)

[Список литературы 15](#_Toc50566297)

# Введение

**Расстояние Левенштейна** — метрика, измеряющая разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (а именно вставки, удаления, замены), необходимых для превращения одной последовательности символов в другую. В общем случае, операциям, используемым в этом преобразовании, можно назначить разные цены [1].

**Расстояние Левенштейна** широко используется в теории информации и компьютерной лингвистике.

Примеры использования.

* Для исправления ошибок в слове.
* В биоинформатике для сравнения генов, белков и прочих.
* Для сравнение текстовых файлов.

Цель лабораторной работы: изучить и применить на практике алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, а также получить практические навыки реализации этих алгоритмов и сравнить их между собой.

## В ходе выполнения лабораторной работы требуется решить следующие задачи.

* Дать описание расстояний Левенштейна и Дамерау-Левенштейна
* Реализовать алгоритм нахождения расстояния Левенштейна матричным способом.
* Реализовать алгоритм нахождения расстояния Левенштейна рекурсивным способом.
* Реализовать алгоритм нахождения расстояния Левенштейна рекурсивным способом с заполнением матрицы.
* Реализовать нерекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с заполнением матрицы.
* Сравнить алгоритмы по затраченным ресурсам (времени и памяти).
* Проанализировать полученные результаты и сделать вывод.

# Аналитическая часть

Задача по нахождению расстояния Левенштейна заключается в поиске минимального количества операций *вставки, удаления, замены* для того, чтобы перейти от одной строки к другой.

Для нахождения расстояния Дамерау-Левенштейна добавляется операция *транспозиции* (перестановка соседних символов).

На вход подается две строки - S1 и S2 с длинами N и M, соответственно. Расстояние Левенштейна можно посчитать по формуле:

где m (a, b) – функция сравнения a и b, которая возвращает 0, если они равны и 1, если они не равны. Функция min (a, b, c) – возвращает минимальное из входящих значений.

# Конструкторская часть

На вход алгоритмы принимают две строки. На выходе выдают число – расстояние Левенштейна или Дамерау-Левенштейна, в зависимости от алгоритма.

# Схемы алгоритмов

На рис. 1-4 приведены схемы разработанных алгоритмов:

1.Алгоритм по нахождению расстояния Левенштейна с заполнением матрицы.

2.Рекурсивный алгоритм по нахождению расстояния Левенштейна.

3.Рекурсивый алгоритм по нахождению расстояния Левенштейна с заполнением матрицы.

4.Алгоритм по нахождению расстояния Дамерау-Левенштейна с заполнением матрицы.

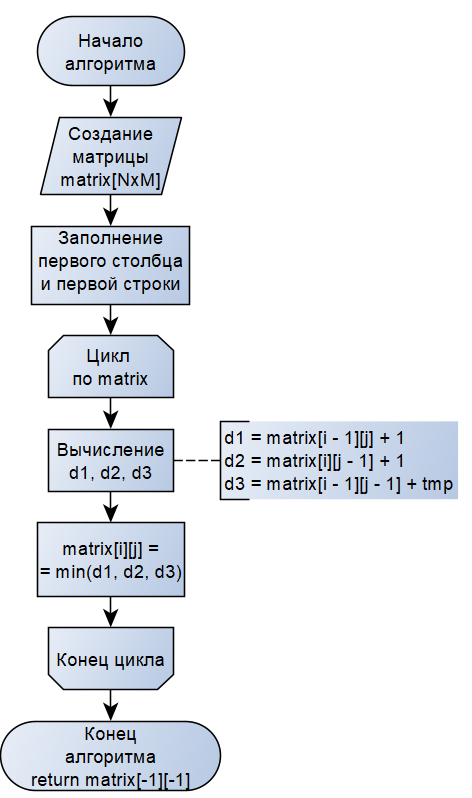
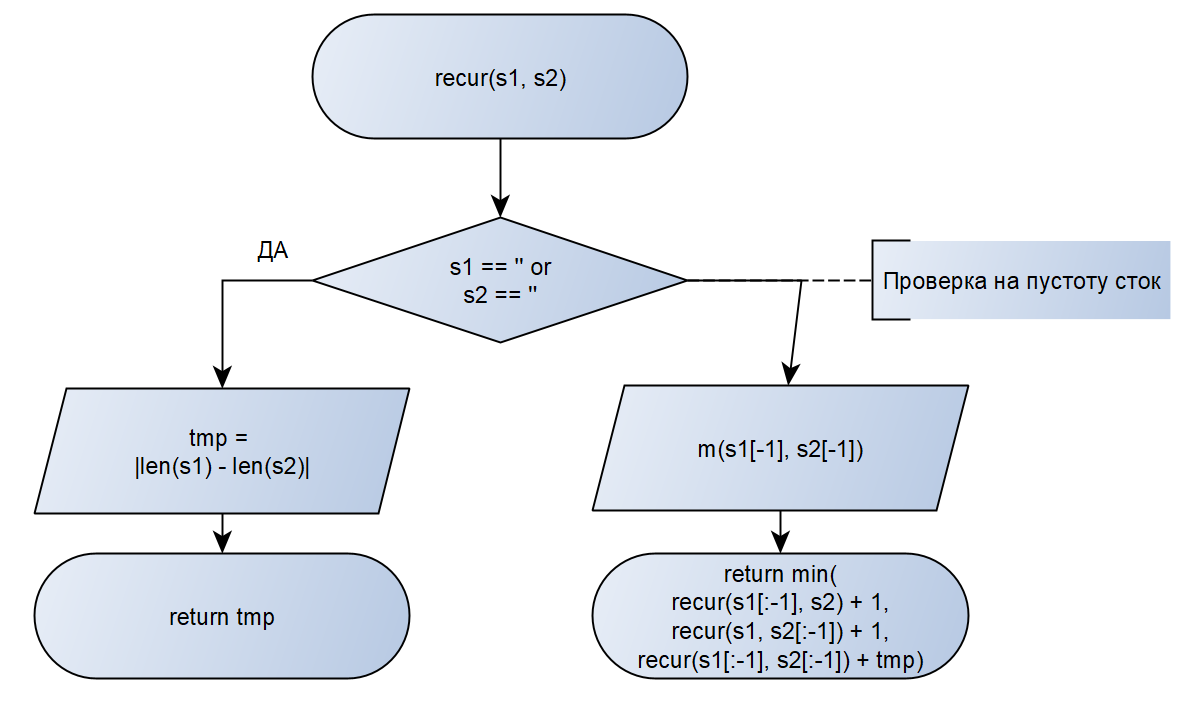
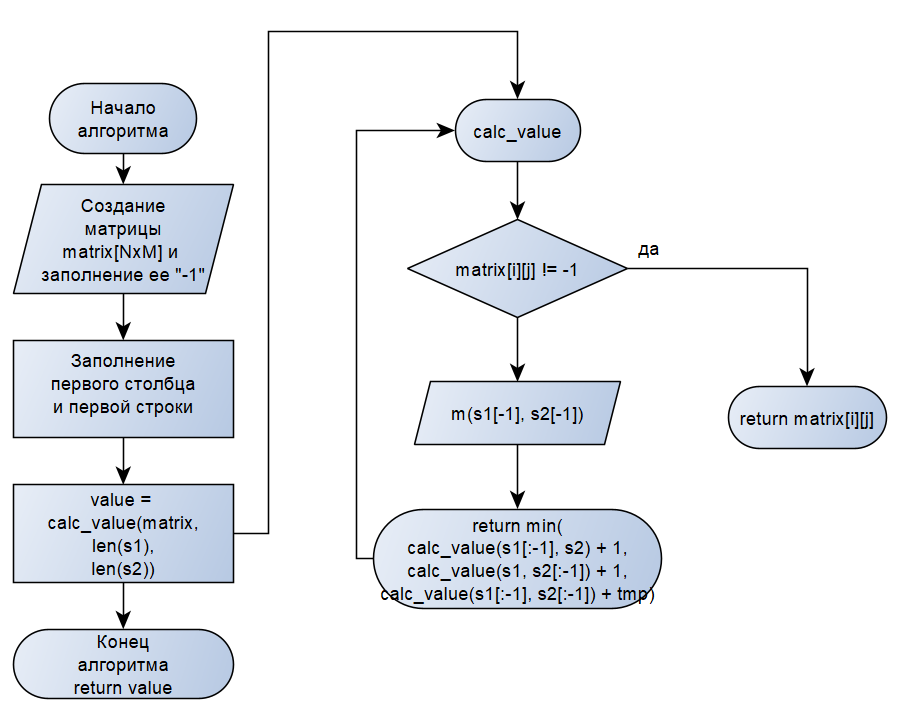


Рисунок 3. Алгоритм по нахождению расстояния Левенштейна

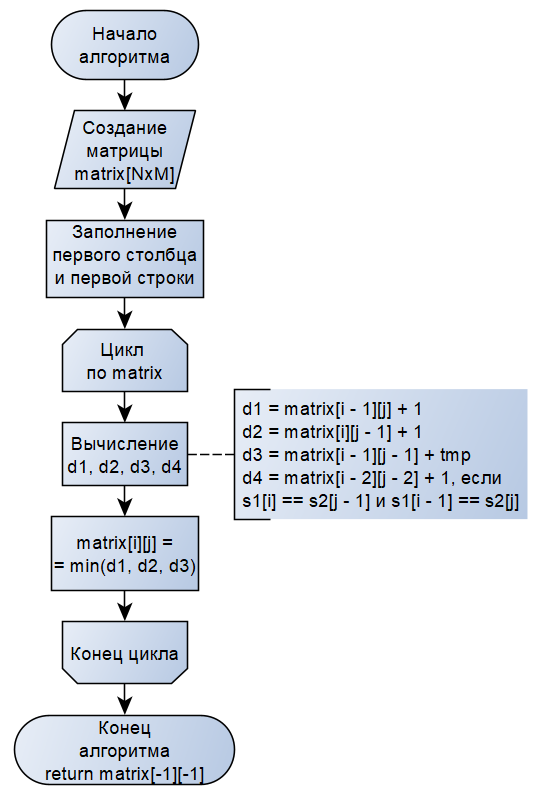


Рисунок

1. 

Рисунок

1. Дамерау-Левенштейн матричный



Рисунок

# Технологическая часть

В качестве языка программирования был выбран python, т.к. данный язык программирования позволяет написать программу за кратчайшее время. Для замера процессорного времени была использована функция process\_time(), стандартной библиотеки python - time.

## Реализация алгоритмов

В листингах 1-4 представлена реализация алгоритмов по нахождению расстояния Левенштейна и Дамерау-Левенштейна. В листинге 5 представлена функция для замера времени выполнения заданной функции на заданном количестве итераций на строках указанной длины.

Листинг 1. Реализация алгоритма по нахождению расстояния Левенштейна с заполнением матрицы

def calc\_dist\_matrix(s1, s2):  
 matr = np.eye(len(s1) + 1, len(s2) + 1)  
  
 for i in range(len(s1) + 1):  
 matr[i][0] = i  
 for j in range(len(s2) + 1):  
 matr[0][j] = j  
  
 for i in range(len(s1)):  
 for j in range(len(s2)):  
 d1 = matr[i + 1][j] + 1  
 d2 = matr[i][j + 1] + 1  
 if s1[i] == s2[j]:  
 d3 = matr[i][j]  
 else:  
 d3 = matr[i][j] + 1  
 matr[i + 1][j + 1] = min(d1, d2, d3)

Листинг 2. Реализация рекурсивного алгоритма по нахождению расстояния Левенштейна

def calc\_dist\_recur(s1, s2):  
 if s1 == **''** or s2 == **''**:  
 return abs(len(s1) - len(s2))  
  
 tmp = 0 if (s1[-1] == s2[-1]) else 1  
 return min(calc\_dist\_recur(s1[:-1], s2) + 1,  
 calc\_dist\_recur(s1, s2[:-1]) + 1,  
 calc\_dist\_recur(s1[:-1], s2[:-1]) + tmp)

Листинг 3. Реализация рекурсивного алгоритма по нахождению расстояния Левенштейна с заполнением матрицы

def calc\_dist\_recur\_matrix(s1, s2):  
 def calc\_value(matr, i, j):  
 if matr[i][j] != -1:  
 return matr[i][j]  
 else:  
 tmp = 0 if (s1[i - 1] == s2[j - 1]) else 1  
 matr[i][j] = min(calc\_value(matr, i - 1, j) + 1,  
 calc\_value(matr, i, j - 1) + 1,  
 calc\_value(matr, i - 1, j - 1) + tmp)  
 return matr[i][j]  
  
 matr = np.full((len(s1) + 1, len(s2) + 1), -1)  
 for i in range(len(s1) + 1):  
 matr[i][0] = i  
 for j in range(len(s2) + 1):  
 matr[0][j] = j  
 value = calc\_value(matr, len(s1), len(s2))

Листинг 4. Реализация алгоритма по нахождению расстояния Дамерау-Левенштейна с заполнением матрицы

def calc\_dist\_damerau(s1, s2):  
 matr = np.eye(len(s1) + 1, len(s2) + 1)  
  
 for i in range(len(s1) + 1):  
 matr[i][0] = i  
 for j in range(len(s2) + 1):  
 matr[0][j] = j  
  
 for i in range(len(s1)):  
 for j in range(len(s2)):  
 d1 = matr[i + 1][j] + 1  
 d2 = matr[i][j + 1] + 1  
 if s1[i] == s2[j]:  
 d3 = matr[i][j]  
 else:  
 d3 = matr[i][j] + 1  
 if s1[i] == s2[j - 1] and s1[i - 1] == s2[j] and i > 0 and j > 0:  
 d4 = matr[i - 1][j - 1] + 1  
 else:  
 d4 = d1  
 matr[i + 1][j + 1] = min(d1, d2, d3, d4)

Листинг 5. Функция подсчета среднего времени выполнения программы для строк длиной length.

def time\_analyze(function, iterations, length=5):  
 t1 = process\_time()  
 for \_ in range(iterations):  
 s1 = random\_string(length)  
 s2 = random\_string(length)  
 function(s1, s2, False)  
 t2 = process\_time()  
 return (t2 - t1) / iterations

## Тестирование функций

Для модульного тестирования реализованных алгоритмах (см. листинги 1-4) была использована стандартная библиотека языка python – unittest. Тесты приведены в листинге 6. Все функции протестированы на пустые входящие строки, а также на различные другие входные данные. Для алгоритма Дамерау – Левенштейна существует дополнительные отдельные тесты на транспозицию.

Листинг 6. Проверка на пустоту строк

def test\_empty(self):  
 self.assertEqual(self.function(**''**, **''**), 0)  
 self.assertEqual(self.function(**'a'**, **''**), 1)  
 self.assertEqual(self.function(**''**, **'a'**), 1)

Листинг 7. Проверка на

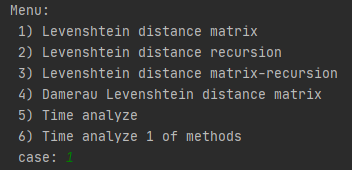
Все тесты пройдены успешно.

# Экспериментальная часть

При запуске программы первое, что видит пользователь, – это интуитивно понятный интерфейс.

## Интерфейс

На рис. 7 представлено главное меню программы. В зависимости от выбранного пункта….



Рисунок

На рис. \_-\_ приведены примеры работы программы при вводе строк «*тело*» и «*столб*» при выборе пунктов меню 1-4.

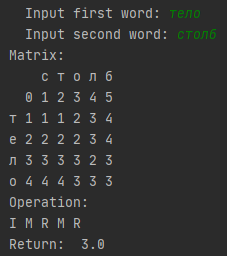
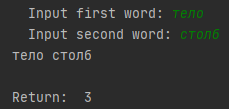


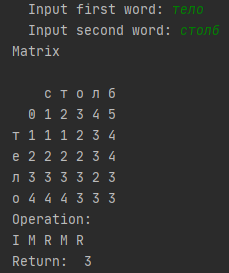
Рисунок – Пример работы алгоритма …. Матричный Левенштейн

1. Рекурсивный Левенштейн



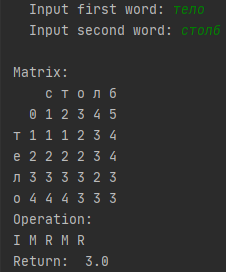
Рисунок

1. Рекурсивный с заполнением матрицы Левенштейн



Рисунок

1. Матричный Дамерау-Левенштейн



Рисунок

Далее можно сравнить алгоритмы по времени. Для этого в программе мы замеряем процессорное время для выполнения n – количества вычислений для строк одинаковой длины.

## Сравнение алгоритмов по времени работы реализаций

Для сравнения в программе необходимо провести замеры процессорное время для выполнения n – количества вычислений для строк одинаковой длины.

Таблица 1. Сравнение алгоритмов по времени

|  |  |  |
| --- | --- | --- |
| Алгоритм | Длина строк | Среднее время, сек |
| Левенштейн матричный | 5 | 0.00008750 |
| Левенштейн матричный | 10 | 0.00029531 |
| Левенштейн рекурсивный | 5 | 0.00123438 |
| Левенштейн рекурсивный | 10 | 5.73437500 |
| Левенштейн рекурсивный с заполнением матрицы | 5 | 0.00012969 |
| Левенштейн рекурсивный с заполнением матрицы | 10 | 0.00046875 |
| Дамерау-Левенштейн | 5 | 0.00008906 |
| Дамерау-Левенштейн | 10 | 0.00032188 |

Все вычисления производились на случайных строках при более чем 1000 итераций, кроме рекурсивных …...

**Вывод**: у матричных алгоритмов время пропорционально квадрату длины строк (при увеличении строки в два раза, время увеличивается в четыре раза). Рекурсивный метод показывает наихудшее время, это связано с повторным вычислением одних и тех же значений (пример ). Матричный Левенштейн и матричный Дамерау-Левенштейн показывают практически одинаковое значение. Матричный Левенштейн немного быстрее, однако необходимо учитывать то, что Дамерау-Левенштейн отвечает на другую задачу.

Сравнение алгоритмов по затраченной памяти**:**

Все значения получены с помощью стандартной библиотеки python – sys.

1. Матричный Левенштейн:

|  |  |
| --- | --- |
| Тип | Размер, байт |
| Матрица N\*M | 56 + 8 \* N \* M |
| Переменные | 4 \* 16 |
| Передача параметров | 50 + n + m |
| **Сумма:** | **154 + 8 \* N \* M + N + M** |

Таблица 2. Необходимая память

1. Рекурсивный Левенштейн

|  |  |
| --- | --- |
| Тип | Размер, байт |
| Переменные | 16 \* max\_depth |
| Передача параметров | (50 + n + m) \* max\_depth \* avr\_len |
| **Сумма:** | **(66 + n + m) \* max\_depth** |

Таблица 3. Необходимая память

1. Рекурсивный с заполнением матрицы Левенштейн

|  |  |
| --- | --- |
| Тип | Размер, байт |
| Матрица | 56 + 8 \* n \* m |
| Переменные | 16 \* max\_depth |
| Передача параметров | (50 + n + m + 56) \* max\_depth |
| **Сумма:** | **56 + 8 \* n \* m + (66+ n + m) \* max\_depth** |

Таблица 4. Необходимая память

1. Матричный Дамерау-Левенштейн

|  |  |
| --- | --- |
| Тип | Размер, байт |
| Матрица | 56 + 8 \* n \* m |
| Переменные | 80 |
| Передача параметров | 50 + n + m |
| **Сумма:** | **186 + 8 \* n \* m + n + m** |

Таблица 5. Необходимая память

В примерах выше max\_depth – максимальная глубина рекурсии (КАКАЯ ?), avr\_len – средняя длина строки

**Вывод:** при большой длине строк, матричные методы занимают огромное количество памяти, в отличии от рекурсивного алгоритма. Сравнивая алгоритмы, можно прийти к тому, что память для рекурсивного алгоритма зависит от n (длины строк), как 2\*n2 + 66\*n, в то время, как память, необходимая для матричного алгоритма – (8 \* n^2 + 2\*n), это говорит нам о том, что до n равном 10 включительно по памяти выигрывает матричный метод, начиная с длины строки 11 и более, рекурсивный алгоритм тратит меньше памяти чем матричный.

Заключение**:**

В ходе работы были изучены алгоритмы нахождения расстояния Левенштейна и Дамерау–Левенштейна различными алгоритмами. Было выполнено сравнение разных алгоритмов по нахождению расстояния Левенштейна по затраченным ресурсам. Было установлено, что рекурсивный алгоритм занимает гораздо меньше памяти при работе со строками большой длины, чем матричные алгоритмы. Однако матричные алгоритмы отмечаются своим быстродействием. Также в ходе работы были реализован программный код алгоритмов по нахождению расстояния Левенштейна.

Список литературы**:**

1. Дж. Макконнелл. Анализ алгоритмов. Активный обучающий подход. –

М.: Техносфера, 2017. – \_\_\_ c.