|  |  |
| --- | --- |
| 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](#_Toc51171214)

[1. Аналитическая часть 4](#_Toc51171217)

[2. Конструкторская часть 5](#_Toc51171218)

[2.1. Схемы алгоритмов 5](#_Toc51171219)

[3. Технологическая часть 8](#_Toc51171220)

[3.1. Реализация алгоритмов 8](#_Toc51171221)

[3.2. Тестирование функций 9](#_Toc51171222)

[4. Экспериментальная часть 11](#_Toc51171223)

[4.1. Интерфейс 11](#_Toc51171224)

[4.2. Сравнение алгоритмов по времени работы реализаций 12](#_Toc51171225)

[4.3. Сравнение алгоритмов по затраченной памяти 13](#_Toc51171226)

[Заключение 15](#_Toc51171227)

[Список литературы 16](#_Toc51171228)

# 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. В частности, оно применимо для решения следующих задач.

1. Для исправления ошибок в слове.
2. В биоинформатике для сравнения генов, белков и прочих.
3. 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.

* 1. Дать описание расстояний Левенштейна и Дамерау-Левенштейна.
  2. Реализовать алгоритм нахождения расстояния Левенштейна матричным способом.
  3. Реализовать алгоритм нахождения расстояния Левенштейна рекурсивным способом.
  4. Реализовать алгоритм нахождения расстояния Левенштейна рекурсивным способом с заполнением матрицы.
  5. Реализовать нерекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с заполнением матрицы.
  6. Сравнить алгоритмы по затраченным ресурсам (времени и памяти).
  7. Проанализировать полученные результаты и сделать вывод о временной и емкостной эффективности реализованных алгоритмов.

# Аналитическая часть

Задача нахождения 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 символов 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.

Расстояние Дамерау-Левенштейна можно посчитать по формуле:

# 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 нахождения расстояния Левенштейна с заполнением матрицы;
2. рекурсивный алгоритм нахождения расстояния Левенштейна;
3. рекурсивый алгоритм нахождения расстояния Левенштейна с заполнением матрицы;
4. алгоритм нахождения расстояния Дамерау-Левенштейна с заполнением матрицы.

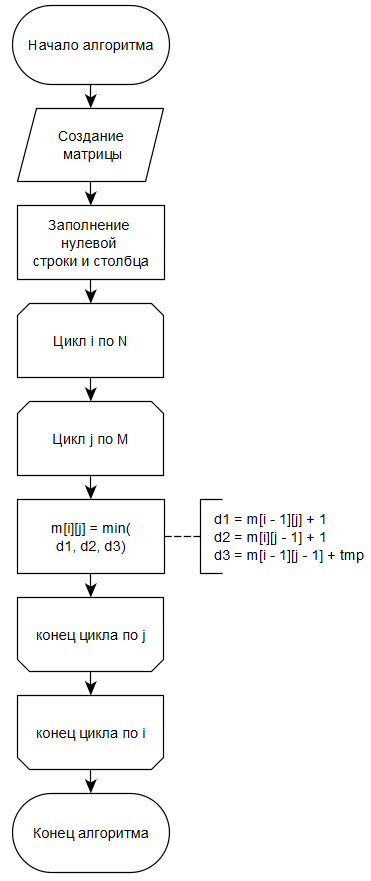


Рисунок 1. *Алгоритм нахождения расстояния Левенштейна с заполнением матрицы*

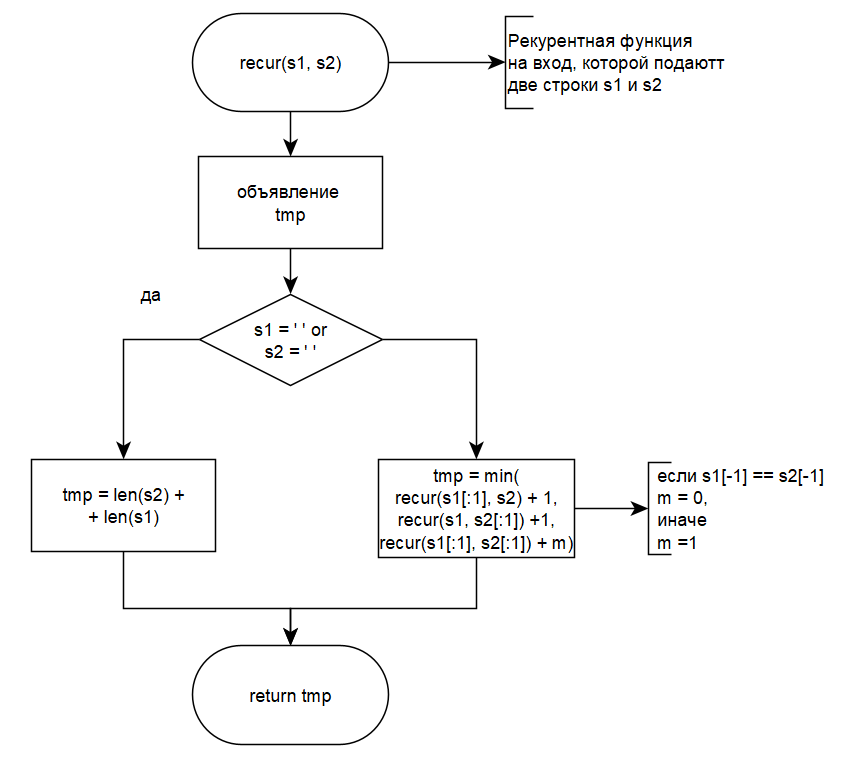


Рисунок 2. *Рекурсивный алгоритм нахождения расстояния Левенштейна*

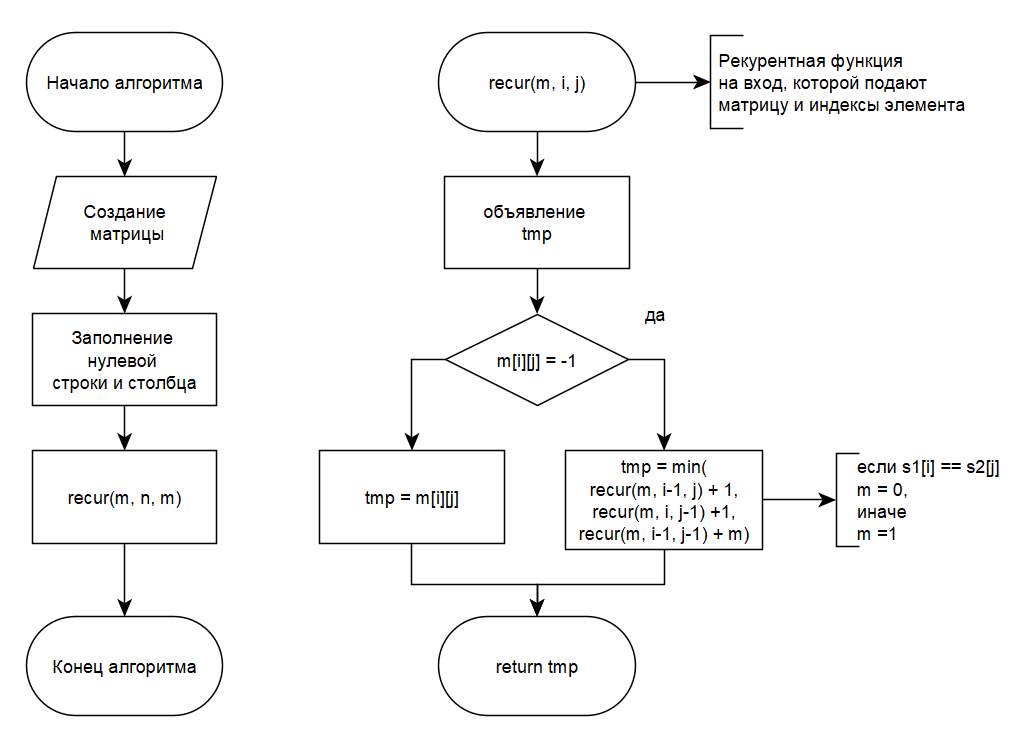
**

Рисунок 3. *Рекурсивный алгоритм нахождения расстояния Левенштейна с заполнением матрицы*

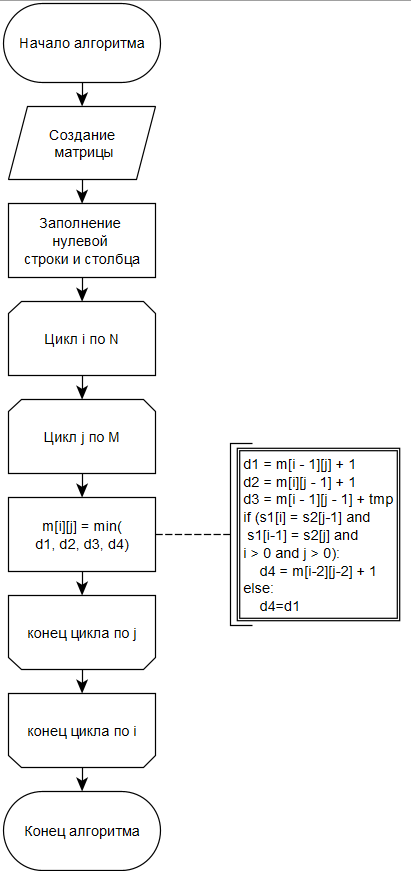
ё

Рисунок 4. *Алгоритм нахождения расстояния Дамерау-Левенштейна с заполнением матрицы.*

# Технологическая часть

В качестве языка программирования был выбран python, т.к. данный язык программирования позволяет написать программу за короткое время. Для замера процессорного времени была использована функция process\_time(), стандартной библиотеки python – time [2].

## Реализация алгоритмов

В листингах 1-4 представлена реализация алгоритмов нахождения расстояний 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 нахождения 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 нахождения 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 нахождения 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 нахождения 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. Модульные тесты приведены в листингах 6-7. Все функции протестированы на пустые входящие строки, а также на различные другие входные данные. Для алгоритма поиска расстояния 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 выполнение операций

def test\_different(self):  
 *# Match* self.assertEqual(self.function(**'a'**, **'a'**), 0)  
 self.assertEqual(self.function(**'c'**, **'c'**), 0)  
 *# Delete* self.assertEqual(self.function(**'ab'**, **'a'**), 1)  
 self.assertEqual(self.function(**'op'**, **'o'**), 1)  
 *# Insert* self.assertEqual(self.function(**'a'**, **'ab'**), 1)  
 self.assertEqual(self.function(**'o'**, **'op'**), 1)  
 *# Replace* self.assertEqual(self.function(**'ab'**, **'aс'**), 1)  
 self.assertEqual(self.function(**'op'**, **'od'**), 1)

Все тесты пройдены успешно.

# Экспериментальная часть

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

## Интерфейс

На рис. 5 представлено главное меню программы. В зависимости от выбранного пункта меню запускается соответствующий алгоритм

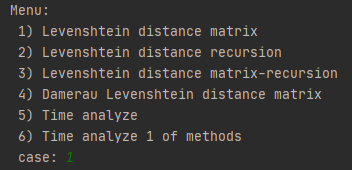


Рисунок 5. Интерфейс – меню

На рис. 6-9 приведены примеры работы программы при вводе строк «*тело*» и «*столб*» при выборе пунктов меню 1-4.

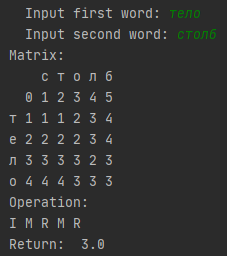


Рисунок 6. Пример работы программы при выборе пункта 1

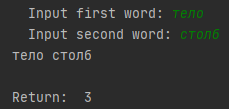


Рисунок 7. Пример работы программы при выборе пункта 2

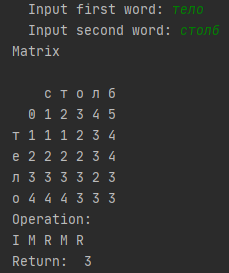


Рисунок 8. Пример работы программы при выборе пункта 3

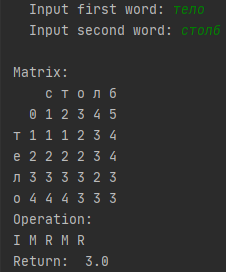


Рисунок 9. Пример работы программы при выборе пункта 4

## Сравнение алгоритмов по времени работы реализаций

Для сравнения в программе необходимо провести замеры процессорного времени 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 |

Замеры времени усреднялись для каждого набора одинаковых экспериментов. Для этого все вычисления производились на случайных строках при 1000-10000 итераций, кроме рекурсивных, для которых количество итераций было в пределах 20-50, это связано с временем выполнения данных алгоритмов.

**Вывод**: у матричных алгоритмов время пропорционально квадрату длины строк (при увеличении строки в два раза, время увеличивается в четыре раза). Рекурсивный алгоритм показывает наихудшее время, это связано с повторным вычислением одних и тех же значений (каждый вызов порождает три новых вызова, у которых могут дублироваться входные значения). Матричные алгоритмы по поиску расстояния Левенштейна и Дамерау-Левенштейна показывают самое лучшее время. Первый алгоритм немного быстрее, однако необходимо учитывать то, что второй отвечает на другую задачу.

* 1. Сравнение алгоритмов по затраченной памяти**:**

Все значения получены с помощью стандартной библиотеки python – sys.

В табл. 2-5 приведены оценки максимальной затрачиваемой реализациями алгоритмов памяти.

Таблица 2. Память, необходимая для реализации алгоритма   
нахождения расстояния Левенштейна с заполнением матрицы

|  |  |
| --- | --- |
| 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 3. Память, необходимая для реализации рекурсивного   
алгоритма нахождения расстояния Левенштейна

|  |  |
| --- | --- |
| Тип | Размер, байт |
| Переменные | 16 \* max\_depth |
| Передача параметров | (50 + n + m) \* max\_depth |
| Возврат | 16 \* max\_depth |
| **Сумма:** | **(82 + n + m) \* max\_depth** |

Таблица 4. Память, необходимая для реализации рекурсивного алгоритма  
 нахождения расстояния Левенштейна с заполнением матрицы

|  |  |
| --- | --- |
| 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 5. Память, необходимая для реализации алгоритма  
нахождения расстояния Дамерау-Левенштейна с заполнением матрицы

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

В примерах выше max\_depth – максимальная глубина рекурсии, которая равна сложению длин входных строк.

**Вывод:** при большой длине строк, матричные методы занимают огромное количество памяти, в отличие от рекурсивного алгоритма. Сравнивая алгоритмы по затрачиваемой памяти, можно прийти к тому, что память для рекурсивного алгоритма зависит от n (длины строк) как 4\*n2 + 164\*n, в то время, как память, необходимая для матричного алгоритма, – (8 \* n^2 + 2\*n + 180), это говорит нам о том, что до n равного 31 включительно по памяти выигрывает матричный алгоритм, а начиная с длины строки 32 и более, рекурсивный алгоритм тратит меньше памяти, чем матричный.

Заключение

В ходе работы были изучены алгоритмы нахождения расстояний Левенштейна и Дамерау–Левенштейна. Реализованы 4 алгоритма поиска этих расстояний, приведен программный код реализации алгоритмов нахождения расстояний.

Было выполнено сравнение разных алгоритмов нахождения 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.

Цель работы достигнута. Алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна применить на практике, получены практические навыки реализации этих алгоритмов.

Список литературы

1. Дж. Макконнелл. Анализ алгоритмов. Активный обучающий подход. –М.: Техносфера, 2017. – 267 c.

2. Официальный сайт Python, документация [Электронный ресурс]. Режим доступа: <https://docs.python.org/3/library/time.html>, свободный (дата обращения: 16.09.20).