**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.

**Расстояние Левенштейна** используется в теории информации и компьютерной лингвистике для:

* Исправления ошибок в слове
* В биоинформатике для сравнения генов, белков и прочих
* 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.

**Задачи**:

* Реализовать алгоритм нахождения расстояния Левенштейна матричным способом
* Реализовать алгоритм нахождения расстояния Левенштейна рекурсивным методом
* Реализовать алгоритм нахождения расстояния Левенштейна рекурсивным методом с заполнением матрицы
* Реализовать алгоритм нахождения расстояния Дамерау-Левенштейна матричным методом
* 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 изображенной на *рис. 1*

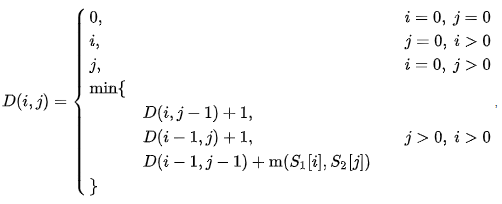


Рисунок 1

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.

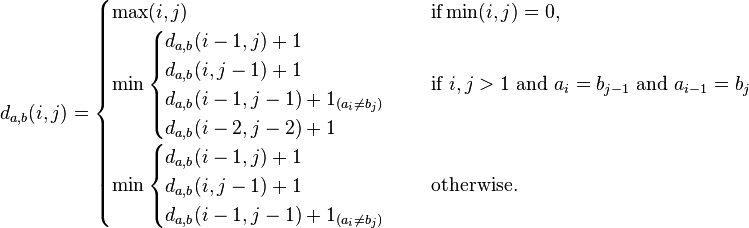


Рисунок 2

**Конструкторская часть**

На вход подаются две строки для любого из алгоритмов.

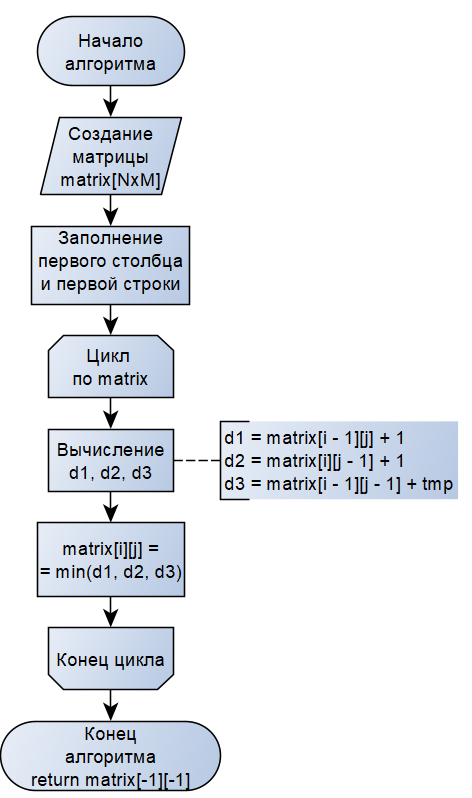
1. Левенштейн матричный 

Рисунок 3

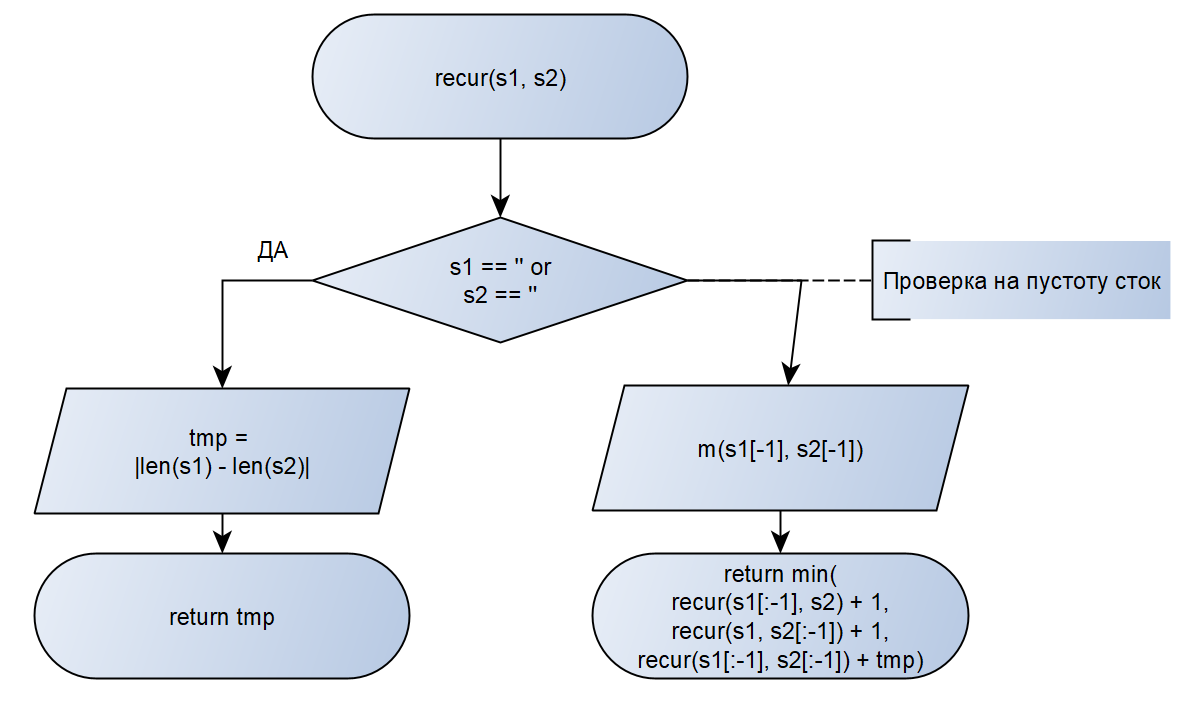
1. Левенштейн рекурсивный 

Рисунок 4

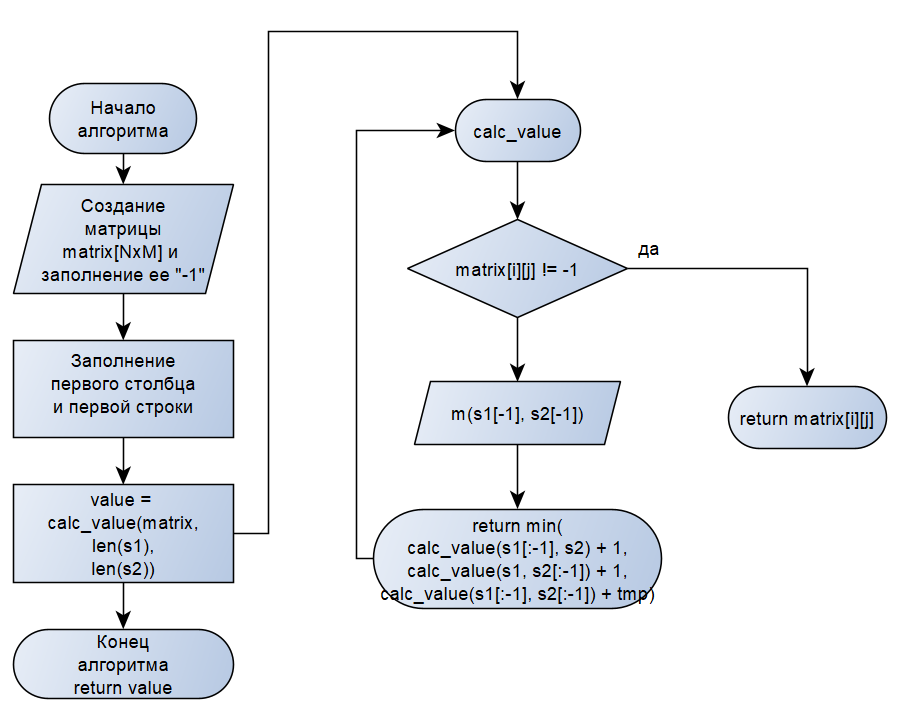
1. Левенштейн рекурсивный с заполнением матрицы 

Рисунок 5

1. Дамерау-Левенштейн матричный

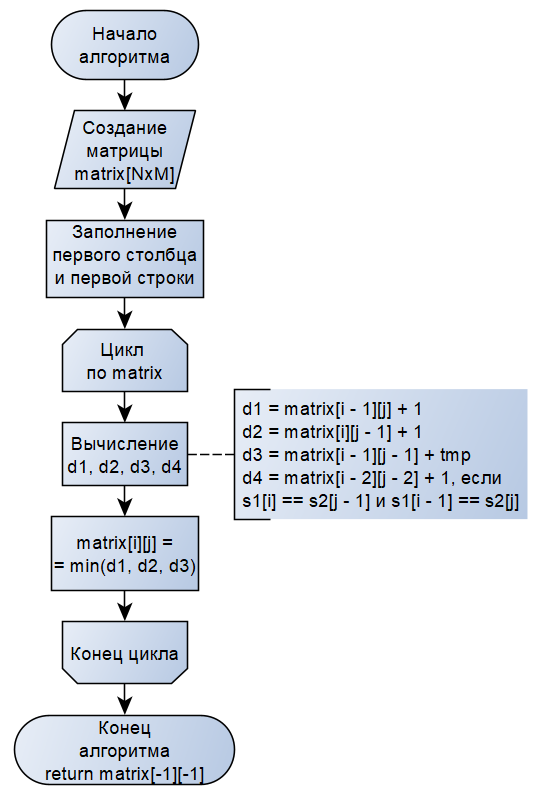


Рисунок 6

**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 стандартная библиотека time.

def calc\_dist\_matrix(s1, s2, printable=False):  
 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 1. Левенштейн матричный

def calc\_dist\_recur(s1, s2, printable=False):  
 if debug or printable:  
 print(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 2. Левенштейн рекурсивный

def calc\_dist\_recur\_matrix(s1, s2, printable=False):  
 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 3. Левенштейн рекурсивный с заполнением матрицы

def calc\_dist\_damerau(s1, s2, printable=False):  
 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 4. Дамерау-Левенштейн матричный

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

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

**Сравнение алгоритмов по времени**

Составим таблицу замеров

|  |  |  |
| --- | --- | --- |
| Метод | 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.

**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. 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:**