

Отчет по лабораторной работе №1  
по курсу "Анализ алгоритмов"  
по теме "Расстояние Левенштейна"

Студент: Барсуков Н.М. ИУ7-56  
Преподаватель: Волкова Л.Л., Строганов Ю.В.

# Содержание

<b>1 Аналитическая часть</b>	<b>2</b>
1.1 Постановка задачи . . . . .	2
1.2 Описание алгоритма . . . . .	2
<b>2 Конструкторский раздел</b>	<b>3</b>
2.1 Алгоритм . . . . .	3
2.1.1 Итеративный алгоритм . . . . .	3
2.1.2 Рекурсивный алгор . . . . .	3
2.2 Типы и структуры данных . . . . .	3
2.3 Структура программы . . . . .	3
<b>3 Технологический раздел</b>	<b>6</b>
3.1 Требования . . . . .	6
3.2 Выбор языка и среды разработки . . . . .	6
3.3 Интерфейс . . . . .	6
3.4 Листинг . . . . .	6
3.4.1 Итеративный Левенштейн . . . . .	6
3.4.2 Левенштейн рекурсивный . . . . .	7
3.4.3 Левенштейн модифицированный . . . . .	8
<b>4 Исследовательский раздел</b>	<b>10</b>
4.1 Сравнение . . . . .	10
4.1.1 Общие результаты замеров . . . . .	10
4.1.2 Детальное сравнение . . . . .	10
<b>5 Вывод</b>	<b>11</b>
<b>6 Заключение</b>	<b>12</b>
<b>7 Список источников</b>	<b>13</b>

# 1 Аналитическая часть

## 1.1 Постановка задачи

Изучить, реализовать и сравнить три версии алгоритма Левенштейна поиска минимального редакционного расстояния

1. Рекурсивный алгоритм с тремя операциями (вставки, удаления, замены)
2. Итеративный алгоритм с тремя операциями
3. Итеративный алгоритм с четырьмя операциями (обмен двух соседних букв)

## 1.2 Описание алгоритма

Расстояние Левенштейна (также редакционное расстояние или дистанция редактирования) между двумя строками в теории информации и компьютерной лингвистике — это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Каждая операция имеет свою оценку штрафа

1. Вставка в S1 (I) - 1
2. Удаление из S1 (D) - 1
3. Замена символа в S1 (R) - 1
4. Совпадение символов в S1 и S2 - 0

## 2 Конструкторский раздел

### 2.1 Алгоритм

Допустим, что существует две строки S1 и S2 над некоторым алфавитом. Длина одной из них - M, второй - N. Для нахождения расстояния Левенштейна между ними D(S1, S2) можно применить следующую формулу (D(S1, S2) == D(M, N)) [Afanasyev92]:

$$D(i, j) = \begin{cases} \max(i, j) & \text{if } \min(i, j) == 0 \\ \min \begin{cases} D(i, j-1) + 1 \\ D(i-1, j) + 1 \\ D(i-1, j-1) + (S1[i] \neq S2[j]) \end{cases} \end{cases}$$

Для случая с 4 операциями формула принимает следующий вид [Vasylenko92]:

$$D(i, j) = \begin{cases} \max(i, j) & \text{if } \min(i, j) == 0 \\ \min \begin{cases} D(i, j-1) + 1 \\ D(i-1, j) + 1 \\ D(i-1, j-1) + (S1[i] \neq S2[j]) \end{cases} & \text{if } i, j > 1 \\ \min \begin{cases} D(i, j-1) + 1 \\ D(i-1, j) + 1 \\ D(i-1, j-1) + (S1[i] \neq S2[j]) \end{cases} \end{cases}$$

#### 2.1.1 Итеративный алгоритм

#### 2.1.2 Рекурсивный алгор

### 2.2 Типы и структуры данных

1. Слова хранятся в виде массивом типа char
2. Результат хранится в виде целового типа int

### 2.3 Структура программы

Пользователя просят выбрать один из двух предоставленных режимов работы

- 1) Поиск редакционного расстояния на 2 строках введенных пользователем
- 2) Поиск редакционного расстояния на 2 случайных сгенерированных строках указанной длины

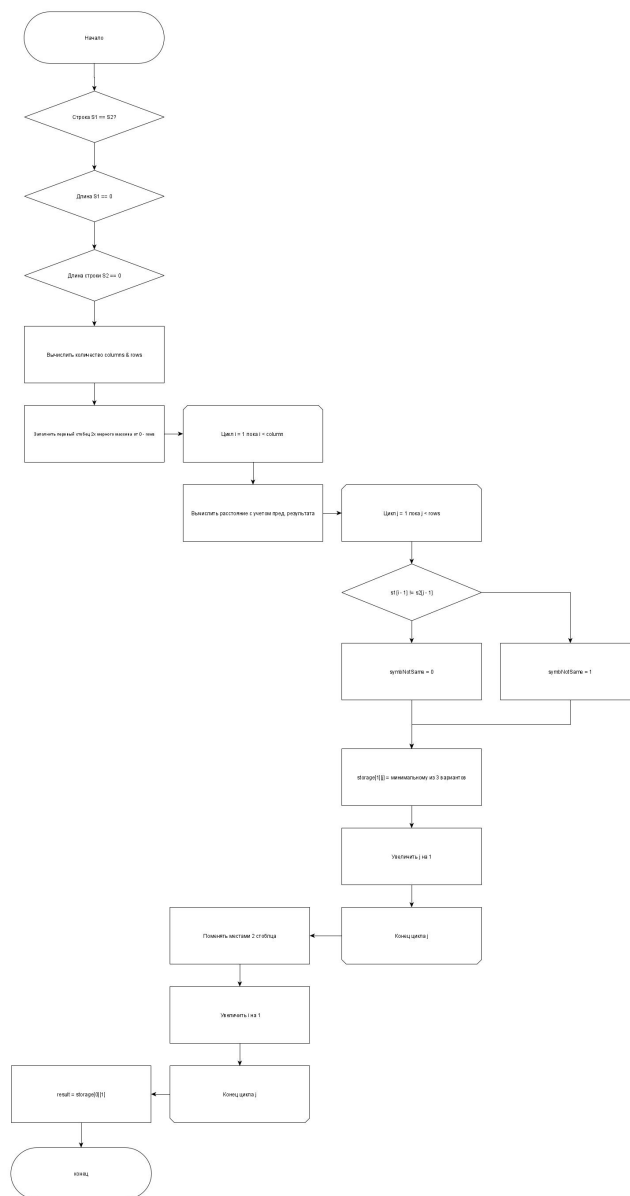


Рис. 1: Блок схема итеративного алгоритма

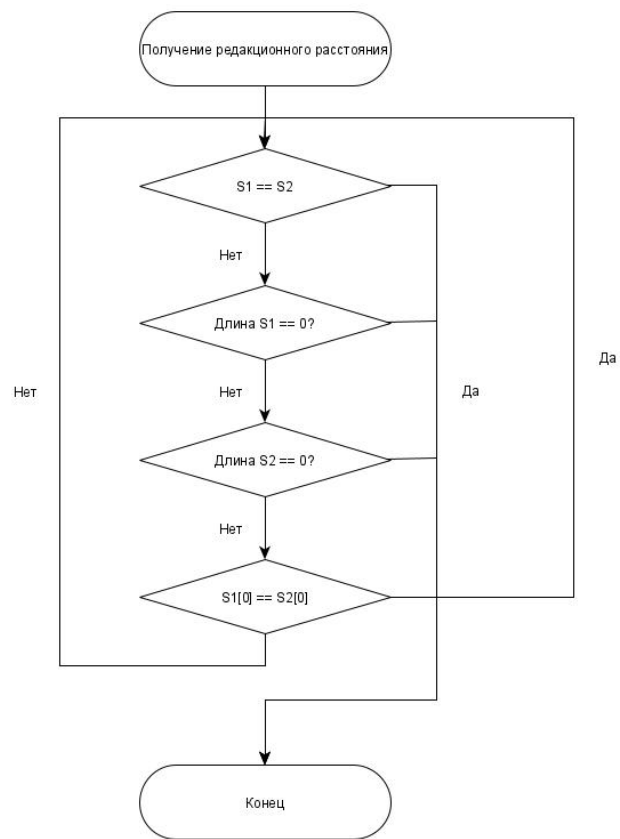


Рис. 2: Блок схема рекурсивного алгоритма

## 3 Технологический раздел

### 3.1 Требования

Данный программный продукт должен возвращать конечному пользователю редакционное расстояние между 2 строк. Редакционное расстояние - это количество операций необходимое для преобразование одной строки в другую.

### 3.2 Выбор языка и среды разработки

Для решения данной поставленной задачи, мной был выбран язык с++ по причине быстрогодействия, по скольку в нашем случаи пользователю необходимо как можно быстрее получить результат выполнения нашего алгоритма.

Так же мной используются среда разработки под названием VS Visual Studio по причине удобства и функциональности. И так же желая излучить данную среду по лучше.

### 3.3 Интерфейс

Интерфейс представляет из себя простую консоль в котором пользователь взаимодействует с помощью ввода команд. Такой тип взаимодействия выбран, по причине простоты разработки и удобства тестирования программы

### 3.4 Листинг

#### 3.4.1 Итеративный Левенштейн

```
"int levenstain3(char* str1 , char* str2) {

    if (strcmp(str1 , str2) == 0) {
        return 0;
    }

    if (strlen(str1) == 0) {
        return strlen(str2);
    }

    if (strlen(str2) == 0) {
        return strlen(str1);
    }

    unsigned columns = strlen(str1) + 1;
    unsigned rows = strlen(str2) + 1;

    int* storage[2];
    storage[0] = new int[rows];
```

```

storage[1] = new int[rows];

for (unsigned i = 0; i < rows; i++) {
    storage[0][i] = i;
}

for (unsigned i = 1; i < columns; i++) {
    storage[1][0] = storage[0][0] + 1;
    for (unsigned j = 1; j < rows; j++) {
        bool symbNotSame = false;
        if (str1[i - 1] != str2[j - 1]) {
            symbNotSame = true;
        }

        storage[1][j] = min(min(
            storage[0][j] + 1, storage[1][j - 1] + 1),
            storage[0][j - 1] + symbNotSame);
    }
    swap(storage[0], storage[1]);
}

int result = storage[0][rows - 1];

delete storage[0];
delete storage[1];

return result;
}"

```

### 3.4.2 Левенштейн рекурсивный

```

"int levenstainRec3(char* str1, char* str2) {

    if (strcmp(str1, str2) == 0) {
        return 0;
    }

    if (strlen(str1) == 0) {
        return strlen(str2);
    }

    if (strlen(str2) == 0) {
        return strlen(str1);
    }

    bool symbNotSame = false;
    if (*str1 != *str2) {
        symbNotSame = true;
    }
}

```



```

int result = min(min(levenstainRec3(str1 + 1, str2) + 1, levenstainRec3(
levenstainRec3(str1 + 1, str2 + 1) + symbNotSame);

return result;
}"

```

### 3.4.3 Левенштейн модифицированный

```

int levenstain4(char* str1, char* str2) {

unsigned lenstr1 = strlen(str1);
unsigned lenstr2 = strlen(str2);

if (lenstr1 < 2 || lenstr2 < 2)
return levenstain3(str1, str2);

unsigned rows = lenstr1 + 1;
unsigned columns = lenstr2 + 1;

int* storage[3];
for (int i = 0; i < 3; i++)
storage[i] = new int[rows];

for (unsigned i = 0; i < rows; i++) {
storage[0][i] = i;
}

storage[1][0] = 1;
for (unsigned i = 1; i < rows; i++) {
bool symbNotSame = false;
if (str2[i - 1] != str1[0]) {
symbNotSame = true;
}
storage[1][i] = min(min(storage[1][i - 1] + 1, storage[0][i] + 1), stor
}

for (unsigned i = 2; i < columns; i++) {
storage[2][0] = storage[1][0] + 1;
bool symbNotSame = false;

if (str2[0] != str1[i - 1]) {
symbNotSame = true;
}

storage[2][1] = min(min(storage[2][0] + 1, storage[1][1] + 1), storage[

for (unsigned j = 2; j < rows; j++) {
symbNotSame = false;

```

```

if (str2[i - 1] != str1[j - 1]) {
symbNotSame = true;
}

storage[2][j] = min(min(min(
storage[1][j] + 1, storage[2][j - 1] + 1),
storage[1][j - 1] + symbNotSame),
storage[0][j - 2] + 1);
}

swap(storage[0], storage[1]);
swap(storage[1], storage[2]);
}

int result = storage[1][rows - 1];

delete storage[0];
delete storage[1];
delete storage[2];

return result;
}

```

## 4 Исследовательский раздел

### 4.1 Сравнение

#### 4.1.1 Общие результаты замеров

Данная таблица содержит в себе общие результаты замеров времени, необходимого для вычисления редакционного расстояния между 2 строками равной длины.

Считается что слова полностью различны.

Время измеряется в секундах

На каждый метод берется 100 итераций вычисления

LEN	Рекурсивный	Итеративный	Модифицированный
2	0.000105387	0.000169814	0.000147626
4	0.00224298	0.000243199	0.0003136
6	0.0621811	0.000329813	0.000431786
8	1.87619	0.000616959	0.000716373
10	56.2996	0.000702292	0.00151893

Исходя из полученных данных, сравнение Рекурсивного метода на строках длинее 8 не имеет смысла, поскольку время выполнения растет экспоненциально. Но как мы можем заметить на строках 2 рекурсивный работает на 0,000064 быстрее итеративного и на 0.00042 быстрее Модифицированного.

#### 4.1.2 Детальное сравнение

Сравним детально время работы классического Левенштейна и модифицированного (с операцией перестановки). На каждый метод берется среднее время 1000 прогонов.

L	Итеративный	Модифицированный
10	6.322e-06	1.112e-05
100	0.000724276	0.00859501
200	0.00286479	0.00383968
300	0.00517626	0.00711481
400	0.0091726	0.0121067
500	0.0151016	0.0191977
1000	0.0540165	0.0753726
2000	0.248242	0.319514
3000	0.480696	0.696582

## 5 Вывод

Рекурсивный алгоритм при более простой реализации работает чрезвычайно долго, что делает его использование нецелесообразным. Итеративный алгоритм значительно превосходит его по эффективности. Алгоритм с добавленной операцией работает дольше, т.к. имеет более сложную логику, но все равно значительно превосходит по скорости рекурсивный алгоритм.

При этом различие между реализациями тем меньше, чем большая разница длины строк.

## 6 Заключение

В ходе работы было проведено сравнение алгоритмов поиска расстояния Левенштейна (рекурсивной и итеративной реализации) и Дамерау-Левенштейна (итеративной реализации). Были исследованы зависимости времени выполнения программ, реализующих данные алгоритмы, от искомого расстояния и от размеров строк для случаев с одинаковой длиной исходных строк и случая, когда одна из строк значительно меньше другой. В ходе исследования были сделаны следующие выводы: 1) Рекурсивная реализация алгоритма Левенштейна выполняется за приемлимое время лишь в случаях, когда размер одной из строк крайне мал (до 10-15 символов) 2) Итеративные реализации алгоритмов поиска расстояний Дамерау-Левенштейна и Левенштейна имеют схожую ассимптотику, но алгоритм поиска расстояния Дамерау-Левенштейна из-за более сложной внутренней логики в среднем работает медленнее.

## **7   Список источников**