МГТУ им. Баумана

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

# Расстояние Левенштейна

Работу выполнил: Рядинский Кирилл, ИУ7-53Б

Преподаватели: Волкова Л.Л.

*Москва, 2021*

# Оглавление

**Введение** **2**

**1 Аналитическая часть** **4**
1.1 Вывод . . . . . 5

**2 Конструкторская часть** **6**
2.1 Схемы алгоритмов . . . . . 6

**3 Технологическая часть** **11**
3.1 Выбор ЯП . . . . . 11
3.2 Реализация алгоритма . . . . . 11

**4 Исследовательская часть** **17**
4.1 Сравнительный анализ на основе замеров времени работы алгоритмов . . . . . 17
4.2 Использование памяти . . . . . 18
4.3 Вывод . . . . . 19

**5 Заключение** **20**

# Введение

**Расстояние Левенштейна** - минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для:

- исправления ошибок в слове
- сравнения текстовых файлов утилитой diff
- в биоинформатике для сравнения генов, хромосом и белков

Задачами данной лабораторной являются:

1. изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
2. применение метода динамического программирования для матричной реализации указанных алгоритмов;
3. получение практических навыков реализации указанных алгоритмов: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;
4. сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);

5. экспериментальное подтверждение различий во временно́й эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;

6. описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

# 1 | Аналитическая часть

Задача по нахождению расстояния Левенштейна заключается в поиске минимального количества операций вставки/удаления/замены для превращения одной строки в другую.

При нахождении расстояния Дамерау — Левенштейна добавляется операция транспозиции (перестановки соседних символов).

**Действия обозначаются так:**

1. D (англ. delete) — удалить,
2. I (англ. insert) — вставить,
3. R (replace) — заменить,
4. M(match) - совпадение.

Пусть $S_1$ и $S_2$ — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min( & \\ D(i,j-1)+1, & \\ D(i-1,j)+1, & j > 0, i > 0 \\ D(i-1,j-1)+m(S_1[i],S_2[j]) & \\ ), & \end{cases}$$

где $m(a,b)$ равна нулю, если $a = b$ и единице в противном случае; $min\{\, a,b,c\}$ возвращает наименьший из аргументов.

Расстояние Дамерау-Левенштейна вычисляется по следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & i > 0, j = 0 \\ j, & i = 0, j > 0 \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[i]), \\ D(i-2,j-2)+m(S_1[i],S_2[i]), \end{cases} & \text{, если } i,j > 0 \\ & \text{и } S_1[i] = S_2[j-1] \\ & \text{и } S_1[i-1] = S_2[j] \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[i]), \end{cases} & \text{, иначе} \end{cases}$$

## 1.1 Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который является модификаций первого, учитывающего возможность перестановки соседних символов.

# 2 | Конструкторская часть

**Требования к вводу**

1. На вход подаются две строки
2. uppercase и lowercase буквы считаются разными

**Требования к программе:**

1. Две пустые строки - корректный ввод, программа не должна аварийно завершаться

## 2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы алгоритмов.

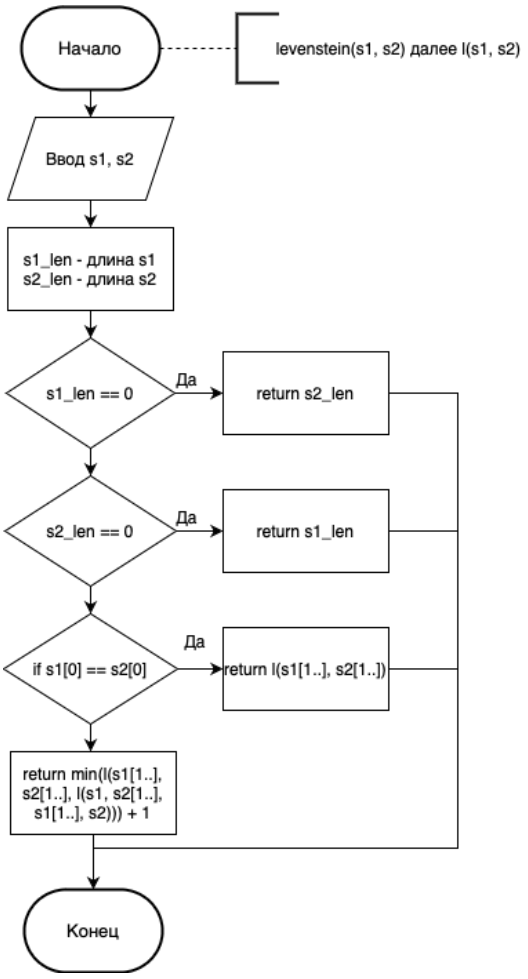


Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

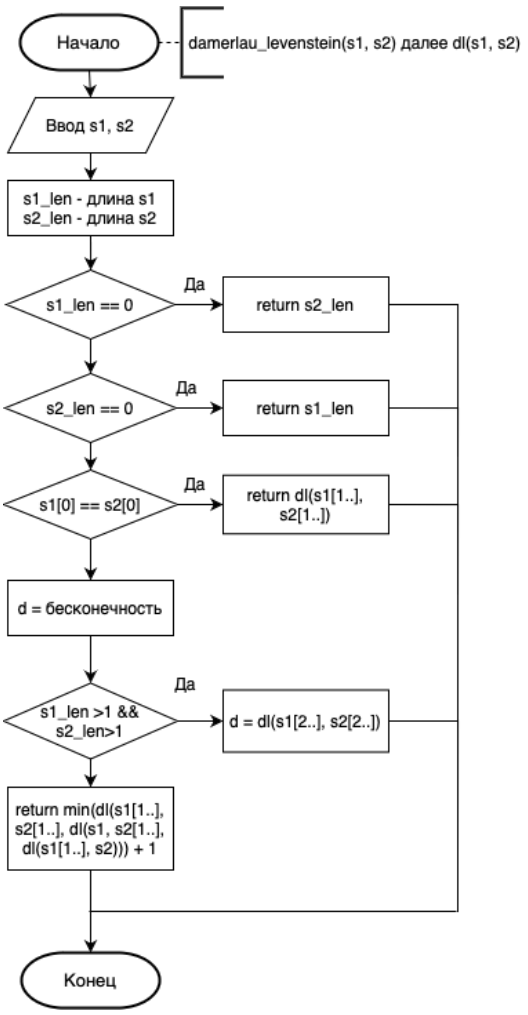


Рис. 2.2: Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

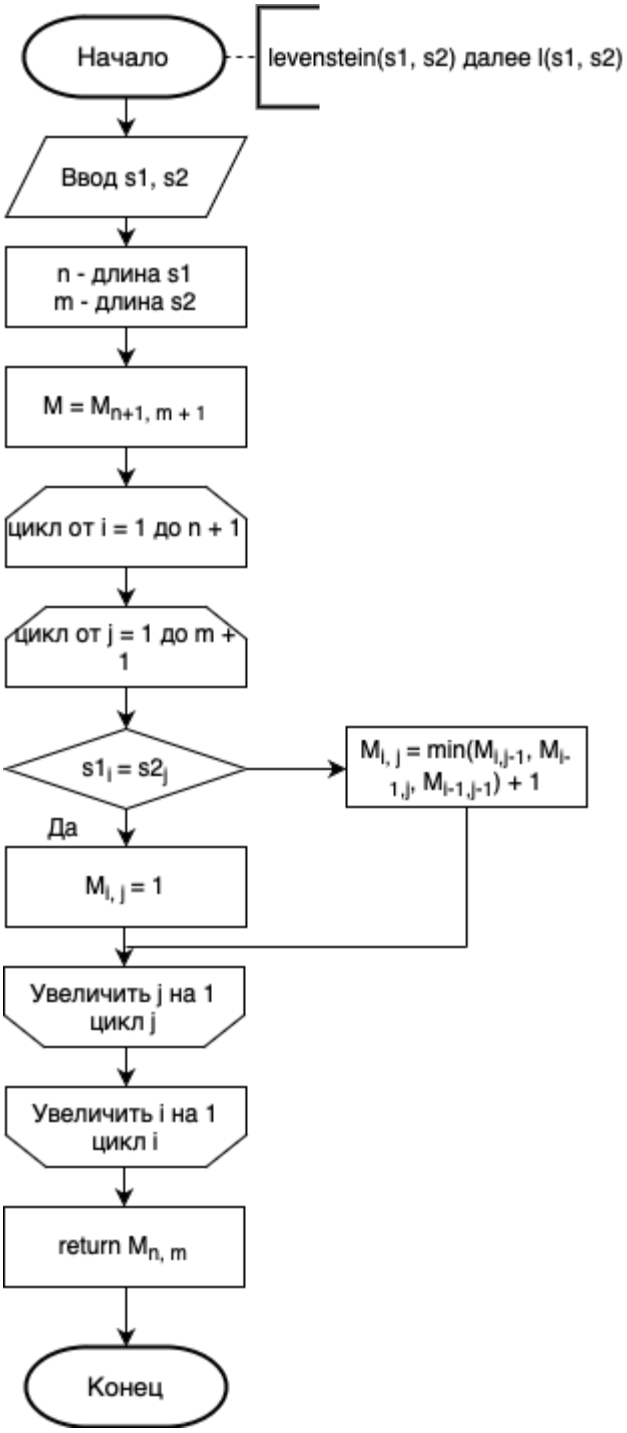


Рис. 2.3: Схема матричного алгоритма нахождения расстояния Левенштейна

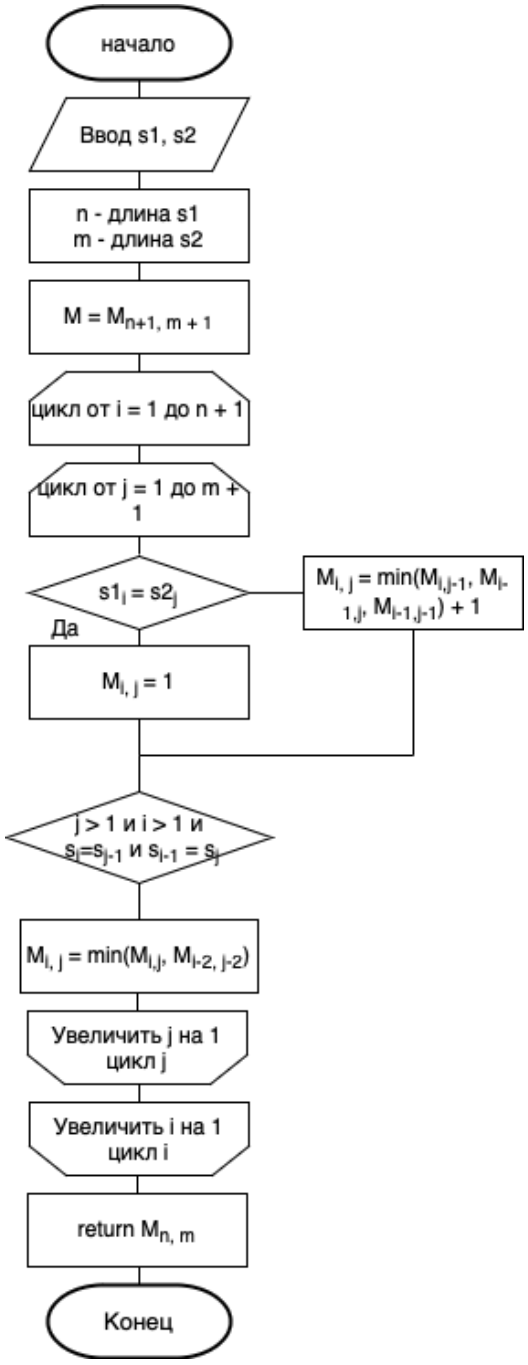


Рис. 2.4: Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

# 3 | Технологическая часть

## 3.1 Выбор ЯП

Для реализации программ я выбрал язык программирования Rust, так как этот язык предоставляет как низкоуровневые интерфейсы, так и высокоуровневые. Также он является таким же быстрым, как и C++, но более безопасен. Среда разработки Visual Studio Code.

## 3.2 Реализация алгоритма

Листинг 3.1: Функция нахождения расстояния Левенштейна рекурсивно

```
pub fn levenstein_rec(s1: &str, s2: &str) -> usize {
  let s1_len = s1.len();
  let s2_len = s2.len();

  if s1_len == 0 {
    return s2_len;
  }

  if s2_len == 0 {
    return s1_len;
  }

  if s1.chars().nth(0) == s2.chars().nth(0) {
    return levenstein_rec(&s1[1..], &s2[1..]);
  }

  let a = levenstein_rec(&s1[1..], &s2[1..]);
```

```
18    let b = levenstein_rec(s1, &s2[1..]);
19    let c = levenstein_rec(&s1[1..], &s2);
20
21    return std::cmp::min(a, std::cmp::min(b, c)) + 1;
22 }
```

Листинг 3.2: Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
pub fn damerlau_levenstein_rec(s1: &str, s2: &str) -> usize {
  let s1_len = s1.len();
  let s2_len = s2.len();

  if s1_len == 0 {
    return s2_len;
  }

  if s2_len == 0 {
    return s1_len;
  }

  if s1.chars().nth(0) == s2.chars().nth(0) {
    return damerlau_levenstein_rec(&s1[1..], &s2[1..]);
  }

  let a = damerlau_levenstein_rec(&s1[1..], &s2[1..]);
  let b = damerlau_levenstein_rec(s1, &s2[1..]);
  let c = damerlau_levenstein_rec(&s1[1..], &s2);

  let mut d = usize::MAX;
  if s1_len > 1 && s2_len > 1 {
    d = damerlau_levenstein_rec(&s1[2..], &s2[2..]);
  }

  return std::cmp::min(d, std::cmp::min(a, std::cmp::min(b, c))) + 1;
}
```

Листинг 3.3: Функция нахождения расстояния Левенштейна матрично

```
pub fn levenstein_iter(word1: &str, word2: &str) -> usize {
  // getting length of words
  let w1 = word1.chars().collect::<Vec<_>>();
  let w2 = word2.chars().collect::<Vec<_>>();

  let word1_length = w1.len() + 1;
  let word2_length = w2.len() + 1;

  let mut matrix = vec![vec![0; word1_length]; word2_length
     ];

  for i in 1..word1_length {
    matrix[0][i] = i;
  }
  for j in 1..word2_length {
    matrix[j][0] = j;
  }

  for j in 1..word2_length {
    for i in 1..word1_length {
      let x: usize = if w1[i - 1] == w2[j - 1] {
        matrix[j - 1][i - 1]
      } else {
        1 + std::cmp::min(
          std::cmp::min(matrix[j][i - 1], matrix[j - 1][i])
              ,
          matrix[j - 1][i - 1],
        )
      };
      matrix[j][i] = x;
    }
  }
  return matrix[word2_length - 1][word1_length - 1];
}
```

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
pub fn damerlau_levenstein_iter(word1: &str, word2: &str)
    -> usize {
    // getting length of words
    let w1 = word1.chars().collect::<Vec<_>>();
    let w2 = word2.chars().collect::<Vec<_>>();

    let word1_length = w1.len() + 1;
    let word2_length = w2.len() + 1;

    let mut matrix = vec![vec![0; word1_length];
        word2_length];

    for i in 1..word1_length {
        matrix[0][i] = i;
    }
    for j in 1..word2_length {
        matrix[j][0] = j;
    }

    for j in 1..word2_length {
        for i in 1..word1_length {
            let x: usize = if w1[i - 1] == w2[j - 1] {
                matrix[j - 1][i - 1]
            } else {
                1 + std::cmp::min(
                    std::cmp::min(matrix[j][i - 1], matrix[
                        j - 1][i]),
                    matrix[j - 1][i - 1],
                )
            };

            matrix[j][i] = x;

            if (j > 1) && (i > 1) && (w1[i - 1] == w2[j -
                2]) && (w1[i - 2] == w2[j - 1]) {
                matrix[j][i] = std::cmp::min(x, matrix[j -
                    2][i - 2] + 1);
            }
```

```
34            }
35        }
36        return matrix[word2_length - 1][word1_length - 1];
37    }
```

| Строка 1 | Строка 2 | Ожидаемый результат | |
|---|---|---|---|
| | | Левенштейн | Дамерау-Левенштейн |
| Take | Took | 3 | 3 |
| Art | Atr | 2 | 1 |
| car | city | 3 | 3 |
| head | ehda | 3 | 2 |
| laptop | notebook | 7 | 7 |
| peek | peeks | 1 | 1 |
| rain | pain | 1 | 1 |

Таблица 3.1: Функциональные тесты

В Таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна и Дамерау-Левенштейна.

# 4 | Исследовательская часть

## 4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведен замер времени работы каждого из алгоритмов.

| Длина слов | Лев. рек. | Лев. итер. | Дам. -Лев. рек. | Дам. -Лев. итер. |
|---|---|---|---|---|
| 1 | 8,50E-08 | 2,48E-07 | 3,20E-10 | 5,50E-07 |
| 2 | 1,08E-07 | 3,07E-07 | 4,80E-08 | 4,93E-07 |
| 3 | 3,20E-07 | 4,26E-07 | 2,77E-07 | 4,91E-07 |
| 4 | 1,30E-06 | 4,88E-07 | 1,50E-06 | 5,00E-07 |
| 5 | 6,90E-06 | 7,41E-07 | 8,40E-06 | 5,08E-07 |
| 6 | 3,65E-05 | 8,12E-07 | 4,68E-05 | 5,31E-07 |
| 7 | 1,97E-04 | 9,86E-07 | 2,64E-04 | 5,14E-07 |
| 8 | 1,00E-03 | 1,10E-06 | 1,51E-03 | 5,20E-07 |
| 9 | 5,92E-03 | 1,22E-06 | 8,72E-03 | 5,09E-07 |

Таблица 4.1: Время работы алгоритмов (в секундах)

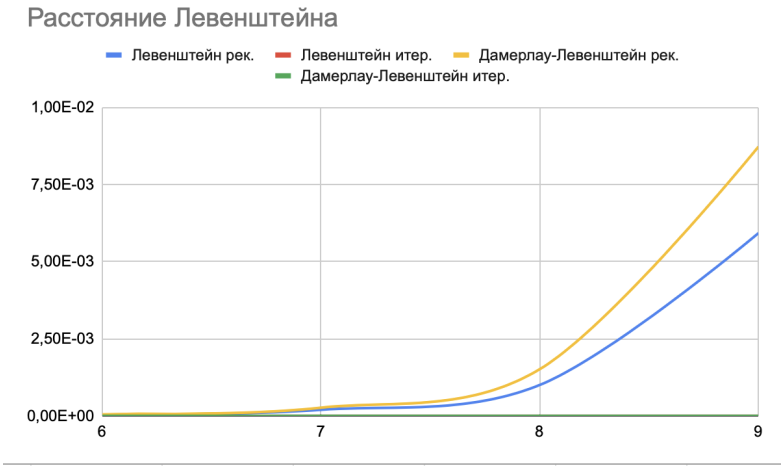


Рис. 4.1: Зависимость времени работы алгоритмов от длины строки.

Наиболее эффективными по времени при маленькой длине слова являются рекурсивные реализации алгоритмов, но как только увеличивается длина слова, их эффективность резко снижается, что обусловлено большим количеством повторных рассчетов. Время работы алгоритма, использующего матрицу, намного меньше благодаря тому, что в нем требуется только (m + 1)*(n + 1) операций заполнения ячейки матрицы. Также установлено, что алгоритм ДамерауЛевенштейна работает немного дольше алгоритма Левенштейна, т.к. в нем добавлены дополнительные проверки, однако алгоритмы сравнимы по временной эффективности.

## 4.2 Использование памяти

Алгоритмы Левенштейна и Дамерау-Левенштейна не отличаются по использованию памяти, соответственно достаточно рассмотреть рекурсивный и матричный реализации этих алгоритмов.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, соответственно, максимальный расход

$$(Size(S_1) + Size(S_2) \cdot (2 \cdot Size(\text{string}) + Size(\text{int}))) \tag{4.1}$$

Где Size - функция, возвращающая размер аргумента; string - строковый тип, int - целочисленный тип.

Использование памяти при итеративной реалзиации теоритически равно

$$(Size(S_1 + 1) \cdot Size(S_2 + 1)) \cdot Size(int) + 2 \cdot Size(string) \tag{4.2}$$

К сожалению, на моей машине не возможны детальные замеры потребления памяти.

## 4.3 Вывод

Рекурсивный алгоритм Левенштейна работает на порядок дольше итеративных реализаций, время его работы увеличивается в геометрической прогрессии. На словах длиной 9 символов, матричная реализация превосходит рекурсивную в 4800 раз. Рекурсивные алгоритмы Левенштейна и Дамерау - Левенштейна сопостовимы по времени.

# 5 | Заключение

Был изучен метод динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна. Также изучены алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками, получены практические навыки раелизации указанных алгоритмов в матричной и рекурсивных версиях.

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработаного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк.

В результате исследований я пришел к выводу, что матричная реализация данных алгоритмов заметно выигрывает по времени при росте длины строк, следовательно более применима в реальных проектах.