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 «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе №1 по дисциплине "Анализ Алгоритмов"

Тема Расстояние Левенштейна и Дамерау-Левенштейна

Студент Нисуев Н.Ф.

Группа ИУ7-52Б

Преподаватель Волкова Л. Л., Строганов Д.В.

2024 г.

Содержание

Введение

Расстояние Левенштейна и Дамерау-Левентшейна — это минимальное количество операций вставки одного символа, удаления одного символа, замены одного символа на другой и транспозиции (перестановки двух соседних символов) в случае расстояния Дамерау-Левенштейна, необходимых для превращения одной строки в другую [1]. Расстояние Левенштейна используется для:

- исправления ошибок в словах;
- поиска дубликатов текстов;
- для сравнения генов, хромосом и белков в биоинформатике;
- сравнения текстовых файлос помощью утилиты diff.

Цель лабораторной работы — сравнение алгоритмов расстояния Левенштейна и Дамерау-Левенштейна. Для достижения поставленной цели необходимо выполнить следующие задачи:

- изучить расстояния Левенштейна и Дамерау-Левенштейна;
- реализовать указвнные алгоритмы поиска расстояний (два алгоритма в матричной версии и один из алгоритмов в рекурсивной версии)
- провести сравнительный анализ линейной и рекурсивной реализаций алгоритмов по затраченному процессорному времени и памяти на основе экспериментальных данных;
- описать и обосновать полученные результаты в отчете о выполненной лабораторной работе.

1 Аналитическая часть

В данном разделе будут рассмотрены алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна.

1.1 Описание алгоритмов

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

1.1.1 Расстояние Левенштейна

Для двух строк S_1 и S_2, представленных в виде списков символов, длинной M и N соответственно, расстояние Левенштейна рассчитывается по рекуррентной формуле 1.1:

$$D(S_1, S_2) = \begin{cases} len(S_1), & \text{если } len(S_2) = 0 \\ len(S_2), & \text{если } len(S_1) = 0 \\ D(tail(S_1), tail(S_2)), & \text{если } head(S_1) = head(S_2) \\ 1 + min \begin{cases} D(tail(S_1), S_2), \\ D(S_1, tail(S_2)), \\ D(tail(S_1), tail(S_2)), \end{cases} & \text{иначе} \end{cases} \tag{1.1}$$

где:

- $len(S)$ — длина списка S;
- $head(S)$ — первый элемент списка S;
- $tail(S)$ — список S без первого элемента;

1.1.2 Расстояние Дамерау-Левенштейна

В алгоритме поиска расстояния Дамерау-Левенштейна, помимо вставки, удаления, и замены присутствует операция транспозиции(перестановки двух соседних символов). Расстояние Дамерау-Левенштейна может быть вычисленно по рекуррентной формуле 1.2:

$$D(S1,S2) = \begin{cases} len(S_1), & \text{если } len(S_2) = 0 \\ len(S_2), & \text{если } len(S_1) = 0 \\ D(tail(S_1), tail(S_2)), & \text{если } head(S_1) = head(S_2) \\ 1 + min \begin{cases} D(tail(S1), S2), \\ D(S1, tail(S2)), \\ D(tail(S1), tail(S2)), \end{cases} \\ 1 + min \begin{cases} D(tail(tail(S1)), tail(tail(S2))), & (1) \\ D(tail(S1), S2), \\ D(S1, tail(S2)), \\ D(tail(S1), tail(S2)), \end{cases} \end{cases} \tag{1.2}$$

(1): если head(S1) = head(tail(S2)) и head(S2) = head(tail(S1));

ВЫВОД

В данном разделе были рассмотрены два основных алгоритма для вычисления расстояний между строками: расстояние Левенштейна и расстояние Дамерау-Левенштейна.

2 Конструкторская часть

В данном разделе будут приведены схемы алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна, приведено описание используемых типов данных, оценки памяти, а также описана структура ПО.

2.1 Представление алгоритмов

На вход алгоритмов подаются строки S_1 и S_2, на выходе — искомое расстояние.

На рис. 2.1 — 2.3 приведены схемы рекурсивного и матричных алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

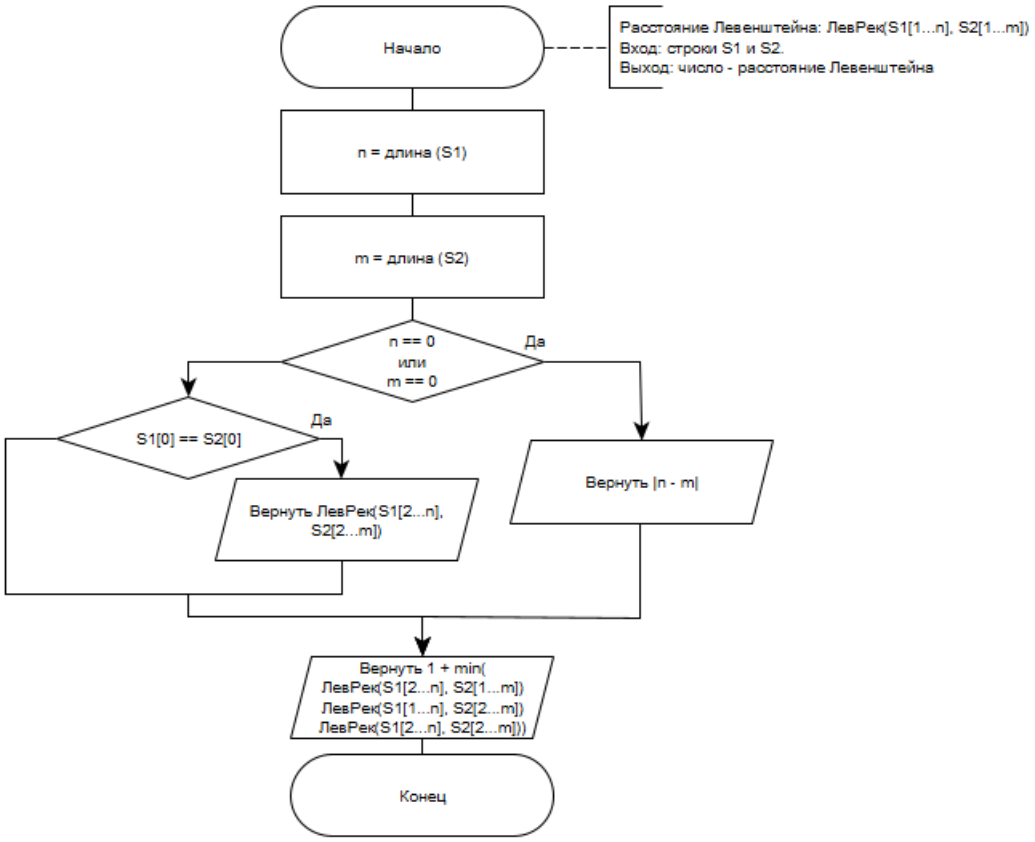

Рисунок 2.1 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна

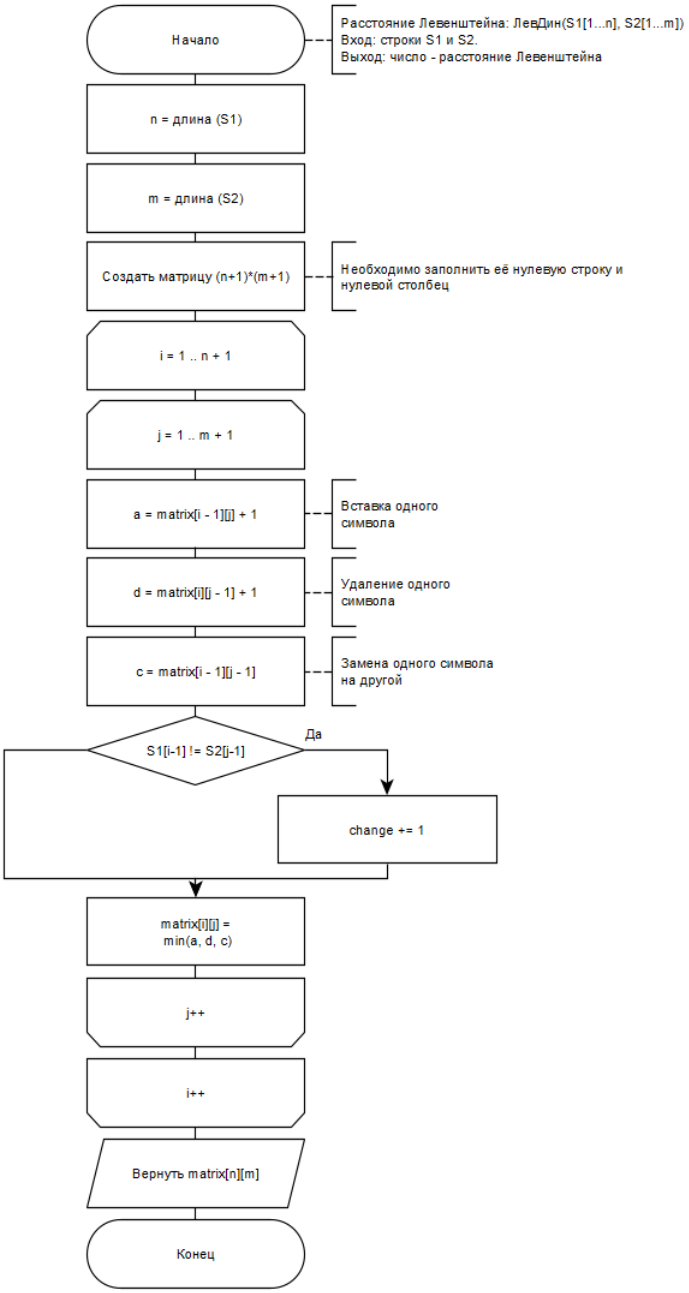

Рисунок 2.2 – Схема динамического алгоритма нахождения расстояния Левенштейна

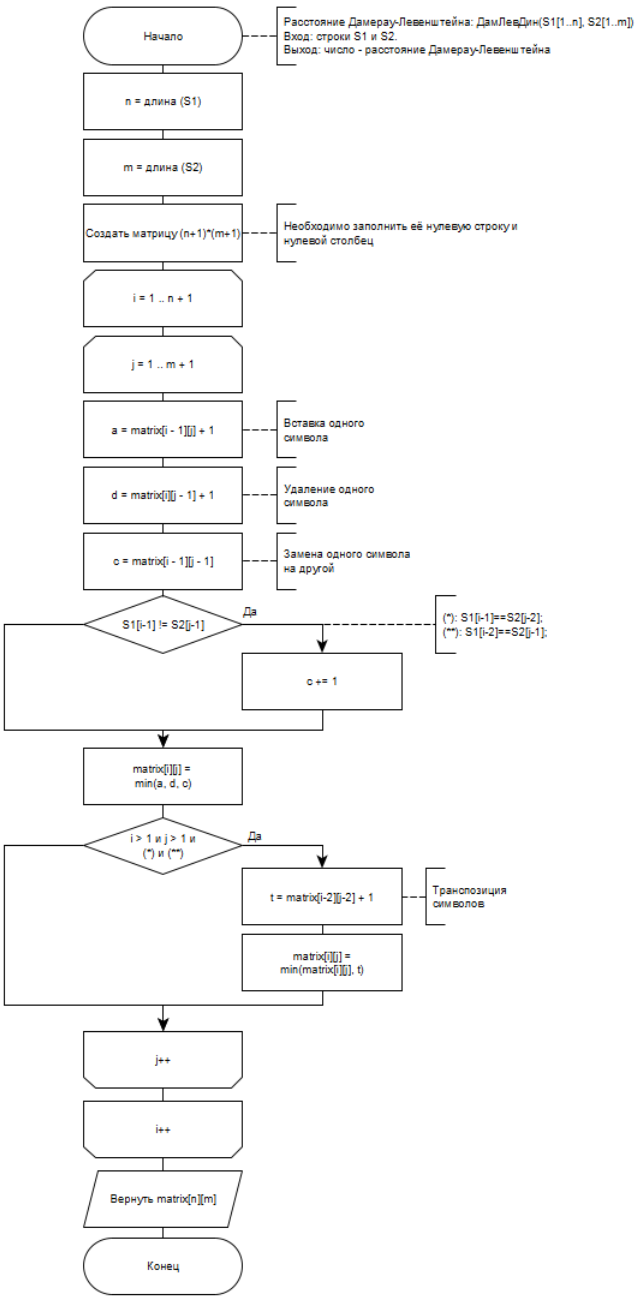

Рисунок 2.3 – Схема динамического алгоритма нахождения расстояния Дамерау-Левенштейна

ВЫВОД

В данном разделе были представлены схемы алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации, листинги кода.

3.1 Требования к программному обеспечению

Входные данные: две строки на русском или английском языке в любом регистре.

Выходные данные: искомое расстояние для выбранного метода и матрицы расстояний для матричных реализаций.

3.2 Средства реализации

В данной работе для реализации был выбран язык программирования *Rust* [2]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного языка наличием функции вычисления процессорного времени в библиотеке *cpu-time* [3].

3.3 Реализация алгоритмов

В листингах 3.1 - 3.3 представлены реализации алгоритмов нахождения расстояния Левенштейна и Дамерау–Левенштейна.

Листинг 3.1 – Функция нахождения расстояния Левенштейна рекурсивно

```rust
pub fn recursion_levenshtein(s1: &str, s2: &str) -> usize {
    if s1.is_empty() || s2.is_empty() {
        return (s1.len() as i32 - s2.len() as i32).abs() as usize;
    }

    let mut s1_chars = s1.chars();
    let mut s2_chars = s2.chars();

    let first_char_s1 = s1_chars.next().unwrap();
    let first_char_s2 = s2_chars.next().unwrap();

    if first_char_s1 == first_char_s2 {
        return recursion_levenshtein(s1_chars.as_str(),
           s2_chars.as_str());
    }

    1 + *[
        recursion_levenshtein(s1_chars.as_str(), s2),
        recursion_levenshtein(s1, s2_chars.as_str()),
        recursion_levenshtein(s1_chars.as_str(), s2_chars.as_str()),
    ]
    .iter()
    .min()
    .unwrap()
}
```

Листинг 3.2 – Функция нахождения расстояния Левенштейна динамически

```rust
pub fn table_levenshtein(s1: &str, s2: &str) -> usize {
    let len_s1 = s1.chars().count();
    let len_s2 = s2.chars().count();

    let mut table = vec![vec![0; len_s2 + 1]; len_s1 + 1];

    for i in 0..=len_s1 {
        table[i][0] = i;
    }
    for j in 0..=len_s2 {
        table[0][j] = j;
    }
```

```
13
14      let len_s1 = s1.len();
15      let len_s2 = s2.len();
16
17      let mut table = vec![vec![0; len_s2 + 1]; len_s1 + 1];
18
19      for i in 0..=len_s1 {
20          table[i][0] = i;
21      }
22      for j in 0..=len_s2 {
23          table[0][j] = j;
24      }
25
26      for i in 1..=len_s1 {
27          for j in 1..=len_s2 {
28              let cost = if s1.chars().nth(i - 1) == s2.chars().nth(j -
                    1) {
29                  0
30              } else {
31                  1
32              };
33
34              table[i][j] = *[
35                  table[i - 1][j] + 1,
36                  table[i][j - 1] + 1,
37                  table[i - 1][j - 1] + cost,
38              ]
39              .iter()
40              .min()
41              .unwrap();
42          }
43      }
44
45      table[len_s1][len_s2]
46  }
```

Листинг 3.3 – Функция нахождения расстояния Дамерау–Левенштейна динамически

```rust
pub fn damerau_levenshtein(s1: &str, s2: &str) -> usize {
    let len_s1 = s1.len();
    let len_s2 = s2.len();

    let mut table = vec![vec![0; len_s2 + 1]; len_s1 + 1];

    for i in 0..=len_s1 {
        table[i][0] = i;
    }
    for j in 0..=len_s2 {
        table[0][j] = j;
    }

    for i in 1..=len_s1 {
        for j in 1..=len_s2 {
            let cost = if s1.chars().nth(i - 1) == s2.chars().nth(j - 1) {
                0
            } else {
                1
            };

            table[i][j] = *[
                table[i - 1][j] + 1,
                table[i][j - 1] + 1,
                table[i - 1][j - 1] + cost,
            ]
            .iter()
            .min()
            .unwrap();

            if i > 1
                && j > 1
                && s1.chars().nth(i - 1) == s2.chars().nth(j - 2)
                && s1.chars().nth(i - 2) == s2.chars().nth(j - 1)
            {
                table[i][j] = table[i][j].min(table[i - 2][j - 2] + 1);
            }
        }
```

```
39      }
40
41      table[len_s1][len_s2]
42  }
```

ВЫВОД

В данном разделе были рассмотрены требования к программному обеспечению, используемые средства реализации, а также приведены листинги кода для вычисления расстояний Левенштейна (на основе рекурсивного и динамического алгоритмов) и Дамерау-Левенштейна (на основе динамического алгоритма).

4 Исследовательская часть

4.1 Технические характеристики

Характеристики используемого оборудования:

- Операционная система — Windows 11 Home [4]
- Память — 16 Гб.
- Процессор — 12th Gen Intel(R) Core(TM) i7-12700H @ 2.30 ГГц [5]

4.2 Описание используемых типов данных

Используемые типы данных:

- строка — последовательность символов типа *str*;
- длина строки — целое число типа *usize*;
- матрица — двумерный массив типа *usize*.

4.3 Оценка памяти

Рекурсивный алгоритм Левенштейна не использует явных структур данных для хранения промежуточных результатов. Каждый вызов функции обрабатывает небольшую часть строк и вызывает сам себя несколько раз. В худшем случае глубина рекурсии составляет:

$$(len(S_1) + len(S_2)). \tag{4.1}$$

При этом каждый вызов сохраняет несколько локальных переменных: две переменные типа *char* и две переменные типа *str*. В результате максимальный расход памяти выражается как:

$$(len(S_1) + len(S_2)) \cdot (2 \cdot size(\text{char}) + 2 \cdot size(\text{str})), \tag{4.2}$$

где $size$ обозначает функцию, вычисляющую размер параметра.

Алгоритм, реализованный с использованием динамического программирования, использует двумерную матрицу размером $(len(S_1) + 1) \times (len(S_2) + 1)$. Эта матрица хранит промежуточные результаты — расстояние Левенштейна для всех возможных подстрок. Кроме того, в памяти хранятся 5 переменных типа $usize$ и 2 переменные типа str. Общий объём памяти, требуемый для этого алгоритма, можно выразить как:

$$(len(S_1) + 1) \cdot (len(S_2) + 1) \cdot size(\text{usize}) + 5 \cdot size(\text{usize}) + 2 \cdot size(\text{str}). \quad (4.3)$$

Таким образом, по памяти алгоритм динамического программирования уступает рекурсивному алгоритму: в динамическом подходе память растёт как произведение длин строк, тогда как в рекурсивном — как их сумма.

Алгоритм Дамерау-Левенштейна, также реализованный с применением динамического программирования, структурно похож на алгоритм Левенштейна. Основное отличие состоит в добавлении операции перестановки соседних символов. Для этого используется та же матрица размером $(len(S_1)+1)\dot{(}len(S_2)+1)$, как и в алгоритме Левенштейна.

Несмотря на добавление операции перестановки, объём используемой памяти остаётся на том же уровне, что и у обычного алгоритма Левенштейна, так как не требуется дополнительного пространства для обработки перестановок. Каждая ячейка матрицы заполняется ровно один раз.

4.4 Время выполнения алгоритмов

Результаты замеров времени работы алгоритмов нахождения расстояний Левенштейна и Дамерау–Левенштейна приведены в таблице 4.1. На рисунке 4.1 приведены графики зависимости времени от количества букв для каждого из алгоритмов. Замеры времени проводились на строках одинаковой длины и с разным набором букв. Каждое значение получено путем взятия среднего из 1000 измерений.

Таблица 4.1 – Время работы алгоритмов (в наносекундах)

Длина строк	Лев рек.	Лев дин.	Дам-Лев дин.
1	47	349	203
2	78	481	291
3	277	645	390
4	1389	881	710
5	7968	2617	812
6	42.5e+03	2617	1108
7	217.5e+03	1713	1462
8	1.18e+09	2147	1966

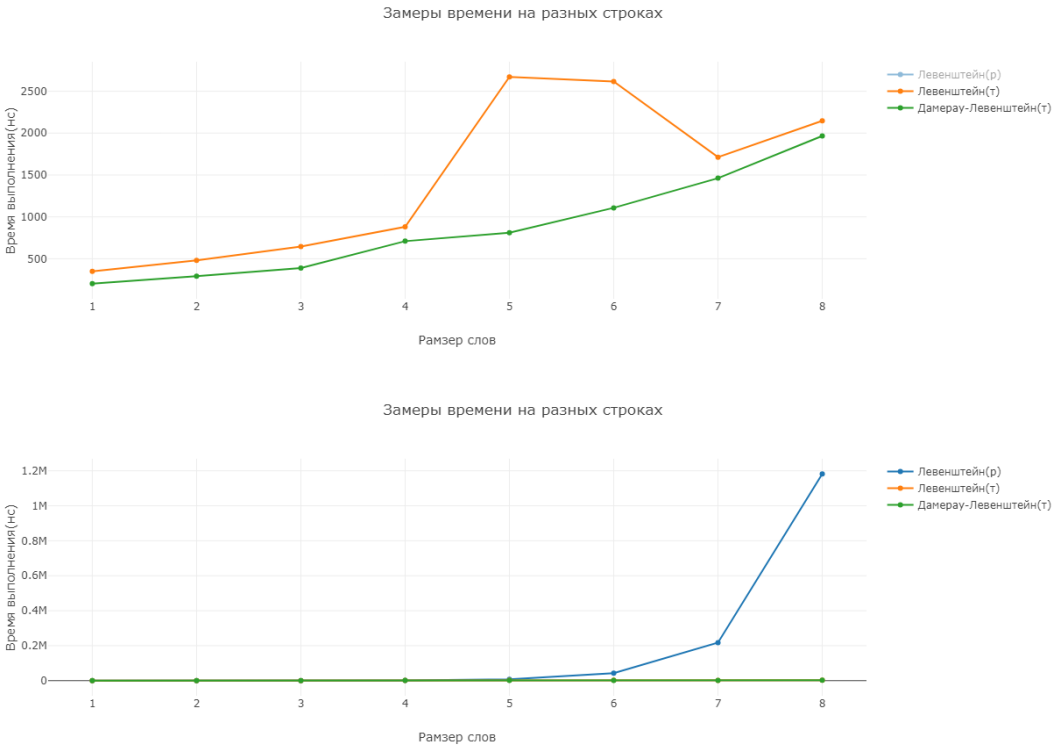

Рисунок 4.1 – Сравнение алгоритмов по времени

Наиболее эффективными являются алгоритмы, использующие динамический подход (матрицу), так как в рекурсивных алгоритмах большое количество повторных расчетов.

4.5 Вывод

Рекурсивный алгоритм для вычисления расстояния Левенштейна по времени работает значительно медленнее, чем его динамический аналог. Также важно отметить, что динамические алгоритмы для расчёта расстояний Левенштейна и Дамерау-Левенштейна по времени выполнения сопоставимы между собой и примерно одинаковы.

Анализ использования памяти показывает, что рекурсивный алгоритм требует меньше памяти по сравнению с алгоритмом на основе динамического программирования. В случае динамического алгоритма для расчёта расстояния Дамерау-Левенштейна, несмотря на добавление операции перестановки символов, потребление памяти остаётся на уровне алгоритма Левенштейна, поскольку дополнительное пространство для хранения результатов перестановок не требуется.

Заключение

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций алгоритмов вычисления расстояния между строками с помощью разработанного программного обеспечения на основе измерений процессорного времени для строк разной длины.

Исследования показали, что матричная реализация этих алгоритмов значительно превосходит рекурсивную по времени выполнения при увеличении длины строк, но требует больше памяти.

В ходе выполнения данной лабораторной работы были решены следующие задачи: В ходе выполнения данной лабораторной работы были решены следующие задачи:

- изучены алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
- применены методы динамического программирования для матричной реализации указанных алгоритмов;
- получены практические навыки реализации указанных алгоритмов: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;
- проведен сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
- описаны и обоснованы полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

Список использованных источников

[1] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. – М.: Доклады АН СССР, 1965. Т. 163. С. 845–848. (дата обращения: 10.09.2024)

[2] The Rust Programming Language [Электронный ресурс]. Режим доступа: `https://doc.rust-lang.org/book/` (дата обращения: 08.09.2024).

[3] cpu-time documentation [Электронный ресурс]. Режим доступа: `https://docs.rs/cpu-time/1.0.0/cpu_time/` (дата обращения: 10.09.2024).

[4] Windows technical documentation for developers and IT pros [Электронный ресурс]. URL: `https://learn.microsoft.com/en-us/windows/` (дата обращения: 10.09.2024).

[5] Intel® Core™ i7-12700H Processor [Электронный ресурс]. URL: `https://ark.intel.com/content/www/us/en/ark/products/132228/intel-core-i7-12700h-processor-24m-cache-up-to-4-70-ghz.html` (дата обращения: 10.09.2024).