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»

Лабораторная работа № 1
по дисциплине "Анализ алгоритмов"

Тема Расстояние Левенштейна и Дамерау-Левенштейна.

Студент Перестоpонин П.Г.

Группа ИУ7-53Б

Преподаватели Строганов Ю.В., Волкова Л.Л.

Москва, 2020

Оглавление

Введение

Расстояние Левенштейна - это минимальное количество операций, необходимых для превращения одной строки в другую, где операции:

- вставка одного символа;
- удаление одного символа;
- замены одного символа на другой.

Расстояние Левенштейна находит практическое применение во многих сферах:

- алгоритмы нечеткого поиска
- сравнения текстовых файлов
- сравнения генов, хромосом и белков в биоинформатике

Целью данной лабораторной работы является реализация рекурсивных и итеративных алгоритмов Левенштейна и Дамерау-Левенштейна.

Задачами данной лабораторной являются:

1. изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
2. применение техник динамического программирования для реализации указанных алгоритмов;
3. получение практических навыков реализации указанных алгоритмов: двух итеративных алгоритмов (Левенштейна и Дамерау-Левенштейна) и алгоритма Левенштейна в 2 рекурсивных версиях (с мемоизацией и без нее);
4. сравнительный анализ итеративной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
5. экспериментальное подтверждение различий во временно́й эффективности различных реализаций исследуемых алгоритмов 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.

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 следующим образом:

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:

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

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. прописные и строчные буквы считаются разными.

Требования к программе:

1. две пустые строки - корректный ввод, программа не должна аварийно завершаться;
2. программа должна уметь обрабатывать слова на русском языке.

2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы алгоритмов.

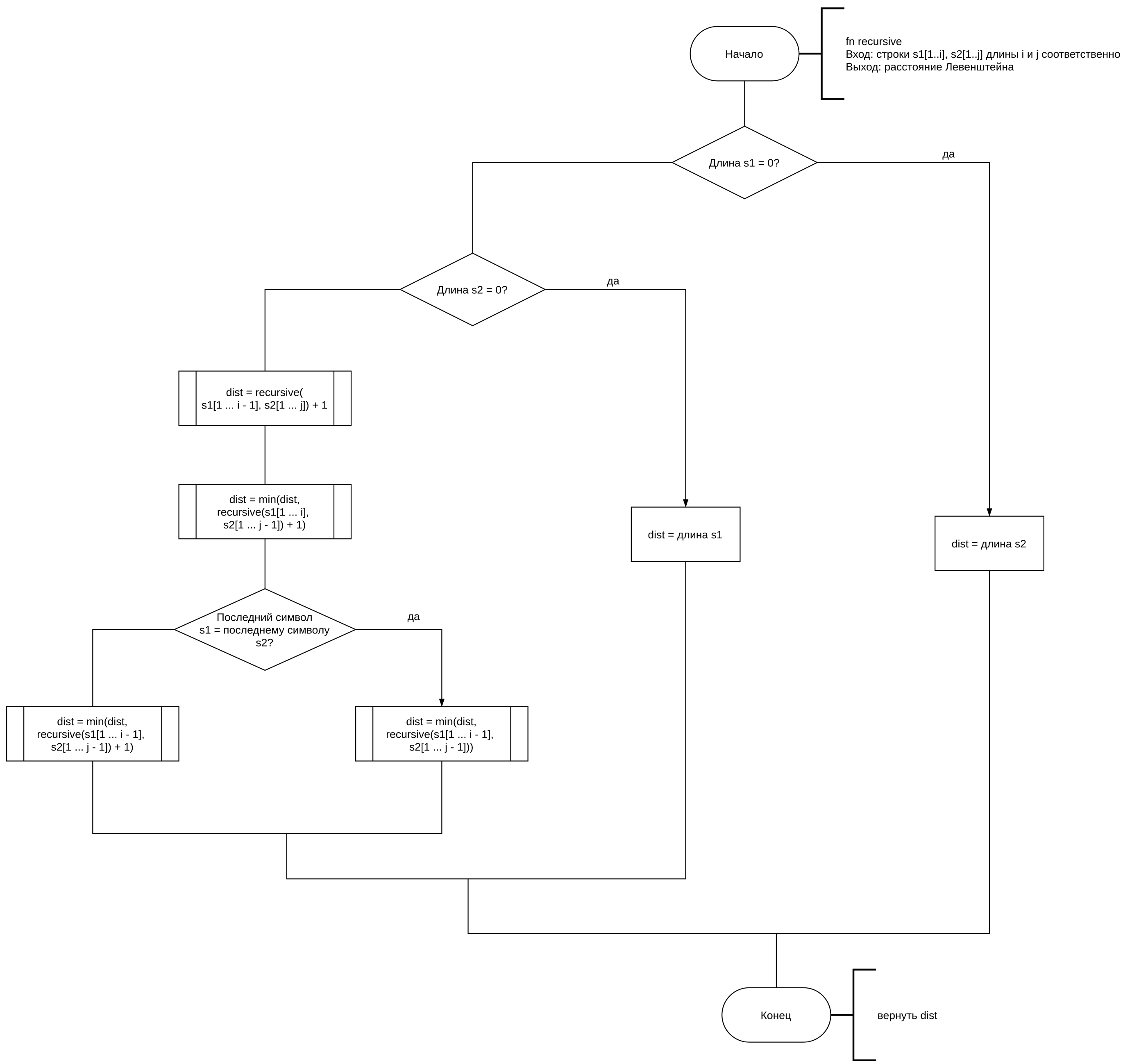


Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

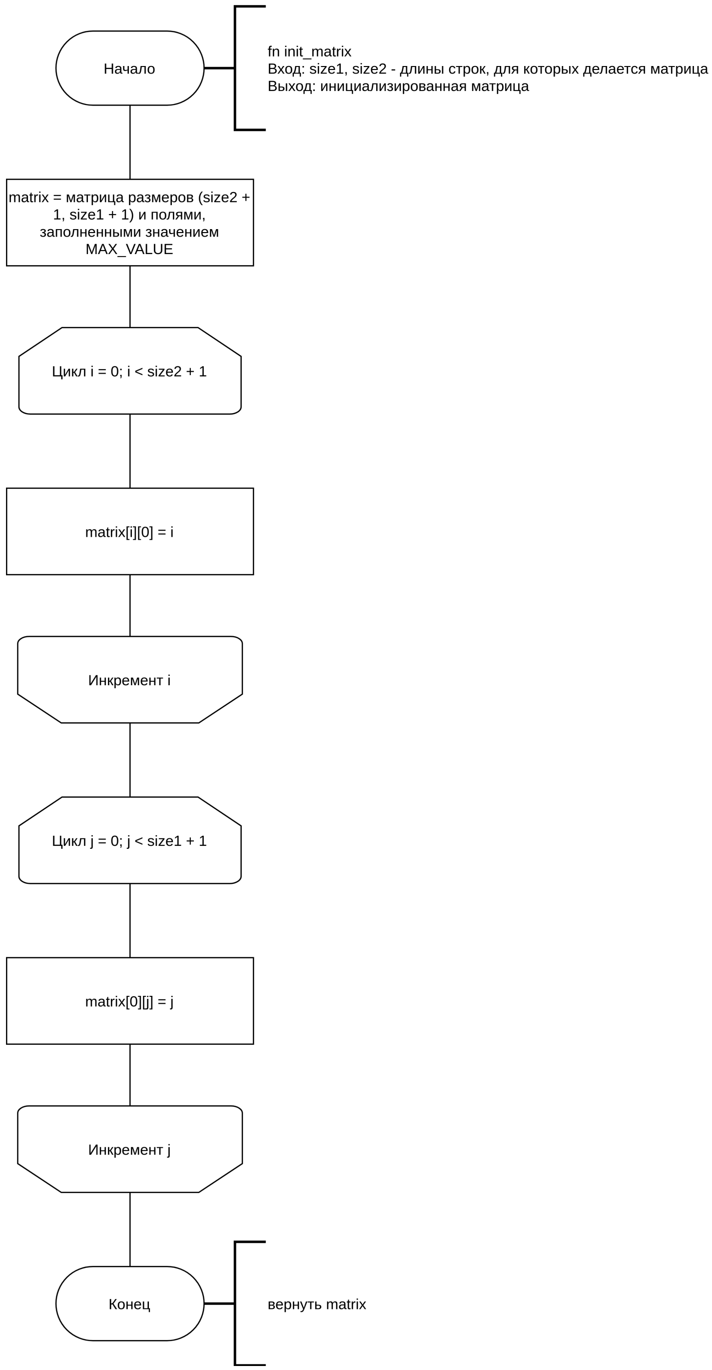


Рис. 2.2: Схема алгоритма инициализации матрицы для итеративных и рекурсивного с мемоизацией алгоритмов

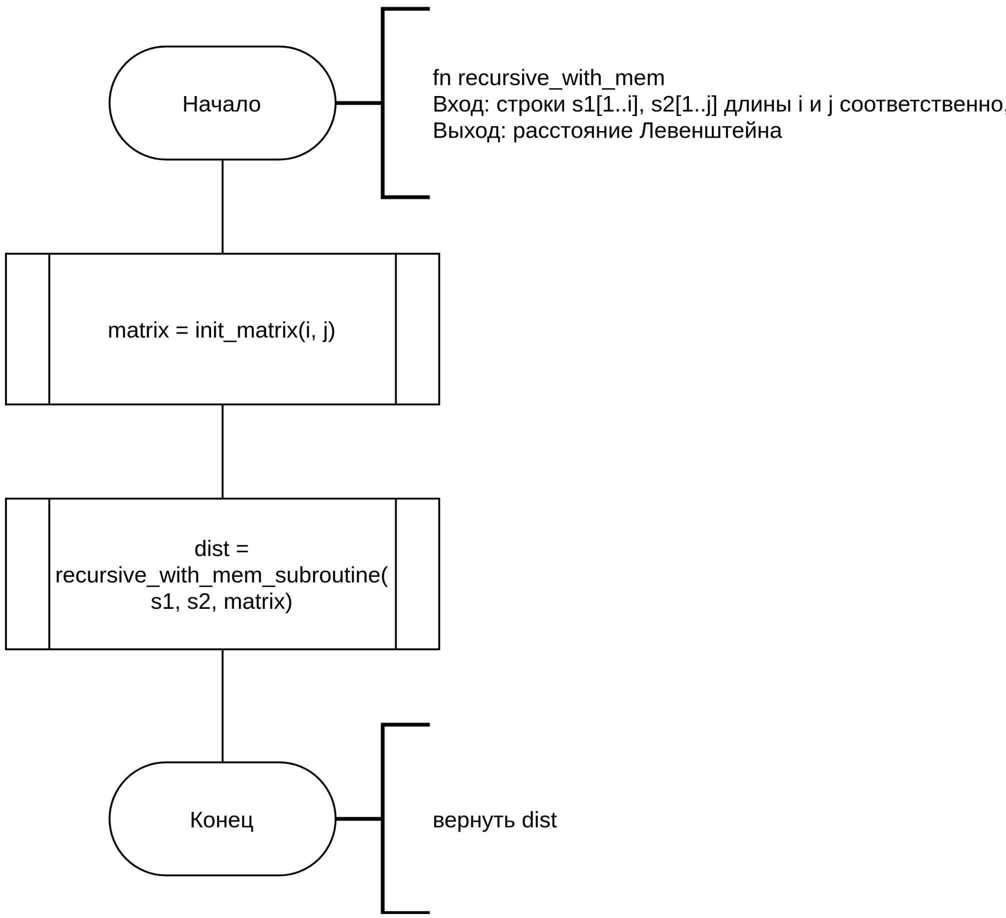


Рис. 2.3: Схема рекурсивного алгоритма с мемоизацией нахождения расстояния Левенштейна

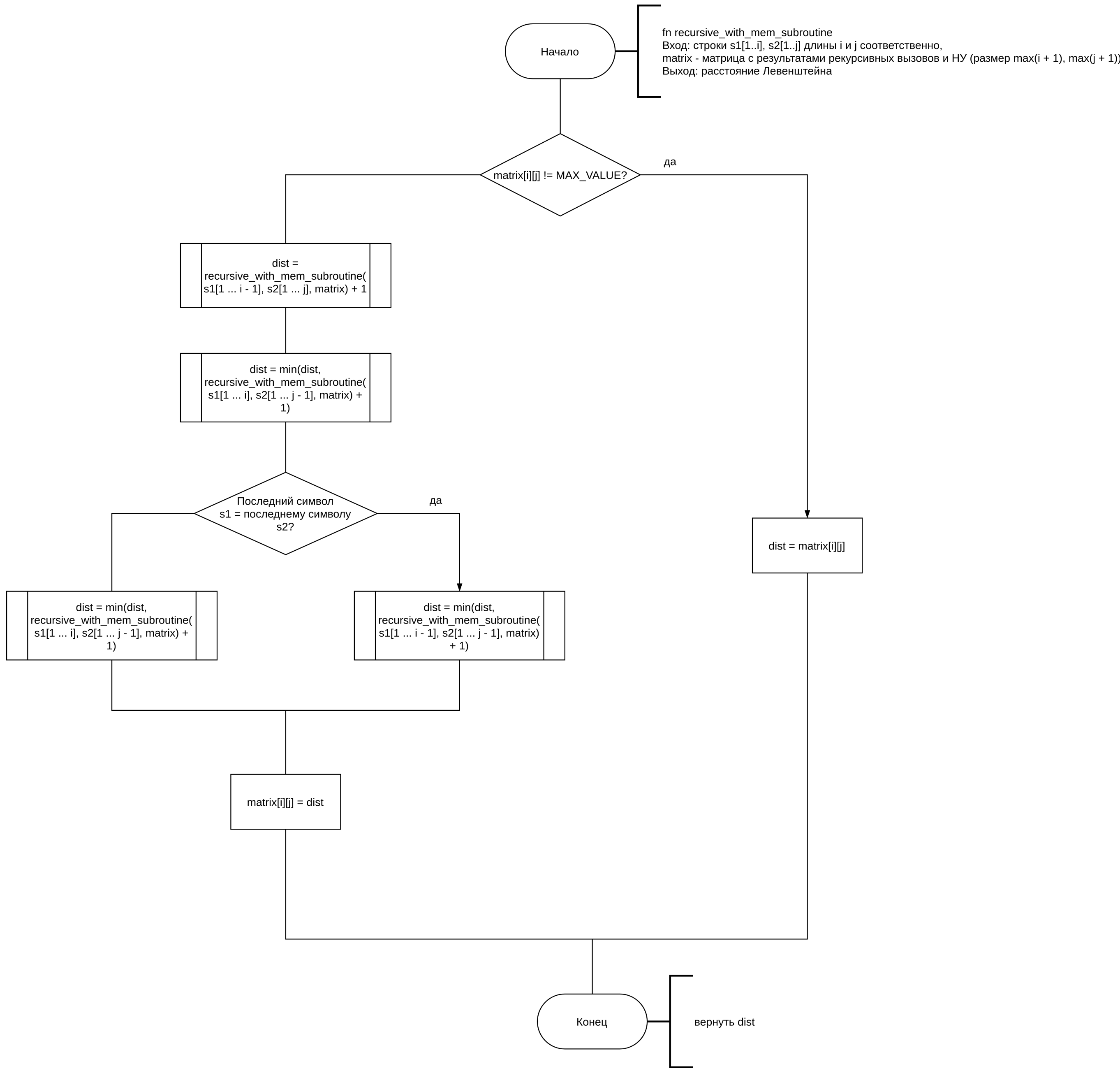


Рис. 2.4: Схема процедуры рекурсивного алгоритма с мемоизацией нахождения расстояния Дамерау-Левенштейна

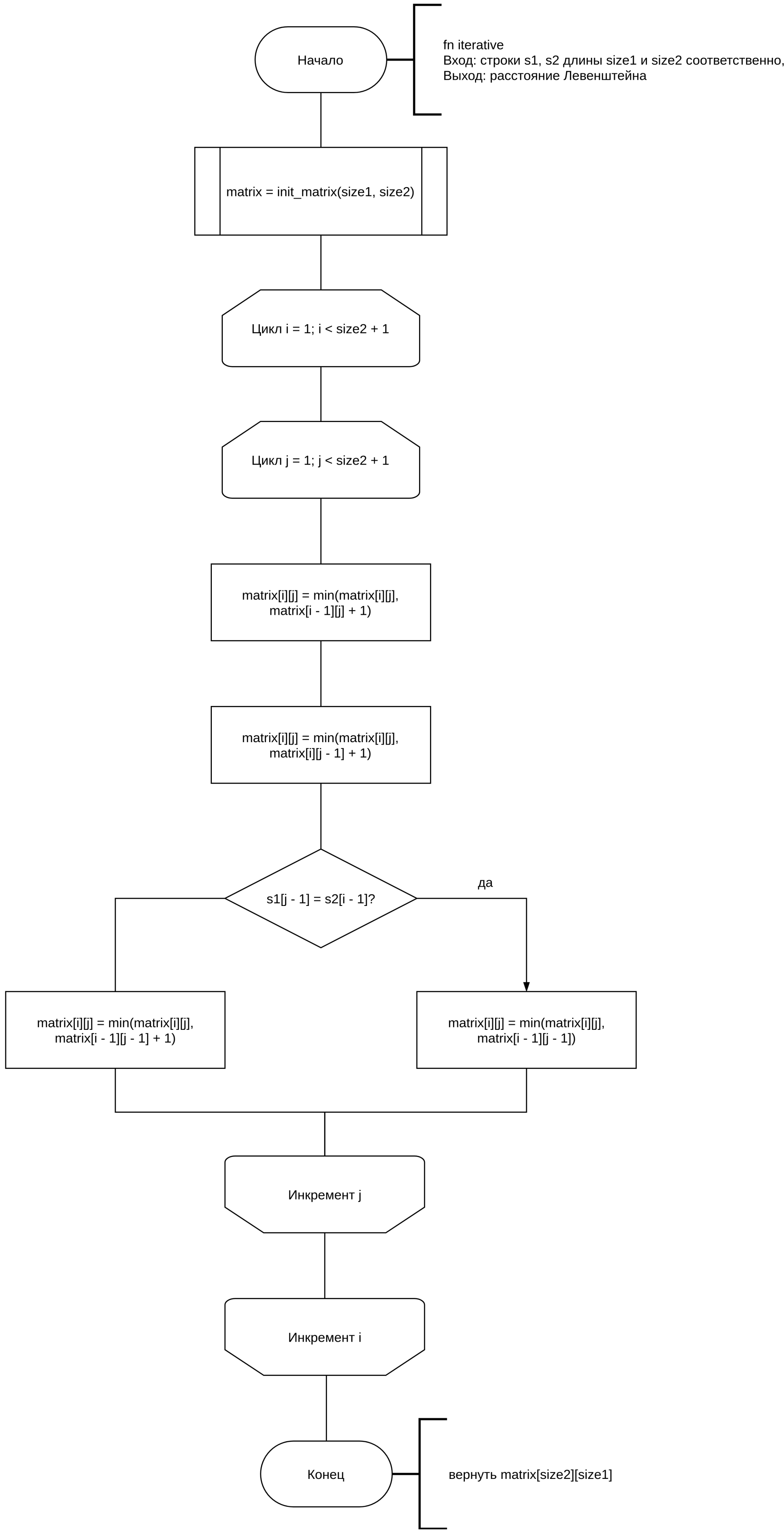


Рис. 2.5: Схема итеративного алгоритма нахождения расстояния Левенштейна

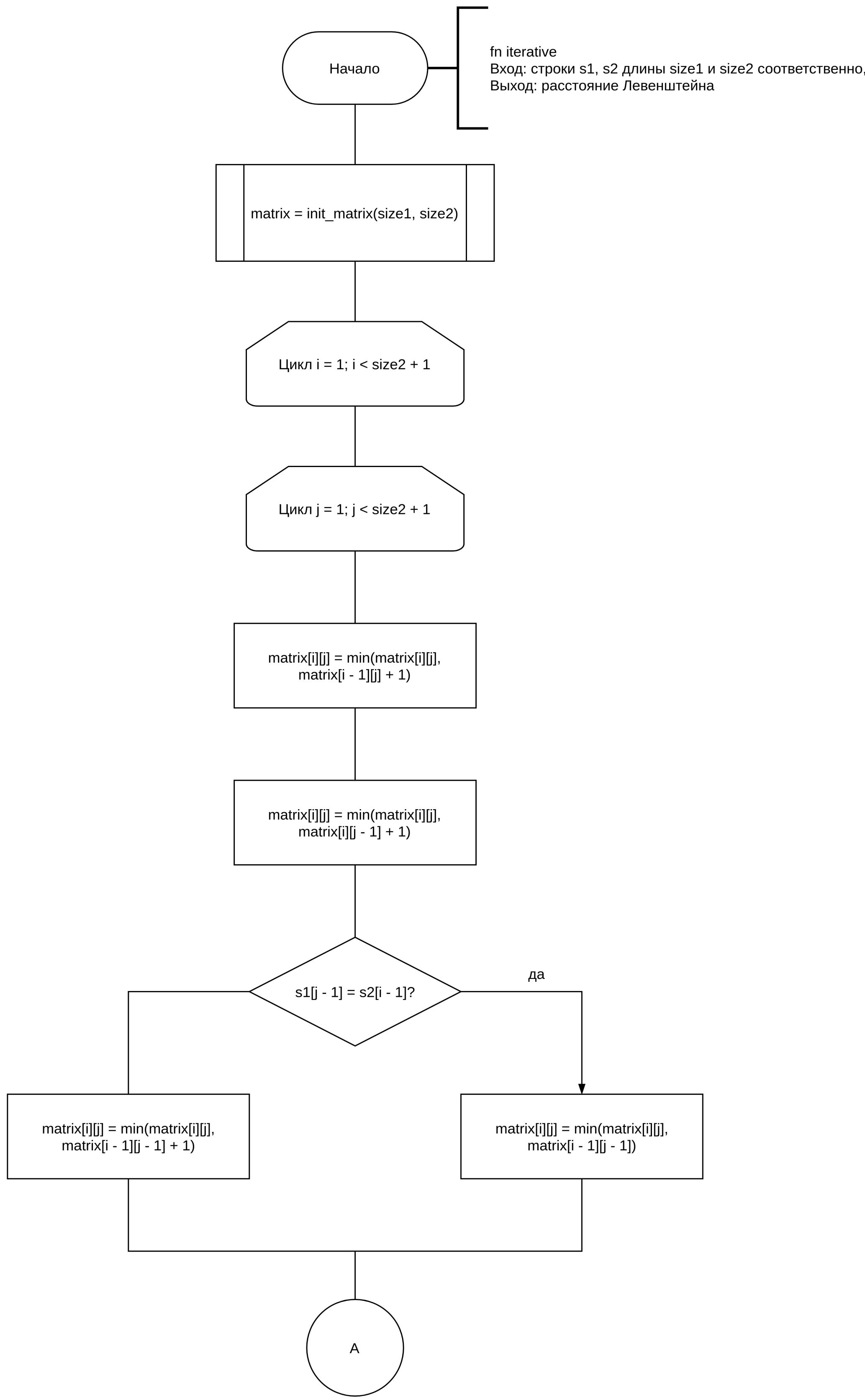


Рис. 2.6: Схема итеративного алгоритма нахождения расстояния Дамерау-Левенштейна

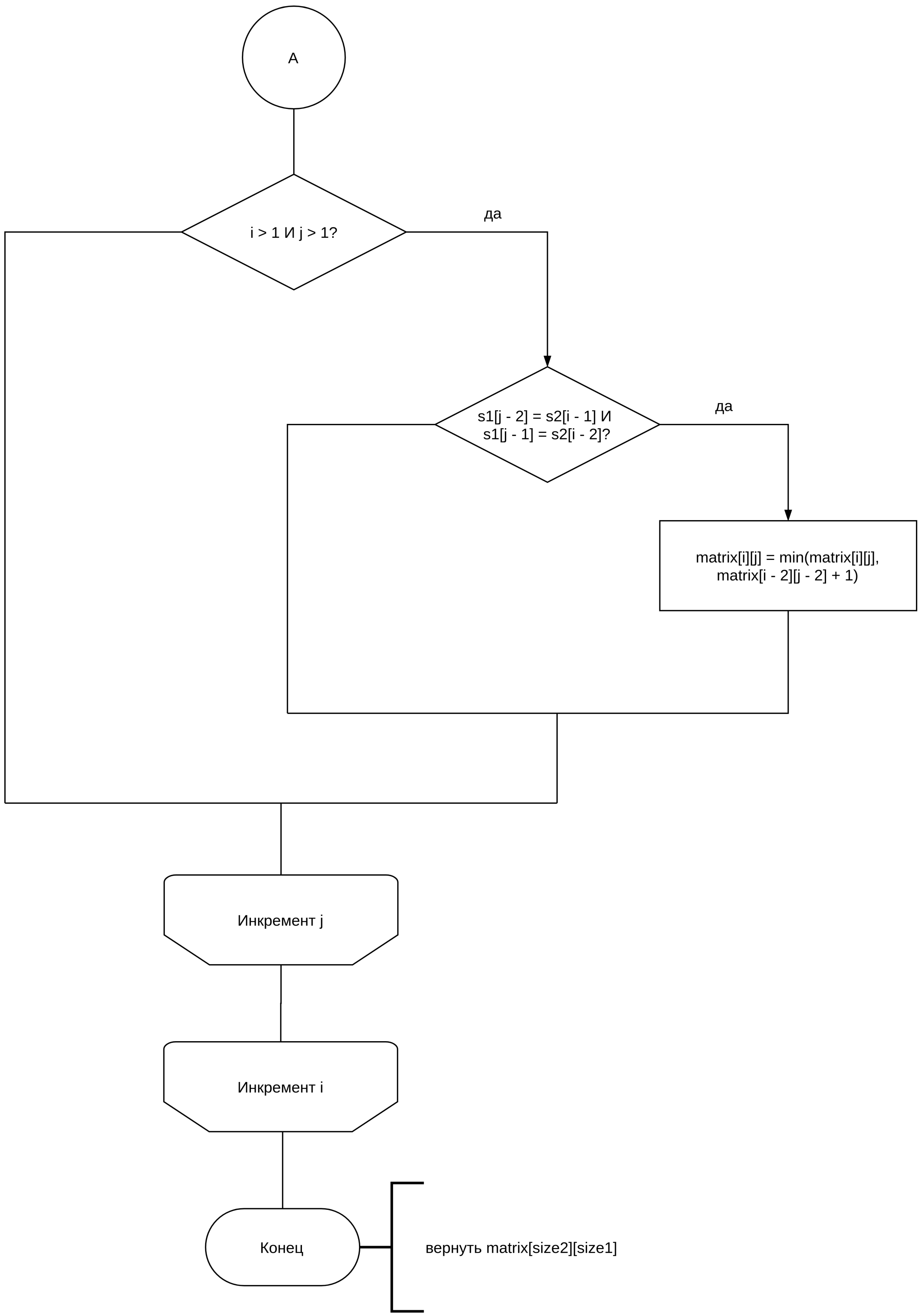


Рис. 2.7: Продолжение схемы матричного алгоритма нахождения расстояния Дамерау-Левенштейна

3 Технологическая часть

3.1 Выбор ЯП

Для реализации программ я выбрал язык программирования Rust, потому что мне нравится его скорость и возможности. среда разработки - Vim с настроенным RLS (Rust Language Server).

3.2 Реализация алгоритма

Листинг 3.1: Функция реализующая рекурсивный алгоритм нахождения расстояния Левенштейна

```
pub fn recursive(s1: &[char], s2: &[char]) -> (usize, usize) {
    _recursive(s1, s2, 0)
}

fn _recursive(s1: &[char], s2: &[char], depth: usize) -> (usize, usize) {
    if s1.len() == 0 {
        return (s2.len(), depth);
    } else if s2.len() == 0 {
        return (s1.len(), depth);
    }

    // insertion
    let (best_score, max_depth) = _recursive(&s1[..(s1.len() - 1)], s2,
        depth + 1);
    let best_score = best_score + 1;
    // deletion
    let (score, cur_depth) = _recursive(s1, &s2[..(s2.len() - 1)], depth +
        1);
    let (best_score, max_depth) = (min(best_score, score + 1), max(cur_depth
        , max_depth));
    // match/replace
    let (score, cur_depth) = _recursive(&s1[..(s1.len() - 1)], &s2[..(s2.len
        () - 1)], depth + 1);
    let score = if s1[s1.len() - 1] == s2[s2.len() - 1] { score } else {
        score + 1 };
```

```
    let (best_score, max_depth) = (min(best_score, score), max(cur_depth,
        max_depth));
    (best_score, max_depth)
}
```

Листинг 3.2: Функция инициализации матрицы для алгоритмов с мемоизацией

```
fn init_matrix(matrix: &mut [vec<usize>]) {
    if matrix.len() == 0 {
        return;
    }

    for i in 0..matrix.len() {
        matrix[i][0] = i;
    }

    for j in 0..matrix[0].len() {
        matrix[0][j] = j;
    }
}
```

Листинг 3.3: Функция реализующая рекурсивный алгоритм с мемоизацией нахождения расстояния Левенштейна

```
pub fn recursive_with_mem(s1: &[char], s2: &[char]) -> (usize, usize, vec<
    vec<usize>>) {
    let mut matrix = vec![vec![usize::max; s1.len() + 1]; s2.len() + 1];
    init_matrix(&mut matrix);
    let res = _recursive_with_mem(s1, s2, 0, &mut matrix);
    (res.0, res.1, matrix)
}

fn _recursive_with_mem(s1: &[char], s2: &[char], depth: usize, matrix: &mut
    [vec<usize>]) -> (usize, usize) {
    if matrix[s2.len()][s1.len()] != usize::max {
        return (matrix[s2.len()][s1.len()], depth);
    }

    // insertion
    let (best_score, max_depth) = _recursive_with_mem(&s1[..(s1.len() - 1)],
        &s2, depth + 1, matrix);
    let best_score = best_score + 1;
    // deletion
    let (score, cur_depth) = _recursive_with_mem(&s1, &s2[..(s2.len() - 1)],
        depth + 1, matrix);
    let (best_score, max_depth) = (min(best_score, score + 1), max(cur_depth
        , max_depth));
    // match/replace
```

```
    let (score, cur_depth) = _recursive_with_mem(&s1[..(s1.len() - 1)], &s2
        [..(s2.len() - 1)], depth + 1, matrix);
    let score = if s1[s1.len() - 1] == s2[s2.len() - 1]  { score } else {
        score + 1 };
    let (best_score, max_depth) = (min(best_score, score), max(cur_depth,
        max_depth));
    matrix[s2.len()][s1.len()] = best_score;
    (best_score, max_depth)
}
```

Листинг 3.4: Функция реализующая итеративный алгоритм нахождения расстояния левенштейна

```
pub fn iterative(s1: &[char], s2: &[char]) -> (usize, vec<vec<usize>>) {
    let mut matrix = vec![vec![usize::max; s1.len() + 1]; s2.len() + 1];
    init_matrix(&mut matrix);

    for i in 1..(s2.len() + 1) {
        for j in 1..(s1.len() + 1) {
            // insertion
            matrix[i][j] = min(matrix[i][j], matrix[i - 1][j] + 1);
            // deletion
            matrix[i][j] = min(matrix[i][j], matrix[i][j - 1] + 1);
            // match/replace
            let score = if s1[j - 1] == s2[i - 1] { matrix[i - 1][j - 1] }
                else { matrix[i - 1][j - 1] + 1 };
            matrix[i][j] = min(matrix[i][j], score);
        }
    }

    (matrix[s2.len()][s1.len()], matrix)
}
```

Листинг 3.5: Функция реализующая итеративный алгоритм нахождения расстояния дамерау-левенштейна

```
pub fn iterative_dl(s1: &[char], s2: &[char]) -> (usize, vec<vec<usize>>) {
    let mut matrix = vec![vec![usize::max; s1.len() + 1]; s2.len() + 1];
    init_matrix(&mut matrix);

    for i in 1..(s2.len() + 1) {
        for j in 1..(s1.len() + 1) {
            // insertion
            matrix[i][j] = min(matrix[i][j], matrix[i - 1][j] + 1);
            // deletion
            matrix[i][j] = min(matrix[i][j], matrix[i][j - 1] + 1);
            // match/replace
            let score = if s1[j - 1] == s2[i - 1] { matrix[i - 1][j - 1] }
                else { matrix[i - 1][j - 1] + 1 };
```

```
                matrix[i][j] = min(matrix[i][j], score);

                if i > 1 && j > 1 {
                    let transition_condition = s1[j - 1] == s2[i - 2] && s1[j - 2] == s2[i - 1];
                    if transition_condition {
                        matrix[i][j] = min(matrix[i][j], matrix[i - 2][j - 2] + 1);
                    }
                }
            }
        }

    (matrix[s2.len()][s1.len()], matrix)
}
```

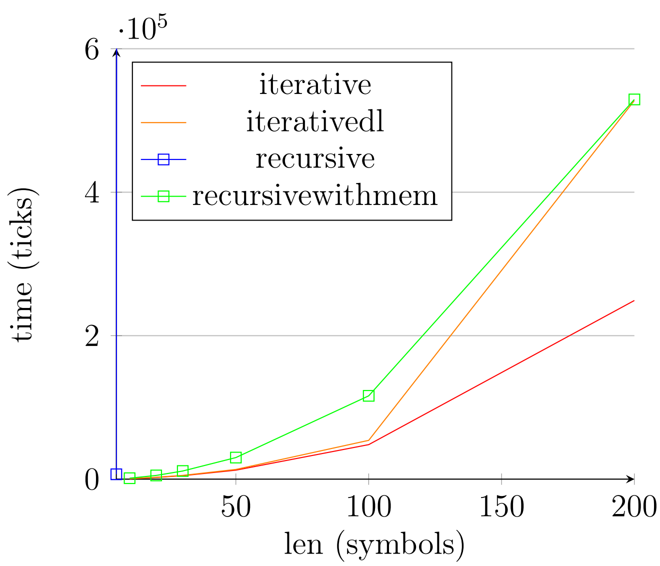
4 Исследовательская часть

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведен замер времени работы каждого из алгоритмов.

Таблица 4.1: время работы алгоритмов (в наносекундах)

len	recursive	recWithMem	iter	iterDl
10	32766430	1313	634	681
20	$> 10^{10}$	5157	2367	2582
30	$> 10^{10}$	11342	4813	5207
50	$> 10^{10}$	30066	12518	13533
100	$> 10^{10}$	116134	48111	54078
200	$> 10^{10}$	529335	249057	527797



Из экспериментов видно, что без применения техник динамеческого программирования даже на сравнительно небольших длинах строк (20) алгоритм выдает сильно худшие результаты в сравнении с алгоритмами, запоминающими результаты своих вычислений.

4.2 Тестовые данные

Проведем тестирование программы. в столбцах "Ожидание"и "Результат-4 числа соответствуют рекурсивному алгоритму нахождения расстояния Левенштейна, рекурсивному алгоритму с мемоизацией нахождения расстояния левенштейна, итеративному алгоритму нахождения расстояния Левенштейна и итеративному алгоритму нахождения расстояния Дамерау-Левенштейна.

Таблица 4.2: Таблица тестовых данных

№	Первое слово	Второе слово	Описание	Ожидание	Результат
1	добро	нутро	только замены	3 3 3 3	3 3 3 3
2	бро	добро	только удаления	2 2 2 2	2 2 2 2
3	доброта	добро	только вставки	2 2 2 2	2 2 2 2
4	доброта	одброат	только смены мест	4 4 4 2	4 4 4 2
5	дброта	добрта	вставки и удаления	2 2 2 2	2 2 2 2
6	изменить	подмена	все вместе	6 6 6 6	6 6 6 6

Заключение

В рамках данной лабораторной работы были изучены и реализованы рекурсивные и итеративные версии алгоритмов Левенштейна и Дамерау-Левенштейна.

Также были при переходе от рекурсивного алгоритма к рекурсивного с мемоизацией были применены техники динамического программирования, что позволило заметно улучшить скорость работы алгоритма.

Было сравнительно показано и экспериментально 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 становится очевидно, что алгоритмы, не использующие техники динамического программирования делают очень много лишней работы, в следствие чего работают значительно медленнее, чем алгоритмы, использующие соответствующие техники. В реальной жизни все задачи, имеющие рекурсивное соотношение, так или иначе используют техники динамического программирования.

Литература

[1] Курс по алгоритмам и структурам данных на образовательной платформе Coursera. Режим доступа: `https://www.coursera.org/specializations/data-structures-algorithms`. Дата обращения: 15.09.2020.

[2] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады АН СССР, 1965. Т. 163. С. 845 - 848.

[3] Гасфилд. Строки, деревья и последовательности в алгоритма. Информатика и вычислительная биология. Невский Диалект БВХ-Петербург, 2003.