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»

Отчёт

по лабораторной работе №1

Название: Расстояния Левенштейна и Дамерау-Левенштейна

Дисциплина: Анализ алгоритмов

Студент	ИУ7-55Б		И. Е. Афимин
	(Группа)	(Подпись, дата)	(И.О. Фамилия)
Преподаватель			Л.Л. Волкова
		(Подпись, дата)	(И.О. Фамилия)

Москва, 2020

Содержание

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 решения следующих задач:

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

Целью данной лабораторной работы является изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна.

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

1.1 Расстояние Левенштейна и Дамерау-Левенштейна

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.

Операции I, D, R имеют штраф 1, а операция M - 0.

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:

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

1.2 Вывод

В данном разделе были рассмотрены расстояния Левенштейна и Дамерау-Левенштейна. Расстояние Дамерау-Левенштейна, учитывающее возможность перестановки соседних символов, является модификацией расстояния Левенштейна.

2 Конструкторская часть

В данном разделе будут рассмотрены требования к функциональности ПО, схемы алгоритмов и опредены способы тестирования.

2.1 Требования к функциональности ПО

В данной работе требуется обеспечить следующую минимальную функциональность консольного приложения.

1) возможность считать две строки;
2) вывод расстояний Левенштейна и Дамерау-Левенштейна между строками.

2.2 Тесты

Тестирование ПО будет проводиться методом чёрного ящика. Необходимо проверить работу системы на тривиальных случаях (одна или обе строки пустые, строки полностью совпадают) и несколько нетривальных случаев.

2.3 Схемы алгоритмов

Ниже будут представлены схемы алгоритмов поиска растояния Левенштейна:

1) нерекурсивного с заполнением матрицы (рисунок 2.1);
2) рекурсивного без заполнения матрицы (рисунок 2.2);
3) рекурсивного с заполнением матрицы (рисунок 2.3).

Также будет представлена схема нерекурсивного алгоритма поиска растояния Дамерау-Левенштейна (рисунок 2.4).

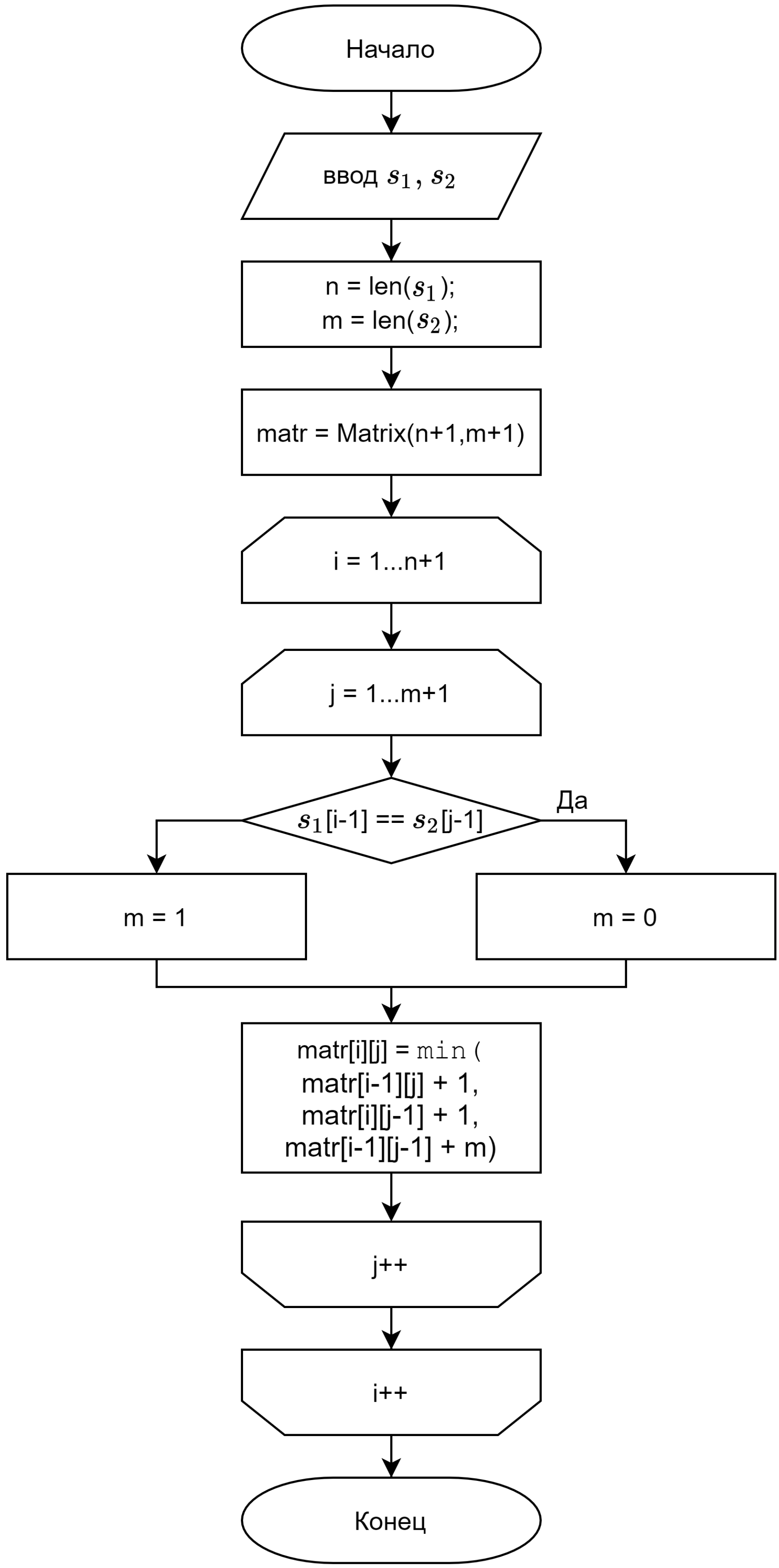

Рисунок 2.1 — Схема нерекурсивного поиска с заполнением матрицы

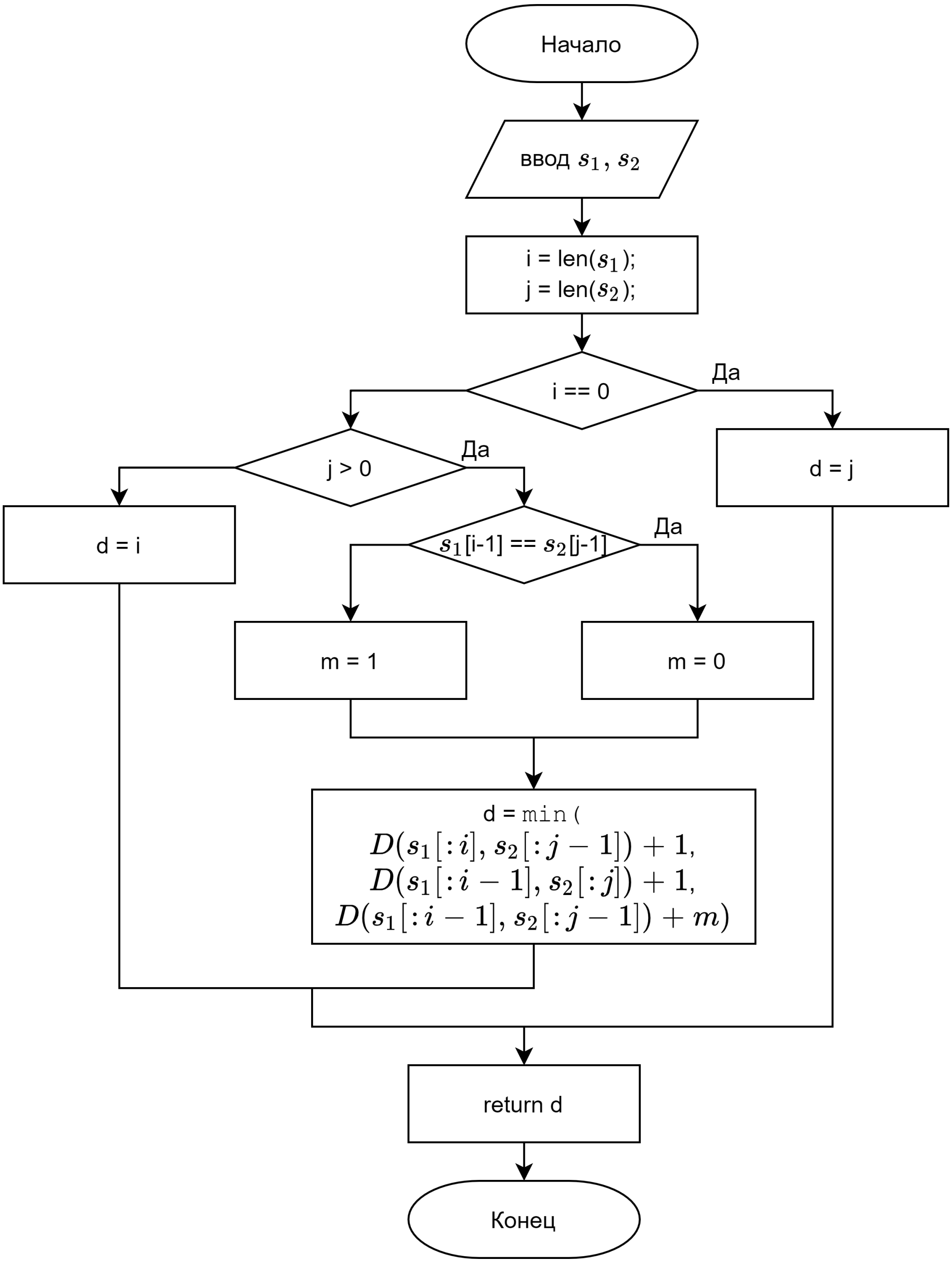

Рисунок 2.2 — Схема рекурсивого поиска без заполнения матрицы

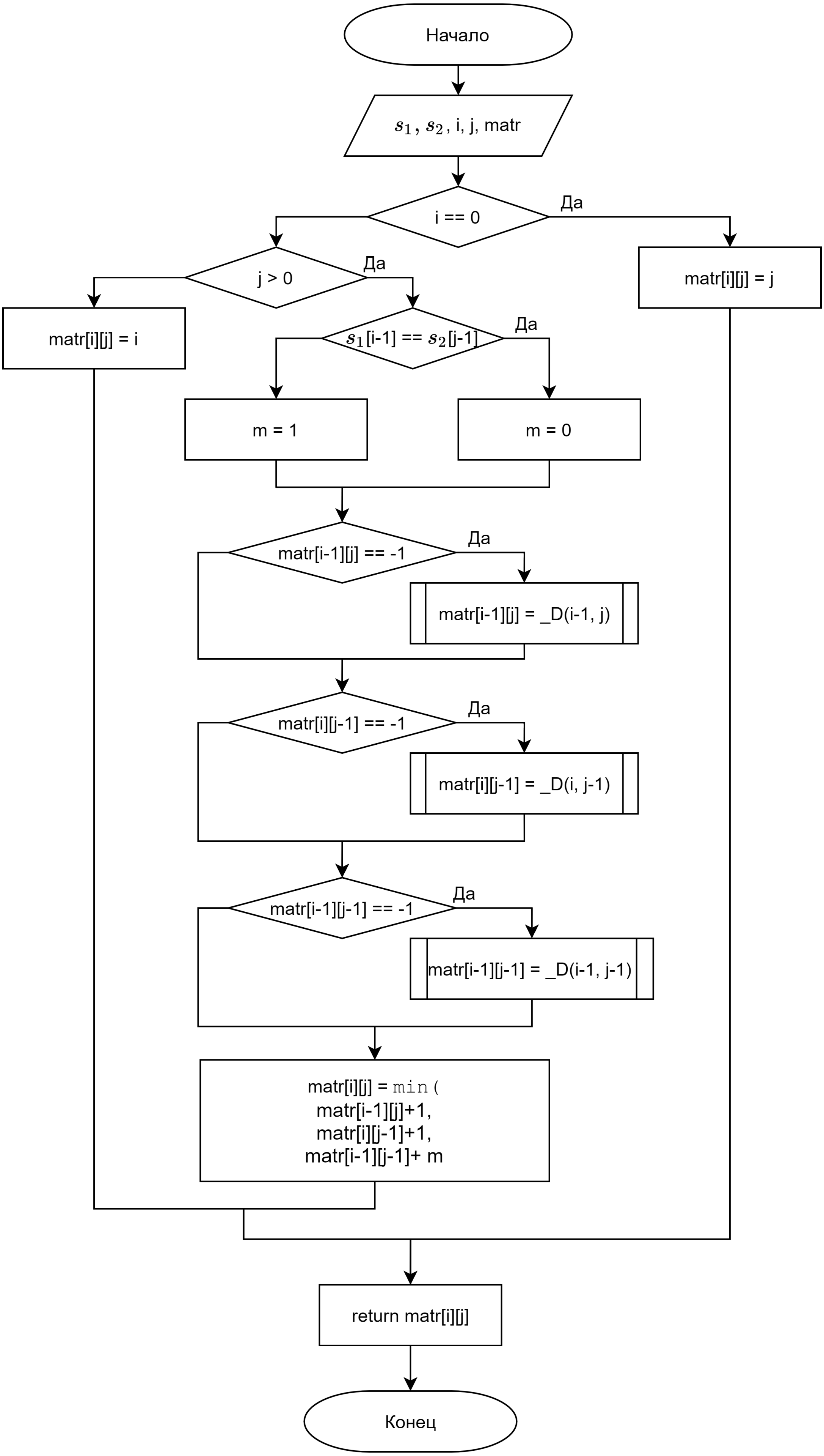

Рисунок 2.3 — Схема рекурсивого поиска с заполнением матрицы

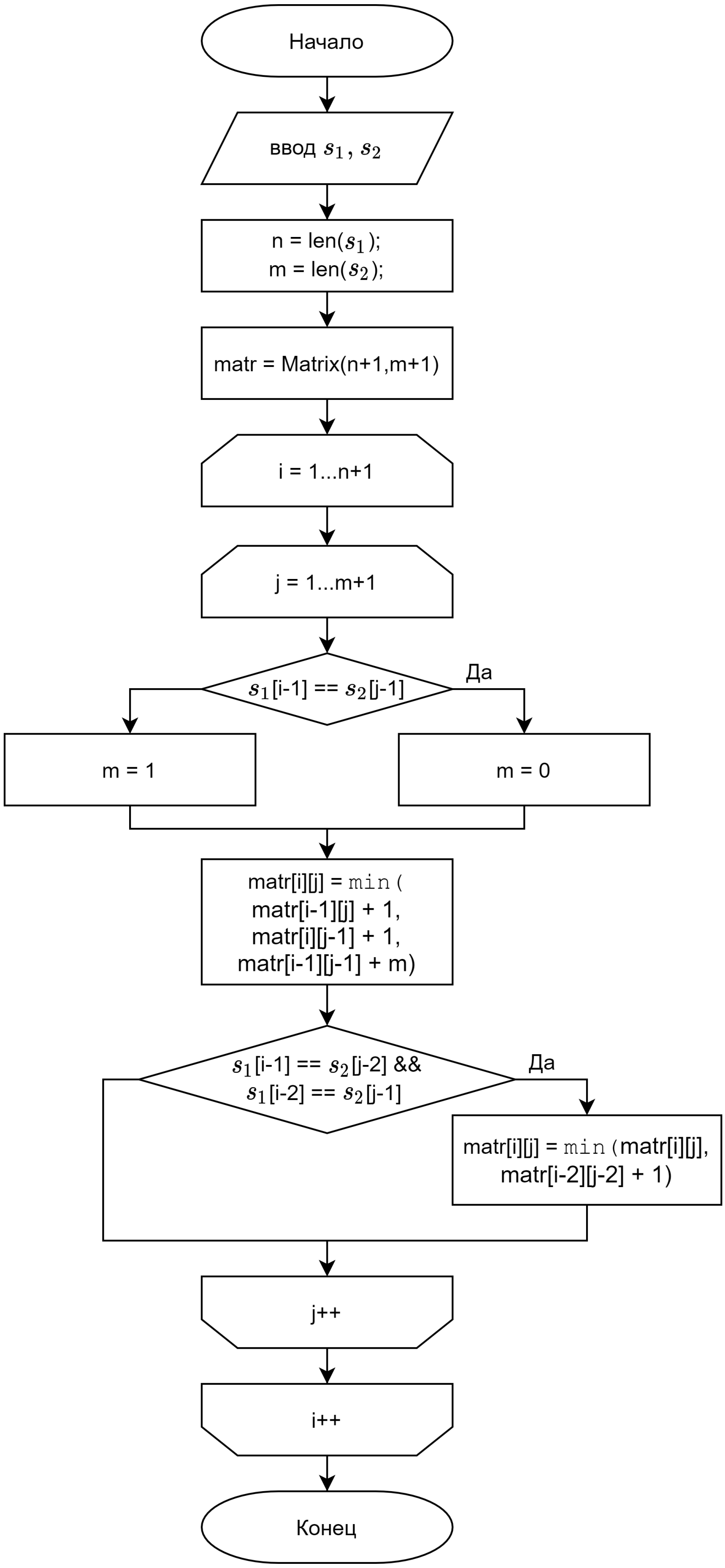

Рисунок 2.4 — Схема нерекурсивного поиска растояния Дамерау-Левенштейна

3 Технологическая часть

В данном разделе будут выбраны средства реплизации ПО, представлен листинг кода и проведён теоритический анализ максимальной затрачиваемой памяти.

3.1 Средства реализации

Для реализации программ я выбрал язык программирования C++, так как имею большой опыт работы с ним. Для замера процессорного времени была использована ассемблерная вставка[1].

3.2 Листинг программы

Ниже представлены листинги кода поиска растояния Левенштейна:

1) нерекурсивного с заполнением матрицы (листинг 3.1);
2) рекурсивного без заполнения матрицы (листинг 3.2);
3) рекурсивного с заполнением матрицы (листинг 3.3).

И код функции поиска растояния Дамерау-Левенштейна (листинг 3.4).

Листинг 3.1 — Функция нахождения расстояния Левенштейна матрично

```
int LevDist(string str1, string str2)
{
    vector<vector<int>> *DistMatr = CreateMatrForLevDist(str1.size() + 1, str2.size() + 1);

    for (int i = 1; i < (*DistMatr).size(); i++)
    {
        for (int j = 1; j < (*DistMatr)[i].size(); j++)
        {
            (*DistMatr)[i][j] = min((*DistMatr)[i - 1][j] + 1,
                                    min((*DistMatr)[i][j - 1] + 1,
                                        (*DistMatr)[i - 1][j - 1]+ (str1[i - 1] == str2[j - 1] ? 0 : 1)));
        }
    }

    int dist = (*DistMatr)[str1.size()][str2.size()];

    FreeLevDistMatr(DistMatr);

    return dist;
}
```

Листинг 3.2 — Функция нахождения расстояния Левенштейна рекурсивно без матрицы

```
int RLevDist(string str1, string str2)
{
    if (str1 == "" or str2 == "")
    {
        return abs((int)(str1.size() - str2.size()));
    }

    return min(RLevDist(str1.substr(0, str1.size() - 1), str2) + 1,
            min(RLevDist(str1, str2.substr(0, str2.size() - 1)) + 1,
                RLevDist(str1.substr(0, str1.size() - 1), str2.substr(0, str2.size() - 1))
                + (str1[str1.size() - 1] == str2[str2.size() - 1] ? 0 : 1)));
}
```

Листинг 3.3 — Функция нахождения расстояния Левенштейна рекурсивно с матрицей

```
int RMatrLevDist_RECURSION2(string str1, string str2, vector<vector<int>> *DistMatr)
{
    if (str1.size() == 0)
    {
        (*DistMatr)[str1.size()][str2.size()] = str2.size();
    }
    else if (str2.size() == 0)
    {
        (*DistMatr)[str1.size()][str2.size()] = str1.size();
    }
    else
    {
        if ((*DistMatr)[str1.size()][str2.size() - 1] == -1)
        {
            (*DistMatr)[str1.size()][str2.size() - 1] =
                RMatrLevDist_RECURSION2(str1, str2.substr(0, str2.size() - 1), DistMatr);
        }

        if ((*DistMatr)[str1.size() - 1][str2.size()] == -1)
        {
            (*DistMatr)[str1.size() - 1][str2.size()] =
                RMatrLevDist_RECURSION2(str1.substr(0, str1.size() - 1), str2, DistMatr);
        }

        if ((*DistMatr)[str1.size() - 1][str2.size() - 1] == -1)
        {
            (*DistMatr)[str1.size() - 1][str2.size() - 1] =
                RMatrLevDist_RECURSION2(str1.substr(0, str1.size() - 1),
```

```
                                                                   str2.substr(0,
                                                                       str2.size() - 1),
                                                                       DistMatr);
        }

        int value = (str1[str1.size() - 1] == str2[str2.size() - 1] ? 0 : 1);
        (*DistMatr)[str1.size()][str2.size()] = min((*DistMatr)[str1.size() -
            1][str2.size()] + 1, min((*DistMatr)[str1.size()][str2.size() - 1] + 1,
            (*DistMatr[str1.size() - 1][str2.size() - 1] + value));

    }
    return (*DistMatr)[str1.size()][str2.size()];
}
int RMatrLevDist(string str1, string str2)
{
    vector<vector<int>> *DistMatr = CreateMatrForLevDist2(str1.size() + 1,
        str2.size() + 1);

    int dist = RMatrLevDist_RECURSION2(str1, str2, DistMatr);

    FreeLevDistMatr(DistMatr);

    return dist;
}
```

Листинг 3.4 — Функция нахождения расстояния Дамерау-Левенштейна матрично

```
int Dam_LevDist(string str1, string str2)
{
    vector<vector<int>> *DistMatr = CreateMatrForLevDist(str1.size() + 1,
        str2.size() + 1);

    for (int i = 1; i < (*DistMatr).size(); i++)
    {
        for (int j = 1; j < (*DistMatr)[i].size(); j++)
        {
            (*DistMatr)[i][j] = min((*DistMatr)[i - 1][j] + 1,
                                    min((*DistMatr)[i][j - 1] + 1,
                                        (*DistMatr)[i - 1][j - 1] + (str1[i - 1] ==
                                            str2[j - 1] ? 0 : 1)));

            if ((i > 1 and j > 1) and str1[i - 1] == str2[j - 2] and str1[i - 2] ==
                str2[j - 1])
            {
                (*DistMatr)[i][j] = min((*DistMatr)[i][j], (*DistMatr)[i - 2][j - 2]
                    + 1);
            }
```

```
        }
    }

    int dist = (*DistMatr)[str1.size()][str2.size()];

    FreeLevDistMatr(DistMatr);

    return dist;
}
```

3.3 Тестирование

В таблице 3.1 отображён возможный набор тестов для тестирования методом чёрного ящика. В столбцах "Ожидаемый результат"и "Полученный результат"4 числа соответсвуют рекурсивному алгоритму без матрицы нахождения расстояния Левенштейна, рекурсивному алгоритму с матрицей нахождения расстояния Левенштейна, матричному алгоритму нахождению расстоянию Левенштейна, матричному алгоритму нахождения расстояние Дамерау-Левенштейна.

Таблица 3.1 — Таблица тестовых данных

№	строка 1	строка 2	Ожидаемый результат	Фактический результат
1	kot	skat	2 2 2 2	2 2 2 2
2	zxc	cxz	2 2 2 2	2 2 2 2
3	sok	kokos	3 3 3 3	3 3 3 3
4	qwerty	asdfgh	6 6 6 6	6 6 6 6
5	qwerty	qewryt	4 4 4 2	4 4 4 2

3.4 Сравнительный анализ потребляемой памяти

С точки зрения использования памяти алгоритмы Левенштейна и Дамерау-Левенштейна не отличаются, следовательно, достаточно рассмотреть лишь разницу рекурсивной и матричной реализаций данных методов.

Использование памяти на строках s_1, s_2 длиной n и m соответственно при использовании матрицы теоритически определяется формулой (3.1):

$$V = (n+1)(m+1)sizeof(int) + 4sizeof(size\_t) + 2sizeof(char*) + sizeof(char)(n+m) \quad (3.1)$$

Максимальный расход памяти памяти на строках s_1, s_2 длиной n и m соответственно при использовании рекурсии определяется максимальной глибиной стека вызовов, которая теоритически определяется формулой (3.2):

$$V = sizeof(char)(n+m) + (n+m)(2sizeof(char*) + 3sizeof(size\_t)) \quad (3.2)$$

4 Исследовательская часть

В данном разделе будут проведены эксперименты для проведения сравнительного анализа алгоритмов по затрачиваемому процессорному времени[2] и максимальной используемой памяти.

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведен замер времени работы каждой реализации на строках равной длинны, с их случайным заполнением. В таблице 4.1 приняты следующие обозначения:

1) LevRecursion – расстояние Левенштейна (рекурсивно, без матрицы) ;
2) LevMatrix – расстояние Левенштейна (матрично, без рекурсии) ;
3) LevRecursionMatrix – расстояние Левенштейна (матрично, с рекурсией);
4) DamLevMatrix – расстояние Дамерау-Левенштейна (матрично).

Графики по таблице изображены на рисунках 4.1.

Таблица 4.1 — Время работы реализации алгоритмов (в наносекундах)

len	LevMatrix	LevRecursion	LevRecursionMatrix	DamLevMatrix
1	4982	3024	6445	5064
2	4993	14832	10188	5082
3	5681	62872	14739	5725
4	6325	302576	20268	6382
5	7819	1504108	27811	7960

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 Дамерау-Левенштейна работает немного дольше алгоритма Левенштейна, т.к. в нем добавлены дополнительные проверки, однако алгоритмы сравнимы по временной эффективности.

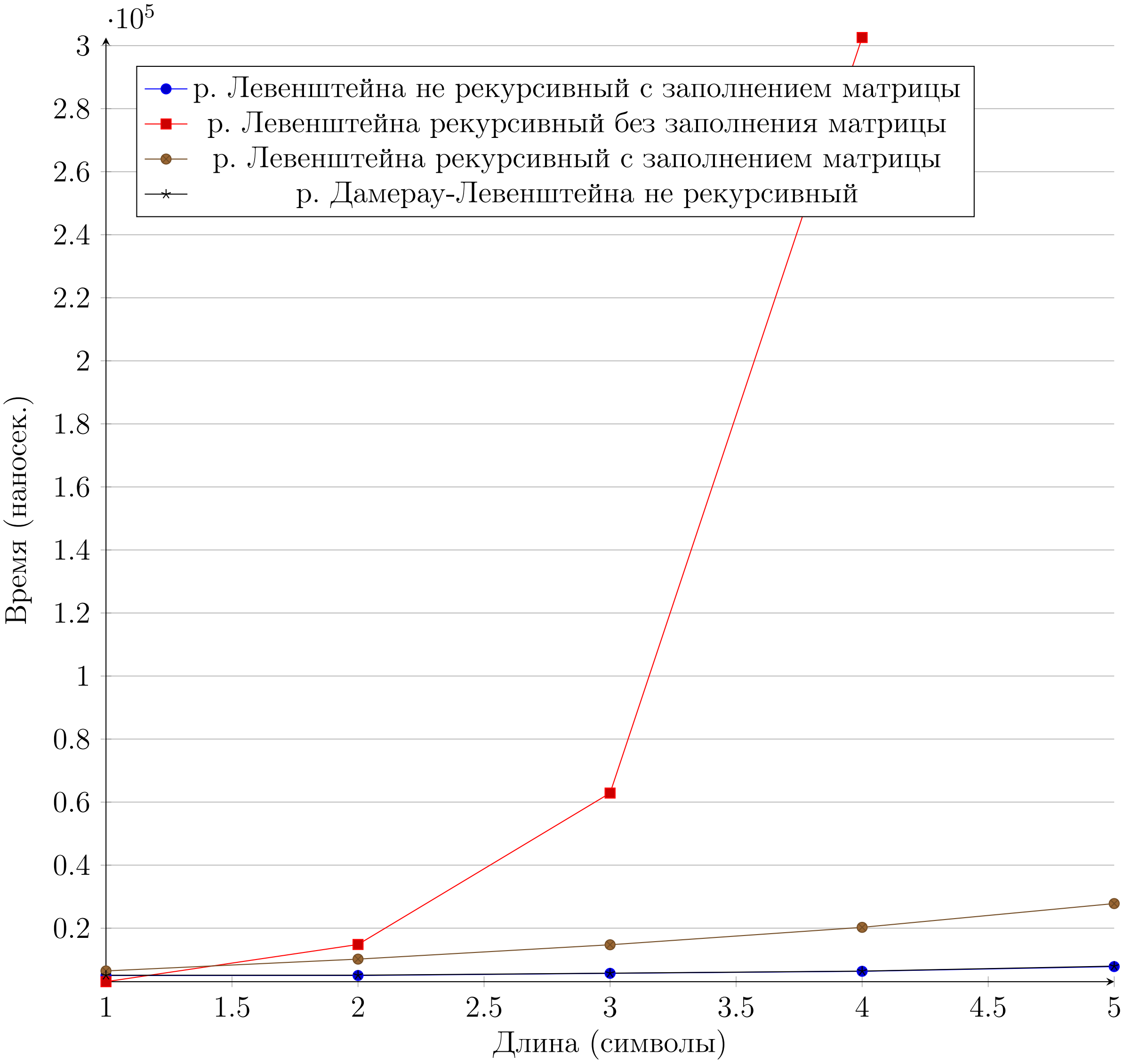

Рисунок 4.1 — График зависимости времени работы алгоритмов от длин строк

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.

Список использованных источников

1. Ассемблерные вставки в AVR-GCC. // [Электронный ресурс]. Режим доступа: http://we.easyelectronics.ru/AVR/assemblernye-vstavki-v-avr-gcc.html, (дата обращения: 03.10.2020).

2. C/C++: как измерять процессорное время. // [Электронный ресурс]. Режим доступа: https://habr.com/ru/post/282301/, (дата обращения: 03.10.2020).