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 №3

Название: Исследование трудоемкости сортировок

Дисциплина: Анализ алгоритмов

Студент ИУ7-52Б _____ Н. В. Ляпина
(Группа) (Подпись, дата) (И.О. Фамилия)

Преподователь _____ Л.Л. Волкова
(Подпись, дата) (И.О. Фамилия)

Москва, 2021

Содержание

Введение

Алгоритм сортировки – это алгоритм для упорядочивания элементов в списке. Входом является последовательность из n элементов: $a_1, a_2, ..., a_n$. Результатом работы алгоритма сортировки является перестановка исходной последовательности $a'_1, a'_2, ..., a'_n$, такая что $a'_1 \leqslant a'_2 \leqslant ... \leqslant a'_n$, где $\leqslant$ – отношение порядка на множестве элементов списка. Поля, служащие критерием порядка, называются ключом сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся какие-либо данные, никак не влияющие на работу алгоритма.

В данной лабораторной работе рассмотриваются алгоритмы:

1) сортировка плавная ;
2) сортировка расчёской;
3) сортировка слиянием.

Цель лабораторной работы:
изучение трудоемкости алгоритмов сортировки и их реализация.

Задачи лабораторной работы:

1) выбрать инструменты для замера процессорного времени выполнения реализации алгоритмов;
2) изучить алгоритмы сортировки расчёской, слиянием и плавной сортировки;
3) реализовать:
    а) плавную сортировку;
    б) сортировку расчёской;
    в) сортировку слиянием;
4) дать оценку трудоёмкости в лучшем, произвольном и худшем случае;
5) провести замеры процессорного времени работы для лучшего, худшего и произвольного случая.

1 Аналитический раздел

1.1 Плавная сортировка (Smoothsort)

Плавная сортировка – алгоритм сортировки выбором, разновидность пирамидальной сортировки, разработанная Э. Дейкстрой. Но, в отличии от пирамидальной сортировки, в которой используется двоичная куча, здесь исползуется специальная куча, полученная с помощью чисел Леонардо.

Числа Леонардо

Числа Ленонардо - последовательность чисел, задаваемая зависимостью:

$$L(n) = \begin{cases} 1, & \text{если } n = 0; \\ 1, & \text{если } n = 1; \\ L(n-1) + L(n-2) + 1, & \text{если } n > 1, \end{cases} \tag{1.1}$$

где n – индекс в массиве чисел Леонардо.

Абсолютно любое целое число можно представить в виде суммы чисел Леонардо, имеющих разные порядковые номера. Массив из n элементов не всегда можно представить в виде одной кучи Леонардо, но любой массив можно разделить на несколько подмассивов, которые будут соответствовать разным числам Леонардо.

При разбиении массива важно учитывать, что:

1) Каждая куча Чисел Леонардо представляет собой несбалансированное бинарное дерево;

2) Корень каждой такой кучи – это последний элемент соответствующего подмассива;

3) Любой узел кучи со всеми своими потомками также представляет из себя леонардову кучу меньшего порядка.

Алгоритм

1) Создаем из массива кучу леонардовых куч, каждая из которых является сортирующим деревом;

а) Перебираем элементы массива слева-направо;

б) Проверяем, можно ли с помощью текущего элемента объединить две крайние слева кучи в уже имеющейся куче леонардовых куч:

— Если да, то объединяем две крайние слева кучи в одну, текущий элемент становится корнем этой кучи, делаем просейку для объединённой кучи;

— Если нет, то добавляет текущий элемент в качестве новой кучи (состоящей пока из одного узла) в имеющуюся кучу леонардовых куч.

2) Извлекает из куч текущие максимальные элементы, которые перемещакем в конец неотсортированной части массива:

а) Ищем максимумы в леонардовых кучах. Так как на предыдущем этапе для куч постоянно делалась просейка, максимумы находятся в корнях этих куч;

б) Найденный максимум (который является корнем одной из куч) меняем местами с последним элементом массива (который является корнем самой последней кучи);

в) После этого обмена куча, в который был найден максимум перестала быть сортирующим деревом. Поэтому делаем для неё просейку;

г) В последней куче удаляем корень, в результате чего эта куча распадается на две кучи;

д) После перемещения максимального элемента в конец, отсортированная часть массива увеличилась, а неотсортированная часть уменьшилась;

е) Повторить пункты $a - b$

1.2 Сортировка расчёской

Основная идея сортировки расчёской заключается в том, чтобы изначально брать самое большое расстояние между сравниваемыми элементами. Затем, по мере упорядочивания массива сужать это расстояние до минимального. Таким образом, мы как бы расчёсываем массив сначала широким гребнем, потом гребнем поменьше. Этот принцип отражается в названии сортировки. Первоначальный разрыв между сравниваемыми элементами лучше брать с учётом специальной величины, называемой фактором уменьшения, оптимальное значение которой равно примерно 1,247.

Сначала расстояние между элементами максимально, то есть равно размеру массива минус один. Затем, пройдя массив с этим шагом, необходимо поделить шаг на фактор уменьшения и пройти по списку вновь. Так продолжается до тех пор, пока разность индексов не достигнет единицы. В этом случае сравниваются соседние элементы как и в сортировке пузырьком, но такая итерация одна.

1.3 Сортировка слиянием

Алгоритм использует принцип «разделяй и властвуй»: задача разбивается на подзадачи меньшего размера, которые решаются по отдельности, после чего их решения комбинируются для получения решения исходной задачи. Конкретно процедуру сортировки слиянием можно описать следующим образом:

1) Если в рассматриваемом массиве один элемент, то он уже отсортирован — алгоритм завершает работу.

2) Иначе массив разбивается на две части, которые сортируются рекурсивно.

3) После сортировки двух частей массива к ним применяется процедура слияния, которая по двум отсортированным частям получает исходный отсортированный массив.

Вывод

Были рассмотрены алгоритмы сортировки расчёской, слиянием и плавная сортировка. Каждый имеет свою особенность, а конкретно сложность работы в лучшем/худшем случаях.

2 Констукторский раздел

Алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна можно реализовать несколькими способами. В данном разделе будут рассмотрены схемы алгоритмов, требования к функциональности ПО, и определены способы тестирования.

2.1 Разработка алгоритмов

Ниже будут представлены схемы алгоритмов поиска растояния Дамерау-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 Левенштейна (рисунок 2.4).

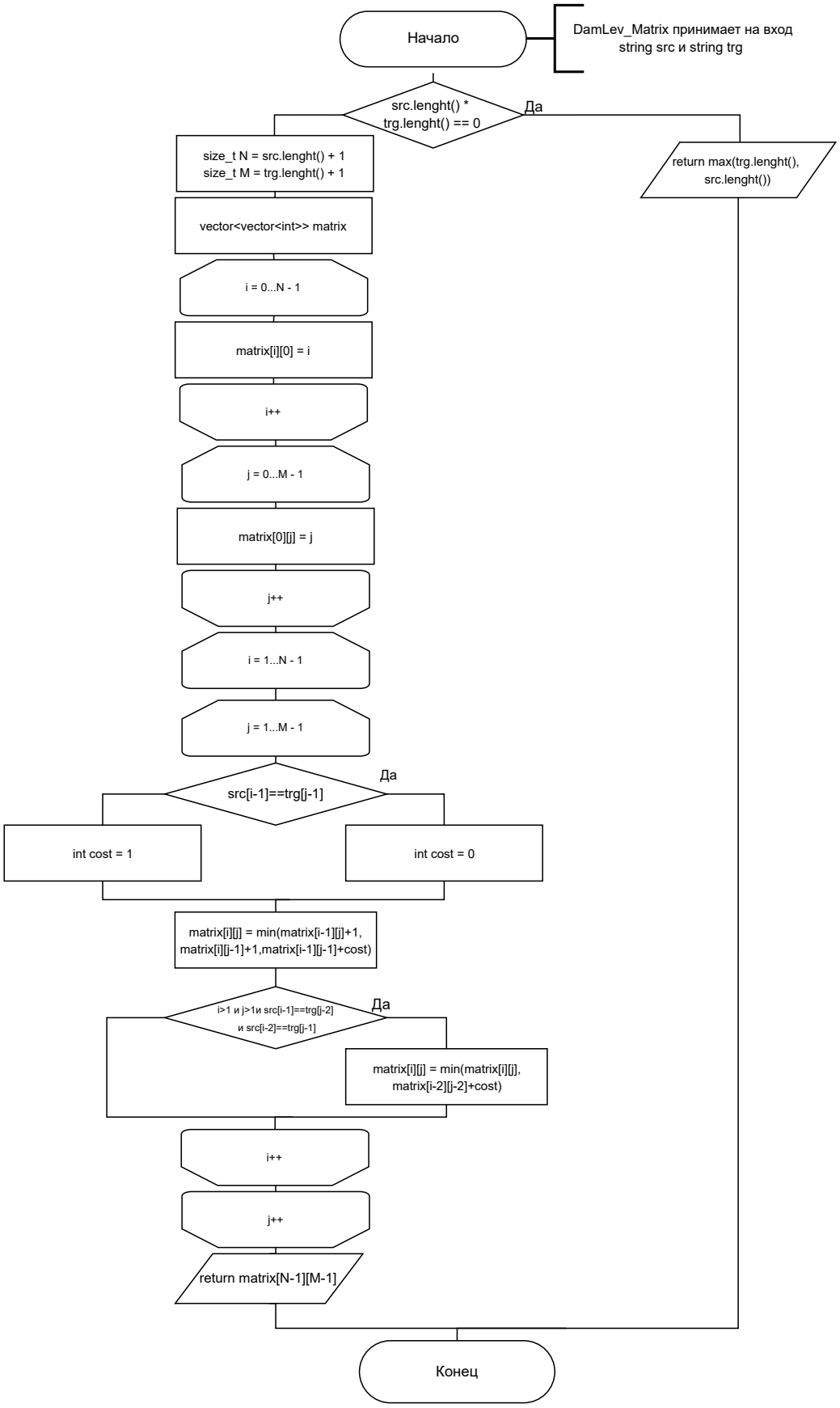

Рисунок 2.1 — Схема нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

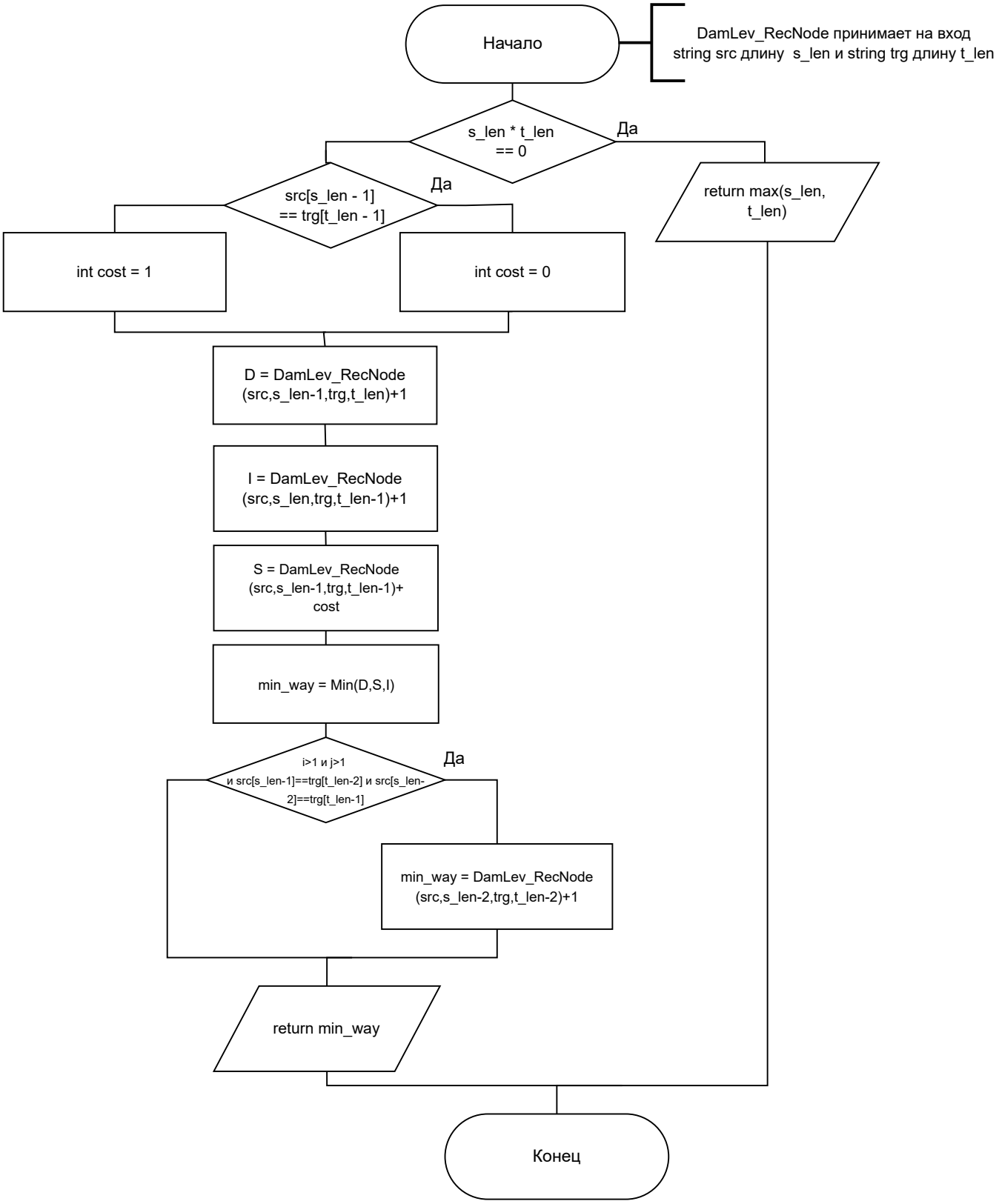

Рисунок 2.2 — Схема рекурсивого алгоритма нахождения расстояния Дамерау-Левенштейна

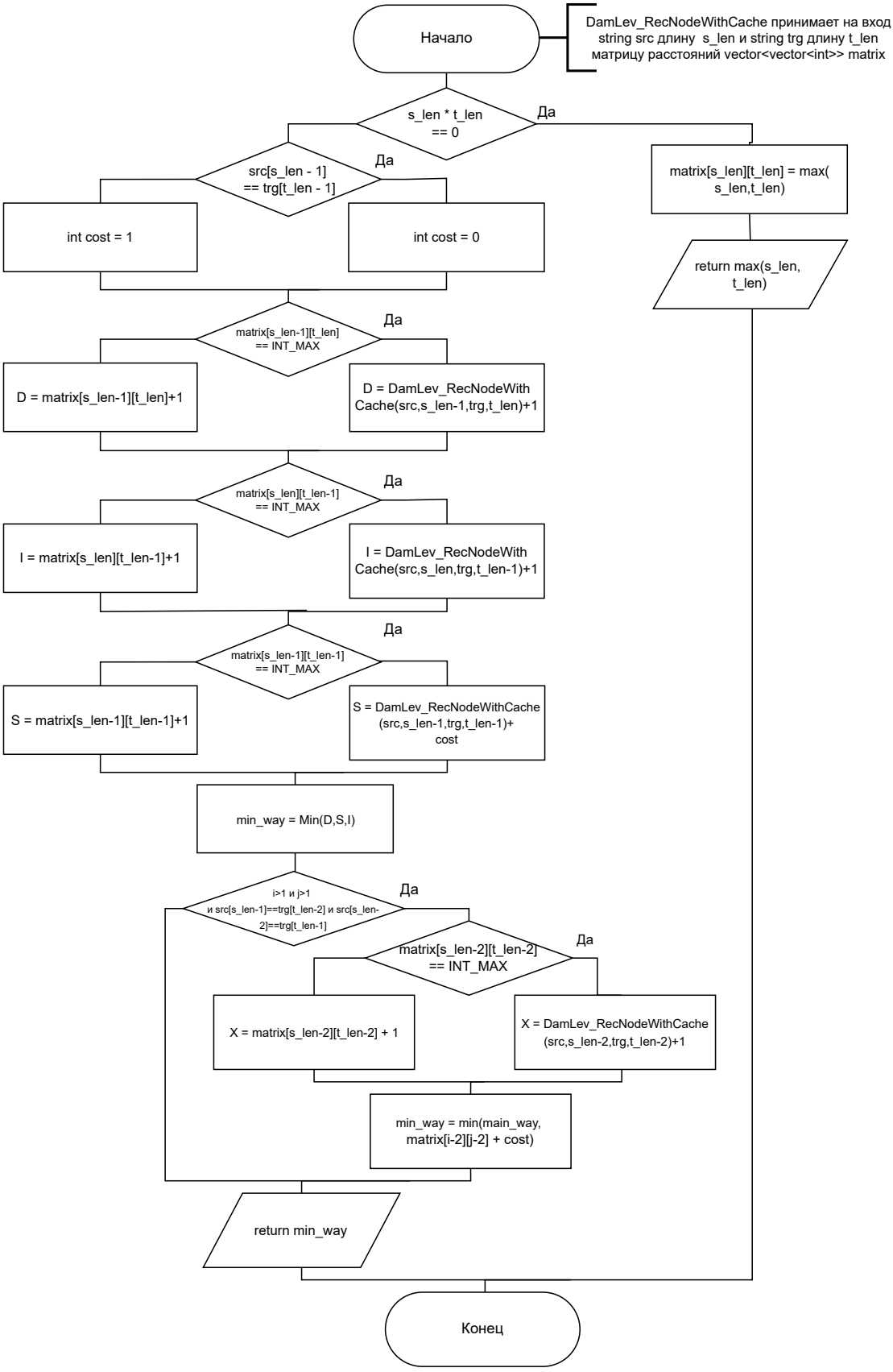

Рисунок 2.3 — Схема рекурсивого алгоритма нахождения расстояния Дамерау-Левенштейна с кэшированием

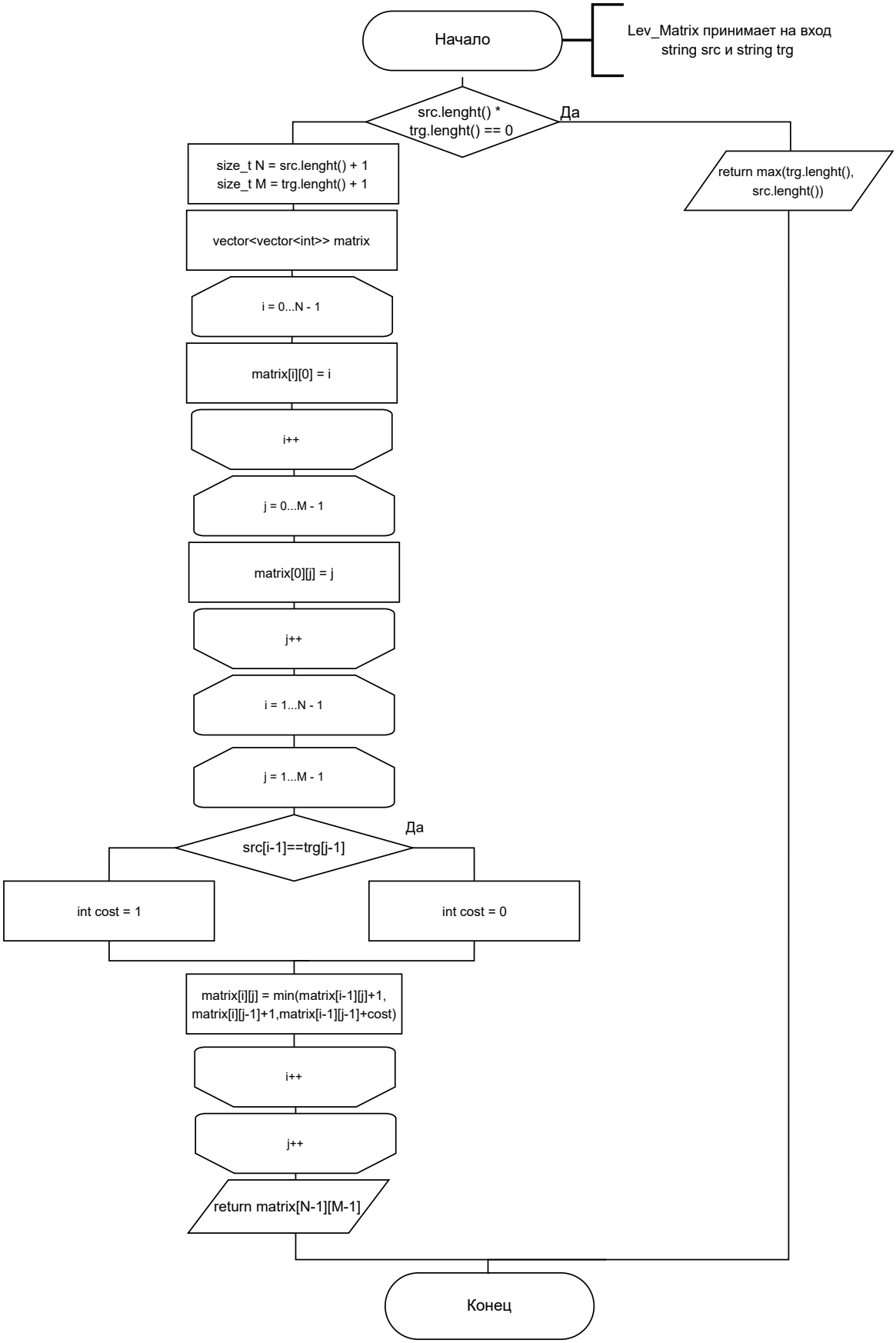

Рисунок 2.4 — Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

2.2 Требования к функциональности ПО

В данной работе требуется обеспечить следующую функциональность.

1) Пользовательский режим:
    а) возможность считать две строки;
    б) вывод расстояний Левенштейна и Дамерау-Левенштейна между строками.
2) Тестовый режим:
    а) возможность замера процессорного времени реализации каждого алгоритма для строк, заданнных внутри программы;
3) Экспериментальный режим:
    а) вывод графиков с процессорным временем работы алгоритмов для варьирующихся строк.

2.3 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.

Вывод

В данном разделе были разработаны схемы используемых алгоритмов и требования к функциональности программы.

3 Технологический раздел

В данном разделе представлены средства, использованные в процессе разработки для реализации задачи, а также листинг кода программы. Кроме того показаны результаты тестирования и анализа затрачиваемой памяти.

3.1 Средства реализации

Для реализации поставленной задачи был использован язык C++, так как имеется большой опыт работы с ним. Для измерения процессорного времени была использована ассемблерная вставка.

3.2 Листинг программы

В приведенных ниже листингах представлены следующие реализации:

1) нерекурсивный алгоритм нахождения расстояния Левенштейна с использованием матрицы (листинг 3.1 3.2);

2) нерекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с использованием матрицы (листинг 3.3 3.4);

3) рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна (листинг 3.5);

4) рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с использованием матрицы (кэшированием) (листинг 3.6 3.7).

Листинг 3.1 — Функция поиска расстояния Левенштейна нерекурсивно

```
1
2  int Lev_Matrix(string src, string trg)
3  {
4      if (src.length() * trg.length() == 0)
5      {
6          return max(trg.length(), src.length());
7      }
8
9      size_t N = src.length() + 1;
10     size_t M = trg.length() + 1;
11
12     vector<vector<int>> matrix (N);
13
14     for (size_t i = 0; i < N; i++)
15         matrix[i].resize(M);
16
17     for (size_t i = 0; i < N; i++)
18         matrix[i][0] = i;
```

Листинг 3.2 — Функция поиска расстояния Левенштейна нерекурсивно

```
    for (size_t j = 0; j < M; j++)
        matrix[0][j] = j;

    for (size_t i = 1; i < N; i++)
    {
        for (size_t j = 1; j < M; j++)
        {
            int cost = src[i - 1] == trg[j - 1] ? 0 : 1;
            matrix[i][j] = CheckMin(matrix[i - 1][j] + 1, matrix[i][j - 1] + 1,
                matrix[i - 1][j - 1] + cost);
        }
    }
    return matrix[N - 1][M - 1];
}
```

Листинг 3.3 — Функция поиска расстояния Дамерау-Левенштейна с использованием матрицы

```

int DamLev_Matrix(string src, string trg)
{
    if (src.length() * trg.length() == 0)
    {
        return max(trg.length(), src.length());
    }

    size_t N = src.length() + 1;
    size_t M = trg.length() + 1;

    vector<vector<int>> matrix (N);

    for (size_t i = 0; i < N; i++)
        matrix[i].resize(M);

    for (size_t i = 0; i < N; i++)
        matrix[i][0] = i;

    for (size_t j = 0; j < M; j++)
        matrix[0][j] = j;

    for (size_t i = 1; i < N; i++)
    {
```

Листинг 3.4 — Функция поиска расстояния Дамерау-Левенштейна с использованием матрицы

```
        for (size_t j = 1; j < M; j++)
        {
            int cost = src[i - 1] == trg[j - 1] ? 0 : 1;
            matrix[i][j] = CheckMin(matrix[i - 1][j] + 1, matrix[i][j - 1] + 1,
                matrix[i - 1][j - 1] + cost);

            if (i > 1 && j > 1 && src[i - 1] == trg[j - 2] && src[i - 2] == trg[j -
                1])
                matrix[i][j] = min(matrix[i][j], matrix[i - 2][j - 2] + cost);
        }
    }
    return matrix[N - 1][M - 1];
}
```

Листинг 3.5 — Функция рекурсивного поиска расстояния Дамерау-Левенштейна

```

static int DamLev_RecNode(string src, size_t s_len, string trg, size_t t_len)
{
    if (s_len * t_len == 0)
        return max(s_len, t_len);

    int cost = src[s_len - 1] == trg[t_len - 1] ? 0 : 1;

    int D = DamLev_RecNode(src, s_len - 1, trg, t_len) + 1;
    int I = DamLev_RecNode(src, s_len, trg, t_len - 1) + 1;
    int S = DamLev_RecNode(src, s_len - 1, trg, t_len - 1) + cost;

    int min_way = CheckMin(D, I, S);

    if (s_len > 1 && t_len > 1 && src[s_len - 1] == trg[t_len - 2] && src[s_len - 2]
       == trg[t_len - 1])
        min_way = min(min_way, DamLev_RecNode(src, s_len - 2, trg, t_len - 2) + 1);
    return min_way;
}

int DamLev_Recursion(string src, string trg)
{
    return DamLev_RecNode(src, src.length(), trg, trg.length());
}
```

Листинг 3.6 — Функция рекурсивного поиска расстояния Дамерау-Левенштейна с кэшированием

```
1
2  static int DamLev_RecNodeWithCache(string src, size_t s_len, string trg, size_t
       t_len, vector<vector<int>> &matrix)
3  {
4      if (s_len * t_len == 0)
5      {
6          int res = max(s_len, t_len);
7          matrix[s_len][t_len] = res;
8          return max(s_len, t_len);
9      }
10
11     int cost = src[s_len - 1] == trg[t_len - 1] ? 0 : 1;
12
13     int D = INT_MAX;
14     if (matrix[s_len - 1][t_len] == INT_MAX)
15     {
16         D = (DamLev_RecNodeWithCache(src, s_len - 1, trg, t_len, matrix) + 1);
17     }
18     else
19         D = matrix[s_len - 1][t_len] + 1;
20
21     int I = INT_MAX;
22     if (matrix[s_len][t_len - 1] == INT_MAX)
23     {
24         I = (DamLev_RecNodeWithCache(src, s_len, trg, t_len - 1, matrix) + 1);
25     }
26     else
27         I = matrix[s_len][t_len - 1] + 1;
28
29     int S = INT_MAX;
30     if (matrix[s_len - 1][t_len - 1] == INT_MAX)
31     {
32         S = (DamLev_RecNodeWithCache(src, s_len - 1, trg, t_len - 1, matrix) + cost);
33     }
34     else
35         S = matrix[s_len - 1][t_len - 1] + cost;
36
37     int min_way = CheckMin(D, I, S);
```

Листинг 3.7 — Функция рекурсивного поиска расстояния Дамерау-Левенштейна с кэшированием

```
    if (s_len > 1 && t_len > 1 && src[s_len - 1] == trg[t_len - 2] && src[s_len - 2]
       == trg[t_len - 1])
    {
        int X = INT_MAX;
        if (matrix[s_len - 2][t_len - 2] == INT_MAX)
        {
            X = DamLev_RecNodeWithCache(src, s_len - 2, trg, t_len - 2, matrix) + 1;
        }
        else
            X = matrix[s_len - 2][t_len - 2] + 1;
        min_way = min(min_way, X);
    }
    if (matrix[s_len][t_len] == INT_MAX || min_way < matrix[s_len][t_len])
        matrix[s_len][t_len] = min_way;
    return min_way;
}

int DamLev_RecursionWithCache(string src, string trg)
{
    size_t N = src.length() + 1;
    size_t M = trg.length() + 1;

    vector<vector<int>> matrix (N);

    for (size_t i = 0; i < N; i++)
    {
        matrix[i].resize(M);
        for (size_t j = 0; j < M; j++)
            matrix[i][j] = INT_MAX;
    }

    int res = DamLev_RecNodeWithCache(src, src.length(), trg, trg.length(), matrix);
    return res;
}
```

3.3 Тестирование

В таблице 3.1 отображён возможный набор тестов для тестирования методом чёрного ящика, результаты которого, представленные на рисунке 3.1, подтверждают прохождение программы перечисленных тестов.

Таблица 3.1 — Тесты проверки корректности программы

№	строка 1	строка 2	Ожидаемый результат (Л, Д-Л)	Фактический результат
1	"скат"	"скат"	0, 0	0, 0
2	"кот"	"скат"	2, 2	2, 2
3	"скат"	"скат"	0, 0	0, 0
4	"скат"	"сакт"	2, 1	2, 1

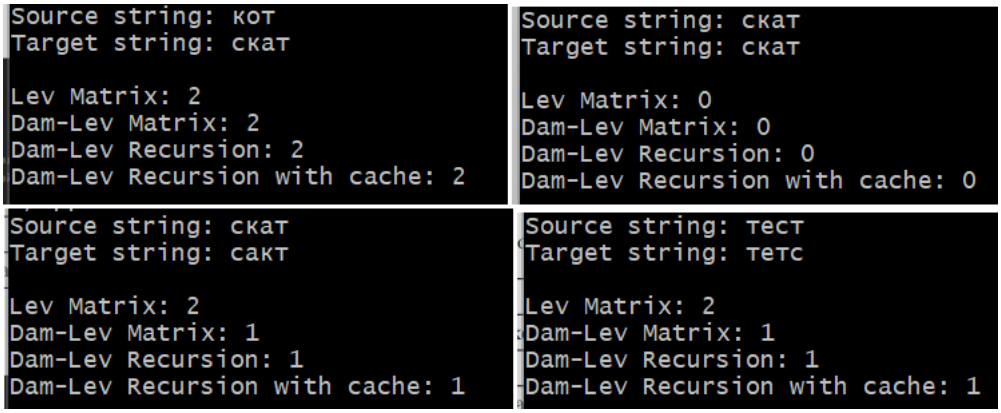

Рисунок 3.1 — Результаты тестирования

3.4 Сравнительный анализ потребляемой памяти

Аналитически посчитаем затрачиваемую память на строках s_1, s_2 длиной n и m соответственно. Так как, с точки зрения памяти алгоритмы Левенштейна и Дамерау-Левенштейна не отличаются, достаточно рассмотреть лишь разные реализации данных алгоритмов.

Использование памяти для матричного алгоритма теоритически определяется формулой (3.1):

$$V = 2(n+1)sizeof(int) + 2sizeof(int) + 2sizeof(size\_t) + sizeof(char)(n+m) \tag{3.1}$$

Где:

1) $2(n+1)sizeof(int)$ - память под две строки (замена матрицы в моей реализации);
2) $2sizeof(int)$ - память под размеры строк;
3) $2sizeof(size\_t)$ - память под итераторы;
4) $sizeof(char)(n+m)$ - память под сами строки как аргументы функции.

Использование памяти для рекурсивного алгоритма зависит от максимальной глубины стека вызовов. Так как эта величина равна сумме длин входящих строк, то теоритически определяется формулой (3.2):

$$V = sizeof(char)(n+m) + (n+m)(2sizeof(char*) + 3sizeof(int)) \tag{3.2}$$

Где:

1) $sizeof(char)(n+m)$ - память под аргументы функции;

2) $(n+m)(2sizeof(char*)+3sizeof(int))$ - выделенные рекурсией ресурсы умноженные на максимальную глубину стека вызовов.

Вывод

Были разработаны и протестированы спроектированные алгоритмы, приведены расчеты сравнительного анализа потребляемой памяти, а также указаны средства реализации поставленной задачи.

4 Экспериментальный раздел

В данном разделе будут проведены эксперименты для проведения сравнительного анализа алгоритмов по затрачиваемому процессорному времени.

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

В рамках данного проекта были проведёны следующие эксперименты:

1) сравнение нерекурсивных алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна на строках длиной от 5 до 10 с шагом 1 (график 4.1);

2) сравнение рекурсивного алгоритма поиска расстояния Дамуерау-Левенштейна и рекурсивного алгоритма Дамерау-Левенштейна с кэшированием на строках длиной от 0 до 12 с шагом 1 (график 4.2).

Тестирование проводилось на компьютере с процессором Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz под управлением Windows 11 с 8 Гб оперативной памяти.

Ниже представлены результаты замеров (таблица 4.1) для всех 4 алгоритмов [1].

Таблица 4.1 — Замеры времени (в секундах) для строк различной длины

Длина строк	Л. матричный	Д.Л. матричный	Д.Л. рекурсивный	Д.Л. с кэшем
5	2.5e-0.5	0.002	5.3e-05	0.003
6	3.4e-0.5	0.009	6.7e-05	0.015
7	4.3e-0.5	0.031	8.5e-05	0.046
8	5.3e-0.5	0.203	0.000101	0.203
9	7e-0.5-	1.093	0.00013	1.265
10	9.2-0.5	5.937	0.00015	6.625
20	0.0003	-	0.0006	-
40	0.001	-	0.002	-
60	0.002	-	0.004	-
80	0.004	-	0.007	-
100	0.007	-	0.012	-
120	0.0108	-	0.018	-
140	0.015	-	0.024	-
160	0.019	-	0.037	-

[1] Прочерк в таблице означает, что для заданных значений тестирование не проводилось

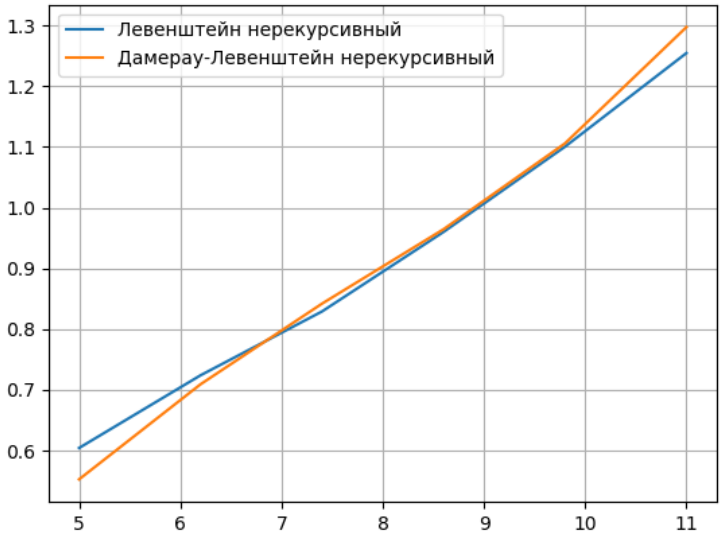

Рисунок 4.1 — Зависимость времени работы алгоритмов от длин строк

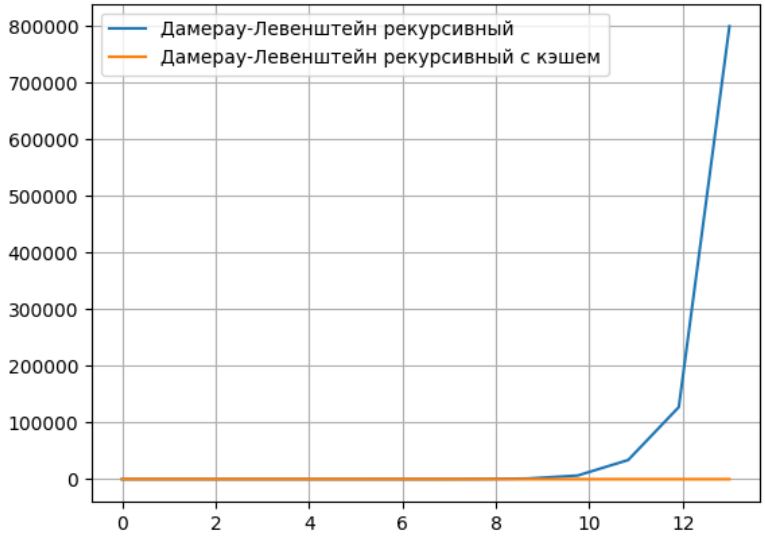

Рисунок 4.2 — Зависимость времени работы алгоритмов от длин строк

4.2 Вывод

В данном разделе были поставлены эксперименты по замеру времени выполнения каждого из алгоритмов. Рекурсивные алгоритмы по нахождению редакционного растояния работают дольше чем матричные на несколько порядков. Время работы таких реализаций увеличивается в геометрической прогрессии. С этой проблемой помогает справляться кэширование, но такой подход увеличивает затраты в памяти.

Следовательно, матричная реализация более применима в реальных проектах.

Заключение

В ходе работы были изучены и реализованы алгоритмы нахождения расстояния Левенштейна (не рекурсивный с заполнением матрицы, рекурсивный без заполнения матрицы, рекурсивный с заполнением матрицы) и Дамерау-Левенштейна (не рекурсивный с заполнением матрицы).

Цель работы была достигнута.

Выполненные задачи:

1) выбраны инструменты для замера процессорного времени выполнения реализации алгоритмов;

2) изучены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна;

3) реализованы:

   а) нерекурсивный метод поиска расстояния Левенштейна;

   б) нерекурсивный метод поиска расстояния Дамерау-Левенштейна;

   в) рекурсивный метод поиска расстояния Дамерау-Левенштейна;

   г) рекурсивный с кэшированием метод поиска расстояния Дамерау-Левенштейна;

4) замерены процессорное время и потребленную память для всех реализованных алгоритмов;

5) проведены анализ работы программы по времени и по памяти, выяснить влияющие на них характеристики.

Список источников

1) Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений сиволов. - М.: Доклады АН СССР, 1965. Т.163. С.845-848.

2) Нечеткий поиск в словаре с универсальным автоматом Левенштейна. Режим доступа: https://habr.com/ru/post/275937/ (дата обращения: 10.09.2021).

3) Нечеткий поиск в тексте и словаре. Режим доступа: https://habr.com/ru/post/114997/ (дата обращения: 11.09.2021).

4) Functools — Higher-order functions and operations on callable objects. Режим доступа: https://docs.python.org/3/library/functools.html (дата обращения: 11.09.2021).

5) Time — Time access and conversions Режим доступа: https://docs.python.org/3/library/time.html (дата обращения: 11.09.2021).