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 курсу «Анализ Алгоритмов»

на тему: «Динамическое программирование»

Студент группы ИУ7-54Б _____ (Подпись, дата) Спирин М. П. (Фамилия И.О.)

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

Преподаватель _____ (Подпись, дата) Строганов Ю. В.. (Фамилия И.О.)

Москва — 2023 г.

Содержание

Введение

В данной лабораторной работе будет рассмотрено расстояние Левенштейна. Данное расстояние показывает минимальное количество операций (вставка, удаление, замена), которое необходимо для преобразования одной строки в другую. Это расстояние помогает определить схожесть двух строк.

Впервые задачу поставил в 1965 году советский математик Владимир Левенштейн при изучении последовательностей 0 – 1, впоследствии более общую задачу для произвольного алфавита связали с его именем.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для решения следующих задач:

— исправление ошибок в слове(в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);

— сравнение текстовых файлов утилитой diff;

— для сравнения геномов, хромосом и белков в биоинформатике.

Метод динамического программирования был предложен и обоснован Р. Беллманом в начале 1960-х годов [1]. Первоначально метод создавался в целях существенного сокращения перебора для решения целого ряда задач экономического характера, формулируемых в терминах задач целочисленного программирования. Однако Р. Беллман и Р. Дрейфус показали, что он применим к достаточно широкому кругу задач, в том числе к задачам поиска расстояния Левенштейна и Дамерау-Левенштейна.

Целью данной лабораторной работы является изучение алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

Для поставленной цели необходимо выполнить следующие задачи.

1) Описать алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна.

2) Создать программное обеспечение, реализующее следующие алгоритмы:

- нерекурсивный алгоритм поиска расстояния Левенштейна;
- нерекурсивный алгоритм поиска расстояния Дамерау-Левенштейна;
- рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна;
- рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кэшированием.

3) Выбрать инструменты для замера процессорного времени необходимого для выполнения реализаций алгоритмов.

4) Провести анализ затрат реализаций алгоритмов по времени и по памяти, определить влияющие на них факторы.

1 Аналитическая часть

1.1 Расстояние Левенштейна

Расстояние Левенштейна (редакционное расстояние, дистанция редактирования) — метрика, измеряющая разность между двумя последовательностями символов. Расстояние Левенштейна — это минимальное количество редакторских операций вставки (I, от англ. insert), замены (R, от англ. replace) и удаления (D, от англ. delete), необходимых для преобразования одной строки в другую [2]. Стоимость зависит от вида операции:

1) $w(a, b)$ — цена замены символа a на b;

2) $w(\lambda, b)$ — цена вставки символа b;

3) $w(a, \lambda)$ — цена удаления символа a.

Будем считать стоимость каждой вышеизложенной операции равной 1:

— $w(a, b) = 1$, $a \neq b$, в противном случае замена не происходит;

— $w(\lambda, b) = 1$;

— $w(a, \lambda) = 1$.

Введем понятие совпадения символов — M (от англ. match). Его стоимость будет равна 0, то есть $w(a, a) = 0$.

Введем в рассмотрение функцию $D(i, j)$, значением которой является редакционное расстояние между подстроками $S_1[1...i]$ и $S_2[1...j]$.

Расстояние Левенштейна между двумя строками S_1 и S_2 длиной M

и N соответственно рассчитывается по рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & \text{i} = 0, \text{j} = 0 \\ i, & \text{j} = 0, \text{i} > 0 \\ j, & \text{i} = 0, \text{j} > 0 \\ min\{D(i,j-1)+1, & \text{i} > 0, \text{j} > 0 \\ \quad D(i-1,j)+1, & \\ \quad D(i-1,j-1)+m(S_1[i],S_2[j])\}, & \end{cases} \tag{1.1}$$

где сравнение символов строк S_1 и S_2 рассчитывается по формуле:

$$m(a,b) = \begin{cases} 0 & \text{если a} = \text{b}, \\ 1 & \text{иначе.} \end{cases} \tag{1.2}$$

1.1.1 Нерекурсивный алгоритм нахождения расстояния Левенштейна

Рекурсивная реализация алгоритма Левенштейна малоэффективна по времени при больших M и N, так как требуется считать множество промежуточных значений. Для оптимизации можно использовать итерационную реализацию заполнения матрицы промежуточными значениями $D(i,j)$.

В качестве структуры данных для хранения промежуточных значений можно использовать матрицу, имеющую размеры:

$$(N+1) \times (M+1). \tag{1.3}$$

Значения в ячейке $[i,j]$ равно значению $D(S1[1...i], S2[1...j])$. Первый элемент матрицы заполнен нулем. Всю таблицу заполнять в соответствии с формулой (1.1).

Однако матричный алгоритм является малоэффективным по памяти по сравнению с рекурсивным при больших M и N, т.к. множество промежуточных значений $D(i,j)$ хранится в памяти после их использования. Для

оптимизации по памяти рекурсивного алгоритма нахождения расстояния Левенштейна можно использовать кеш, т.е. пару строк, содержащую значения $D(i, j)$, вычисленные в предыдущей итерации, алгоритма и значения $D(i, j)$, вычисляемые в текущей итерации.

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна, названное в честь ученых Фредерика Дамерау и Владимир Левенштейна, — это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна: к трем базовым операциям добавляется операция транспозиции T (от англ. transposition).

Расстояние Дамерау-Левенштейна может быть вычислено по рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & \text{i} = 0, \text{j} = 0, \\ i, & \text{j} = 0, \text{i} > 0, \\ j, & \text{i} = 0, \text{j} > 0, \\ & \text{если i} > 1, \text{j} > 1, \\ min\{D(i,j-1)+1, & S_1[i] = S_2[j-1], \\ \quad D(i-1,j)+1, & S_1[i-1] = S_2[j], \\ \quad D(i-1,j-1)+m(S_1[i],S_2[j]), & \\ \quad D(i-2,j-2)+1\}, & \\ min\{D(i,j-1)+1, & \text{иначе.} \\ \quad D(i-1,j)+1, & \\ \quad D(i-1,j-1)+m(S_1[i],S_2[j])\}, & \end{cases} \quad (1.4)$$

1.2.1 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Рекурсивный алгоритм реализует формулу (1.4), функция D составлена таким образом, что истинны следующее положения:

1) для передачи из пустой строки в пустую требуется ноль операций;

2) для перевода из пустой строки в строку a требуется $|a|$ операций;

3) для перевода из строки a в пустую строку требуется $|a|$ операций;

4) для перевода из строки a в строку b требуется выполнить последовательно некоторое количество операций удаления, вставки, замены, транспозиции в некоторой последовательности.

Если полагать, что *a', b'* — строки a и b без последнего символа соответственно, а *a", b"*— строки a и b без двух последних символов, то цена преобразования из строки a в b выражается из элементов, представленных ниже:

— сумма цены преобразования строки a' в b и цены проведения операции удаления, которая необходима для преобразования a' в a;

— сумма цены преобразования строки a в b' и цены проведения операции вставки, которая необходима для преобразования b' в b;

— сумма цены преобразования из a' в b' и операции замены, предполагая, что a и b оканчиваются на разные символы;

— сумма цены преобразования из a'' в b'' и операции перестановки, предполагая, что длины a'' и b'' больше 1 и последние два символа a'', поменянные местами, совпадут с двумя последними символами b'';

— цена преобразования из a' в b', предполагая, что a и b оканчиваются на один и тот же символ.

Минимальной стоимостью преобразования будет минимальное значение приведенных вариантов.

1.2.2 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с кешированием

Рекурсивная реализация алгоритма Дамерау-Левенштейна малоэффективна по времени при больших M и N из-за необходимости совершать повторные вычисления значений расстояний между подстроками. Для оптимизации алгоритма нахождения расстояния Левенштейна можно использовать матрицу в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой рекурсивное заполнение матрицы $A_{|a|,|b|}$ промежуточными значениями $D(i,j)$, такое хранение промежуточных данных можно назвать кэшем для рекурсивного алгоритма.

1.2.3 Нерекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Рекурсивная реализация алгоритма Левенштейна с кешированием малоэффективна по времени при больших M и N. Для оптимизации можно использовать итерационную реализацию заполнения матрицы промежуточными значениями $D(i,j)$.

В качестве структуры данных для хранения промежуточных значений можно использовать матрицу, имеющую размеры:

$$(N+1) \times (M+1), \tag{1.5}$$

Значение в ячейке $[i,j]$ равно значению $D(S1[1...i], S2[1...j])$. Первый элемент заполнен нулем. Всю таблицу заполняем в соответствии с формулой (1.4).

Вывод

В данном разделе были рассмотрены алгоритмы динамического программирования — алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна, формулы которых задаются рекуррентно, а следовательно, данные алгоритмы могут быть реализованы рекурсивно и итеративно. На вход алгоритмам поступают две строки, которые могут содержать как русские, так и английские буквы, также будет предусмотрен ввод пустых строк.

2 Конструкторская часть

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

2.1 Требования к программному обеспечению

К программе предъявлен ряд функциональных требований: входные данные — две строки, выходные данные — результат работы всех алгоритмов поиска расстояний, целое число.

Кроме того, программа должна соответствовать следующим требованиям:

— наличие интерфейса для выбора действий;

— должна обрабатывать строки;

— возможность обработки строк, включающих буквы как на латинице, так и на кириллице;

— наличие функциональности замера процессорного времени работы реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левен- штейна.

2.2 Разработка алгоритмов

На вход алгоритмов подаются строки S_1 и S_2.

На рисунках 2.1 – 2.2 представлены схемы алгоритма поиска расстояния Левенштейна. На рисунках 2.3 – 2.7 представлены схемы алгоритмов поиска расстояния Дамерау-Левенштейна.

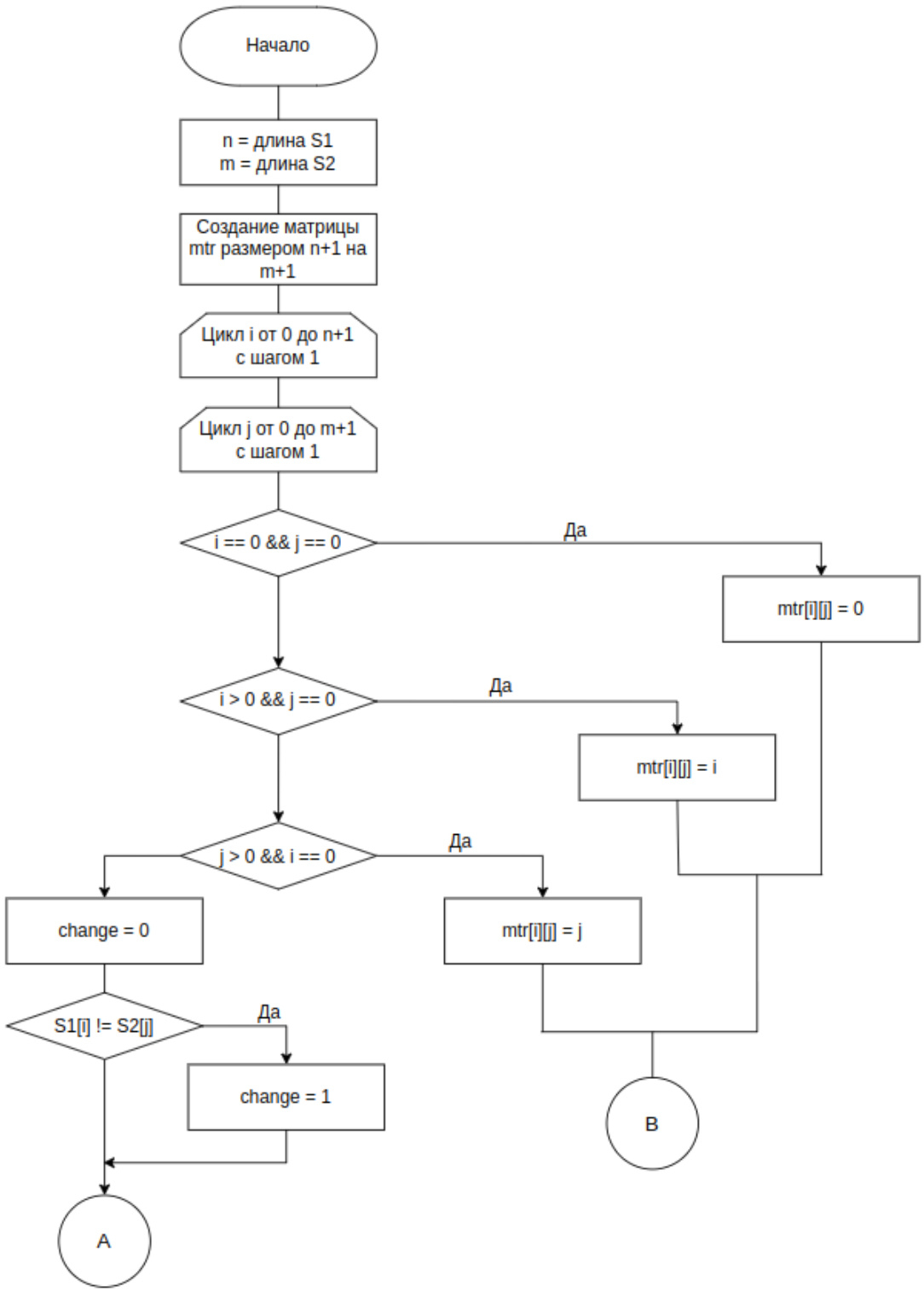

Рисунок 2.1 – Схема 1 нерекурсивного алгоритма нахождения расстояния Левенштейна

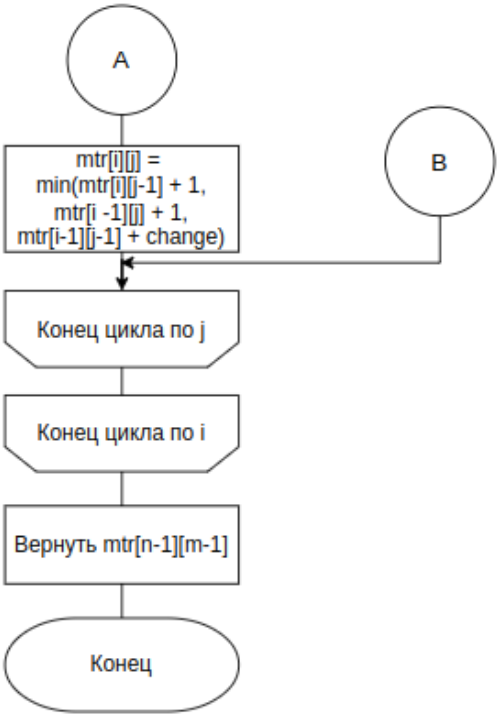

Рисунок 2.2 – Схема 2 нерекурсивного алгоритма нахождения расстояния Левенштейна

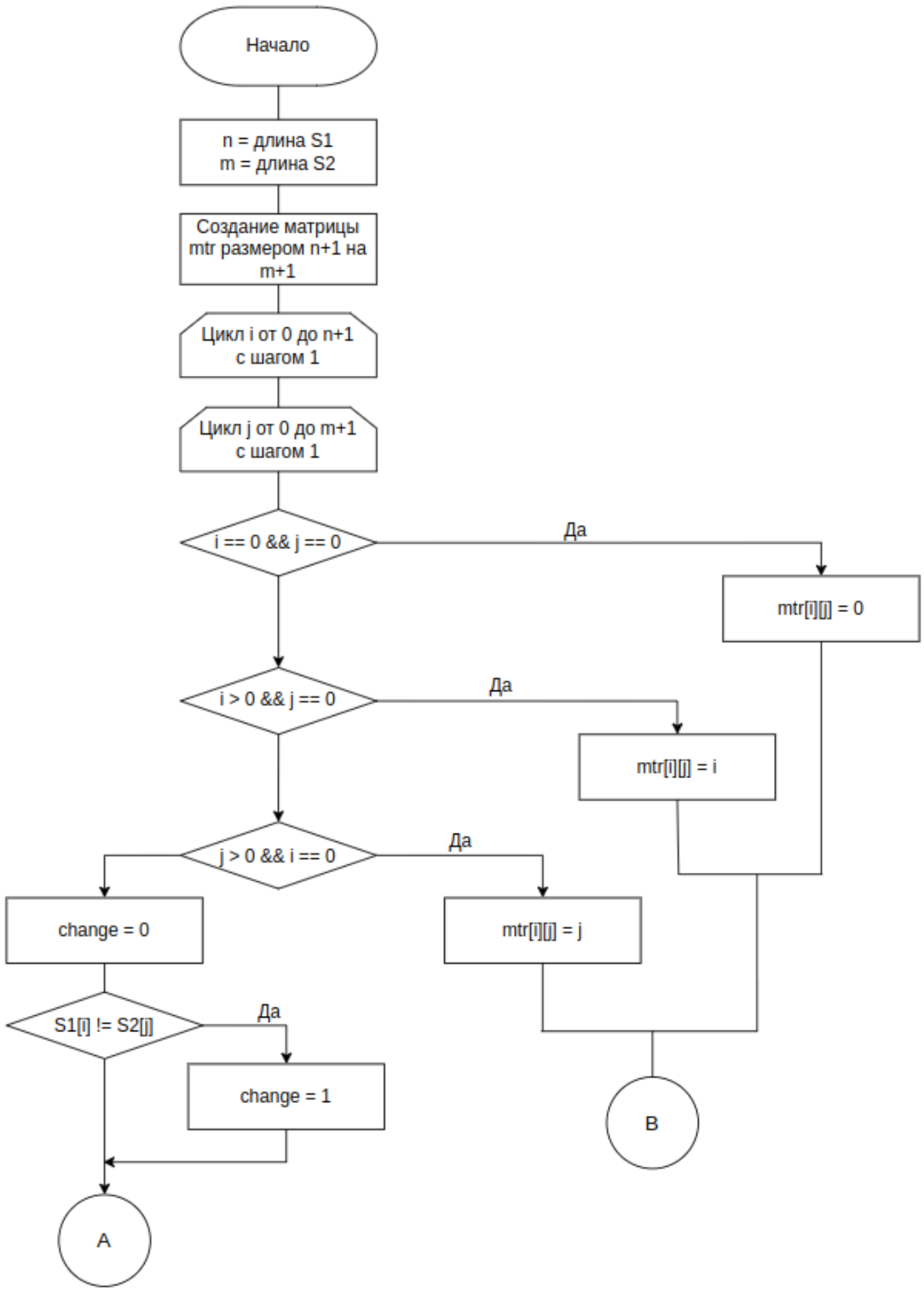

Рисунок 2.3 – Схема 1 нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

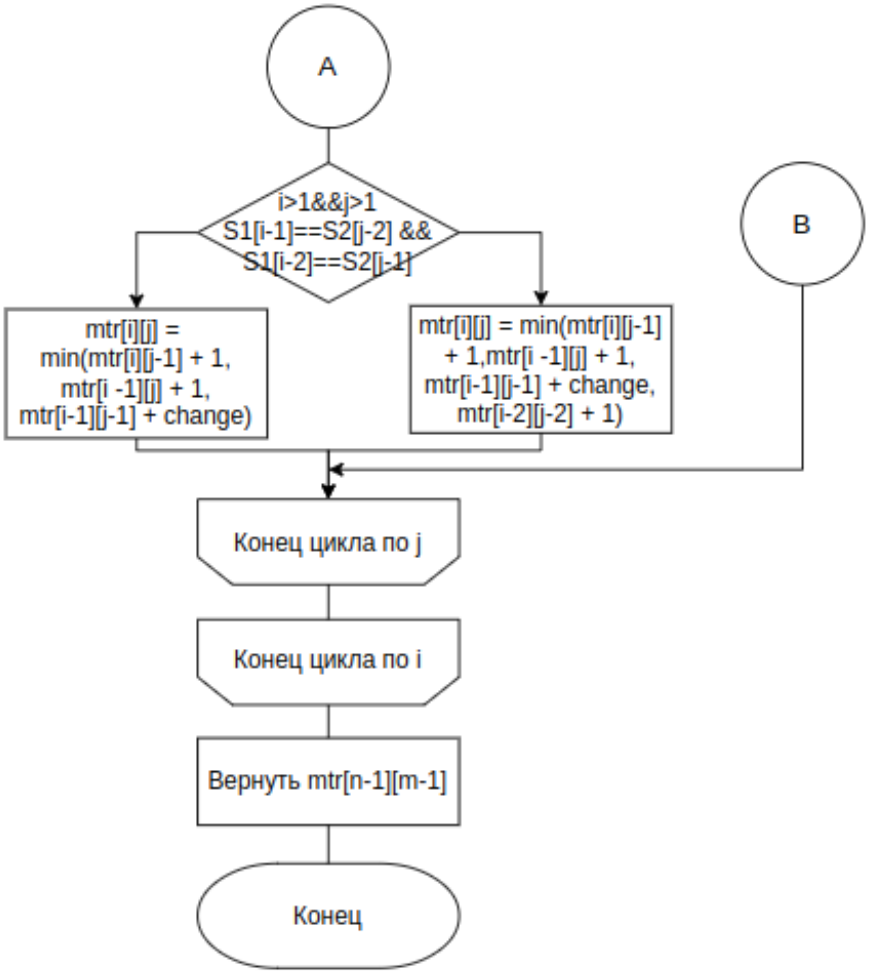

Рисунок 2.4 – Схема 2 нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

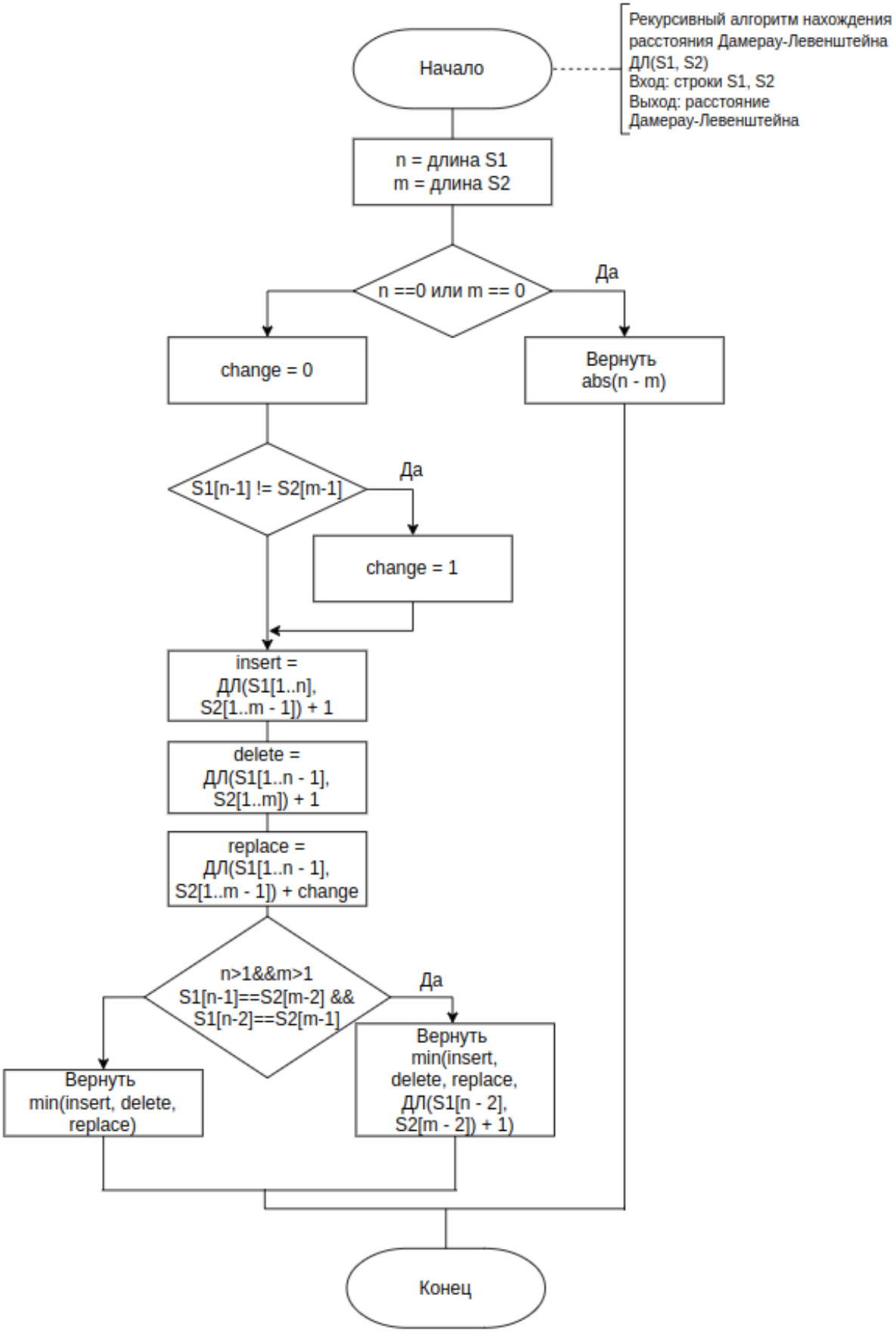

Рисунок 2.5 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

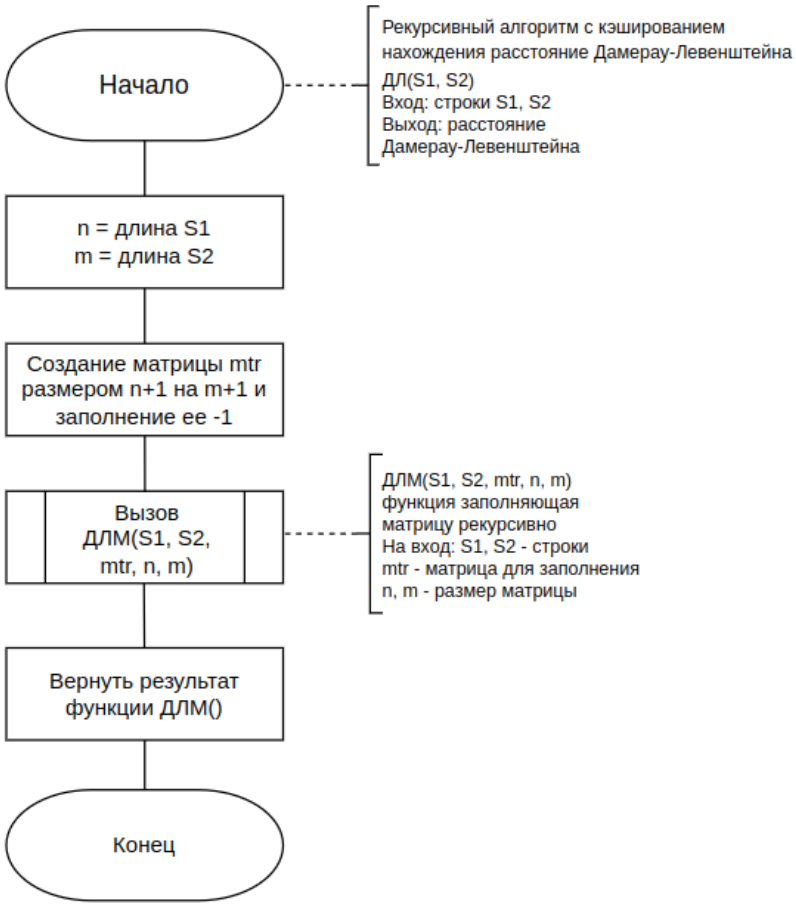

Рисунок 2.6 – Схема 1 рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кешированием

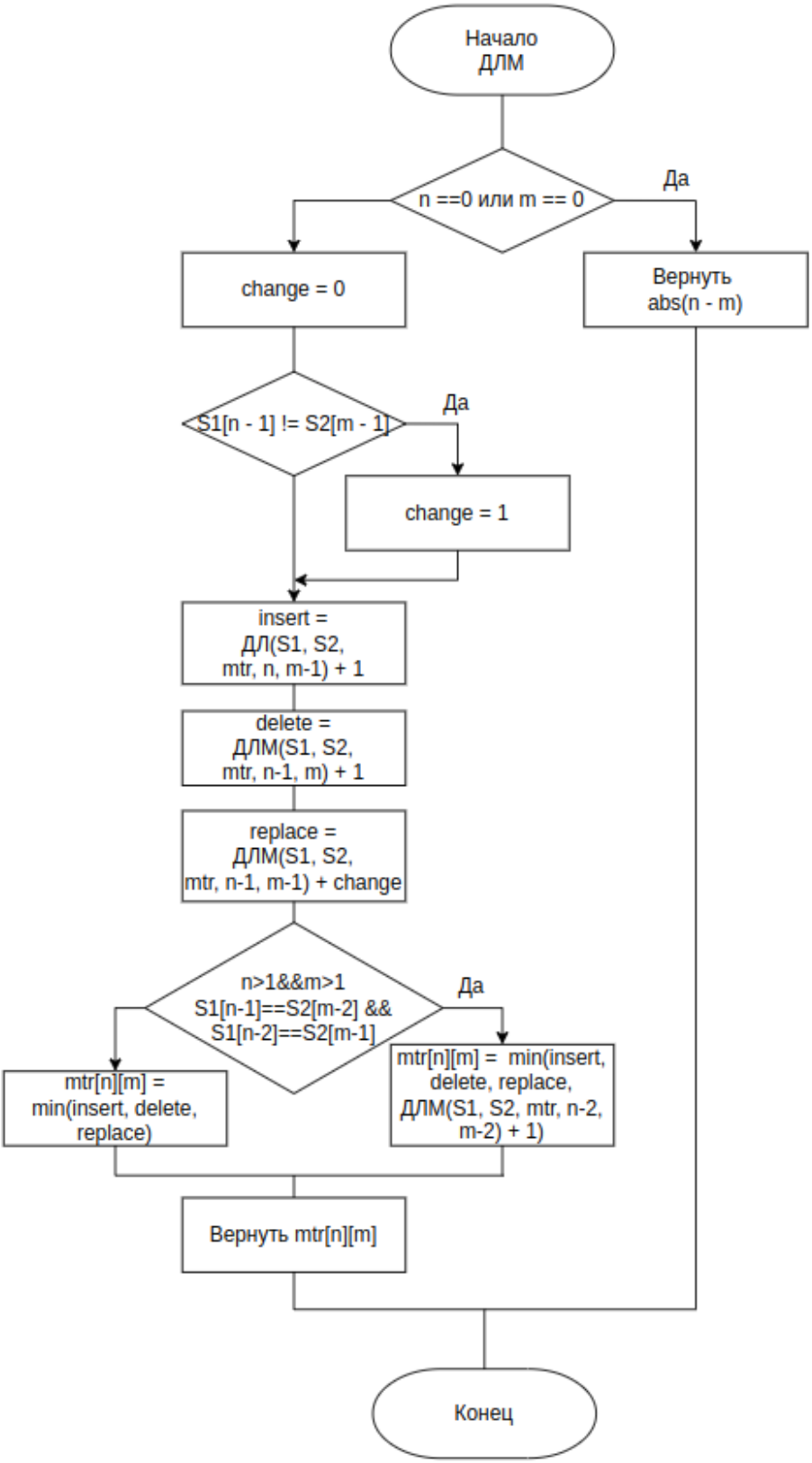

Рисунок 2.7 – Схема 2 рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кешированием

2.3 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

— строка — массив типа *wchar* размером длины строки;

— матрица — двумерный массив значений типа *int*.

Вывод

В данном разделе на основе теоретических данных были построены схемы требуемых алгоритмов, выбраны используемые типы данных.

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации, листинг кода и функциональные тесты.

3.1 Средства реализации

Для реализации данной лабораторной работы был выбран язык $C++$ [3]. Данный выбор был сделан, так как у языка есть встроенный модуль *ctime* для измерения процессорного времени и тип данных, позволяющий хранить как кириллические символы, так и латинские — $std :: wstring$, что необходимо согласно третьему требованию из п.2.1 и соответствуют выдвинутым техническим требованиям.

Время работы было замерено с помощью функции *clock()* из библиотеки *ctime* [4].

3.2 Сведения о модулях программы

Данная программа разбита на следующие модули:

— `main.cpp` — файл, содержащий точку входа в программу, из которой происходит вызов алгоритмов по разработанному интерфейсу;

— `levenshtein.cpp` — файл содержит функции поиска расстояния Левенштейна и Дамерау-Левенштейна;

— `memory.cpp` — файл содержит функции динамического выделения и очищения памяти для матрицы;

— `time.cpp` — файл содержит функции, замеряющее процессорное время алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна;

3.3 Реализация алгоритмов

В листингах 3.1 – 3.4 приведены реализации алгоритмов поиска расстояния Левенштейна (только нерекурсивный алгоритм) и Дамерау-Левенштейна (нерекурсивный, рекурсивный и рекурсивный с кэшированием). В листинге 3.5 приведены реализации алгоритмов выделения памяти под матрицу и очищения памяти из под нее.

Листинг 3.1 – Функция нахождения расстояния Левенштейна с использованием матрицы

```
1  int levenshtein_matrix(std::wstring &first_word, std::wstring
       &second_word)
2  {
3      int first_length = first_word.length();
4      int second_length = second_word.length();
5      int **matrix = alloc_matrix(first_length + 1, second_length
         + 1);
6      for (int i = 0; i < first_length + 1; i++)
7      {
8          for (int j = 0; j < second_length + 1; j++)
9          {
10             if (i == 0 || j == 0)
11             {
12                 matrix[i][j] = abs(i - j);
13             }
14             else
15             {
16                 int equal = 1;
17                 if (first_word[i - 1] == second_word[j - 1])
18                     equal = 0;
19                 matrix[i][j] = std::min({matrix[i - 1][j] + 1,
20                                          matrix[i][j - 1] +
                                             1,
21                                          matrix[i - 1][j -
                                             1] + equal});
22             }
23         }
24     }
25     int res = matrix[first_length][second_length];
26     free_matrix(matrix, first_length + 1);
27     return res;
28 }
```

Листинг 3.2 – Функция нахождения расстояния Дамерау-Левенштейна с использованием матрицы

```
1
2  int damerau_levenshtein_matrix(std::wstring &first_word,
3      std::wstring &second_word)
4  {
5      int first_length = first_word.length();
6      int second_length = second_word.length();
7      int **matrix = alloc_matrix(first_length + 1, second_length
         + 1);
8      for (int i = 0; i < first_length + 1; i++)
9      {
10         for (int j = 0; j < second_length + 1; j++)
11         {
12             if (i == 0 || j == 0)
13             {
14                 matrix[i][j] = abs(i - j);
15             }
16             else
17             {
18                 int equal = 1;
19                 if (first_word[i - 1] == second_word[j - 1])
20                 equal = 0;
21                 matrix[i][j] = std::min({matrix[i - 1][j] + 1,
22                             matrix[i][j - 1] + 1,
23                             matrix[i - 1][j - 1] + equal});
24                 if (i > 1 && j > 1 && first_word[i - 2] ==
                      second_word[j - 1] &&
25                     first_word[i - 1] == second_word[j - 2])
26                 {
27                     matrix[i][j] = std::min(matrix[i][j],
28                                 matrix[i - 2][j - 2] + 1);
29                 }
30             }
31         }
32     }
33     int res = matrix[first_length][second_length];
34     free_matrix(matrix, first_length + 1);
35     return res;
36 }
```

Листинг 3.3 – Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
1  int damerau_lev_rec(std::wstring &first_word, std::wstring
       &second_word,
2      int n, int m)
3  {
4      if (n == 0 || m == 0)
5          return abs(n - m);
6
7      int equal = 1;
8      if (first_word[n - 1] == second_word[m - 1])
9          equal = 0;
10
11     int res = std::min({damerau_lev_rec(first_word,
           second_word, n, m - 1) + 1,
12                         damerau_lev_rec(first_word,
                               second_word, n - 1, m) + 1,
13                         damerau_lev_rec(first_word, second_word,
14                         n - 1, m - 1) + equal});
15
16     if (n > 1 && m > 1 && first_word[n - 1] == second_word[m -
           2] &&
17         first_word[n - 2] == second_word[m - 1])
18     {
19         res = std::min(res, damerau_lev_rec(first_word,
               second_word,
20                                             n - 2, m - 2) + 1);
21     }
22
23     return res;
24 }
25
26 int damerau_lev_recursion(std::wstring &first_word,
       std::wstring &second_word)
27 {
28     return damerau_lev_rec(first_word, second_word,
29                            first_word.length(),
                                  second_word.length());
30 }
```

Листинг 3.4 – Функция нахождения расстояния Дамерау-Левенштейна рекурсивно с кешированием

```
1
2 int damerau_lev_rec_hash(std::wstring &first_word, std::wstring
     &second_word,
3                          int **matrix, int n, int m)
4 {
5     if (matrix[n][m] != -1) {
6         return matrix[n][m];
7     }
8
9     if (n == 0 || m == 0) {
10        matrix[n][m] = std::max(m, n);
11        return matrix[n][m];
12    }
13
14    int equal = 1;
15    if (first_word[n - 1] == second_word[m - 1])
16        equal = 0;
17
18    matrix[n][m] = std::min({damerau_lev_rec_hash(first_word,
        second_word, matrix, n, m - 1) + 1,
19                             damerau_lev_rec_hash(first_word,
                                 second_word, matrix, n - 1, m) + 1,
20                             damerau_lev_rec_hash(first_word,
                                 second_word, matrix,
21                                                  n - 1, m - 1) + equal});
22
23    if (n > 1 && m > 1 && first_word[n - 1] == second_word[m -
        2] &&
24        first_word[n - 2] == second_word[m - 1])
25    {
26        matrix[n][m] = std::min(matrix[n][m],
            damerau_lev_rec_hash(first_word, second_word, matrix,
27                                                  n - 2, m - 2) + 1);
28    }
29
30    return matrix[n][m];
31 }
32
33 int damerau_lev_recursion_hash(std::wstring &first_word,
```

```
    std::wstring &second_word)
34 {
35     int first_length = first_word.length();
36     int second_length = second_word.length();
37 
38     int **matrix = alloc_matrix(first_length + 1, second_length
       + 1);
39     for (int i = 0; i < first_length + 1; i++) {
40         for (int j = 0; j < second_length + 1; j++) {
41             matrix[i][j] = -1;
42         }
43     }
44 
45     int res = damerau_lev_rec_hash(first_word, second_word,
       matrix,
46                                    first_word.length(),
                                          second_word.length());
47 
48     free_matrix(matrix, first_length + 1);
49     return res;
50 }
```

Листинг 3.5 – Функции динамического выделения и очищения памяти под матрицу

```
int **alloc_matrix(int n, int m) {
    if (n < 1 || m < 1) {
        return nullptr;
    }
    int **matrix = new int*[n];
    for (int i = 0; i < n; i++) {
        matrix[i] = new int[m];
    }
    return matrix;
}

int free_matrix(int **matrix, int n) {
    if (n < 1 || matrix == nullptr) {
        return 1;
    }
    for (int i = 0; i < n; i++) {
        delete[] matrix[i];
    }
    delete[] matrix;
    return 0;
}
```

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояний Левенштейна и Дамерау—Левенштейна. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Входные данные		**Расстояние и алгоритм**			
Строка 1	**Строка 2**	**Левенштейна**	**Дамерау-Левенштейна**		
		Итеративный	**Итеративный**	**Рекурсивный**	
				Без кеша	**С кешом**
a	b	1	1	1	1
a	a	0	0	0	0
12	123	1	1	1	1
123	12	1	1	1	1
hello	helol	2	1	1	1
привет	привте	2	1	1	1
слон	слоны	1	1	1	1

Вывод

Были реализованы алгоритмы поиска расстояния Левенштейна итеративно, а также поиска расстояния Дамерау–Левенштейна итеративно, рекурсивно и рекурсивного с кеширования. Проведено тестирование реализаций алгоритмов.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени, представлены далее.

— Процессор: Intel(R) Core(TM) i7-1165G7 CPU 2.80 ГГц.

— Оперативная память: 15 ГБайт.

— Операционная система: Ubuntu 64-разрядная система версии 22.04.3.

При замерах времени ноутбук был включен в сеть электропитания и был нагружен только системными приложениями.

4.2 Демонстрация работы программы

На рисунке 4.1 представлена демонстрация работы разработанного программного обеспечения, а именно показаны результаты вычислений реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна на примере двух строк «клон» и «трон».

```
Выберете команду: 1 - Найти расстояние с помощью алгоритма Левенштейна с матрицами
2 - Найти расстояние с помощью алгоритма Дамерау-Левенштейна с матрицами
3 - Найти расстояние с помощью алгоритма Дамерау-Левенштейна с рекурсией
4 - Найти расстояние с помощью алгоритма Дамерау-Левенштейна с рекурсией и кэшем
5 - Найти среднее время на выполнение
6 - Остановить выполнение
1
Введите первое слово: клон
Введите второе слово: трон
Результат: 2

Выберете команду: 1 - Найти расстояние с помощью алгоритма Левенштейна с матрицами
2 - Найти расстояние с помощью алгоритма Дамерау-Левенштейна с матрицами
3 - Найти расстояние с помощью алгоритма Дамерау-Левенштейна с рекурсией
4 - Найти расстояние с помощью алгоритма Дамерау-Левенштейна с рекурсией и кэшем
5 - Найти среднее время на выполнение
6 - Остановить выполнение
2
Введите первое слово: клон
Введите второе слово: трон
Результат: 2

Выберете команду: 1 - Найти расстояние с помощью алгоритма Левенштейна с матрицами
2 - Найти расстояние с помощью алгоритма Дамерау-Левенштейна с матрицами
3 - Найти расстояние с помощью алгоритма Дамерау-Левенштейна с рекурсией
4 - Найти расстояние с помощью алгоритма Дамерау-Левенштейна с рекурсией и кэшем
5 - Найти среднее время на выполнение
6 - Остановить выполнение
6
```

Рисунок 4.1 – Демонстрация работы программы при поиске расстояний Левенштейна и Дамерау-Левенштейна

4.3 Временные характеристики

Результаты эксперимента замеров по времени приведены в таблице 4.1, в которой есть поля, обозначенные «-». Это обусловлено тем, что для рекурсивной реализации алгоритмов достаточно приведенных замеров для построения графика. По полученным замерам по времени для рекурсивной реализации понятно, что проведения замеров на длин строк больше 9 будет достаточно долгим, поэтому нет смысла проводить замеры по времени рекурсивной реализации алгоритма.

Таблица 4.1 – Замер по времени для строк, размер которых от 1 до 200

Кол-во символов	Время, мс			
Длина строки	Левенштейн-Матрица	Дамерау-Левенштейн матрица	Дамерау-Левенштейн рекурсия	Дамерау-Левенштейн рекурсия с кэшем
1	246.100	239.000	87.700	79.100
2	128.800	125.400	124.600	182.000
3	263.600	267.000	734.600	297.600
4	340.600	426.400	3 664.200	498.300
5	522.200	689.200	17 661.500	723.100
6	746.500	849.400	93 846.500	1 024.300
7	1 064.500	1 138.300	510 799.312	1 409.600
8	1 350.000	1 489.400	2 901 220.000	1 902.600
9	1 727.500	1 880.700	16 431 517.000	2 401.600
10	2 106.500	2 213.500	-	2 879.200
20	7 065.100	7 797.000	-	11 505.800
30	15 460.500	17 161.100	-	26 226.199
40	26 836.400	29 828.600	-	46 470.699
50	41 745.699	49 676.602	-	73 213.797
60	58 991.000	65 434.000	-	104 228.797
70	77 570.398	86 644.500	-	139 459.094
80	101 191.898	113 068.500	-	183 324.000
90	127 196.102	142 163.000	-	231 846.297
100	157 460.797	174 466.797	-	282 203.812
200	643 454.875	711 905.500	-	1 158 021.375

Замеры проводились на одинаковых длин строк от 1 до 200 с различным шагом.

Отдельно сравнивается реализации итеративных алгоритмов поиска расстояний Левенштейна и Дамерау–Левенштейна. Сравнение будет производиться на основе данных, представленных в таблице 4.1. Результат можно рассмотреть на рисунке 4.2.

При длинах строк менее 30 символов разница по времени между итеративными реализациями незначительна, однако при увеличении длины строки алгоритм поиска расстояния Левенштейна затрачивает меньше времени. Это обосновывается тем, что у алгоритма поиска расстояния Дамерау-Левенштейна задействуется дополнительная операция, которая замедляет алгоритм.

Также сравним рекурсивную и итеративную реализации алгоритма

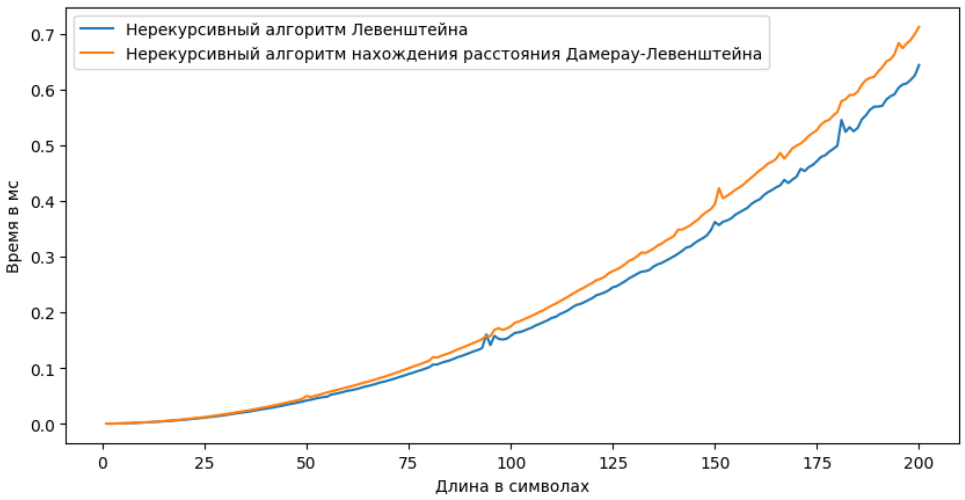

Рисунок 4.2 – Сравнение по времени нерекурсивных реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна

поиска расстояния Дамерау-Левенштейна. Данные представлены в таблице 4.1 и отображены на рисунке 4.3.

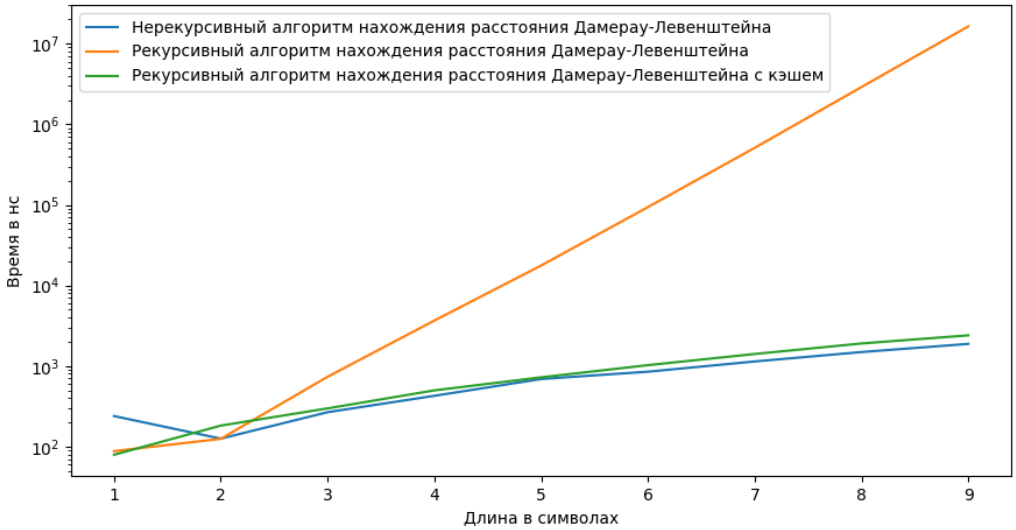

Рисунок 4.3 – Сравнение по времени реализаций алгоритмов поиска расстояния Дамерау-Левенштейна

На рисунке 4.3 продемонстрировано, что рекурсивный алгоритм без кэширования становится менее эффективным по времени (на 4 порядка больше времени при длине строк равной 8 элементов), чем итеративный и рекурсивный с кэшированием.

Кроме того, согласно данным, приведенным в таблице 4.1, рекурсивный алгоритм без кэширования при длинах строк более 10 элементов не пригодны к использованию в силу экспоненциально роста затрат процес-

сорного времени, в то время, как затраты итеративных алгоритмов и рекурсивного алгоритма с кэшированием по времени линейны.

4.4 Характеристики по памяти

Введем следующие обозначения:

— n — длина строки S_1;

— m — длина строки S_2;

— $size()$ — функция вычисляющая размер в байтах;

— $string$ — строковый тип;

— int — целочисленный тип;

— $size\_t$ — беззнаковый целочисленный тип.

Максимальная глубина стека вызовов при рекурсивной реализации нахождения расстояния Дамерау-Левенштейна равна сумме входящих строк, а на каждый вызов требуется 2 дополнительные переменные, соответственно, максимальный расход памяти вычисляется по формуле:

$$(n + m) \cdot (2 \cdot size(string) + 3 \cdot size(int) + 2 \cdot sizeof(size\_t)), \qquad (4.1)$$

где:

— $2 \cdot size(string)$ — хранение двух строк;

— $2 \cdot size(size\_t)$ — хранение размеров строк;

— $2 \cdot size(int)$ — дополнительные переменные;

— $size(int)$ — адрес возврата.

Для рекурсивного алгоритма с кешированием поиска расстояния Дамерау-Левенштейна будет теоретически схож с расчетом в формуле (4.1),

но также учитывается матрица, соответственно, максимальный расход памяти вычисляется по формуле:

$$(n+m)\cdot(2\cdot size(string)+3\cdot size(int)+2\cdot size(size\_t))+ \\ +(n+1)\cdot(m+1)\cdot size(int). \tag{4.2}$$

Использование памяти при итеративной реализации алгоритма поиска расстояния Левенштейна теоретически вычисляется по формуле:

$$(n+1)\cdot(m+1)\cdot size(int)+2\cdot size(string)+2\cdot size(size\_t)+ \\ +size(int**)+(n+1)\cdot size(int*)+2\cdot size(int), \tag{4.3}$$

где

— $2\cdot size(string)$ — хранение двух строк;

— $2\cdot size(size\_t)$ — хранение размеров матрицы;

— $(n+1)\cdot(m+1)\cdot size(int)$ — хранение матрицы;

— $size(int**)+(n+1)\cdot size(int*)$ — указатель на матрицу;

— $size(int)$ — дополнительная переменная для хранения результата;

— $size(int)$ — адрес возврата.

Использование памяти при итеративной реализации алгоритма поиска расстояния Дамерау-Левенштейна теоретически вычисляется по формуле:

$$(n+1)\cdot(m+1)\cdot size(int)+2\cdot size(string)+2\cdot size(size\_t)+ \\ +size(int**)+(n+1)\cdot size(int*)+3\cdot size(int), \tag{4.4}$$

где

— $2*size(string)$ — хранение двух строк;

— $2\cdot size(size\_t)$ — хранение размеров матрицы;

— $(n+1)\cdot(m+1)\cdot size(int)$ — хранение матрицы;

— $size(int**)+(n+1)\cdot size(int*)$ — указатель на матрицу;

— $2 \cdot size(int)$ — дополнительные переменные;

— $size(int)$ — адрес возврата.

По расходу памяти итеративные алгоритмы проигрывают рекурсивным: максимальный размер используемой памяти в итеративном растет как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк.

По формулам 4.1 – 4.3 затрат по памяти в программе были написаны соответствующие функции для подсчета расходуемой памяти, результаты расчетов, которых представлены в таблице 4.2, где размеры строк находятся в диапазоне от 10 до 200 с шагом 10.

Таблица 4.2 – Замер памяти для строк, размером от 10 до 200

Len	lev	dl	dl_rec	dl_rec_cash
10	668	672	1 840	2 420
20	2 028	2 032	3 680	5 620
30	4 188	4 192	5 520	9 620
40	7 148	7 152	7 360	14 420
50	10 908	10 912	9 200	20 020
60	15 468	15 472	11 040	26 420
70	20 828	20 832	12 880	33 620
80	26 988	26 992	14 720	41 620
90	33 948	33 952	16 560	50 420
100	41 708	41 712	18 400	60 020
110	50 268	50 272	20 240	70 420
120	59 628	59 632	22 080	81 620
130	69 788	69 792	23 920	93 620
140	80 748	80 752	25 760	106 420
150	92 508	92 512	27 600	120 020
160	105 068	105 072	29 440	134 420
170	118 428	118 432	31 280	149 620
180	132 588	132 592	33 120	165 620
190	147 548	147 552	34 960	182 420
200	163 308	163 312	36 800	200 020

Из данных, приведенных в таблице 4.2, понятно, что рекурсивные алгоритмы являются более эффективными по памяти, так как используется только память под локальные переменные, передаваемые аргументы и возвращаемое значение, в то время как итеративные алгоритмы затрачивают память линейно пропорционально длинам обрабатываемых строк.

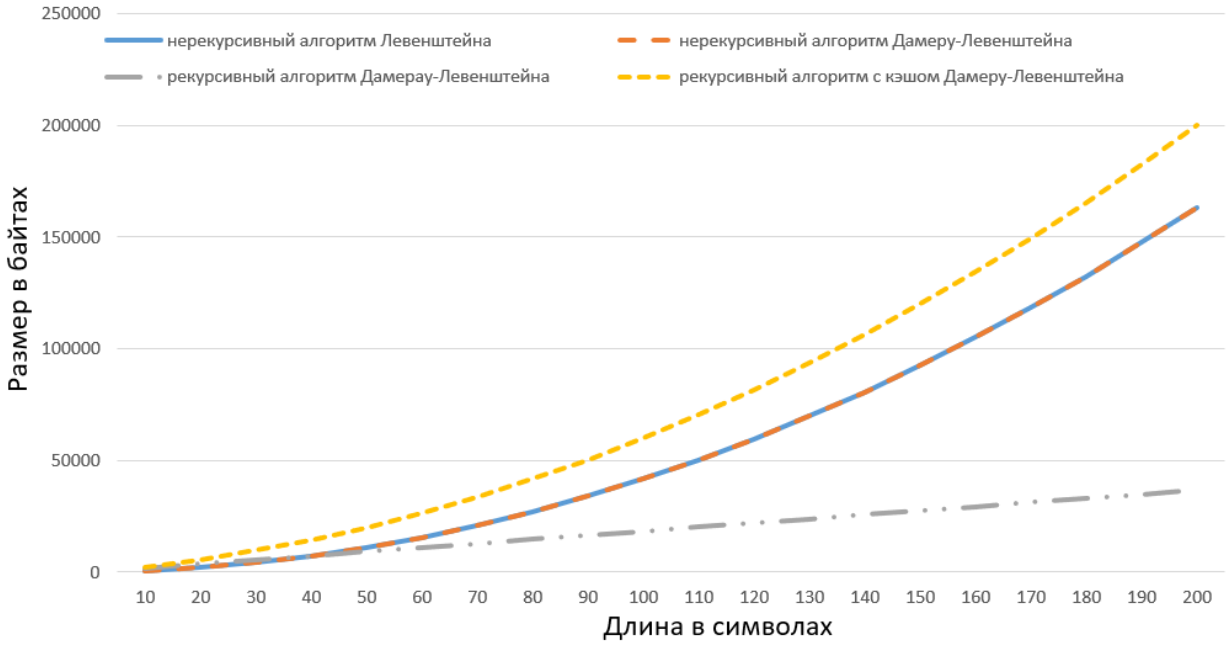

Рисунок 4.4 – Сравнение по памяти реализаций алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна — итеративной и рекурсивной реализации

Из рисунка 4.5 понятно, что рекурсивная реализация алгоритма поиска расстояния Дамерау-Левенштейна более эффективна по памяти, чем итеративная.

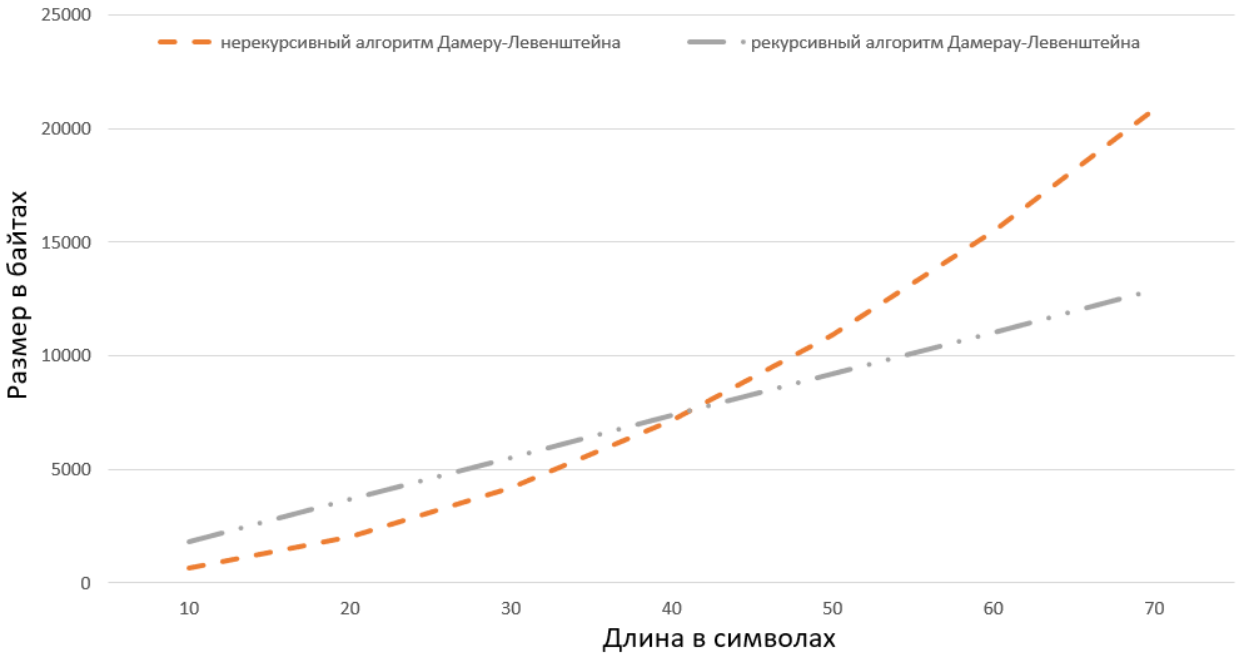

Рисунок 4.5 – Сравнение по памяти реализаций алгоритмов поиска расстояния Дамерау-Левенштейна — итеративной и рекурсивной реализации

4.5 Вывод

В данном разделе было произведено сравнение количества затраченного времени и памяти реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна. Наименее затратным по времени оказалась итеративная реализация алгоритма нахождения расстояния Левенштейна. Наименее затратным по памяти оказался рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна.

Приведенные характеристики показывают, что рекурсивная реализация алгоритма поиска расстояния Дамерау-Левенштейна в 30 раз проигрывает по времени итеративной реализации при строках длиной 5, однако выигрывает по памяти в 5 раз при строках длиной 200. Несмотря на выигрыш по памяти при росте длины строки по сравнению с нерекурсивным алгоритмом, рекурсивный алгоритм следует использовать лишь для малых размерностей строк (1 – 4 символа), так как рост времени на выполнение делает его неприменимым для более длинных строк.

Написанная реализация рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кэшированием проигрывает нерекурсивной реализации по времени при всех строках, длина которых больше 1, и всегда проигрывает по памяти. Из этого можно сделать вывод, что нерекурсивный алгоритм поиска расстояния Дамерау-Левенштейна стоит использовать вместо рекурсивных реализаций во всех рассмотренных случаях.

Сравнивая нерекурсивные алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна, стоит учесть, что во время печати очень часто возникают ошибки, связанные с транспозицией букв, из-за чего для поиска меры разницы двух строк стоит использовать алгоритмы поиска расстояния Дамерау-Левенштейна, несмотря на то, что приведенные реализации алгоритмов либо требуют больше времени, либо используют больше памяти, чем представленная реализация алгоритма поиска расстояния Левенштейна.

Заключение

В результате исследования было определено, что время работы реализаций алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна растет в геометрической прогрессии при увеличении длин строк. Лучшие показатели по времени дает нерекурсивная реализация алгоритма нахождения расстояния Дамерау-Левенштейна и его рекурсивная реализация с кэшированием, использование которых приводит к 21-кратному превосходству по времени работы уже на длине строки в 4 символа за счет сохранения необходимых промежуточных вычислений.

В результате оценки лучшим по памяти является рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна. Нерекурсивные алгоритмы дали одинаковые результаты, используя памяти меньше, чем рекурсивный алгоритм. Худшей по памяти оказался рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кешированием.

Цель данной лабораторной работы была достигнута — были изучены алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна.

Для достижения поставленных целей были выполнены все задачи.

1) Описаны алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна;

2) Создано программное обеспечение, реализующее следующие алгоритмы:

   — нерекурсивный метод поиска расстояния Левенштейна;
   — нерекурсивный метод поиска расстояния Дамерау-Левенштейна;
   — рекурсивный метод поиска расстояния Дамерау-Левенштейна;
   — рекурсивный поиска расстояния Дамерау-Левенштейна с кэшированием.

3) Выбраны инструменты для замера процессорного времени требуемого для выполнения реализаций алгоритмов.

4) Проведен анализ затрат работы программы по времени и по памяти, определены влияющие на них факторы.

Список использованных источников

1. В. Ульянов М. Ресурсно-эффективные компьютерные алгоритмы: учебное пособие. — М.: Издательство «Наука», ФИЗМАТЛИ, 2007.

2. И. Левенштейн В. Двоичные коды с исправлением выпадений, вставок и замещений символов. — М.: Издательство «Наука», Доклады АН СССР, 1965. Т. 163.

3. Документация по Microsoft C++ [Электронный ресурс]. — Режим доступа: `https://learn.microsoft.com/ru-ru/cpp/?view=msvc-170&viewFallbackFrom=vs-2017` (дата обращения: 26.09.2023).

4. C library function clock() [Электронный ресурс]. — Режим доступа: `https://www.tutorialspoint.com/c_standard_library/c_function_clock.htm` (дата обращения: 26.09.2023).