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"

Тема Расстояние Левенштейна и Дамерау-Левенштейна

Студент Коротыч М. Д.

Группа ИУ7-55Б

Преподаватели _____

Москва — 2021 г.

Оглавление

Введение **2**

1 Аналитическая часть **4**

1.1 Рекурсивный алгоритм 5

1.2 Матричный алгоритм 6

1.3 Рекурсивный алгоритм с заполнением матрицы 6

1.4 Расстояние Дамерау — Левенштейна 7

1.5 Вывод 7

2 Конструкторская часть **8**

2.1 Схема алгоритма нахождения расстояния Левенштейна . . . 8

2.2 Схема алгоритма нахождения расстояния Дамерау — Левенштейна 13

2.3 Вывод 15

3 Технологическая часть **16**

3.1 Требования к ПО 16

3.2 Средства реализации 16

3.3 Демонстрация работы 16

3.4 Листинг кода 17

3.5 Вывод 22

4 Исследовательская часть **23**

4.1 Технические характеристики 23

4.2 Время выполнения алгоритмов 23

4.3 Использование памяти 27

4.4 Вывод 30

Заключение **32**

Список литературы **33**

Введение

Расстояние Левенштейна (редакционное расстояние) — метрика, измеряющая разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (вставки, удаления, замены), необходимых для превращения одной последовательности символов в другую. В общем случае, операциям, используемым в этом преобразовании, можно назначить разные цены. Широко используется в теории информации и компьютерной лингвистике.

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 и его обобщения применяются:

1. для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);

2. для сравнения текстовых файлов утилитой `diff` и ей подобными;

3. в биоинформатике.

Расстояние Дамерау — Левенштейна (названо в честь учёных Фредерика Дамерау и Владимира 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.

Целью данной лабораторной работы является изучение, реализация и исследование алгоритмов нахождения расстояний Левенштейна и Дамерау - Левенштейна. Задачами данной лабораторной работы являются:

- изучение алгоритмов нахождения расстояния Левенштейна и Дамерау - Левенштейна;

- сравнение и выявление достоинств и недостатков рассмотренных алгоритмов;

- применение методов динамического программирования для реализации алгоритмов;
- получение практических навыков реализации алгоритмов Левенштейна и Дамерау — Левенштейна;
- сравнительный анализ алгоритмов на основе экспериментальных данных;
- подготовка отчёта по лабораторной работе.

1 Аналитическая часть

Расстояние Левенштейна между двумя строками — это минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую.

Цены операций могут зависеть от вида операции (вставка, удаление, замена) и/или от участвующих в ней символов. В общем случае имеются следующие стоимости:

- $w(a, b)$ — цена замены символа a на символ b;
- $w(\lambda, b)$ — цена вставки символа b;
- $w(a, \lambda)$ — цена удаления символа a.

Для решения задачи о редакционном расстоянии необходимо найти последовательность замен, которая минимизирует суммарную цену. Расстояние Левенштейна является частным случаем этой задачи при:

- $w(a, a) = 0$;
- $w(a, b) = 1,\ a \neq b$;
- $w(\lambda, b) = 1$;
- $w(a, \lambda) = 1$.

1.1 Рекурсивный алгоритм

Расстояние Левенштейна между двумя строками a и b может быть вычислено по формуле 1.1, где $|a|$ означает длину строки a; $a[i]$ — i-ый символ строки a , функция $D(i,j)$ определена как:

$$D(i,j) = \begin{cases} 0 & \text{i} = 0, \text{j} = 0 \\ i & \text{j} = 0, \text{i} > 0 \\ j & \text{i} = 0, \text{j} > 0 \\ \min\{ & \\ \quad D(i,j-1)+1 & \\ \quad D(i-1,j)+1 & \text{i} > 0, \text{j} > 0 \\ \quad D(i-1,j-1)+m(a[i],b[j]) & (1.2) \\ \} & \end{cases}, \qquad (1.1)$$

а функция 1.2 определена как:

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

Рекурсивный алгоритм реализует формулу 1.1. Функция D составлена из следующих соображений:

1. для перевода 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 требуется $|a|$ операций.

Для перевода из строки a в строку b требуется выполнить некоторое количество операций (удаление, вставка, замена) в некоторой последовательности. Полагая, что a', b' — строки a и b без последнего символа соответственно, цена преобразования из строки a в строку b может быть выражена как:

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

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

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

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

Минимальной ценой преобразования будет минимальное значение приведенных вариантов.

1.2 Матричный алгоритм

Прямая реализация формулы 1.1 может быть малоэффективна по времени исполнения при больших i, j, т. к. множество промежуточных значений $D(i, j)$ вычисляются заново множество раз подряд. Для оптимизации нахождения расстояния Левенштейна можно использовать матрицу для хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы $A_{(|a|+1)\times(|b|+1)}$ значениями $D(i, j)$.

1.3 Рекурсивный алгоритм с заполнением матрицы

Рекурсивный алгоритм заполнения можно оптимизировать по времени выполнения с использованием матричного алгоритма. Суть данного метода заключается в параллельном заполнении матрицы при выполнении рекурсии. В случае, если рекурсивный алгоритм выполняет прогон для данных, которые ещё не были обработаны, результат нахождения расстояния заносится в матрицу. В случае если обработанные ранее данные встречаются

снова, для них расстояние не находится и алгоритм переходит к следующему шагу.

1.4 Расстояние Дамерау — Левенштейна

Расстояние Дамерау — Левенштейна может быть найдено по формуле 1.3, которая задана как

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j) = 0, \\ \min\{ & \\ \quad d_{a,b}(i,j-1) + 1, & \\ \quad d_{a,b}(i-1,j) + 1, & \\ \quad d_{a,b}(i-1,j-1) + m(a[i],b[j]), & \text{иначе} \\ \quad \begin{cases} d_{a,b}(i-2,j-2) + 1, & \text{если } i,j > 1; \\ & a[i] = b[j-1]; \\ & b[j] = a[i-1] \\ \infty, & \text{иначе} \end{cases} & \\ \} & \end{cases}, \tag{1.3}$$

Формула выводится так же, как и формула (1.1). Как и в случае с рекурсией, прямое применение этой формулы неэффективно по времени исполнения, поэтому аналогично методу из 1.3 добавляется матрица для хранения промежуточных значений рекурсивной формулы.

1.5 Вывод

Формулы Левенштейна и Дамерау — Левенштейна для расчёта расстояния между строками задаются рекурсивно, а следовательно, алгоритмы могут быть реализованы рекурсивно или итерационно.

2 Конструкторская часть

2.1 Схема алгоритма нахождения расстояния Левенштейна

На рисунке 2.1 приведена схема рекурсивного алгоритма нахождения расстояния Левенштейна.

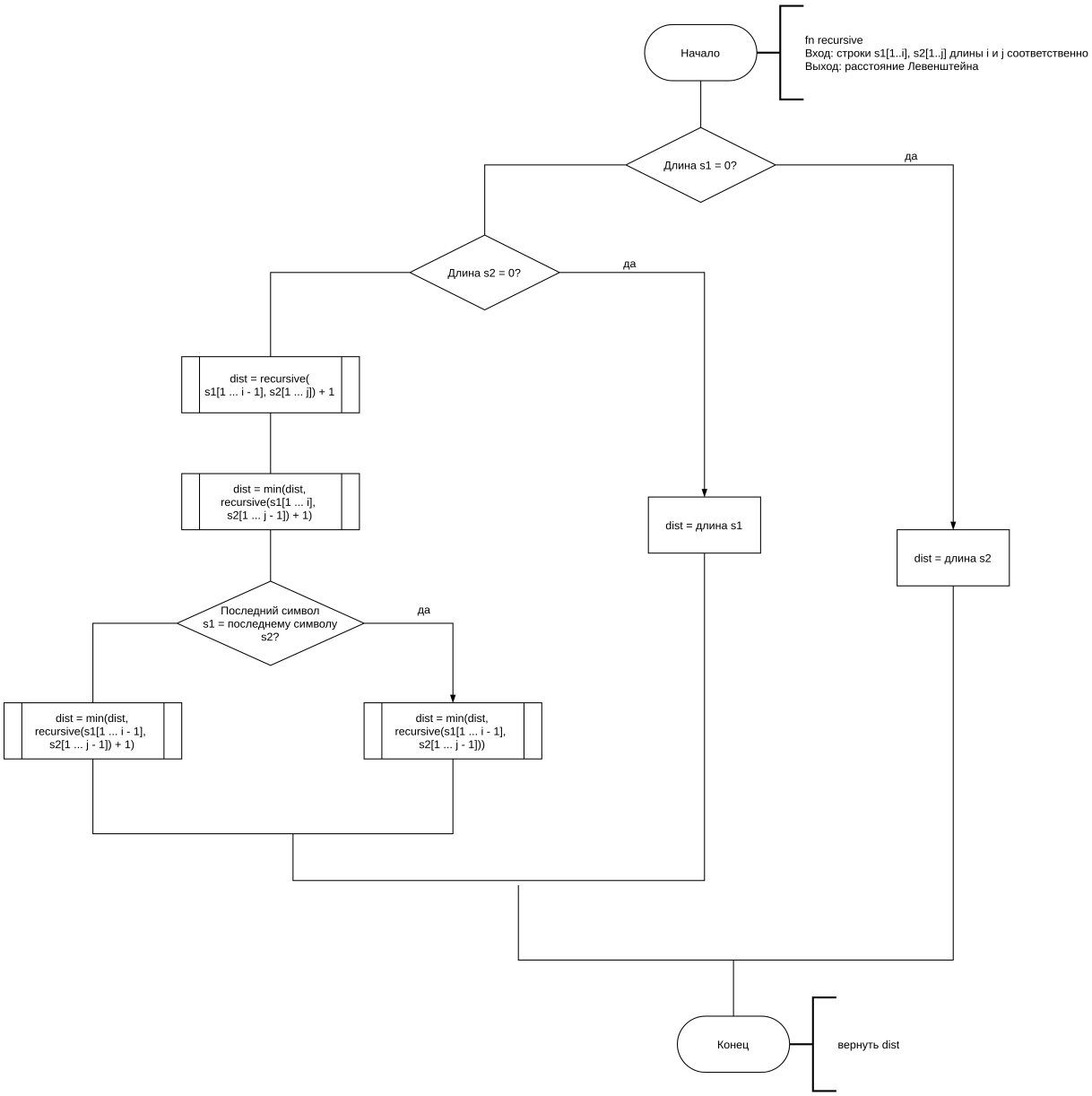

Рисунок 2.1 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна

На рисунке 2.2 схема алгоритма инициализации матрицы.

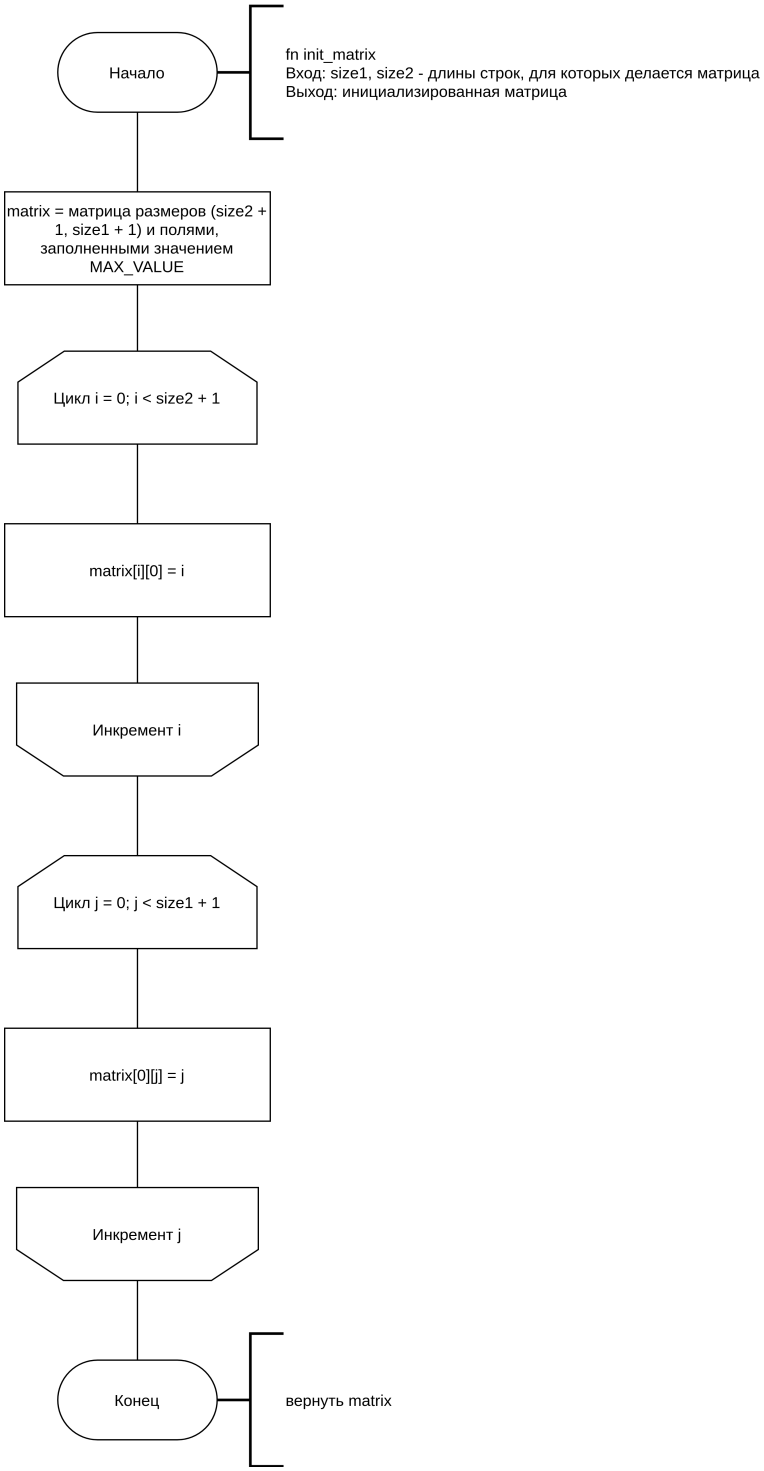

Рисунок 2.2 – Схема алгоритма инициализации матрицы

На рисунках 2.3 и 2.4 приведена схема рекурсивного алгоритма нахождения расстояния Левенштейна с заполнением матрицы.

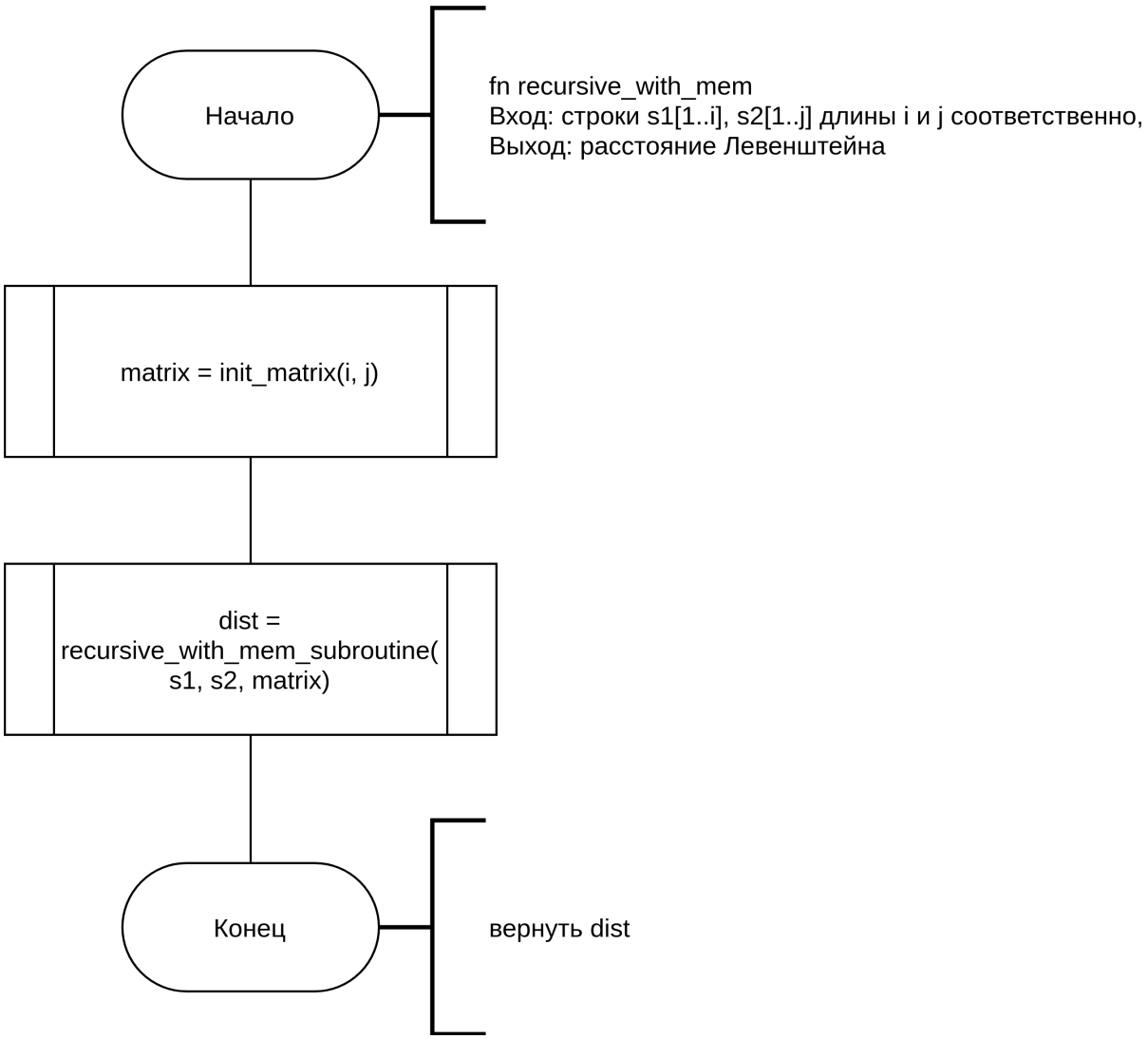

Рисунок 2.3 – Схема матрично-рекурсивного алгоритма нахождения расстояния Левенштейна

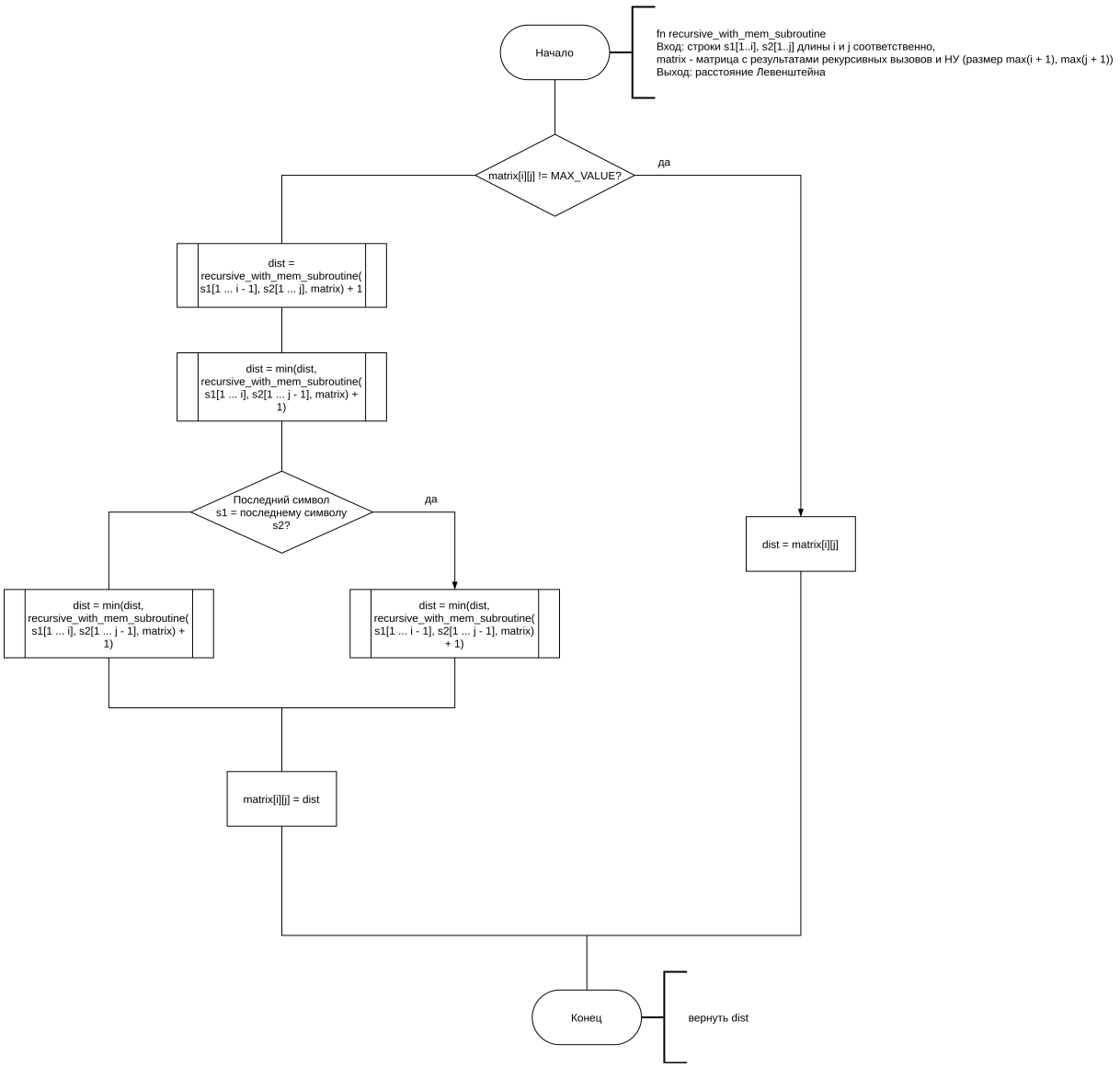

Рисунок 2.4 – Схема матрично-рекурсивного алгоритма нахождения расстояния Дамерау - Левенштейна

На рисунке 2.5 приведена схема алгоритма нахождения расстояния расстояния Левенштейна с заполнением матрицы.

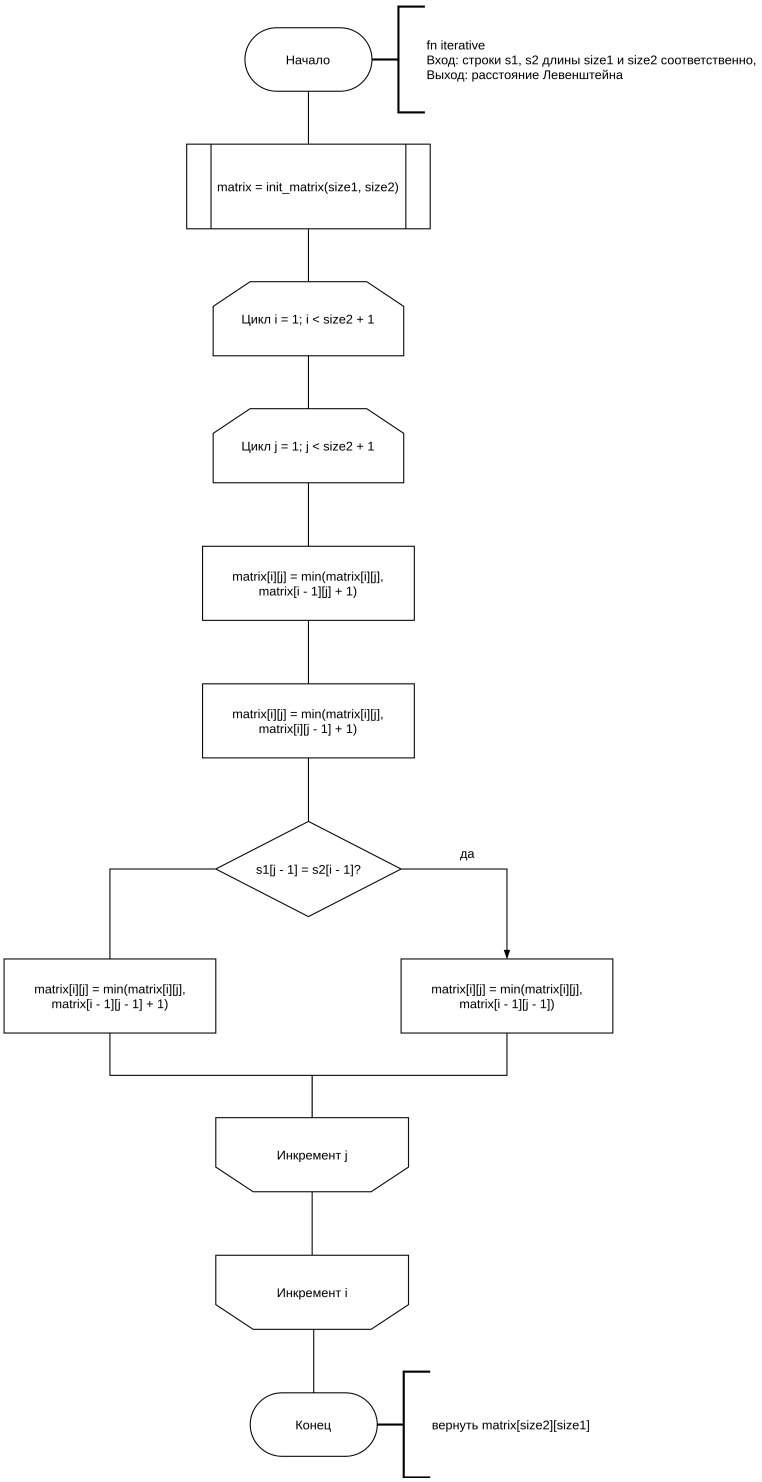

Рисунок 2.5 – Схема матричного алгоритма нахождения расстояния Левенштейна

2.2 Схема алгоритма нахождения расстояния Дамерау — Левенштейна

На рисунках 3.1 и 2.7 приведена схема матричного алгоритма нахождения расстояния Дамерау - Левенштейна.

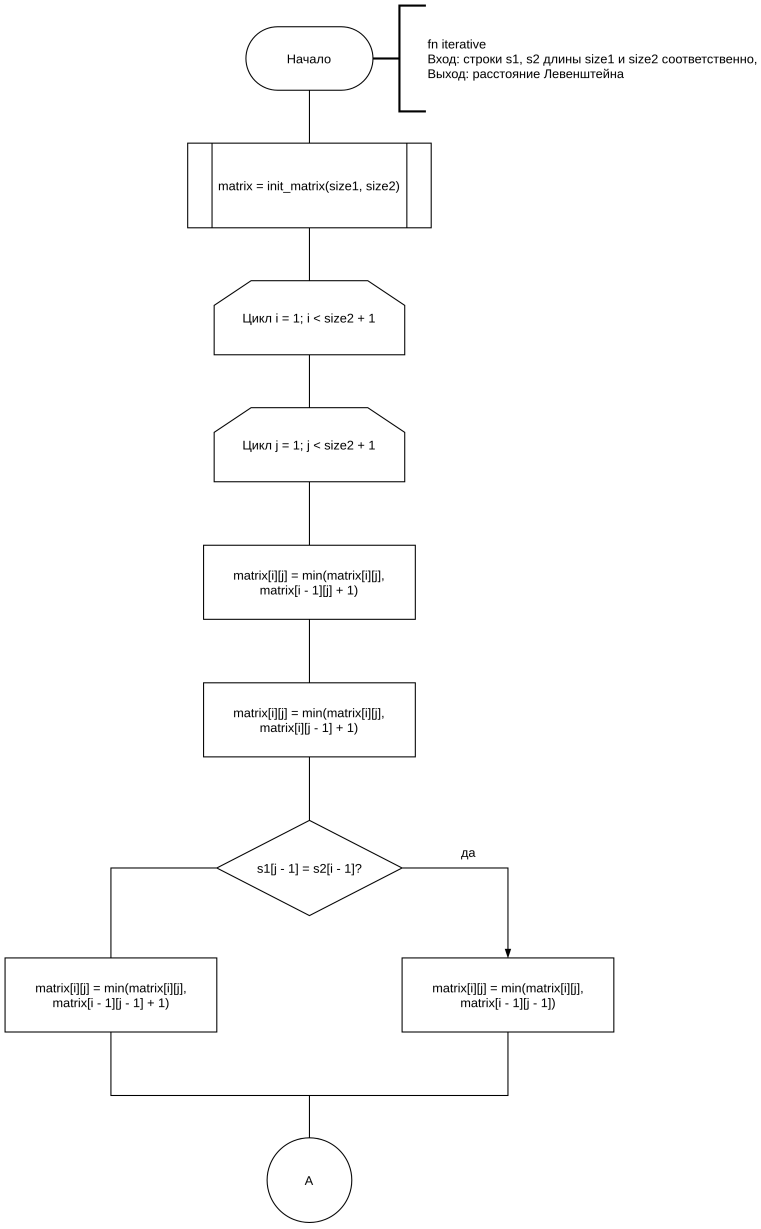

Рисунок 2.6 – Схема матричного алгоритма нахождения расстояния Дамерау - Левенштейна

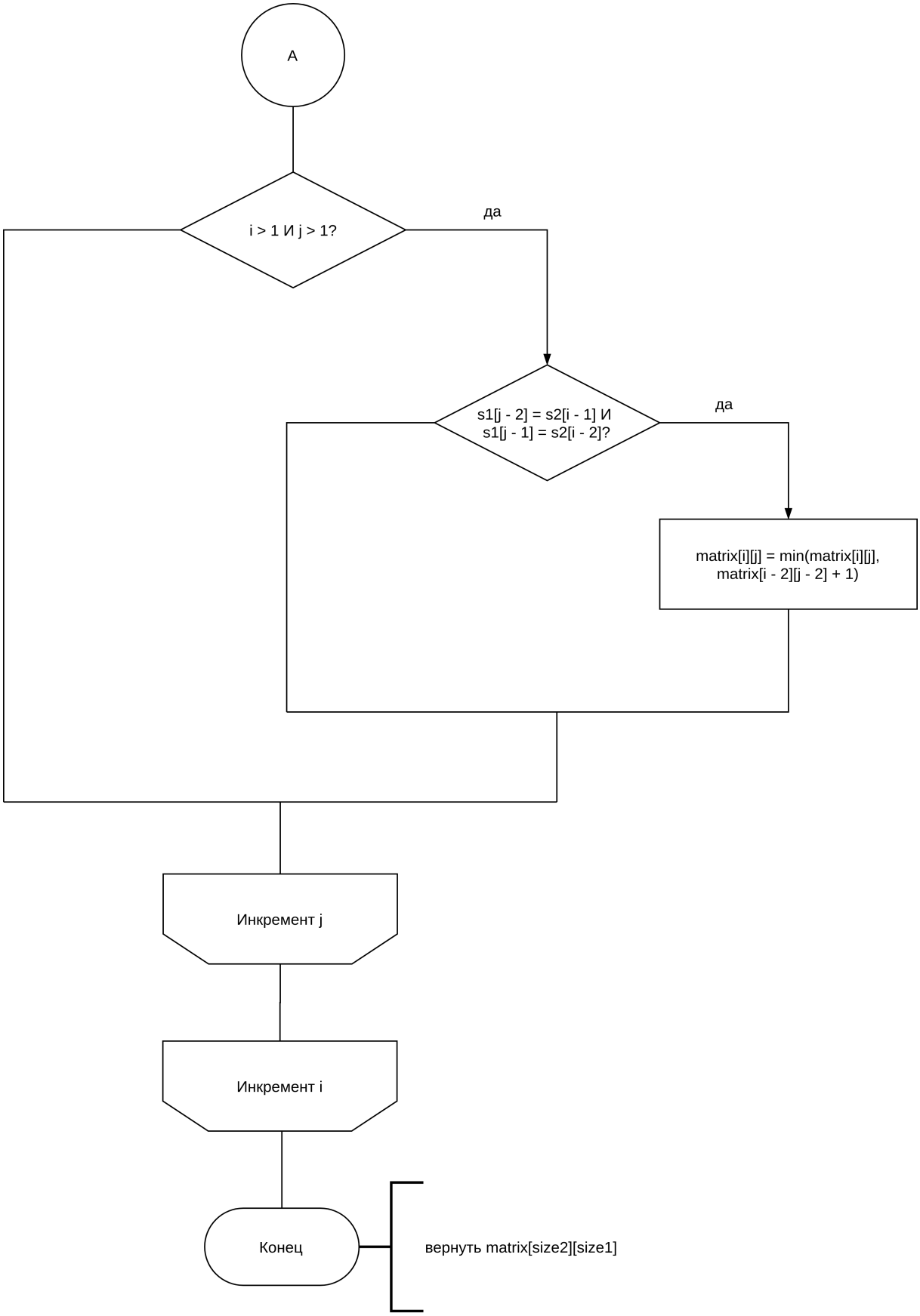

Рисунок 2.7 – Продолжение схемы матричного алгоритма нахождения расстояния Дамерау-Левенштейна

2.3 Вывод

На основе теоретических данных, полученных из аналитического раздела были построены схемы требуемых алгоритмов.

3 Технологическая часть

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

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подаются две строки на английском языке в любом регистре;
- на выходе — искомое расстояние для всех четырёх методов и матрицы расстояний для всех методов, за исключением рекурсивного.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран ЯП C++. Данный выбор обусловлен наличием большого опыта в написании кода и знанием структуры этого языка программирования.

3.3 Демонстрация работы

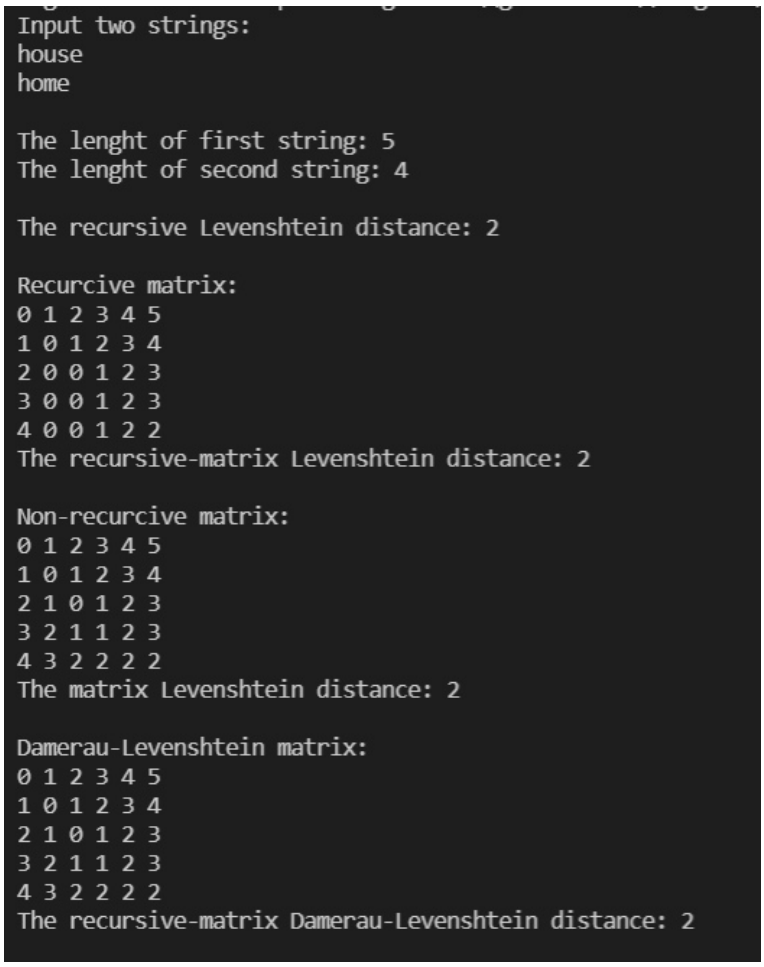

Рисунок 3.1 – Результат работы программы

3.4 Листинг кода

В листинге 3.1 приведена реализация алгоритмов нахождения расстояния Левенштейна и Дамерау — Левенштейна, а также вспомогательные функции.

Листинг 3.1 – Листинг с алгоритмами

```
#include <algorithm>
#include <iostream>
#include <climits>
#include <cmath>
#include <string>
#include <vector>
#include "getCPUTime.h"

std::vector<std::vector<int>> init_matrix(std::vector<std::vector<int>> &matrix,
    std::string str1, std::string str2)
```

```
{
    int n = str1.length(), m = str2.length();

    for (int i = 0; i < m + 1; i++)
        matrix[i][0] = i;
    for (int j = 0; j < n + 1; j++)
        matrix[0][j] = j;

    return matrix;
}

int sbrt(std::vector<std::vector<int>> &matrix, std::string str1, std::string str2)
{
    int n = str1.length(), m = str2.length();

    if (matrix[m][n] != INT_MAX)
        return matrix[m][n];
    else
    {
        int dist = sbrt(matrix, str1.substr(0, n - 1), str2) + 1;
        dist = std::min(dist, sbrt(matrix, str1, str2.substr(0, m - 1)) + 1);
        dist = std::min(dist, sbrt(matrix, str1[n - 1] == str2[m - 1] ? str1.substr(0, n
            - 1) : str1, str2.substr(0, m - 1)) + 1);
        matrix[m][n] = dist;
        return dist;
    }
}

int lev_dist_rec_mtrx(std::string str1, std::string str2) // -
{
    std::vector<std::vector<int>> matrix(str2.length() + 1,
        std::vector<int>(str1.length() + 1, INT_MAX));
    matrix = init_matrix(matrix, str1, str2);

    return sbrt(matrix, str1, str2);
}

int lev_dist_rec(std::string str1, std::string str2) //
{
    int n = str1.length(), m = str2.length();
    if (!n)
        return m;
    else
        if (!m)
            return n;
        else
        {
            int dist = lev_dist_rec(str1.substr(0, n - 1), str2) + 1;
```

```
56            dist = std::min(dist, lev_dist_rec(str1, str2.substr(0, m - 1)) + 1);
57            dist = str1[n - 1] == str2[m - 1]? std::min(dist, lev_dist_rec(str1.substr(0,
                  n - 1), str2.substr(0, m - 1)))
58                                                          : std::min(dist,
                                                              lev_dist_rec(str1.substr(0,
                                                              n - 1), str2.substr(0, m -
                                                              1)) + 1);
59            return dist;
60        }
61 }
62
63 int lev_dist_mtrx(std::string str1, std::string str2)
64 {
65     int n = str1.length(), m = str2.length();
66     std::vector<std::vector<int>> matrix(m + 1, std::vector<int>(n + 1, INT_MAX));
67     matrix = init_matrix(matrix, str1, str2);
68
69     for (int i = 1; i < m + 1; i++)
70         for (int j = 1; j < n + 1; j++)
71         {
72             matrix[i][j] = std::min(matrix[i][j], matrix[i - 1][j] + 1);
73             matrix[i][j] = std::min(matrix[i][j], matrix[i][j - 1] + 1);
74             matrix[i][j] = std::min(matrix[i][j], str1[j - 1] == str2[i - 1]? matrix[i -
                  1][j - 1] : matrix[i - 1][j - 1] + 1);
75         }
76
77     return matrix[m][n];
78 }
79
80 int damlev_dist_mtrx(std::string str1, std::string str2)
81 {
82     int n = str1.length(), m = str2.length();
83     std::vector<std::vector<int>> matrix(m + 1, std::vector<int>(n + 1, INT_MAX));
84     matrix = init_matrix(matrix, str1, str2);
85
86     for (int i = 1; i < m + 1; i++)
87         for (int j = 1; j < n + 1; j++)
88         {
89             matrix[i][j] = std::min(matrix[i][j], matrix[i - 1][j] + 1);
90             matrix[i][j] = std::min(matrix[i][j], matrix[i][j - 1] + 1);
91             matrix[i][j] = std::min(matrix[i][j], str1[j - 1] == str2[i - 1] ? matrix[i -
                  1][j - 1] : matrix[i - 1][j - 1] + 1);
92
93             if (i > 1 && j > 1 && str1[j - 2] == str2[i - 1] && str1[j - 1] == str2[i -
                  2])
94                 matrix[i][j] = std::min(matrix[i][j], matrix[i - 2][j - 2] + 1);
95         }
96
```

```
97      return matrix[m][n];
98  }
99
100 void input(std::string &str1, std::string &str2)
101 {
102     std::cout << "Input two strings:" << std::endl;
103     try
104     {
105         std::cin >> str1;
106         std::cin >> str2;
107     }
108     catch (const std::exception &e)
109     {
110         std::cerr << e.what() << '\n';
111     }
112 }
113
114 void output(std::string str1, std::string str2)
115 {
116     std::cout << "\nThe lenght of first string: " << str1.length() << std::endl;
117     std::cout << "The lenght of second string: " << str2.length() << std::endl;
118 }
119
120 int main(void)
121 {
122     std::setbuf(stdout, nullptr);
123     std::string str1, str2;
124     input(str1, str2);
125     output(str1, str2);
126     double start_time, end_time;
127
128     start_time = getCPUTime();
129     int s = lev_dist_rec(str1, str2);
130     end_time = getCPUTime();
131
132     std::cout << "\nThe recursive Levenshtein distance: " << s << std::endl;
133     std::cout << "CPU time used for the recursive Levenshtein distance in seconds: " <<
            end_time - start_time /** std::pow(10, 6)*/ << std::endl;
134
135     start_time = getCPUTime();
136     s = lev_dist_rec_mtrx(str1, str2);
137     end_time = getCPUTime();
138
139     std::cout << "\nThe recursive-matrix Levenshtein distance: " << s << std::endl;
140     std::cout << "CPU time used for the recursive-matrix Levenshtein distance in
            seconds: " << end_time - start_time /** std::pow(10, 6)*/ << std::endl;
141
142     start_time = getCPUTime();
```

```
143     s = lev_dist_mtrx(str1, str2);
144     end_time = getCPUTime();
145 
146     std::cout << "\nThe matrix Levenshtein distance: " << s << std::endl;
147     std::cout << "CPU time used for the matrix Levenshtein distance in seconds: " <<
            end_time - start_time /** std::pow(10, 6)*/ << std::endl;
148 
149     start_time = getCPUTime();
150     s = damlev_dist_mtrx(str1, str2);
151     end_time = getCPUTime();
152 
153     std::cout << "\nThe recursive-matrix Damerau-Levenshtein distance: " << s <<
            std::endl;
154     std::cout << "CPU time used for the matrix Damerau-Levenshtein distance in seconds:
            " << end_time - start_time /** std::pow(10, 6)*/ << std::endl;
155     return 0;
156 }
```

3.5 Вывод

Были разработаны и протестированы алгоритмы: нахождения расстояния Левенштейна рекурсивно, с заполнением матрицы и матрично-рекурсивно, а также нахождения расстояния Дамерау — Левенштейна с заполнением матрицы.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Microsoft Windows 10x86_64.
- Память: 8 ГБ.
- Процессор: Intel Core i7-8550U.

Тестирование проводилось на ноутбуке, включённом в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, окружением, а также непосредственно системой тестирования.

4.2 Время выполнения алгоритмов

Алгоритмы тестировались при помощи функции `GetProccessTime` из `Windows.h`.

В листинге 4.1 приведена реализация функции `GetProccessTime`.

Листинг 4.1 – Листинг с алгоритмами

```
#include "getCPUTime.h"

/*
 * Author: David Robert Nadeau
 * Site: http://NadeauSoftware.com/
 * License: Creative Commons Attribution 3.0 Unported License
 * http://creativecommons.org/licenses/by/3.0/deed.en_US
 */
#if defined(_WIN32)
#include <Windows.h>

#elif defined(__unix__) || defined(__unix) || defined(unix) || (defined(__APPLE__) &&
    defined(__MACH__))
#include <unistd.h>
```

```
14 #include <sys/resource.h>
15 #include <sys/times.h>
16 #include <time.h>
17
18 #else
19 #error "Unable to define getCPUTime( ) for an unknown OS."
20 #endif
21
22
23 /**
24  * Returns the amount of CPU time used by the current process,
25  * in seconds, or -1.0 if an error occurred.
26  */
27 double getCPUTime() {
28 #if defined(_WIN32)
29     /* Windows -------------------------------------------------- */
30     FILETIME createTime;
31     FILETIME exitTime;
32     FILETIME kernelTime;
33     FILETIME userTime;
34     if (GetProcessTimes(GetCurrentProcess(), &createTime, &exitTime, &kernelTime,
        &userTime) != -1) {
35         ULARGE_INTEGER li = {{userTime.dwLowDateTime, userTime.dwHighDateTime }};
36         return li.QuadPart / 10000000.;
37     }
38
39 #elif defined(__unix__) || defined(__unix) || defined(unix) || (defined(__APPLE__) &&
    defined(__MACH__))
40     /* AIX, BSD, Cygwin, HP-UX, Linux, OSX, and Solaris --------- */
41
42 #if defined(_POSIX_TIMERS) && (_POSIX_TIMERS > 0)
43     /* Prefer high-res POSIX timers, when available. */
44     {
45         clockid_t id;
46         struct timespec ts;
47 #if _POSIX_CPUTIME > 0
48         /* Clock ids vary by OS. Query the id, if possible. */
49         if (clock_getcpuclockid(0, &id) == -1)
50 #endif
51 #if defined(CLOCK_PROCESS_CPUTIME_ID)
52             /* Use known clock id for AIX, Linux, or Solaris. */
53             id = CLOCK_PROCESS_CPUTIME_ID;
54 #elif defined(CLOCK_VIRTUAL)
55         /* Use known clock id for BSD or HP-UX. */
56         id = CLOCK_VIRTUAL;
57 #else
58         id = (clockid_t) - 1;
59 #endif
```

```
60          if (id != (clockid_t) - 1 && clock_gettime(id, &ts) != -1)
61              return (double) ts.tv_sec + (double) ts.tv_nsec / 1000000000.0;
62      }
63  #endif
64
65  #if defined(RUSAGE_SELF)
66      {
67          struct rusage rusage;
68          if (getrusage(RUSAGE_SELF, &rusage) != -1)
69              return (double) rusage.ru_utime.tv_sec + (double) rusage.ru_utime.tv_usec /
                    1000000.0;
70      }
71  #endif
72
73  #if defined(_SC_CLK_TCK)
74      {
75          const double ticks = (double) sysconf(_SC_CLK_TCK);
76          struct tms tms;
77          if (times(&tms) != (clock_t) - 1)
78              return (double) tms.tms_utime / ticks;
79      }
80  #endif
81
82  #if defined(CLOCKS_PER_SEC)
83      {
84          clock_t cl = clock();
85          if (cl != (clock_t) - 1)
86              return (double) cl / (double) CLOCKS_PER_SEC;
87      }
88  #endif
89
90  #endif
91
92      return -1; /* Failed. */
93  }
```

В таблице 4.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна и Дамерау — Левенштейна. Все тесты пройдены успешно.

Таблица 4.1 – Функциональные тесты

Строка 1	**Строка 2**	**Ожидаемый результат**	
		Левенштейн	**Дамерау — Левенштейн**
clean	cleaner	2	2
father	fandem	3	3
woman	water	4	4
program	friend	6	6
house	girl	5	5
computer	comupter	2	1
road	orda	3	2
think	thought	5	5
funny	funny	0	0
minute	moment	5	5
pupil	cut	5	5
day	days	1	1
member	morning	6	6
death	health	2	2
education	question	4	4
room	moor	2	2
car	city	3	3
air	area	3	3

Результаты замеров приведены в таблице 4.2. В данной таблице для значений, для которых тестирование не выполнялось, в поле результата находится NaN.

4.3 Использование памяти

Алгоритмы нахождения расстояний Левенштейна и Дамерау — Левенштейна не отличаются друг от друга с точки зрения использования памяти, следовательно, достаточно рассмотреть лишь разницу рекурсивной и матричной реализаций этих алгоритмов.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, при этом для каждого вызова рекурсии в моей реализации требуется:

- 4 локальные переменные беззнакового типа, в моем случае: $4 \cdot 8 = 32$

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

Длина строк	**Время, нс**			
	Recursive	**RecMem**	**Iterative**	**IterativeDL**
10	32766430	1313	634	681
20	NaN	5157	2367	2582
30	NaN	11342	4813	5207
50	NaN	30066	12518	13533
100	NaN	116134	48111	52078
200	NaN	529335	249057	527797

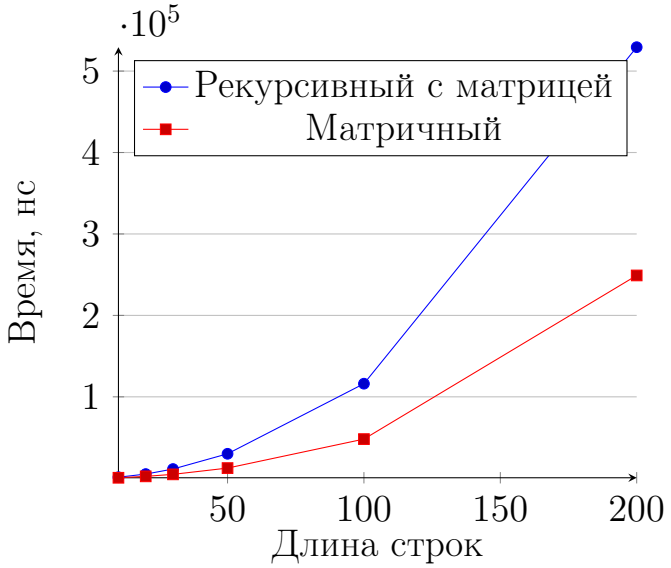

Рисунок 4.1 – Зависимость времени работы алгоритма вычисления расстояния Левенштейна от длины строк (рекурсивная с заполнением матрицы и матричная реализации)

байта;

- 2 аргумента типа строка: $2 \cdot 16 = 32$ байта;
- адрес возврата: 8 байт;
- место для записи возвращаемого функцией значения: 8 байт.

Таким образом получается, что при обычной рекурсии на один вызов требуется (4.1):

$$M_{percall} = 32 + 32 + 8 + 8 = 80 \tag{4.1}$$

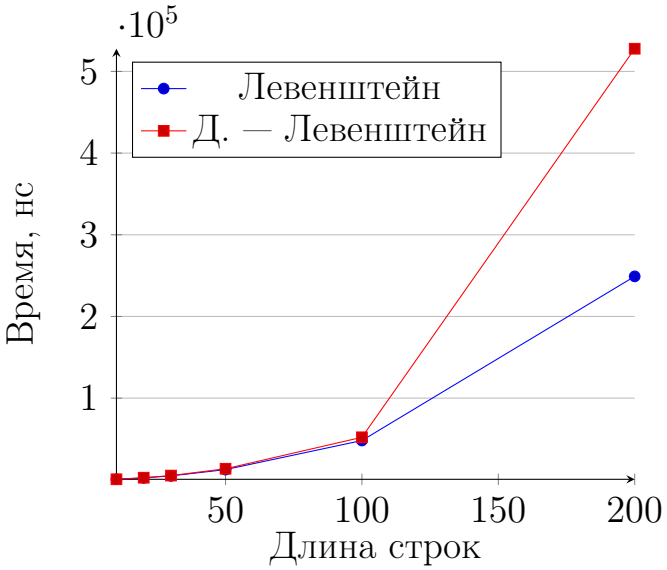

Рисунок 4.2 – Зависимость времени работы матричных реализаций алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна

Следовательно память, расходуемая в момент, когда стек вызовов максимален, равна (4.2):

$$M_{recursive} = 80 \cdot depth \tag{4.2}$$

где $depth$ - максимальная глубина стека вызовов, которая равна (4.3):

$$depth = |S_1| + |S_2| \tag{4.3}$$

где S_1, S_2 - строки.

Если мы используем рекурсивный алгоритм с заполнением матрицы матрицы, то для каждого вызова рекурсии добавляется новый аргумент - ссылка на матрицу - размером 8 байт. Также в данном алгоритме требуется память на саму матрицу, размеры которой: $m = |S_1|+1, n = |S_2|+1$. Размер элемента матрицы равен размеру беззнакового целого числа, используемого в моей реализации, то есть 8 байт. Отсюда выходит, что память, которая тратится на хранение матрицы (4.4):

$$M_{Matrix} = (|S_1| + 1) \cdot (|S_2| + 1) \cdot 8 \tag{4.4}$$

Таким образом, при рекурсивной реализации требуемая память равна

(4.5):

$$M_{recursive} = 88 \cdot depth + M_{Matrix} \tag{4.5}$$

где M_{Matrix} взято из соотношения 4.4.

Память, требуемая для при итеративной реализации, состоит из следующего:

- 4 локальные переменные беззнакового типа, в моем случае: $4 \cdot 8 = 32$ байта;
- 2 аргумента типа строка: $2 \cdot 16 = 32$ байта;
- адрес возврата: 8 байт;
- место для записи возвращаемого функцией значения: 8 байт;
- матрица: M_{Matrix} из соотношения 4.4.

Таким образом общая расходуемая память итеративных алгоритмов (4.6):

$$M_{iter} = M_{Matrix} + 80 \tag{4.6}$$

где M_{Matrix} определяется из соотношения 4.4.

4.4 Вывод

Рекурсивный алгоритм нахождения расстояния Левенштейна работает на порядок дольше итеративных реализаций, время его работы увеличивается в геометрической прогрессии. На словах длиной 10 символов, матричная реализация алгоритма нахождения расстояния Левенштейна превосходит по времени работы рекурсивную на несколько порядков. Рекурсивный алгоритм с заполнением матрицы превосходит простой рекурсивный и сравним по времени работы с матричными алгоритмами. Алгоритм нахождения расстояния Дамерау — Левенштейна по времени выполнения сопоставим с алгоритмом нахождения расстояния Левенштейна. В нём добавлена дополнительная проверка, позволяющая находить ошибки пользователя, связанные с неверным порядком букв, в связи с чем он работает

незначительно дольше, чем алгоритм нахождения расстояния Левенштейна.

Заключение

В ходе выполнения лабораторной работы была проделана следующая работа:

- были теоретически изучены алгоритмы нахождения расстояний Левенштейна и Дамерау–Левенштейна;
- для некоторых реализаций были применены методы динамического программирования, что позволило сделать алгоритмы быстрее;
- были практически реализованы алгоритмы в 2 вариантах: рекурсивном и итеративном;
- на основе полученных в ходе экспериментов данных были сделаны выводы по поводу эффективности всех реализованных алгоритмов;
- был подготовлен отчёт по ЛР.

Список литературы

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

[2] Процессор Intel Core 17-85500 [Электронный ресурс] Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/122589/intel-core-i7-8550u-processor-8m-cache-up-to-4-00-ghz.html (дата обращения: 14.09.2021)

[3] ЛаТеХ для продвинутых: Как контролировать положение плавающих объектов "floats"? [Электронный ресурс] Режим доступа: http://mydebianblog.advanced-floats.html (дата обращения: 01.10.2021)

[4] GetProcessTimes function (processthreadsapi.h) [Электронный ресурс] Режим доступа: https://docs.microsoft.com/en-us/windows/win32/api/processthread processthreadsapi-getprocesstimessyntax (дата обращения: 27.09.2021)

[5] WIndows 10 [Электронный ресурс] Режим доступа: https://www.microsoft.co ru/windows/get-windows-10 (дата обращения: 14.09.2021)