ОглавлениеСодержаниеЛитератураСписок использованных источников
лениеСодержаниеЛитератураСписок использованных источников

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	ИУ7-52Б		Д.В. Батраков
	(Группа)	(Подпись, дата)	(И.О. Фамилия)
Преподователь			Л.Л. Волкова
		(Подпись, дата)	(И.О. Фамилия)

Москва, 2020

Оглавление

ОглавлениеЛитература

Введение

В данной работе требуется изучить и применить алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, а также получить практические навыки реализации указанных алгоритмов.

Расстояния Левенштейна и Дамерау-Левенштейна применяется для следующих задач:

1) для автозамены, в том числе в поисковых системах;
2) в биоинформатике для сравнения цепочек белков, генов и т.д.

1 Аналитический раздел

1.1 Цель и задачи

Цель: реализовать и сравнить по эффективности алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна.

Задачи:

1) дать математическое описание расстояний Левенштейна и Дамерау-Левенштейна;
2) разработать алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна;
3) реализовать алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна;
4) провести эксперименты по замеру времени работы реализованных алгоритмов;
5) проанализировать реализованные алгоритмы по затраченному времени и максимально затраченной памяти.

1.2 Расстояние Левенштейна

Расстояние Левенштейна (или редакционное расстояние) – это минимальное количество редакционных операций, которое необходимо для преобразования одной строки в другую.

Редакционными операциями являются:

1) вставка (I – Insert);
2) удаление (D – Delete);
3) замена (R – Replace);

Также есть операция совпадение (M – Match). Операции I, D, R имеют штраф 1, а операция M – 0.

Пусть s_1 и s_2 — две строки (длиной M и N соответственно) в некотором алфавите V, тогда расстояние Левенштейна можно подсчитать по рекуррентной формуле (1.1):

$$D(s1[1..i],s2[1..j]) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min(s1[1..i],s2[1..j-1]) + 1, & \\ D(s1[1..i-1],s2[1..j]) + 1, & j > 0, i > 0 \\ D(s1[1..i-1],s2[1..j-1]) + M(s1[i], s2[j]), & \end{cases} \quad (1.1)$$

где $s[1..k]$ - подстрока длиной k и $M(a, b) = \begin{cases} 0, & a = b \\ 1, & a \neq b \end{cases}$

В Таблице 1.1 минимальное расстояние между строками "кот"и "скат"равно 2. Последовательность редакторских операций, которая привела к ответу - IMRM.

Таблица 1.1 — Пример работы преобразования строки "кот"в "скат"

	λ	С	К	А	Т
λ	0	1	2	3	4
К	1	1	1	2	3
О	2	2	2	2	3
Т	3	3	3	3	2

1.3 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, добавлена операция транспозиции (перестановки двух соседних символов) (X - exchange). Операция транспозиции возможна, если символы попарно совпадают.

Пусть s_1 и s_2 — две строки (длиной M и N соответственно) в некотором алфавите V, тогда расстояние Дамерау-Левенштейна можно подсчитать по рекуррентной формуле (1.2):

$$D(s1[1..i],s2[1..j]) = \begin{cases} max(i,j), & min(i,j) = 0 \\ min \begin{cases} s1[1..i], s2[1..j-1] + 1 \\ s1[1..i-1], s2[1..j] + 1 \\ s1[1..i-2], s2[1..j-2] + 1 \\ s1[1..i-1], s2[1..j-1] + M(s[i], s[j]) \end{cases} & i, j > 1, s1_{i-1} = s2_j, s1_i = s2_{j-1} \\ min \begin{cases} s1[1..i], s2[1..j-1] + 1 \\ s1[1..i-1], s2[1..j]) + 1 \\ s1[1..i-1], s2[1..j-1] + M(s[i], s[j]) \end{cases} & \text{иначе.} \end{cases} \tag{1.2}$$

2 Констукторский раздел

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

2.1 Разработка 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).

2.2 Требования к функциональности ПО

В данной работе требуется обеспечить следующую минимальную функциональность консольного приложения.

1) Режим ввода:
   а) возможность считать две строки;
   б) вывод расстояний Левенштейна и Дамерау-Левенштейна между строками;
   в) вывод матриц, используемых в вычислении расстояний (если использовались).
2) Экспериментальный режим:
   а) вывод таблицы с процессорным временем работы (Рисунок 4.1).

По умолчанию приложение работает в режиме ввода, для перехода в режим тестирования необходимо указать ключ -t при запуске.

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.

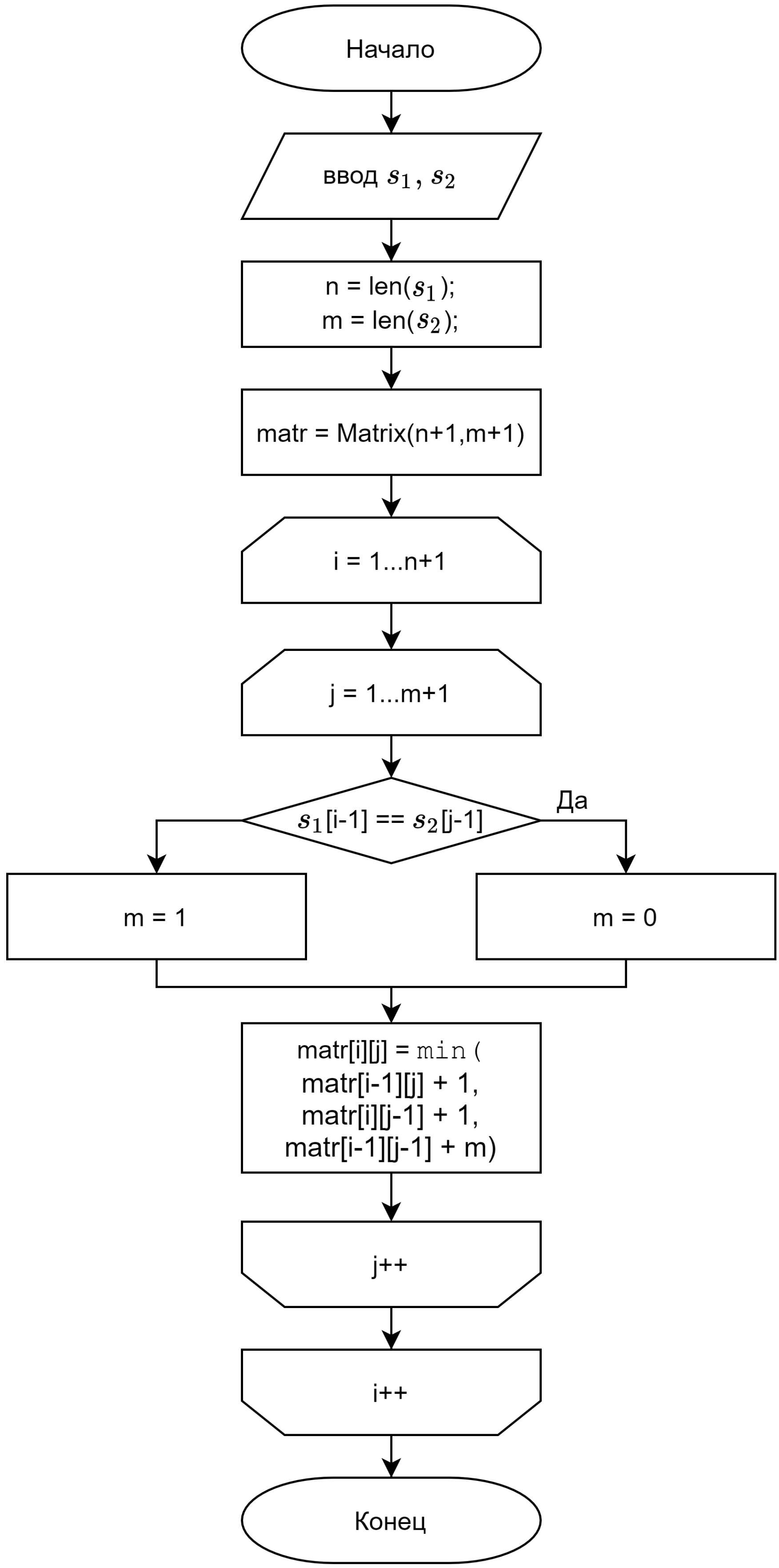

Рисунок 2.1 — Схема нерекурсивного поиска с заполнением матрицы

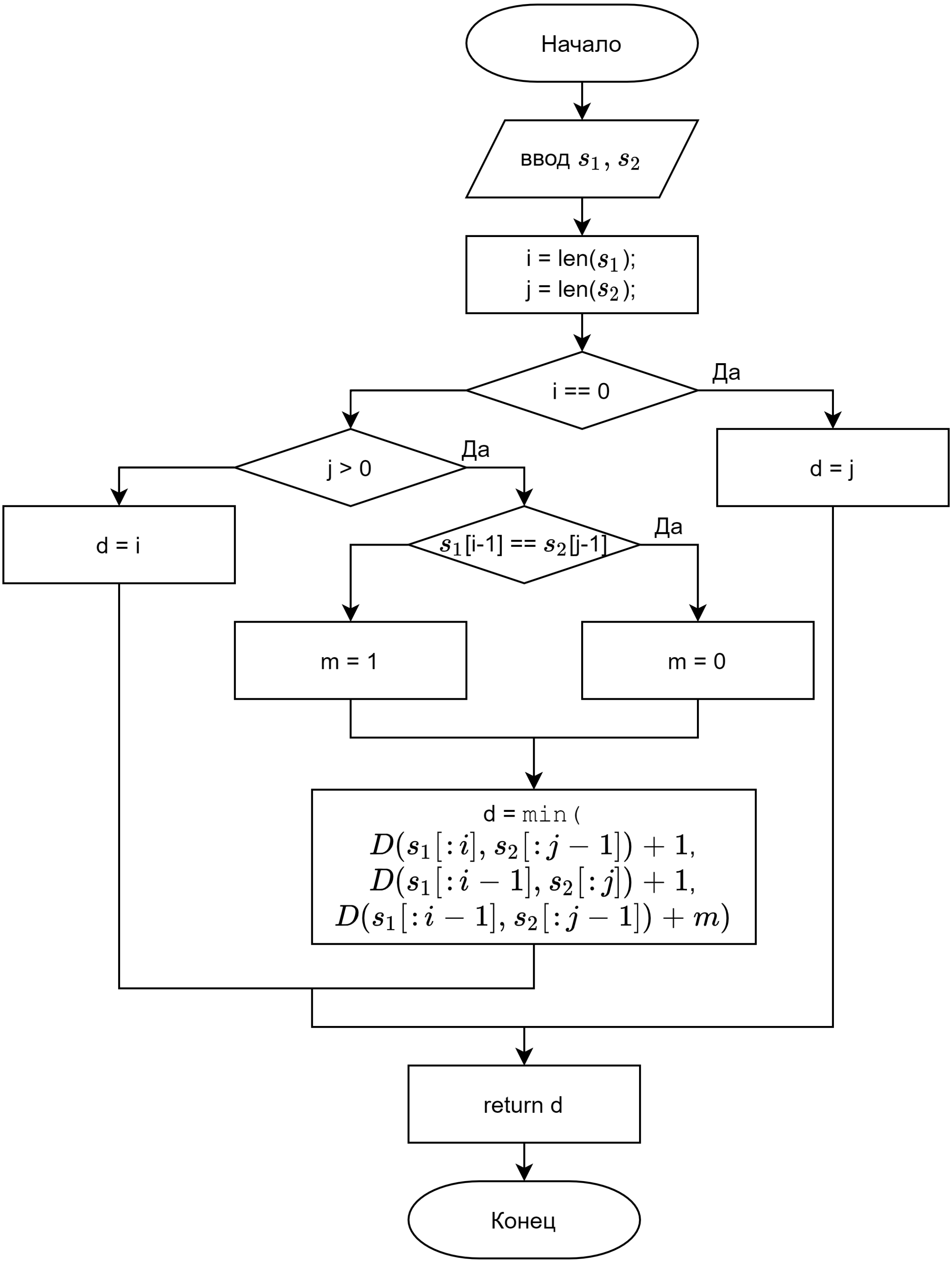

Рисунок 2.2 — Схема рекурсивого поиска без заполнения матрицы

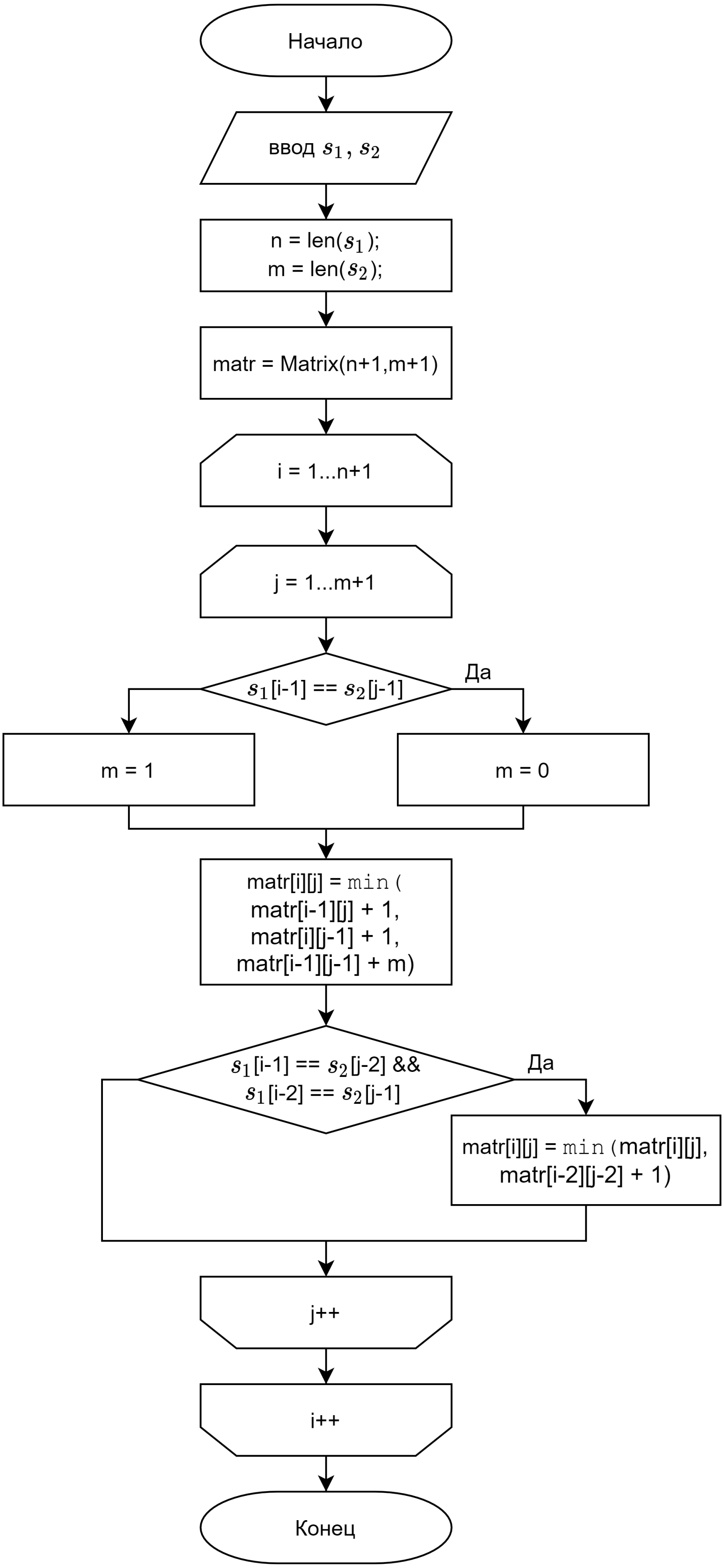

Рисунок 2.3 — Схема рекурсивого поиска с заполнением матрицы

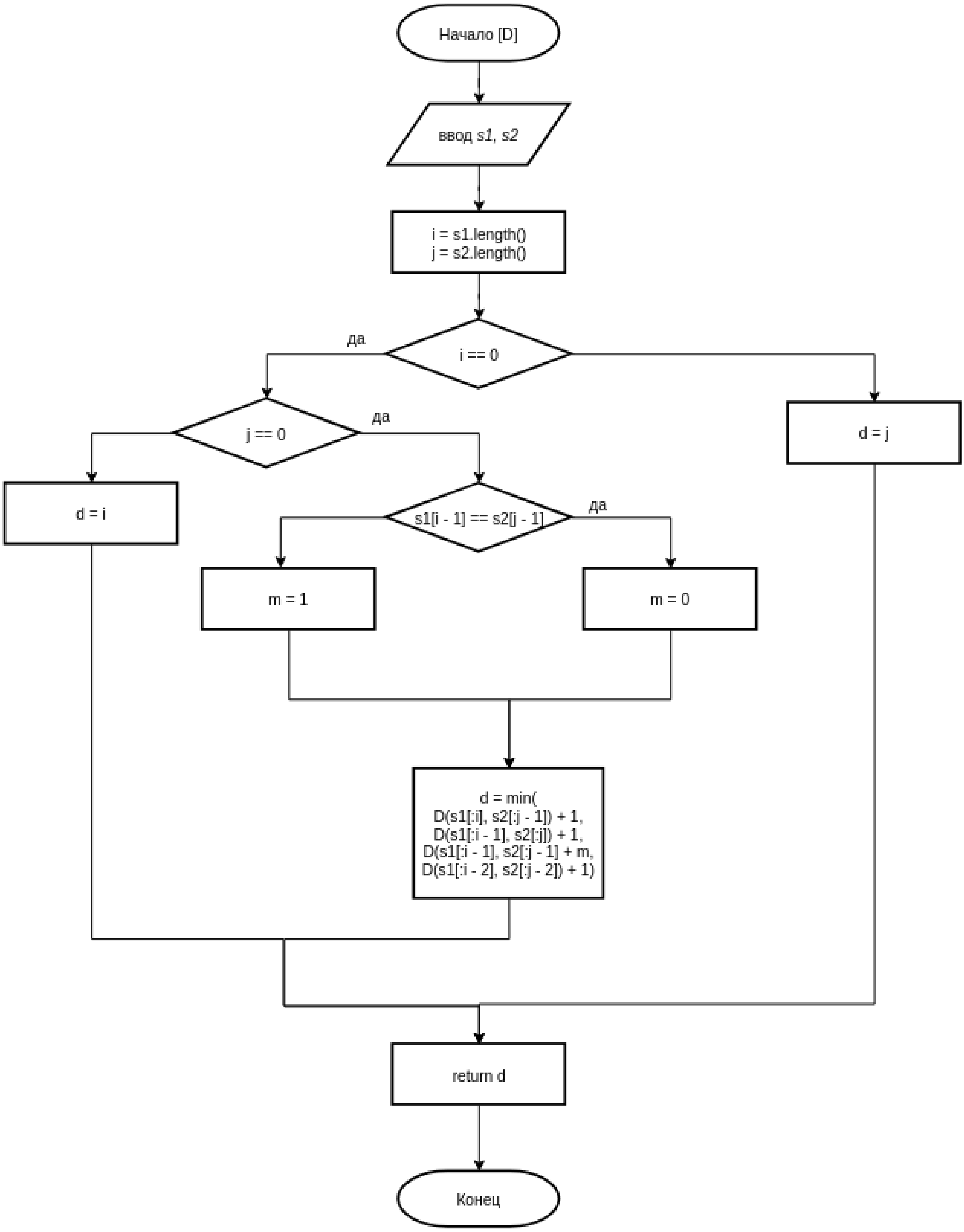

Рисунок 2.4 — Схема поиска р. Дамерау-Левенштейна

3 Технологический раздел

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

3.1 Средства реализации

В данной работе используется язык программирования C++, так как язык позволяет написать программу, работающую относительно быстро. Проект выполнен в IDE CLion.

Для замера процессорного времени была использована функция getCPUTime реализованная David Robert Nadeau для Linux и Windows , использование которой представлено в листинге 3.1.

Листинг 3.1 — Функция замера времени

```

/*
* 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>
#include <sys/resource.h>
#include <sys/times.h>
#include <time.h>

#else
#error "Unable to define getCPUTime( ) for an unknown OS."
#endif

/**
* Returns the amount of CPU time used by the current process,
* in seconds, or -1.0 if an error occurred.
*/
double getCPUTime( )
{
#if defined(_WIN32)
/* Windows -------------------------------------------------- */
FILETIME createTime;
FILETIME exitTime;
```

```
FILETIME kernelTime;
FILETIME userTime;
if ( GetProcessTimes( GetCurrentProcess( ),
&createTime, &exitTime, &kernelTime, &userTime ) != -1 )
{
SYSTEMTIME userSystemTime;
if ( FileTimeToSystemTime( &userTime, &userSystemTime ) != -1 )
return (double)userSystemTime.wHour * 3600.0 +
(double)userSystemTime.wMinute * 60.0 +
(double)userSystemTime.wSecond +
(double)userSystemTime.wMilliseconds / 1000.0;
}

#elif defined(__unix__) || defined(__unix) || defined(unix) || (defined(__APPLE__)
    && defined(__MACH__))
/* AIX, BSD, Cygwin, HP-UX, Linux, OSX, and Solaris --------- */

#if defined(_POSIX_TIMERS) && (_POSIX_TIMERS > 0)
/* Prefer high-res POSIX timers, when available. */
{
clockid_t id;
struct timespec ts;
#if _POSIX_CPUTIME > 0
/* Clock ids vary by OS.  Query the id, if possible. */
if ( clock_getcpuclockid( 0, &id ) == -1 )
#endif
#if defined(CLOCK_PROCESS_CPUTIME_ID)
/* Use known clock id for AIX, Linux, or Solaris. */
id = CLOCK_PROCESS_CPUTIME_ID;
#elif defined(CLOCK_VIRTUAL)
/* Use known clock id for BSD or HP-UX. */
id = CLOCK_VIRTUAL;
#else
id = (clockid_t)-1;
#endif
if ( id != (clockid_t)-1 && clock_gettime( id, &ts ) != -1 )
return (double)ts.tv_sec +
(double)ts.tv_nsec / 1000000000.0;
}
#endif

#if defined(RUSAGE_SELF)
{
struct rusage rusage;
if ( getrusage( RUSAGE_SELF, &rusage ) != -1 )
return (double)rusage.ru_utime.tv_sec +
(double)rusage.ru_utime.tv_usec / 1000000.0;
```

```
}
#endif

#if defined(_SC_CLK_TCK)
{
const double ticks = (double)sysconf( _SC_CLK_TCK );
struct tms tms;
if ( times( &tms ) != (clock_t)-1 )
return (double)tms.tms_utime / ticks;
}
#endif

#if defined(CLOCKS_PER_SEC)
{
clock_t cl = clock( );
if ( cl != (clock_t)-1 )
return (double)cl / (double)CLOCKS_PER_SEC;
}
#endif

#endif

return -1;      /* Failed. */
}
```

3.2 Листинг программы

Ниже представлены листинги кода поиска растояния Левенштейна:

1) нерекурсивного с заполнением матрицы (листинг 3.2);
2) рекурсивного без заполнения матрицы (листинг 3.3);
3) рекурсивного с заполнением матрицы (листинг 3.4);

и код функции поиска растояния Дамерау-Левенштейна (листинг 3.5).

Листинг 3.2 — Функция нерекурсивного поиска с заполнением матрицы

```
size_t Levenshtein::withoutRecursion(const std::string& str1, const std::string&
    str2, bool debugMode) {
    auto len1 = str1.length();
    auto len2 = str2.length();
    auto matrix = Matrix(len1 + 1, len2 + 1);

    for (size_t i = 0; i <= len1; i++)
        matrix[i][0] = i;
    for (size_t j = 0; j <= len2; j++)
```

```
        matrix[0][j] = j;

    for (auto i = 1; i <= len1; i++)
    {
        for (auto j = 1; j <= len2; j++)
        {
            auto m = 1;
            if (str1[i - 1] == str2[j - 1])
                m = 0;

            matrix[i][j] = std::min(matrix[i - 1][j] + 1,
            std::min(matrix[i][j - 1] + 1,
            matrix[i - 1][j - 1] + m));
        }
    }
    if (debugMode) {
        std::cout << "Levenshtein without recursion" << std::endl;
        matrix.print();
    }
    return matrix[len1][len2];
}
```

Листинг 3.3 — Функция рекурсивного поиска без заполнения матрицы

```
size_t Levenshtein::_recursionWithoutCash(const char* str1, size_t i, const char*
    str2, size_t j) {
    size_t res;
    if (i)
    {
        if (j)
        {
            auto m = 1;
            if (str1[i - 1] == str2[j - 1])
            m = 0;

            auto a = _recursionWithoutCash(str1, i, str2, j - 1) + 1;
            auto b = _recursionWithoutCash(str1, i - 1, str2, j) + 1;
            auto c = _recursionWithoutCash(str1, i - 1, str2, j - 1) + m;
            res = std::min(a, std::min(b, c));
        }
        else
            res = i;
    }
    else
        res = j;
    return res;
```

```
}

size_t Levenshtein::recursionWithoutCash(const std::string &str1, const std::string
    &str2) {
    auto i = str1.length();
    auto j = str2.length();
    return _recursionWithoutCash(str1.data(), i, str2.data(), j);
}
```

Листинг 3.4 — Функция рекурсивного поиска с заполнением матрицы

```

size_t Levenshtein::recursionWithCash(const std::string &str1, const std::string
    &str2, bool debugMode) {
    auto n = str1.length();
    auto m = str2.length();

    auto matrix = Matrix(n + 1, m + 1);
    for (size_t i = 0; i <= n; i++)
    {
        for (size_t j = 0; j <= m; j++)
            matrix[i][j] = -1;
    }

    auto res = _recursionWithCash(str1.data(), n, str2.data(), m, matrix);
    if (debugMode) {
        std::cout << "Levenshtein recursion with cash" << std::endl;
        matrix.print();
    }
    return res;
}

size_t Levenshtein::_recursionWithCash(const char* str1, size_t i, const char* str2,
    size_t j, Matrix& matrix) {

    if (!i)
        matrix[i][j] = j;
    else if (!j)
        matrix[i][j] = i;
    else
    {
        if (matrix[i][j - 1] == -1)
            matrix[i][j - 1] = _recursionWithCash(str1, i, str2, j - 1, matrix);
        if (matrix[i - 1][j] == -1)
            matrix[i - 1][j] = _recursionWithCash(str1, i - 1, str2, j, matrix);
        if (matrix[i - 1][j - 1] == -1)
```

```
            matrix[i - 1][j - 1] = _recursionWithCash(str1, i - 1, str2, j - 1,
                matrix);
        matrix[i][j] = std::min(std::min(matrix[i][j - 1], matrix[i - 1][j]) + 1,
                                matrix[i - 1][j - 1] + (str1[i - 1] != str2[j - 1]));
    }

    return matrix[i][j];
}
```

Листинг 3.5 — Функция поиска растояния Дамерау-Левенштейна

```

size_t Levenshtein::recursionDamerauWithoutCash(const std::string &str1, const
    std::string &str2) {
    auto i = str1.length();
    auto j = str2.length();

    return _recursionDamerauWithoutCash(str1.data(), i, str2.data(), j);
}

size_t Levenshtein::_recursionDamerauWithoutCash(const char* str1, size_t i, const
    char* str2, size_t j) {
    size_t res;
    if (i)
    {
        if (j)
        {
            auto m = 1;
            if (str1[i - 1] == str2[j - 1])
                m = 0;

            auto a = _recursionWithoutCash(str1, i, str2, j - 1) + 1;
            auto b = _recursionWithoutCash(str1, i - 1, str2, j) + 1;
            auto c = _recursionWithoutCash(str1, i - 1, str2, j - 1) + m;
            res = std::min(a, std::min(b, c));
            if (i > 1 && j > 1 && str1[i - 1] == str2[j - 2] && str1[i - 2] ==
                str2[j - 1])
                res = std::min(res, _recursionWithoutCash(str1, i - 2, str2, j - 2)
                    + 1);
        }
        else
            res = i;
    }
    else
        res = j;
    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 (р.Л, р.Д-Л)	Фактический результат (р.Л, р.Д-Л)
1	0	0	0, 0	0, 0
2	0	ab	2, 2	2, 2
3	abba	baab	3, 2	3, 2
4	abcd	qwer	4, 4	4, 4

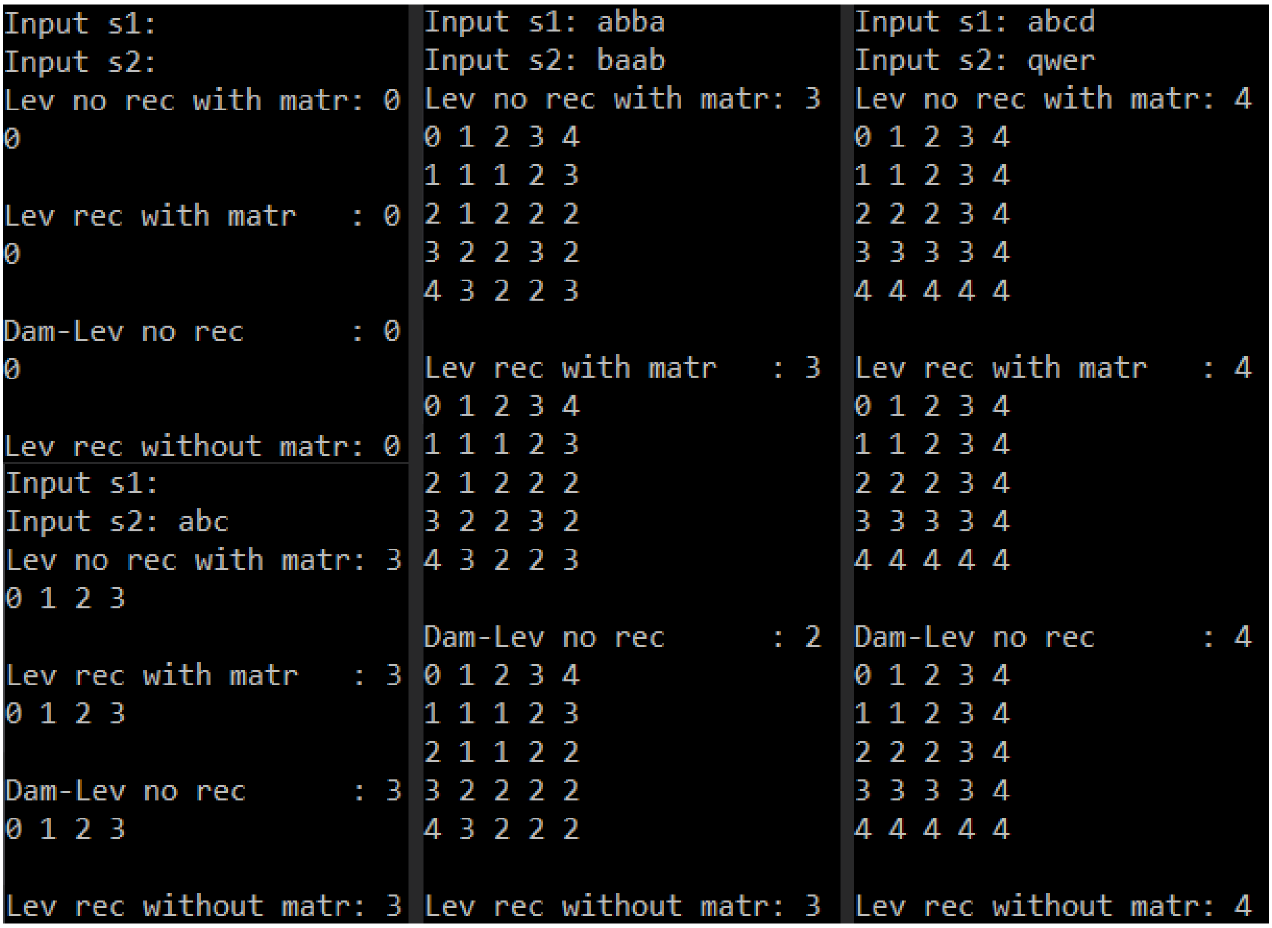

Рисунок 3.1 — Результаты тестирования

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):

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

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):

$$V = 2sizeof(char*) + 6sizeof(size\_t)) \tag{3.2}$$

4 Экспериментальный раздел

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

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

В рамках данного проекта были проведёны следующие эксперименты:

1) сравнение алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна на строках длиной от 0 до 5 с шагом 1 (рисунок 4.1);

2) сравнение алгоритмов [1] поиска расстояния Левенштейна и Дамерау-Левенштейна на строках длиной от 0 до 1000 с шагом 50 (рисунок 4.2).

Эксперимент проводился на стационаном компьютере с процессором Intel(R) Core(TM) i5-7200U CPU 2.50 GHz под управлением Debian 10 с 8 Гб оперативной памяти.

Ниже представлены графики зависимости времени работы алгоритмов от длины входных строк (рисунки 4.3 и 4.4).

Обозначения:

a) levNR - алгоритм Левенштейна без рекурсии.

b) levRNC - алгоритм Левенштейна с рекурсией и без кэша.

c) levRWC - алгоритм Левенштейна с рекурсией и с кэшем.

d) levDamR - алгоритм Дамерау-Левенштейна с рекурсией.

4.2 Вывод

В данном разделе были поставлены эксперименты по замеру времени выполнения каждого из алгоритмов. По итогам замеров не рекурсивный алгоритм нахождения расстояния Левенштейна оказался самым быстродействующим на длинах строк превышающих 3 на 136 % быстрее, чем алгоритм поиска расстояния Левенштейна рекурсивно с заполнением матрицы и на 42 %, чем реализация алгоритм поиска расстояния Дамерау-Левенштейна. На строках длиной менее 3х символов рекурсивная реализиция быстрее матричной, так как не выделяет в куче место под хранение матрицы.

По расходу памяти матричные алгоритмы проигрывают рекурсивному, так как максимальный размер используемой памяти имеет квадратичную ассимптотику (произведение длин строк), в то время как у рекурсивного - линейная (сумма длин строк).

[1] Замеры времени для рекурсивного алгоритма поиска расстояния Левенштейна на строках длиной от 0 до 975 с шагом 75 не проводились, так как уже на строках длиной 10 алгоритм работает 70 034 ms, что в 35 000 раз больше, времени работы алгоритмов с использованием матрицы. Это связано с экспоненциальной асимптотикой времени выполнения данного алгоритма (пропорционально количеству рекурсивных вызовов).

N - no
W - with
R - recursion
C - cash

str_len	levNR	levRNC	levRWC	levDamR
0	1.2962e-05	4.14e-07	7.67e-07	4.02e-07
1	1.002e-06	4.89e-07	8.32e-07	4.67e-07
2	8.86e-07	6.28e-07	8.37e-07	6.18e-07
3	1.174e-06	1.324e-06	1.17e-06	1.178e-06
4	1.403e-06	3.213e-06	1.401e-06	2.972e-06
5	1.713e-06	2.991e-05	1.705e-06	2.1366e-05

Рисунок 4.1 — Результаты замера времени на строках длиной от 0 до 5 в секундах

N - no
W - with
R - recursion
C - cash

str_len	levNR	levRNC	levRWC	levDamR
0	2.661e-06	4e-07	7.18e-07	4.96e-07
75	0.000167288	-1	0.000228132	-1
150	0.000627072	-1	0.000944667	-1
225	0.00134542	-1	0.00213462	-1
300	0.00246606	-1	0.00494261	-1
375	0.00495264	-1	0.00684413	-1
450	0.00539339	-1	0.00866721	-1
525	0.00732033	-1	0.0119194	-1
600	0.00903693	-1	0.016115	-1
675	0.0119257	-1	0.0209858	-1
750	0.0148076	-1	0.0269556	-1
825	0.0175376	-1	0.0331788	-1
900	0.0215238	-1	0.0392752	-1
975	0.0240596	-1	0.0441632	-1

Рисунок 4.2 — Результаты замера времени на строках длиной от 0 до 975 в секундах

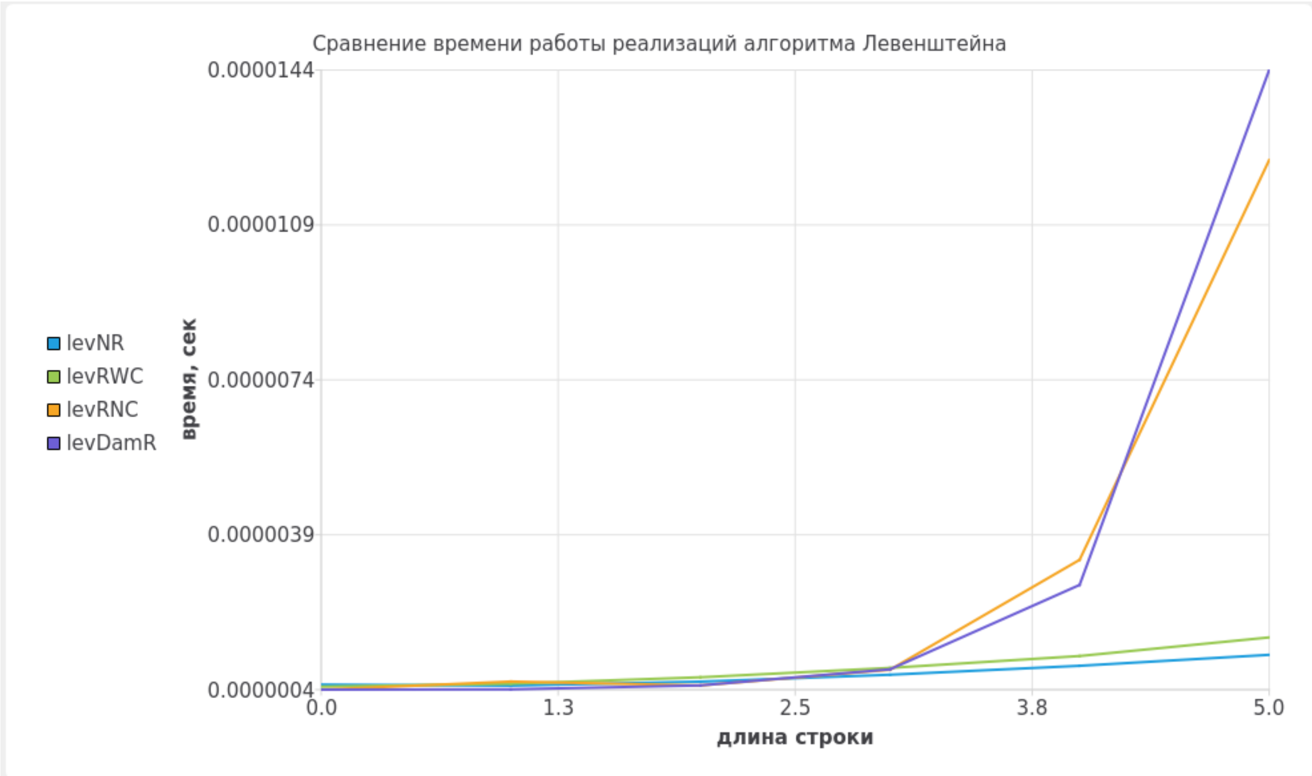

Рисунок 4.3 — График зависимости времени работы алгоритмов от длин строк

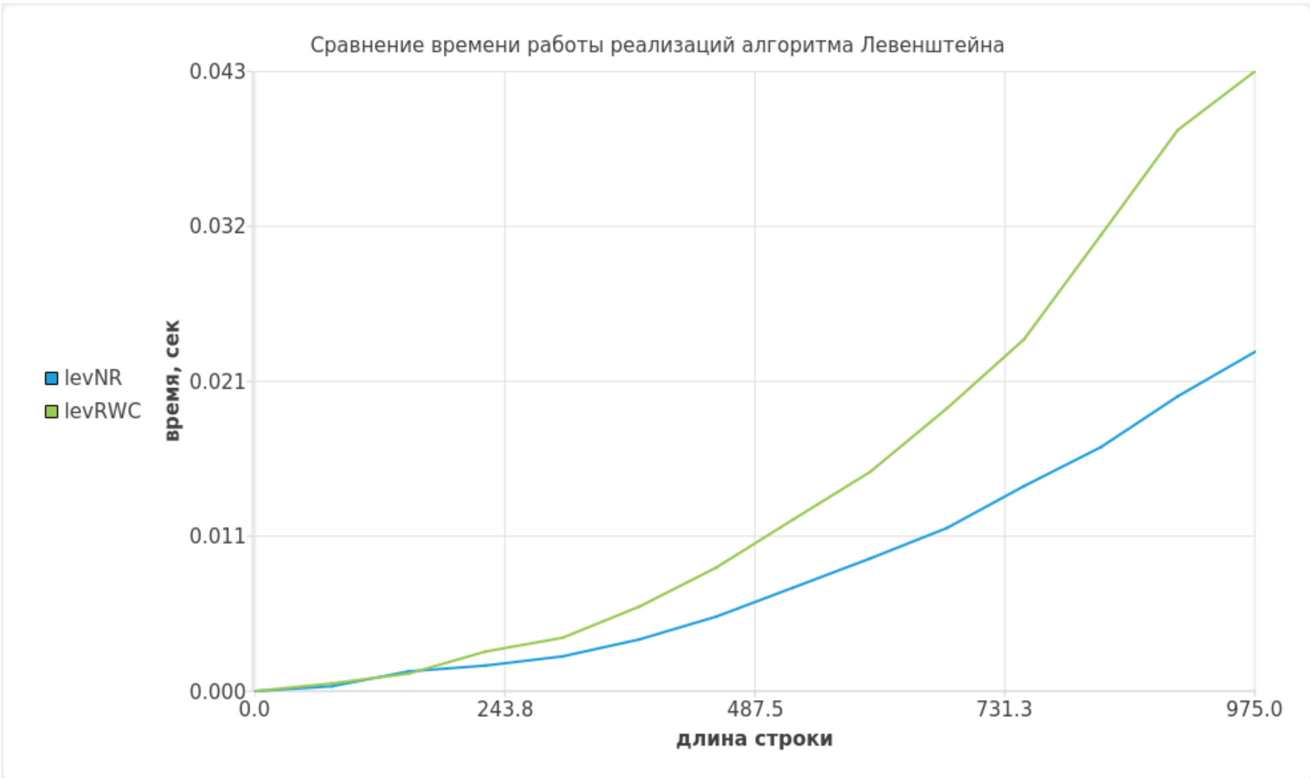

Рисунок 4.4 — График зависимости времени работы алгоритмов от длин строк

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). Выполнено сравнение перечисленных алгоритмов.

В ходе экспериментов по замеру времени работы было установлено, что не рекурсивный алгоритм нахождения расстояния Левенштейна на длинах строк превышающих 3 на 136 % быстрее, чем алгоритм поиска расстояния Левенштейна рекурсивно с заполнением матрицы и на 42 %, чем реализация алгоритм поиска расстояния Дамерау-Левенштейна. На строках длиной менее 3 символов рекурсивная реализиция работает бысрее матричной, так как не выделяет в куче место под хранение матрицы.

Из теоритического анализа максимальной затрачиваемой памяти каждым из алгоритмов представленым в технологической части можно сделать вывод, что реализации с использованием матриц занимают намного больше памяти при обработке длинных строк, чем рекурсивная реализация, так при длине строк 1000 символов, рекурсивный алгоритм теоритически использует в 95.5 раз меньше памяти, чем остальные.

Список использованных источников

1) GetCpuTime. // [Электронный ресурс]. Режим доступа: https://gist.github.com/Randl/45bcca59720f661fa033a67d5f44bff0, (дата обращения: 11.09.2021).

2) IDE CLion 2021. // [Электронный ресурс]. Режим доступа: https://www.jetbrains.com/clion/, (дата обращения: 11.09.2021).

3) processor-i5-7200u. // [Электронный ресурс]. Режим доступа: https://www.intel.com/content/www/us/en/products/processors/core/i5-processors/i5-7200u.html, (дата обращения: 11.09.2021).