Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчёт

по лабораторной работе №1

Название: Расстояния Левенштейна и Дамерау-Левенштейна

Дисциплина: Анализ алгоритмов

Студент	ИУ7-55Б		М.А. Козлов
	(Группа)	(Подпись, дата)	(И.О. Фамилия)
Преподователь			Л.Л. Волкова
		(Подпись, дата)	(И.О. Фамилия)

Москва, 2020

Содержание

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 практики

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);
4) 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):

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

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 словом "кит"и "скат"равно 2. Последовательность редакторских операций, которая привела к ответу - IMRM.

Таблица 1.1 — Пример работы преобразования слова "кит"в "скат"

	λ	С	К	А	Т
λ	0	1	2	3	4
К	1	1	1	2	3
И	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):

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

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 [1] работы.

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.

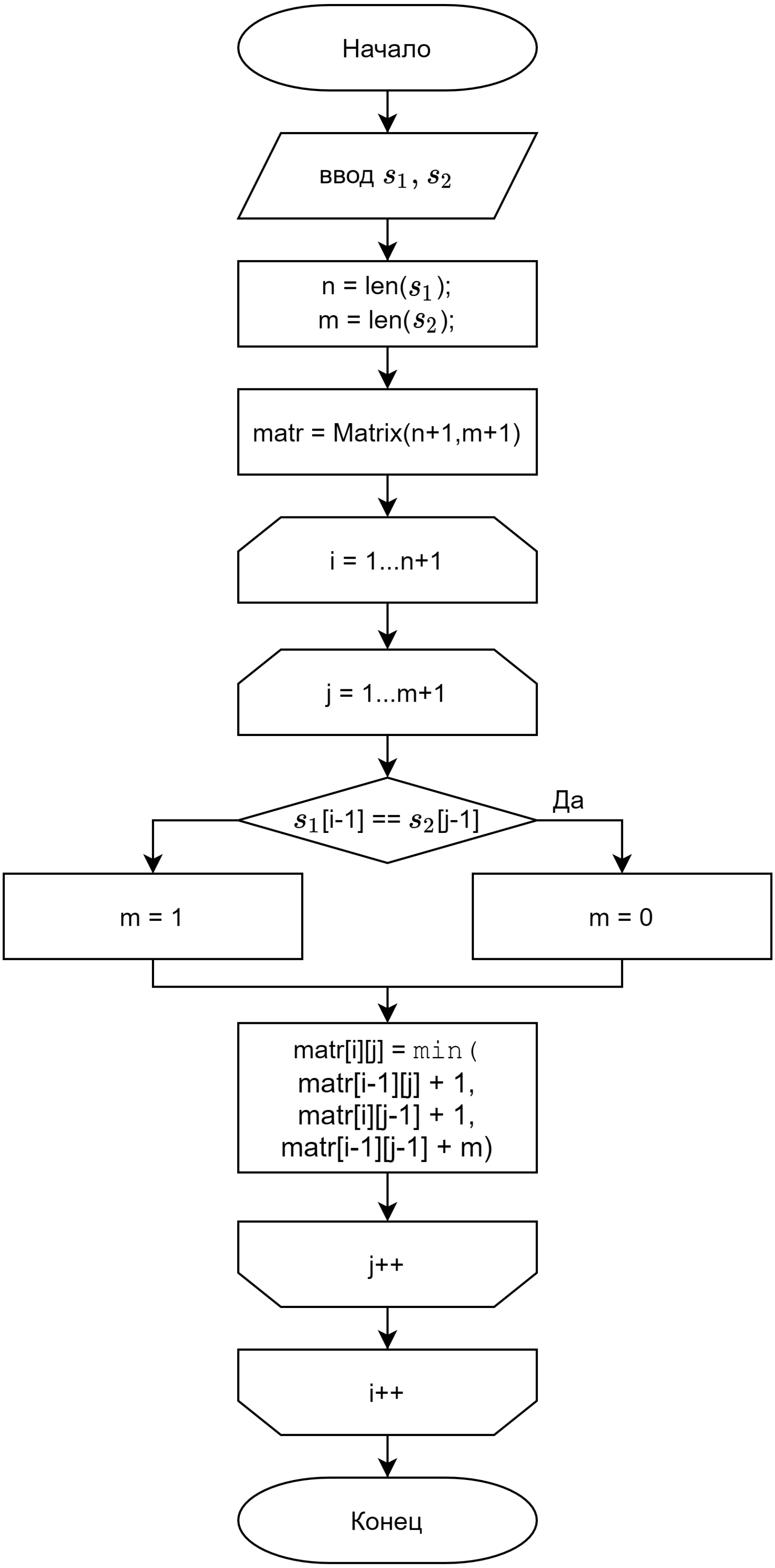


Рисунок 2.1 — Схема нерекурсивного поиска с заполнением матрицы

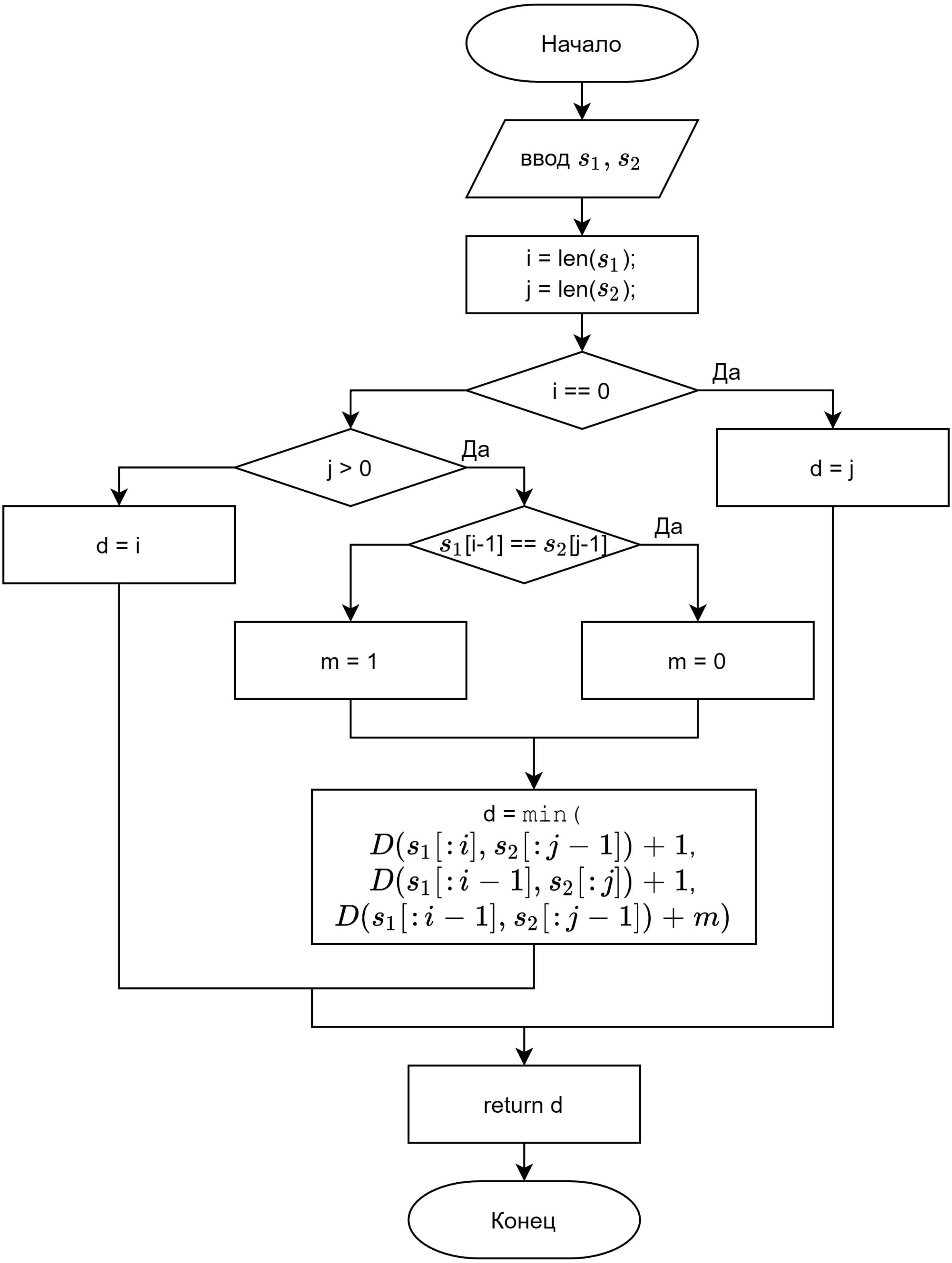


Рисунок 2.2 — Схема рекурсивого поиска без заполнения матрицы

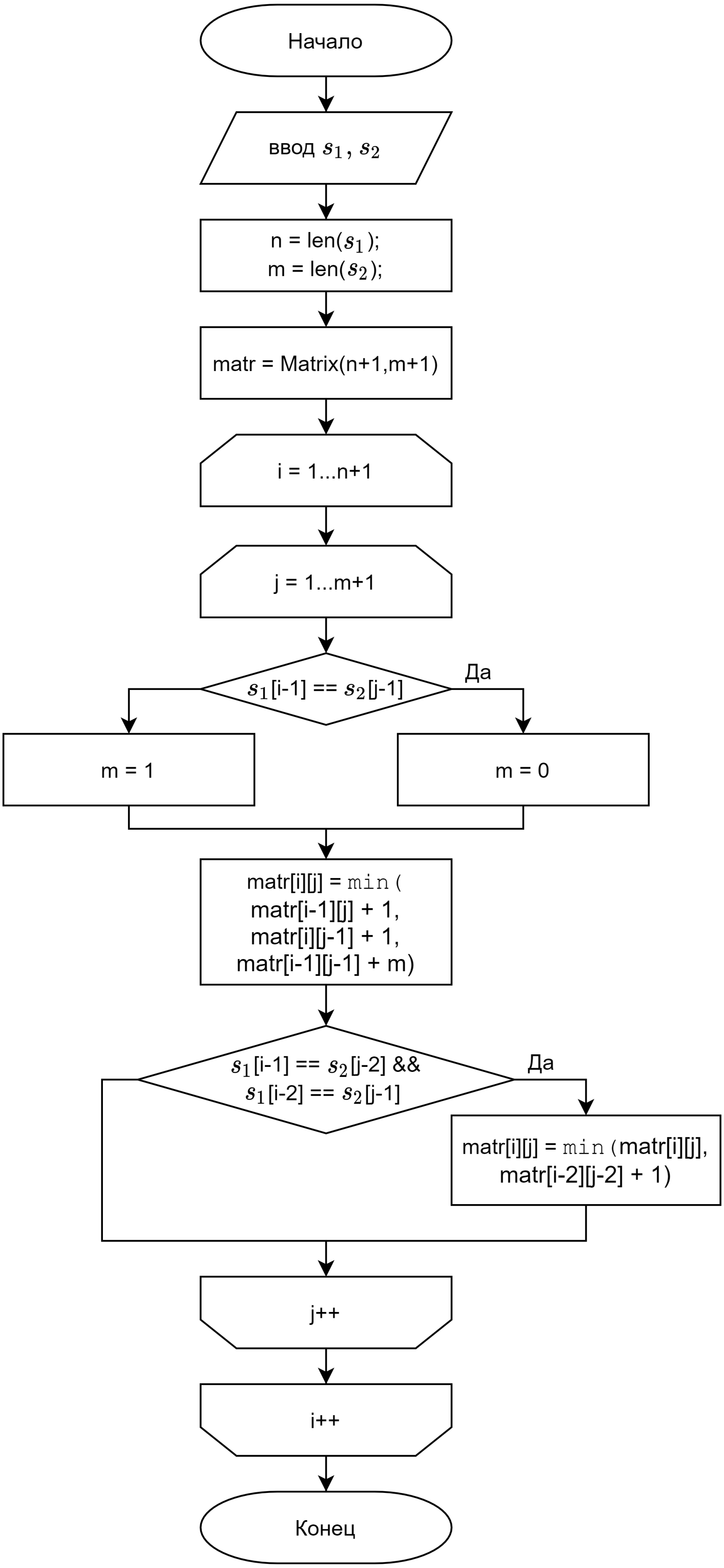


Рисунок 2.3 — Схема рекурсивого поиска с заполнением матрицы

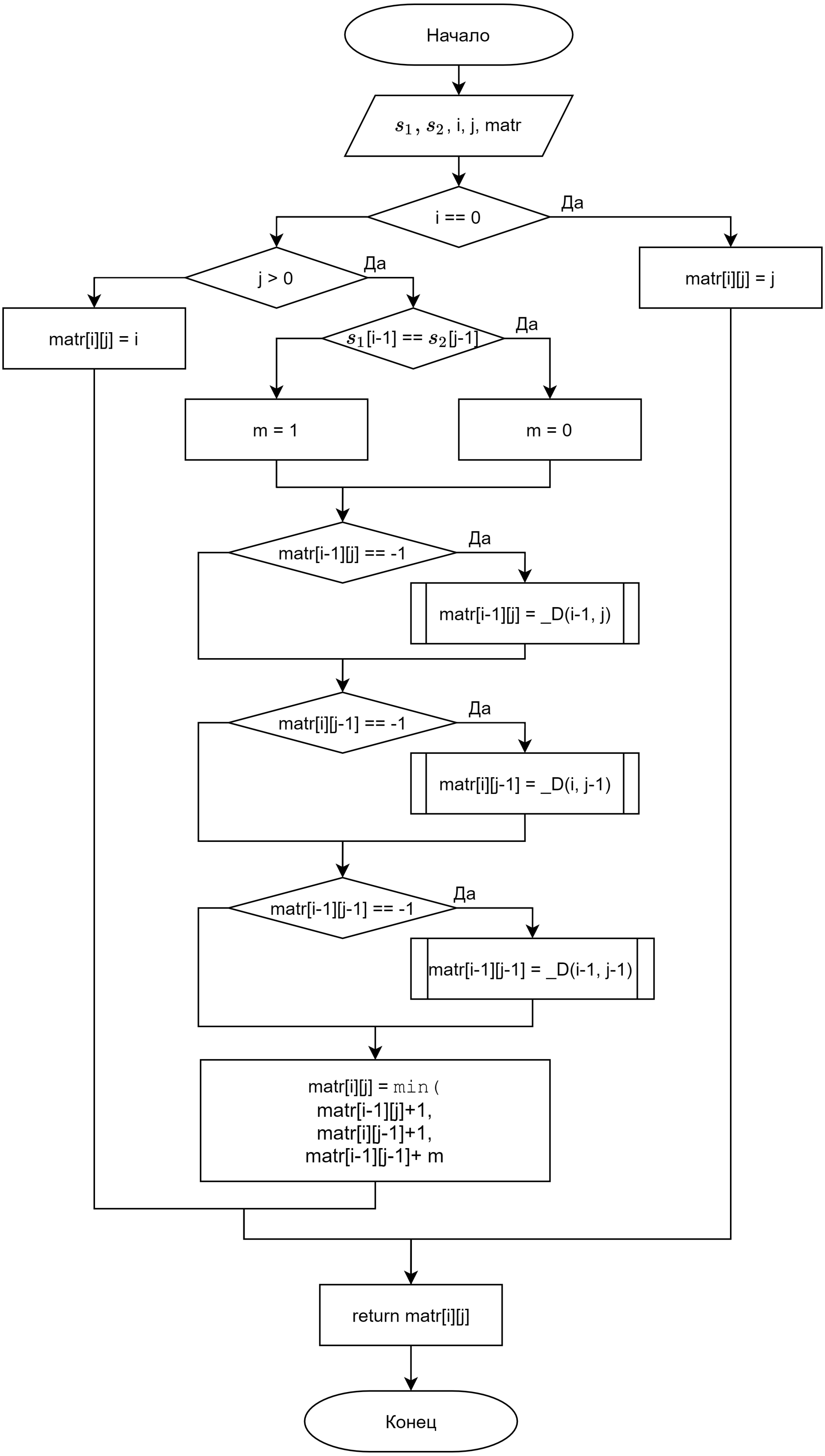


Рисунок 2.4 — Схема поиска р. Дамерау-Левенштейна

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 Visual Studio 2019 [2].

Для замера процессорного времени была использована функция QueryPerformanceCounter [3] из библиотеки WinAPI, использование которой представлено в листинге 3.1.

Листинг 3.1 — Функция замера времени

```
using funcDL = std::size_t( * )(const char*, const char* );
double getTime(funcDL getDL, const char* s1, const char* s2, int samples)
{
    LARGE_INTEGER StartingTime, EndingTime, ElapsedMicroseconds;
    LARGE_INTEGER Frequency;

    QueryPerformanceFrequency(&Frequency);
    QueryPerformanceCounter(&StartingTime);

    // Activity to be timed
    for (size_t i = 0; i < samples; i++)
        getDL(s1, s2);

    QueryPerformanceCounter(&EndingTime);
    ElapsedMicroseconds.QuadPart = EndingTime.QuadPart - StartingTime.QuadPart;

    ElapsedMicroseconds.QuadPart *= 1000000;
    ElapsedMicroseconds.QuadPart /= (Frequency.QuadPart * samples);
    return ElapsedMicroseconds.QuadPart;
}
```

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

```
std::size_t getLevMatr(const char* s1, const char* s2)
{
        auto n = strlen(s1);
        auto m = strlen(s2);
        auto matr = Matrix(n + 1, m + 1);

        for (size_t i = 0; i <= n; i++)
                matr[i][0] = i;

        for (size_t j = 1; j <= m; j++)
                matr[0][j] = j;

        for (size_t i = 1; i <= n; i++)
        {
                for (size_t j = 1; j <= m; j++)
                {
                        matr[i][j] = _min(_min(
                                matr[i - 1][j] + 1,
                                matr[i][j - 1] + 1),
                                matr[i - 1][j - 1] + (s1[i - 1] != s2[j - 1])
                        );
                }
        }

        return matr[n][m];
}
```

Листинг 3.3 — Функция рекурсивного поиска без заполнения матрицы

```
std::size_t getLevRec(const char* s1, const char* s2)
{
        std::size_t i = strlen(s1);
        std::size_t j = strlen(s2);
        return _getLevRec(s1, i, s2, j);
}
std::size_t _getLevRec(const char* s1, size_t i, const char* s2, size_t j)
{
    std::size_t d;
    if (i == 0)
        d = j;
    else if (j == 0)
        d = i;
    else
    {
        d = _min(_min(
            _getLevRec(s1, i, s2, j - 1) + 1,
            _getLevRec(s1, i - 1, s2, j) + 1),
```

```
            _getLevRec(s1, i - 1, s2, j - 1) + (s1[i - 1] != s2[j - 1])
        );
    }
    return d;
}
```

Листинг 3.4 — Функция рекурсивного поиска с заполнением матрицы

```
std::size_t getLevRecMatr(const char* s1, const char* s2)
{
    std::size_t n = strlen(s1);
    std::size_t m = strlen(s2);
    auto matr = Matrix(n + 1, m + 1);
    for (size_t i = 0; i < n + 1; i++)
        for (size_t j = 0; j < m + 1; j++)
            matr[i][j] = -1;
    return _getLevRecMatr(s1, n, s2, m, matr);
}
std::size_t _getLevRecMatr(const char* s1, size_t i, const char* s2, size_t j, Matrix& matr)
{
    if (i == 0)
        matr[i][j] = j;
    else if (j == 0)
        matr[i][j] = i;
    else
    {
        if (matr[i][j - 1] == -1)
            matr[i][j - 1] = _getLevRecMatr(s1, i, s2, j - 1, matr);
        if (matr[i - 1][j] == -1)
            matr[i - 1][j] = _getLevRecMatr(s1, i - 1, s2, j, matr);
        if (matr[i - 1][j - 1] == -1)
            matr[i - 1][j - 1] = _getLevRecMatr(s1, i - 1, s2, j - 1, matr);
        matr[i][j] = _min(_min(matr[i][j - 1], matr[i - 1][j]) + 1, matr[i - 1][j - 1] + (s1[i - 1] != s2[j - 1]));
    }

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

Листинг 3.5 — Функция поиска растояния Дамерау-Левенштейна

```
std::size_t getDamLevMatr(const char* s1, const char* s2)
{
    std::size_t n = strlen(s1);
    std::size_t m = strlen(s2);
    auto matr = Matrix(n + 1, m + 1);

```

```
    for (size_t i = 0; i <= n; i++)
        matr[i][0] = i;

    for (size_t j = 1; j <= m; j++)
        matr[0][j] = j;

    for (size_t i = 1; i <= n; i++)
    {
        for (size_t j = 1; j <= m; j++)
        {
            matr[i][j] = _min(_min(
                matr[i - 1][j] + 1,
                matr[i][j - 1] + 1),
                matr[i - 1][j - 1] + (s1[i - 1] != s2[j - 1]
            ));

            if (i > 1 && j > 1 && s1[i - 1] == s2[j - 2] && s1[i - 2] == s2[j - 1])
                matr[i][j] = _min(matr[i][j], matr[i - 2][j - 2] + 1);
        }
    }

    return matr[n][m];
}
```

3.3 Тестирование

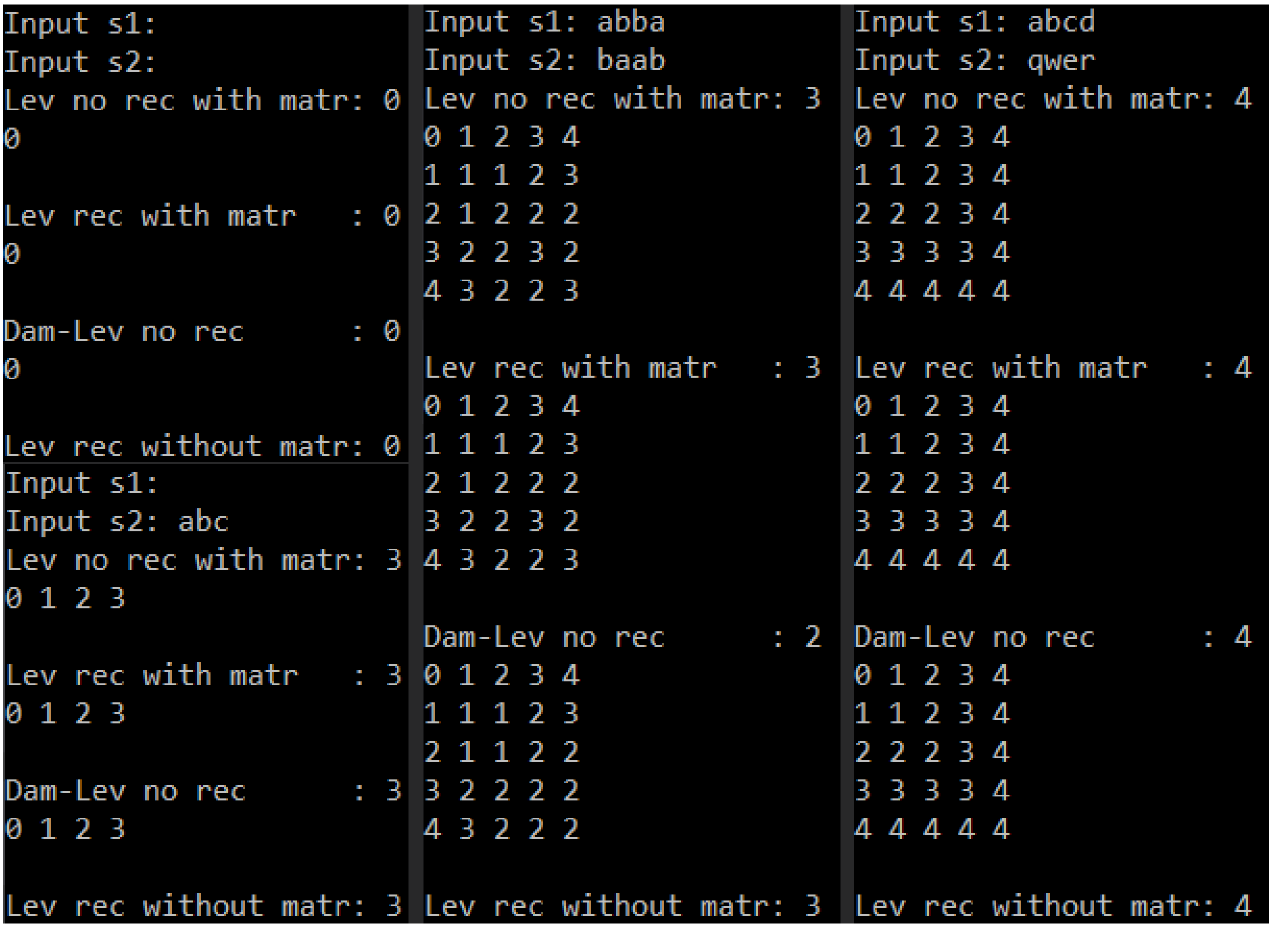
В таблице 3.1 отображён возможный набор тестов для тестирования методом чёрного ящика, результаты которого, представленные на рисунке 3.1, подтверждают прохождение программы перечисленных тестов.

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

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.


```
Input s1:
Input s2:
Lev no rec with matr: 0
0

Lev rec with matr   : 0
0

Dam-Lev no rec      : 0
0

Lev rec without matr: 0
```
```
Input s1:
Input s2: abc
Lev no rec with matr: 3
0 1 2 3

Lev rec with matr   : 3
0 1 2 3

Dam-Lev no rec      : 3
0 1 2 3

Lev rec without matr: 3
```
```
Input s1: abba
Input s2: baab
Lev no rec with matr: 3
0 1 2 3 4
1 1 1 2 3
2 1 2 2 2
3 2 2 3 2
4 3 2 2 3

Lev rec with matr   : 3
0 1 2 3 4
1 1 1 2 3
2 1 2 2 2
3 2 2 3 2
4 3 2 2 3

Dam-Lev no rec      : 2
0 1 2 3 4
1 1 1 2 3
2 1 1 2 2
3 2 2 2 2
4 3 2 2 2

Lev rec without matr: 3
```
```
Input s1: abcd
Input s2: qwer
Lev no rec with matr: 4
0 1 2 3 4
1 1 2 3 4
2 2 2 3 4
3 3 3 3 4
4 4 4 4 4

Lev rec with matr   : 4
0 1 2 3 4
1 1 2 3 4
2 2 2 3 4
3 3 3 3 4
4 4 4 4 4

Dam-Lev no rec      : 4
0 1 2 3 4
1 1 2 3 4
2 2 2 3 4
3 3 3 3 4
4 4 4 4 4

Lev rec without matr: 4
```

Рисунок 3.1 — Результаты тестирования

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

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

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

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

4 Экспериментальный раздел

В данном разделе будут проведены эксперименты для проведения сравнительного анализа алгоритмов по затрачиваемому процессорному времени[1] 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 4 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).

Тестирование проводилось на ноутбуке с процессором Intel(R) Core(TM) i5-7200U CPU 2.50 GHz [4] под управлением Windows 10 с 8 Гб оперативной памяти.

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

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 выигрывает матричные, 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 1000 с шагом 50 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).

	time (ms)			
len(str)	Dam-Lev no rec	Lev no rec with matr	Lev rec with matr	Lev rec without matr
0	0.272	0.291	0.298	0.011
1	0.337	0.302	0.322	0.028
2	0.440	0.332	0.374	0.112
3	15.172	0.572	0.669	0.498
4	0.785	0.743	0.886	2.573

Рисунок 4.1 — Результаты замера времени на строках длиной от 0 до 4

len(str)	Dam-Lev no rec	Lev no rec with matr	Lev rec with matr	Lev rec without matr
0	0.242	0.172	0.295	0.008
50	24.768	20.223	43.307	-1.000
100	96.617	62.595	159.110	-1.000
150	210.250	113.777	310.115	-1.000
200	359.725	213.259	479.764	-1.000
250	809.422	307.855	748.595	-1.000
300	1192.700	417.931	1120.427	-1.000
350	2010.168	540.600	1238.982	-1.000
400	2481.607	960.249	2110.764	-1.000
450	2799.885	1269.255	3245.515	-1.000
500	4121.502	1873.876	3545.871	-1.000
550	7042.492	2115.854	4672.413	-1.000
600	7004.781	2144.970	4966.896	-1.000
650	4965.338	2496.948	6002.317	-1.000
700	5071.567	2886.049	6820.962	-1.000
750	4921.851	3345.728	9194.316	-1.000
800	5869.040	3847.814	10393.363	-1.000
850	6376.064	4272.943	11300.614	-1.000
900	7905.439	6642.079	12616.023	-1.000
950	7861.776	5617.177	13579.861	-1.000
1000	9547.587	6176.626	18423.410	-1.000

Рисунок 4.2 — Результаты замера времени на строках длиной от 0 до 1000

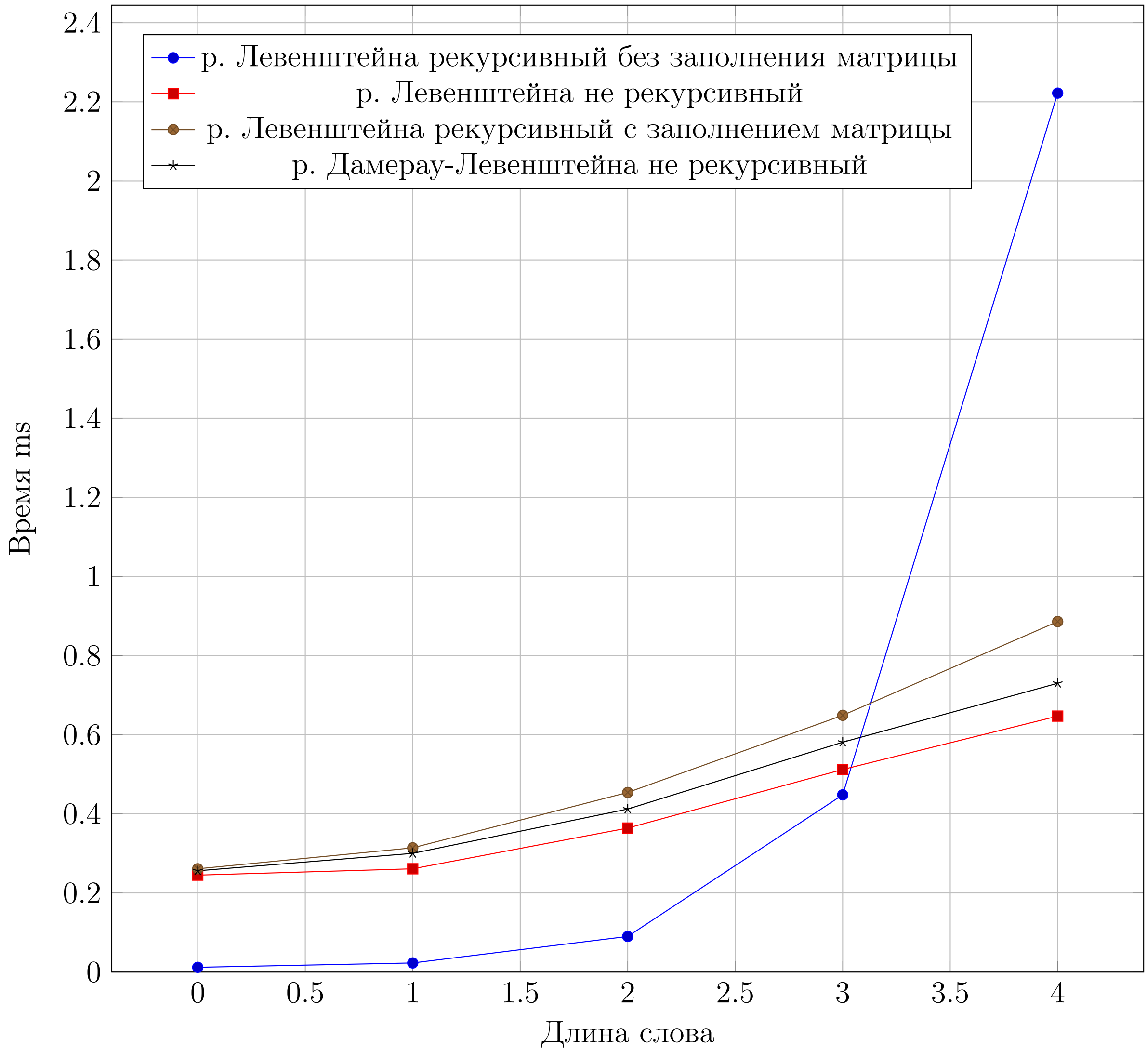


Рисунок 4.3 — График зависимости времени работы алгоритмов от длин строк

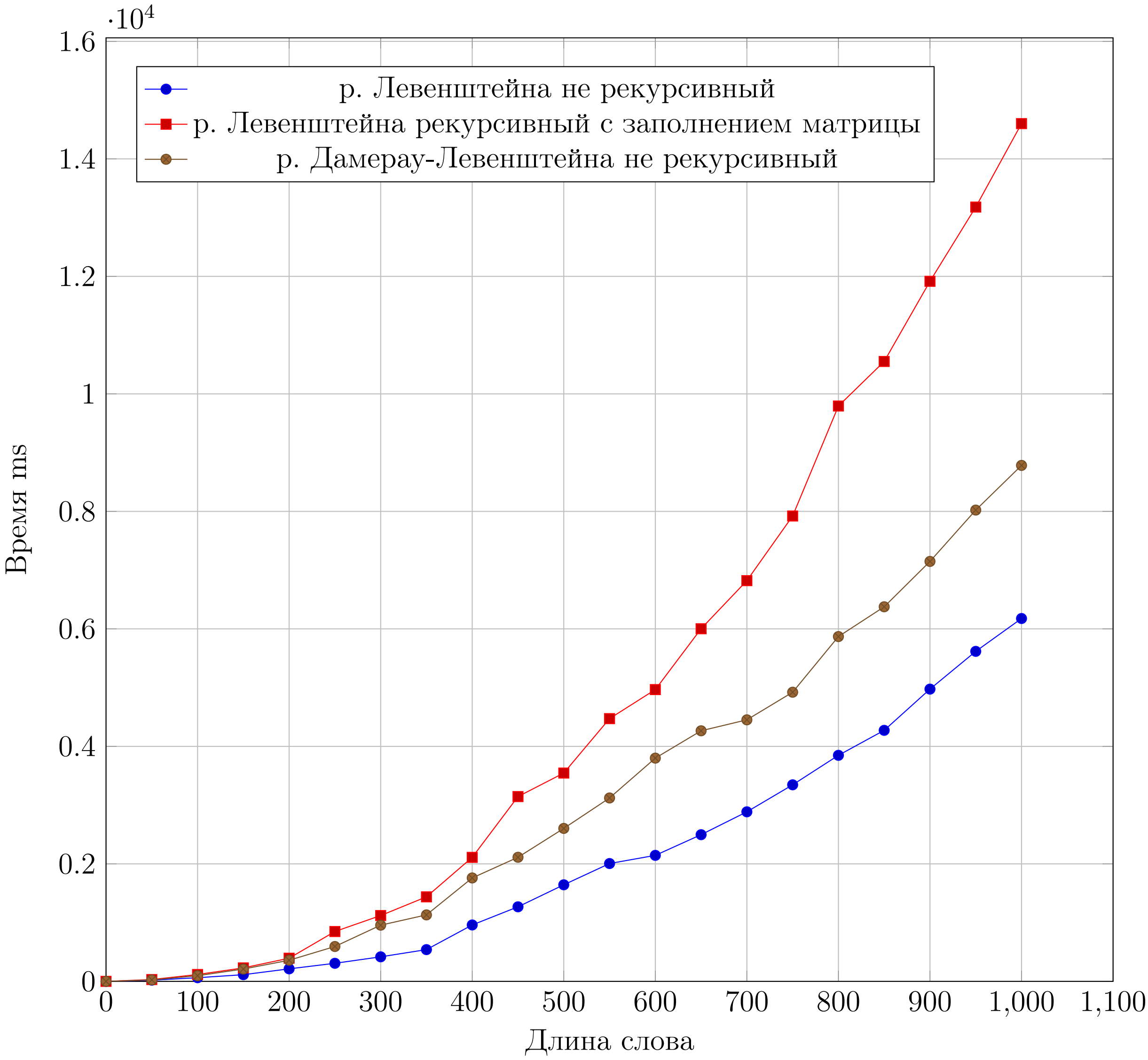


Рисунок 4.4 — График зависимости времени работы алгоритмов от длин строк

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 3х символов рекурсивная реализиция выигрывает матричные, 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. Time Basics. // [Электронный ресурс]. Режим доступа: https://www.gnu.org/software/libc/manual/html_node/Time-Basics.html, (дата обращения: 11.09.2020).

2. IDE Visual Studio 2019. // [Электронный ресурс]. Режим доступа: https://visualstudio.microsoft.com/ru/vs/, (дата обращения: 11.09.2020).

3. Acquiring high-resolution time stamps. // [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/windows/win32/sysinfo/acquiring-high-resolution-time-stamps?redirectedfrom=MSDN, (дата обращения: 11.09.2020).

4. Intel® Core™ i5-7200U Processor. // [Электронный ресурс]. Режим доступа: https://www.intel.com/content/www/us/en/products/processors/core/i5-processors/i5-7200u.html, (дата обращения: 26.09.2020).