**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»

# Лабораторная работа №1 по дисциплине "Анализ Алгоритмов"

**Тема** Расстояние Левенштейна и Дамерау-Левенштейна

**Студент** Светличная А.А.

**Группа** ИУ7-53Б

**Преподаватель** Волкова Л. Л., Строганов Ю.В.

Москва — 2022 г.

# Оглавление

# Введение

В программировании большое количество задач связано с обработкой строк. В данной рабораторной работе будут рассмотрены алгоритмы, использующие расстояние Левенштейна – метрика, позволяющая определить «схожесть» двух строк, вычисляя минимальное количество операций вставки, удаления, замены одного символа на другой, необходимых для превращения одной строки в другую.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для:

1) исправления ошибок в слове;

2) сравнения текстовых файлов утилитой diff;

3) для сравнения генов, хромосом и белков в биоинформатике.

# 1 Аналитическая часть

## 1.1 Цель и задачи

Целью данной работой является получение навыка динамического программирования на примере разработки алгоритмов поиска редакционных расстояний.

Для достижения поставленной цели требуется решить следующие задачи:

1) изучить расстояния Левенштейна и Дамерау-Левенштейна;
2) разработать указнные алгоритмы поиска расстояний;
3) реализовать разработанные алгоритмы;
4) провести сравнительный анализ реализаций алгоритмов по затраченному процессорному времени и памяти на основе экспериментальных данных;
5) описать и обосновать полученные результаты в отчете о выполненной лабораторной работе.

## 1.2 Итерационный алгоритм поиска расстояния Левенштейна

Расстояние Левенштейна [1] – это минимальное количество операций вставки, удаления и замены, необходимох для превращения одной строки в другую.

Цены операций могут зависеть от вида операций (вставка, удаление, замена) и/или от участвующих в ней символов, отражая разную вероятность разных ошибок при вводе текста и т.п. В общем случае

1) $w(a, b)$ – цена замены символа $a$ на $b$, R (от англ. replace);
2) $w(\lambda, b)$ – цена вставки символа $b$, I (от англ. insert);

3) $w(a, \lambda)$ – цена удаления символа $a$, D (от англ. delete).

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

1) $w(a, a) = 0$;

2) $w(a, b) = 1$, $a \neq b$;

3) $w(\lambda, b) = 1$;

4) $w(a, \lambda) = 1$.

Имеется две строки $S_1$ и $S_2$, длинной m и n соответственно. Расстояние Левенштейна рассчитывается по рекуррентной формуле (1.1).

$$D(i,j) = \begin{cases} 0, & \mathrm{i} = 0, \mathrm{j} = 0 \\ i, & \mathrm{j} = 0, \mathrm{i} > 0 \\ j, & \mathrm{i} = 0, \mathrm{j} > 0 \\ min \begin{cases} D(i, j-1) + 1 \\ D(i-1, j) + 1 \\ D(i-1, j-1) + \begin{cases} 0, & \text{если } S_i = S_j, \\ 1, & \text{иначе} \end{cases} \end{cases} & \mathrm{i} > 0, \mathrm{j} > 0 \end{cases} \tag{1.1}$$

## 1.3 Итерационный алгоритм поиска расстояния Дамерау-Левенштейна

В алгоритме поиска расстояния Дамерау-Левенштейна, помимо вставки, удаления, и замены присутствует еще одна редакторская операция - транспозиция T (от англ. transposition).

Расстояние Дамерау-Левенштейна может быть вычисленно по рекуррентной формуле (1.2).

$$D(i,j) = \begin{cases} 0, & \text{i = 0, j = 0,} \\ i, & \text{j = 0, i > 0,} \\ j, & \text{i = 0, j > 0,} \\ \min\{ & \\ \qquad D(i,j-1)+1, & \text{i > 0, j > 0,} \\ \qquad D(i-1,j)+1, & S_i = S_{j-1}, \\ \qquad D(i-2,j-2)+1, & S_{i-1} = S_j, \\ \qquad D(i-1,j-1) + \begin{cases} 0, & \text{если } S_i = S_j, \\ 1, & \text{иначе} \end{cases} & \\ \}, & \\ \min\{ & \\ \qquad D(i,j-1)+1, & \\ \qquad D(i-1,j)+1, & \text{иначе} \\ \qquad D(i-1,j-1) + \begin{cases} 0, & \text{если } S_i = S_j, \\ 1, & \text{иначе} \end{cases} & \\ \} & \end{cases} \tag{1.2}$$

## 1.4 Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна

Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна отличается от его итерационной версии тем, что в нем вместо использования матрицы для хранения предыдущих значений, необходимых для подсчета последующих, эти значения вычисляются каждый раз рекурсивно.

## 1.5 Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с использованием кеша

Данный алгоритм является оптимизацией рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна. Оптимизация заключается в использовании кеша, который представляется собой матрицу, в которую записываются значения, вычисленные на шагах рекурсии. Таким образом, при необходимости вычисления какого-либо нового значения по реккурентной формуле, величины, необходимые для этого не вычисляются каждый раз: сначала проверяется, было ли уже вычислено данное значения, и только лишь в случае, если этого не происходило, выполняются рекурсивные вычисления для его получения.

## Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, формулы которых задаются реккурентно, а следовательно, данные алгоритмы могут быть реализованы рекурсивно и итерационно.

# 2 Конструкторская часть

## 2.1 Описание типов данных

Для реализации алгоритмов Левенштейна и Дамерау-Левенштейна были использованы следующие типы данных:

1) строка - массив типа $char$;
2) длина строки - число типа $int$;
3) матрица - структура, содержащая двумерный массив типа $int$ и размерности типа $int$.

## 2.2 Оценка затрат алгоритмов по памяти

Алгоритмы Левенштейна и Дамерау-Левенштейна не отличаются по использованию памяти, поэтому достаточно рассмотреть итеррационную и рекурсивную реализации одного алгоритма.

Затраты по памяти для итерационного алгоритм поиска расстояния Левенштайна (Дамерау-Левенштейна):

1) длины строк n, m – $2 \cdot sizeof(int)$;
2) матрица – $(n+1) \cdot (m+1) \cdot sizeof(int) + 2 \cdot sizeof(int)$;
3) строки – $(n+m+2) \cdot sizeof(char)$;
4) дополнительные переменные (i, j, res, offset) – $4 \cdot sizeof(int)$;
5) адрес возврата.

Суммарные затраты по памяти:
$(n+1) \cdot (m+1) \cdot sizeof(int) + 7 \cdot sizeof(int) + (n+m+2) \cdot sizeof(char)$

Затраты по памяти для рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна для одного вызова:

1) длины строк n, m – $2 \cdot sizeof(int)$;
2) строки – $(n + m + 2) \cdot sizeof(char)$;
3) дополнительные переменные (res, offset) – $2 \cdot sizeof(int)$;
4) адрес возврата.

Суммарные затраты по памяти (R – количество вызовов рекурсии): $(4 \cdot sizeof(int) + (n + m + 2) \cdot sizeof(char)) \cdot R$

Затраты по памяти для рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кешем для одного вызова:

1) длины строк n, m – $2 \cdot sizeof(int)$;
2) строки – $(n + m + 2) \cdot sizeof(char)$;
3) дополнительные переменные (res, offset) – $2 \cdot sizeof(int)$;
4) матрица – $(n + 1) \cdot (m + 1) \cdot sizeof(int) + 2 \cdot sizeof(int)$;
5) адрес возврата.

Суммарные затраты по памяти (R – количество вызовов рекурсии): $(6 \cdot sizeof(int) + (n+m+2) \cdot sizeof(char)) \cdot R + (n+1) \cdot (m+1) \cdot sizeof(int)$

## 2.3 Описания алгоритмов

На рисунках ниже показаны схемы алгоритмов Левенштейна и Дамерау-Левенштейна.

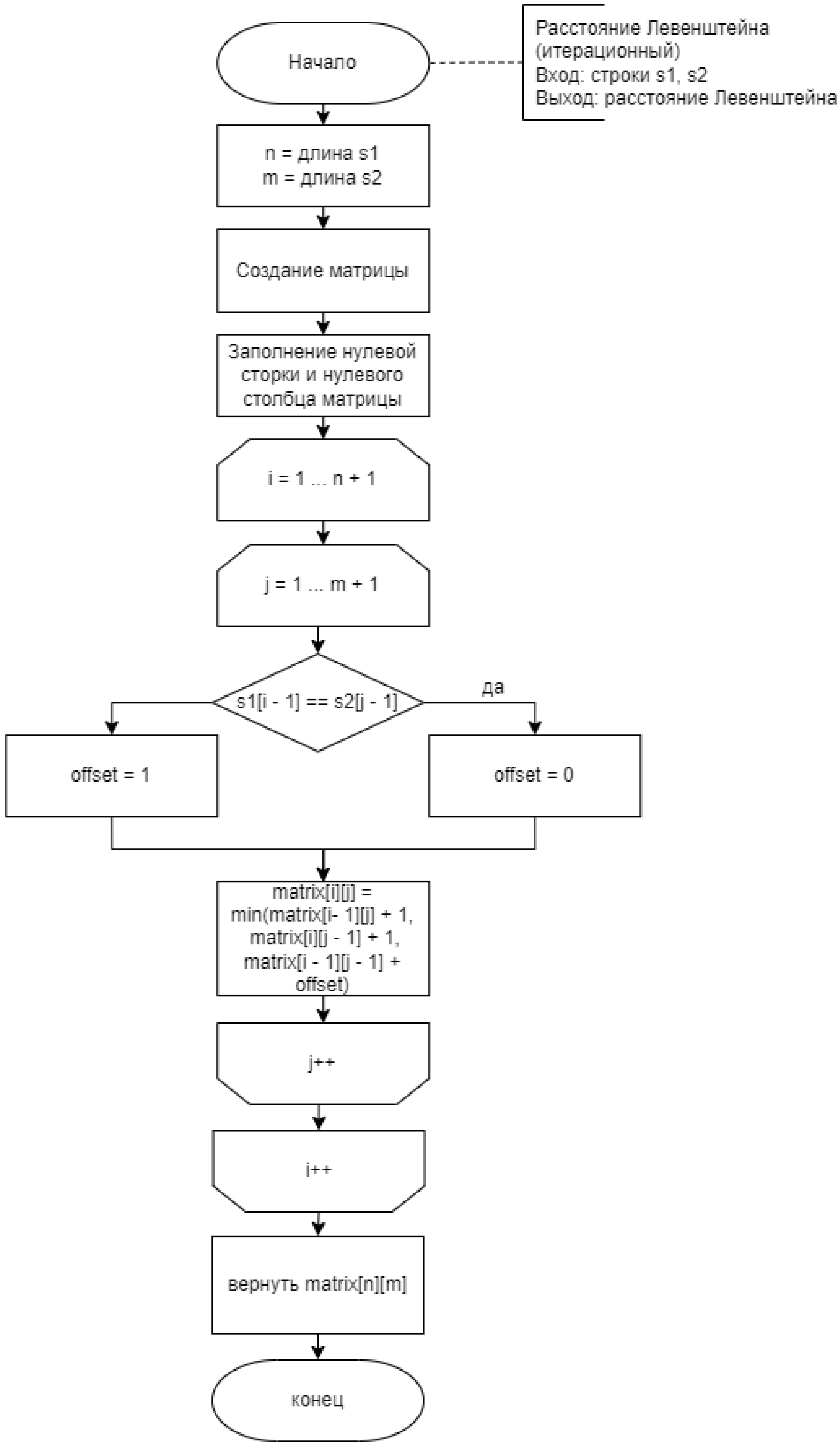


Рисунок 2.1 – Схема итерационного алгоритма поиска расстояния Левенштейна

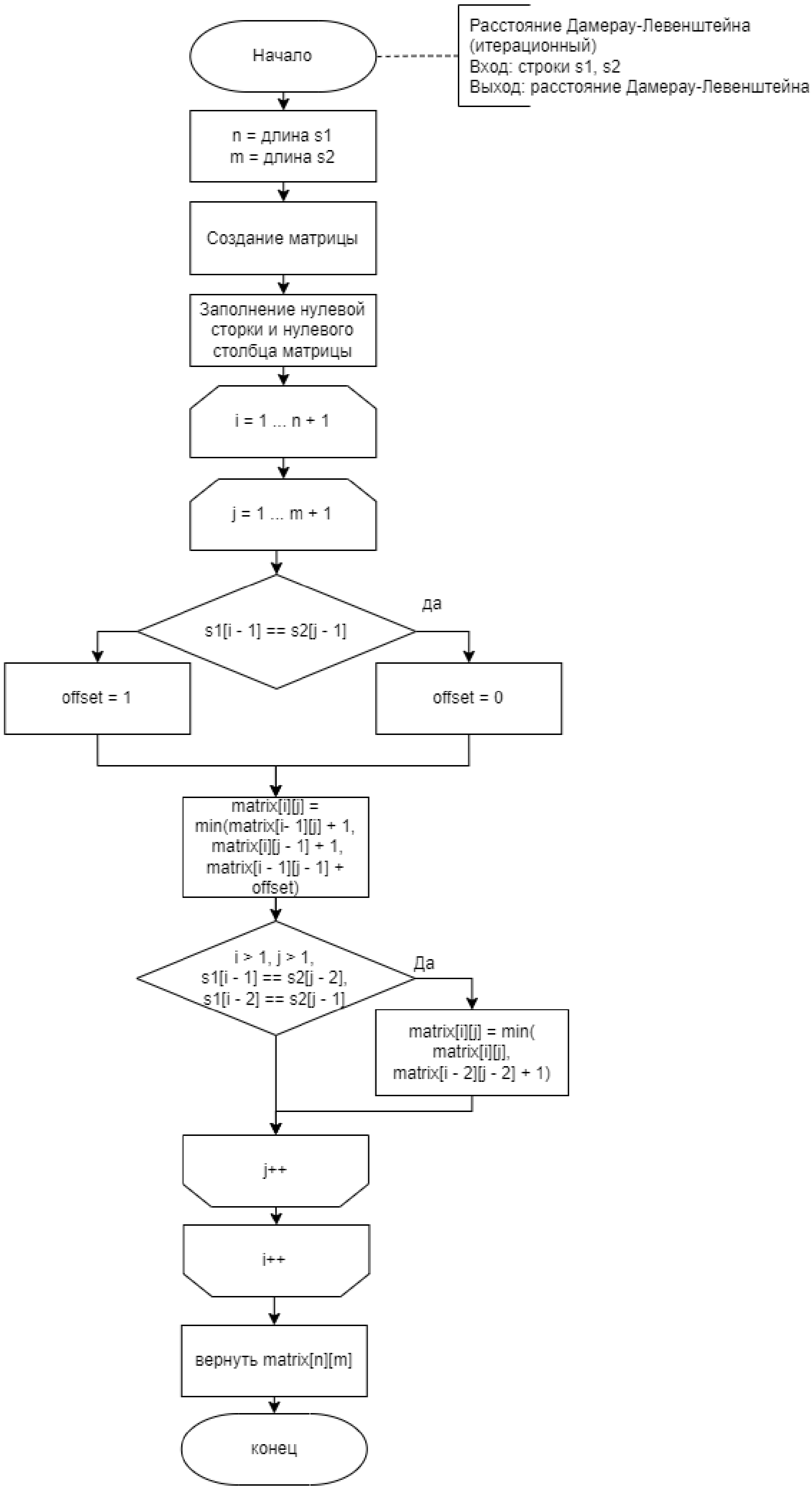


Рисунок 2.2 – Схема итерационного алгоритма поиска расстояния Дамерау-Левенштейна

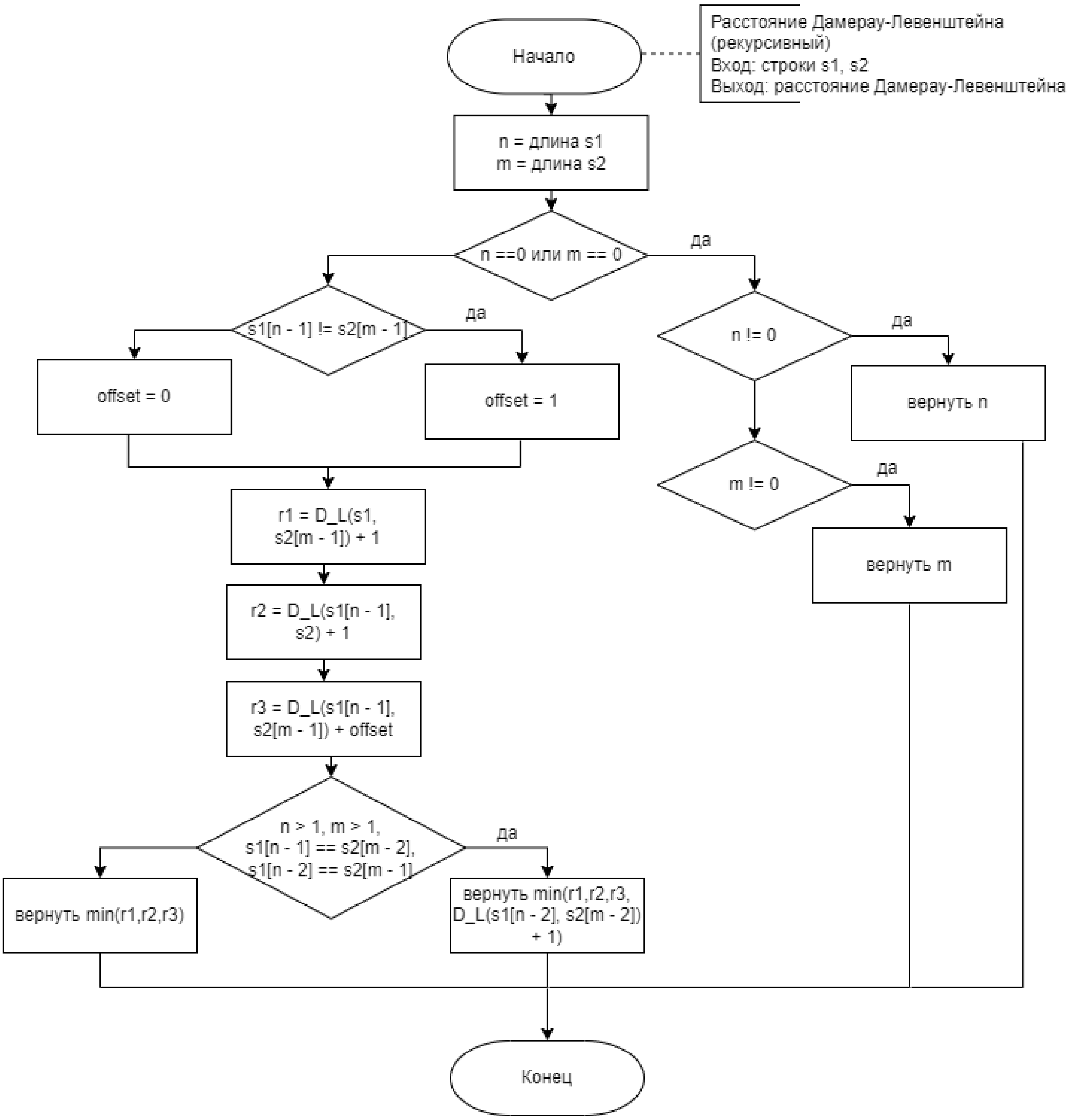


Рисунок 2.3 – Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна

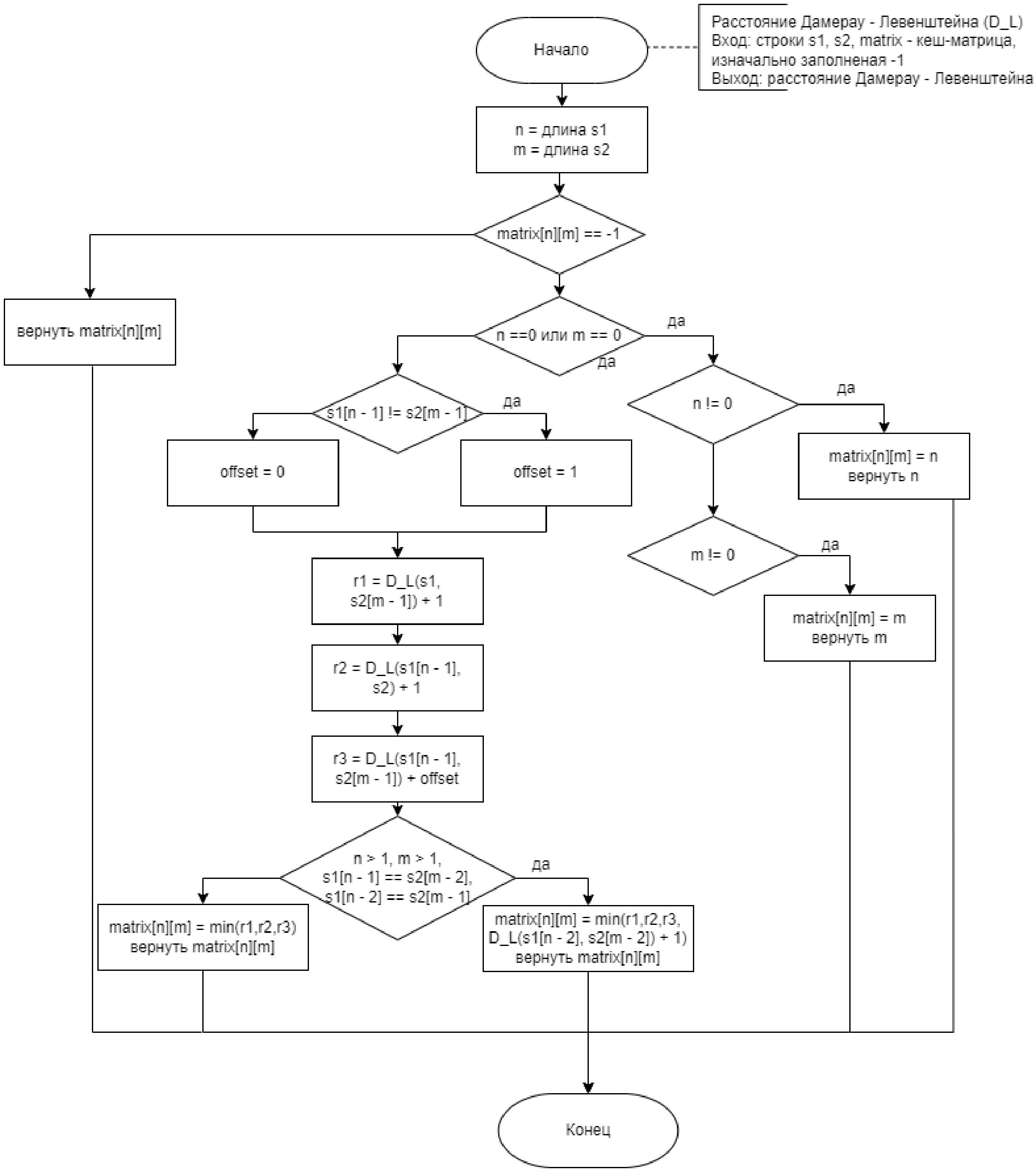


Рисунок 2.4 – Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кешем

# Вывод

В данном разделе на основе теоретических данных были построены схемы требуемых алгоритмов, выбраны используемые типы данных, а также была проведена оценка затрачиваемого объёма памяти.

# 3 Технологическая часть

## 3.1 Требования к программному обеспечению

В программе должна присутствовать возможность:

1) ввода исходных слов, для которых будут находиться расстояния Левенштейна и Дамерау-Левенштейна;

2) поиска искомого расстояния для введеных строк с помощью одного из четырех рассматриваемых алгоритмов;

3) замера процессорного времени выполнения реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

## 3.2 Выбор языка программирования

Для реализации алгоритмов поиска редакционных расстояний был выбран язык программирования C в силу наличия точных библиотек для замеров процессорного времени и быстродейственности языка.

## 3.3 Выбор библиотеки и способа для замера времени

Для замера процессорного времени выполнения реализаций агоритмов была выбрана не стандартная функция библиотеки <time.h> языка C — clock(), которая недостаточно четко работает при замерах небольших промежутков времени, а QueryPerformanceCounter - API-интерфейс, использующийся для получения меток времени с высоким разрешением или измерения интервалов времени.

Для облегчения работы с данным инструментом были самостоятельно написаны обертки-макросы, представленные на листинге 3.1.

Листинг 3.1 – Листинг макросов

```
#define TIMER_INIT \
    LARGE_INTEGER frequency; \
    LARGE_INTEGER t1,t2; \
    double elapsedTime; \
    QueryPerformanceFrequency(&frequency);

#define TIMER_START QueryPerformanceCounter(&t1);

#define TIMER_STOP \
    QueryPerformanceCounter(&t2); \
    elapsedTime=(float)(t2.QuadPat1.QuadPart)/frequency.
       QuadPart/COUNT*MICRO; \
    printf("%lf", elapsedTime);
```

В силу существования явления вытеснения процессов из ядра, квантования процессорного времени все процессорное время не отдается какой-либо одной задаче, поэтому для получения точных результатов необходимо усреднить результаты вычислений: замерить совокупное время выполнения реализации алгоритма N раз и вычислить среднее время выполнения.

## 3.4 Реализации алгоритмов

В листингах 3.2, 3.3, 3.4, 3.5 приведены реализации алгоритмов поиска расстояний Левенштейна (итерационный), Дамерау-Левенштейна (итерационный), Дамерау-Левенштейна (рекурсивный), Дамерау-Левенштейна (рекурсивный с кешем) соответсвенно.

Листинг 3.2 – Листинг итерационного алгоритма поиска расстояния Левенштейна

```
int lev(char *str_1, char *str_2, int print_table_flag)
{
    matrix_t *m = create_matrix(strlen(str_1) + 1, strlen(
       str_2) + 1);

    m->elements[0][0] = 0;
    for (size_t i = 1; i < m->rows; ++i)
        m->elements[i][0] = i;
```

```c
        for (size_t i = 1; i < m->cols; ++i)
            m->elements[0][i] = i;

        for (size_t i = 1; i < m->rows; ++i)
            for (size_t j = 1; j < m->cols; ++j)
        {
                int offset = str_1[i - 1] == str_2[j - 1] ? 0 : 1;
                m->elements[i][j] = min(3, m->elements[i][j - 1] + 1,
                                           m->elements[i - 1][j] + 1,
                                           m->elements[i - 1][j - 1] + offset);
            }

        int res = m->elements[m->rows - 1][m->cols - 1];
        free_matrix(m);

        return res;
    }
```

Листинг 3.3 – Листинг итерационного алгоритма поиска расстояния Дамерау-Левенштейна

```c
    int dameray_lev(char *str_1, char *str_2, int print_table_flag)
    {
        matrix_t *m = create_matrix(strlen(str_1) + 1, strlen(str_2) + 1);

        m->elements[0][0] = 0;
        for (size_t i = 1; i < m->rows; ++i)
            m->elements[i][0] = i;
        for (size_t i = 1; i < m->cols; ++i)
            m->elements[0][i] = i;

        for (size_t i = 1; i < m->rows; ++i)
            for (size_t j = 1; j < m->cols; ++j) {
                int offset = str_1[i - 1] == str_2[j - 1] ? 0 : 1;
                m->elements[i][j] = min(3, m->elements[i][j - 1] + 1,
                                           m->elements[i - 1][j] +
```

```
                                                      1,
                                                  m->elements[i - 1][j - 1] + offset);
                if (j > 1 && i > 1 && str_1[i - 1] == str_2[j - 2] && str_1[i - 2] == str_2[j - 1])
                    m->elements[i][j] = min(2, m->elements[i][j], m->elements[i - 2][j - 2] + 1);
            }

    int res = m->elements[m->rows - 1][m->cols - 1];
    free_matrix(m);

    return res;
}
```

Листинг 3.4 – Листинг рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна

```
int dameray_lev_rec(char *str_1, char *str_2, int len_1, int len_2)
{
    if (len_1 == 0)
        return len_2;
    if (len_2 == 0)
        return len_1;

    int offset = str_1[len_1 - 1] == str_2[len_2 - 1] ? 0 : 1;
    int res = min(3, dameray_lev_rec(str_1, str_2, len_1 - 1, len_2) + 1,
                  dameray_lev_rec(str_1, str_2, len_1, len_2 - 1) + 1,
                  dameray_lev_rec(str_1, str_2, len_1 - 1, len_2 - 1) + offset);
    if (len_1 > 1 && len_2 > 1 && str_1[len_1 - 1] == str_2[len_2 - 2] && str_1[len_1 - 2] == str_2[len_2 - 1])
        res = min(2, res, dameray_lev_rec(str_1, str_2, len_1 - 2, len_2 - 2) + 1);

    return res;
}
```

Листинг 3.5 – Листинг рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кешем

```
int dameray_lev_rec_cache_call(char *str_1, char *str_2, int
    len_1, int len_2, matrix_t *mat)
{
    if (len_1 == 0)
        return (mat->elements)[len_1][len_2] = len_2;
    if (len_2 == 0)
        return (mat->elements)[len_1][len_2] = len_1;

    if (mat->elements[len_1 - 1][len_2] == -1)
        dameray_lev_rec_hash_call(str_1, str_2, len_1 - 1,
            len_2, mat);
    if (mat->elements[len_1][len_2 - 1] == -1)
        dameray_lev_rec_hash_call(str_1, str_2, len_1, len_2 -
            1, mat);
    if (mat->elements[len_1 - 1][len_2 - 1] == -1)
        dameray_lev_rec_hash_call(str_1, str_2, len_1 - 1,
            len_2 - 1, mat);

    int offset = str_1[len_1 - 1] == str_2[len_2 - 1] ? 0 : 1;
    mat->elements[len_1][len_2] = min(3, mat->elements[len_1][
        len_2 - 1] + 1,
                        mat->elements[len_1 - 1][len_2] + 1,
                                            mat->elements[len_1 -
                                                1][len_2 - 1] +
                                                offset);
    if (len_1 > 1 && len_2 > 1 && str_1[len_1 - 1] == str_2[
        len_2 - 2] && str_1[len_1 - 2] == str_2[len_2 - 1]) {
        if (mat->elements[len_1 - 2][len_2 - 2] == -1)
            dameray_lev_rec_hash_call(str_1, str_2, len_1 - 2,
                len_2 - 2, mat);
        mat->elements[len_1][len_2] = min(2, mat->elements[
            len_1][len_2], mat->elements[len_1 - 2][len_2 - 2]
            + 1);
    }

    return mat->elements[len_1][len_2];
}

int dameray_lev_rec_cache(char *str_1, char *str_2)
```

```
    {
        matrix_t *m = create_matrix(strlen(str_1) + 1, strlen(
           str_2) + 1);
        clear_matrix(m);

        int res = dameray_lev_rec_hash_call(str_1, str_2, strlen(
           str_1), strlen(str_2), m);
        free_matrix(m);

        return res;
    }
```

## 3.5 Тестирование алгоритмов

В таблице 3.1 приведены проведенные тесты для функций, реализующих алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна.

Таблица 3.1 – Тестирование функций

| Первое слово | Второе слово | Расстояние Лев. | Расстояние Дам.-Лев. |
|---|---|---|---|
| $\lambda$ | $\lambda$ | 0 | 0 |
| *cat* | $\lambda$ | 3 | 3 |
| *cat* | *cat* | 0 | 0 |
| *cat* | *catdog* | 3 | 3 |
| *hotdog* | *hodtog* | 2 | 1 |

## Вывод

В данном разделе были разработаны алгоритмы поиска расстояний Левенштейна и Дамерау–Левенштейна, а также проведено тестирование.

# 4 Экспериментальная часть

## 4.1 Технические характеристики

Ниже приведены технические характеристики устройства, на котором было проведено тестирование программного обеспечения:

1) операционная система Windows-10, 64-bit;

2) оперативная память 8 ГБ;

3) процессор Intel(R) Core(TM) i3-7020U CPU @ 2.30GHz, 2304 МГц, ядер 2, логических процессоров 4.

## 4.2 Замеры времени

В таблице 4.1 приведены результаты замеров в микросекундах времени алгоритмов для входных строк разной длины.

Таблица 4.1 – Таблица замера времени выполнения алгоритмов на строках, имеющих разные длины

| Длина стоки | Лев. | Дам.-Лев. | Дам.-Лев.(рек.) | Дам.-Лев.(рек. с кэшем) |
|---|---|---|---|---|
| 1 | 1.15 | 1.12 | 0.11 | 0.96 |
| 2 | 1.57 | 1.08 | 0.24 | 1.96 |
| 3 | 2.71 | 2.84 | 1.36 | 1.61 |
| 4 | 1.56 | 2.56 | 6.30 | 3.94 |
| 5 | 2.85 | 3.71 | 44.91 | 4.11 |
| 6 | 5.35 | 6.73 | 161.43 | 5.47 |
| 7 | 6.60 | 7.40 | 1061.13 | 6.12 |
| 8 | 5.43 | 6.27 | 5355.47 | 4.91 |
| 9 | 3.70 | 4.47 | 30525.51 | 8.21 |
| 10 | 5.24 | 6.52 | 180535.97 | 6.95 |

Зависимость времени работы алгоритмов поиска расстояний Левенштейна (итерационный), Дамерау-Левенштейна (итерационный), Дамерау-

Левенштейна (рекурсивный с кешем) от длины входных строк представлена на рисунке 4.1.

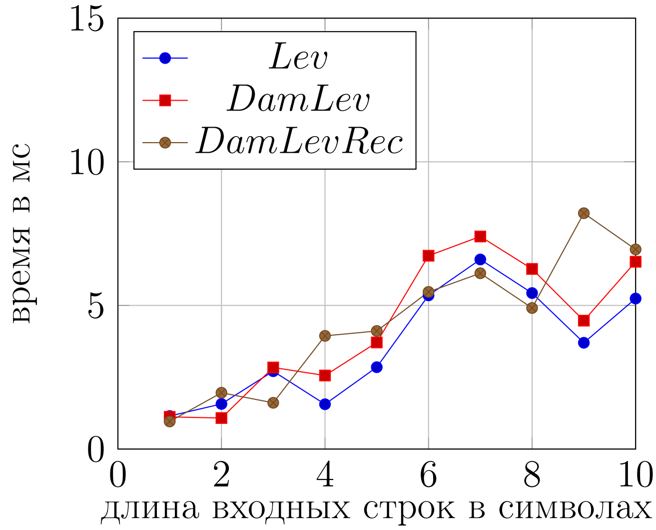


Рисунок 4.1 – Зависимость времени от длины входных строк

Зависимость времени работы алгоритма поиска расстояний Дамерау-Левенштейна (рекурсивный) от длины входных строк представлена на рисунке 4.2.

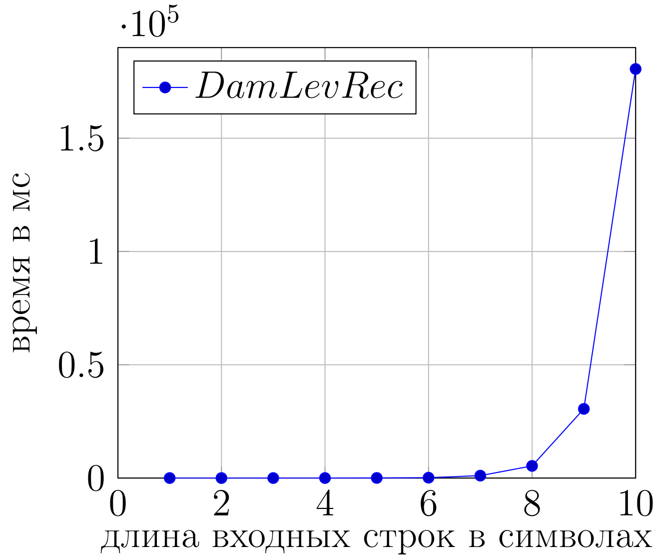


Рисунок 4.2 – Зависимость времени от длины входных строк

# Вывод

Результаты замеров показали, что рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна работает дольше всего. При этом его оптимизация с кешем работает в разы быстрее. Итерационные алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна оказались наиболее быстрыми, причем на длине строки от 1 до 10 элементов итерационный алгоритм Левенштейна в среднем работает немного быстрее, нежели итерационный алгоритм нахождения расстояния Дамера-Левенштейна.

# Заключение

Цель лабораторной работы достигнута – были получены навыки динамического программирования на примере разработки алгоритмов поиска редакционных расстояний. Все задачи решены:

1) изучены расстояния Левенштейна и Дамерау-Левенштейна;
2) разработаны указнные алгоритмы поиска расстояний;
3) реализованы разработанные алгоритмы;
4) была рассчитана их трудоемкость по памяти;
5) проведен сравнительный анализ реализаций алгоритмов по затраченному процессорному времени и памяти на основе экспериментальных данных;
6) описаны и обоснованы полученные результаты в отчете о выполненной лабораторной работе.

На основании проведенных экспериментов было определено, что время работы алгоритмов с увелечением длины строк увеличивается в геометрической прогрессии. Самым медленным по скорости выполнения, но наименне затратным по памяти оказался рекурсивный алгоритм определения расстояния Дамерау-Левенштейна. Самым быстрым же оказался итерационный алгоритм нахождения расстояния Левенштейна, который одновременно с этим, оказался одним из самых затратных по памяти, так как в его реализации дополнительно используется матрица, размером, соответсвующим длине входных строк.

# Список использованных источников


[1] Погорелов, Д. А. Сравнительный анализ алгоритмов редакционного расстояния Левенштейна и Дамерау-Левенштейна / Д. А. Погорелов, А. М. Тарзанов. - Синергия Наук. - 2019. URL: https://elibrary.ru/item.asp?id=36907767 (дата обращения: 25.10.2022).

[2] Microsoft. QueryPerformanceCounter function (profileapi.h). URL: https://learn.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter (дата обращения: 25.10.2022).

[3] The LaTeX project. Core Documentation. URL: https://www.latex-project.org/help/documentation/ (дата обращения: 20.10.2022).