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 «Анализы Алгоритмов»
на тему: «Динамическое программирование»

Студент ИУ7-56Б
(Группа)
Мансуров В. М.
(Фамилия И. О.)

Преподаватели
Волкова Л. Л., Строганов Ю. В.
(Фамилия И. О.)

Москва — 2022 г.

Содержание

Введение

В данной лабораторной работе будет рассмотрено расстояние Левенштейна. Данное расстояние показывает минимальное количество операций (вставка, удаление, замены), которое необходимо для перевода 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 [1] был предложен и обоснован Р. Беллманом в начале 1960-х годов. 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 описание и исследование 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 с кешированием алгоритм поиска расстояния Дамерау-Левенштейна.

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

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

1 Аналитическая часть

1.1 Расстояние Левенштейна

Расстояние Левенштейна [2] (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. Стоимости операций могут зависеть от вида операций:

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:

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

где сравнение символов строк S_1 и S_2 рассчитывается как:

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

1.1.1 Нерекурсивный алгоритм нахождения расстояния Левенштейна

Рекурсивная реализация алгоритма Левенштейна малоэффективна по времени при больших M и N, так как 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:

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

1.2.1 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Рекурсивный алгоритм реализует формулу (1.4), функция D составлена таким образом, что верно следующее.

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. Последовательность поведения любых двух операций можно поменять, порядок поведения операций не имеет никакого значения. 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 по причине проблемы повторных вычислений 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 кешем 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 Функциональные требования

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

2.2 Разработка алгоритмов

На вход алгоритмов подаются строки S_1 и S_2.

На рисунке 2.1 представлена схема алгоритма поиска расстояния Левенштейна. На рисунках 2.2 – 2.5 представлены схемы алгоритмов поиска Дамерау-Левенштейна.

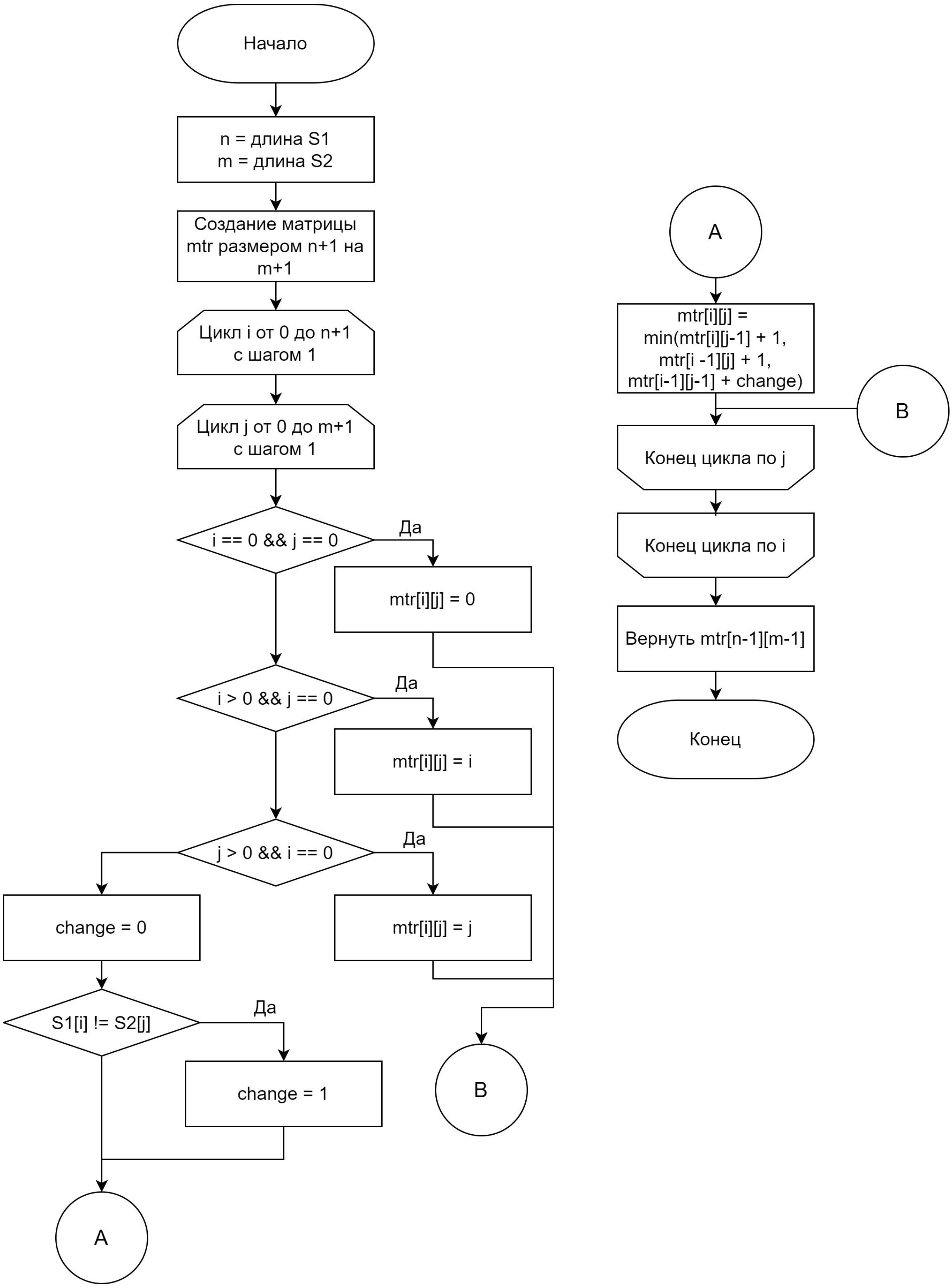

Рисунок 2.1 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

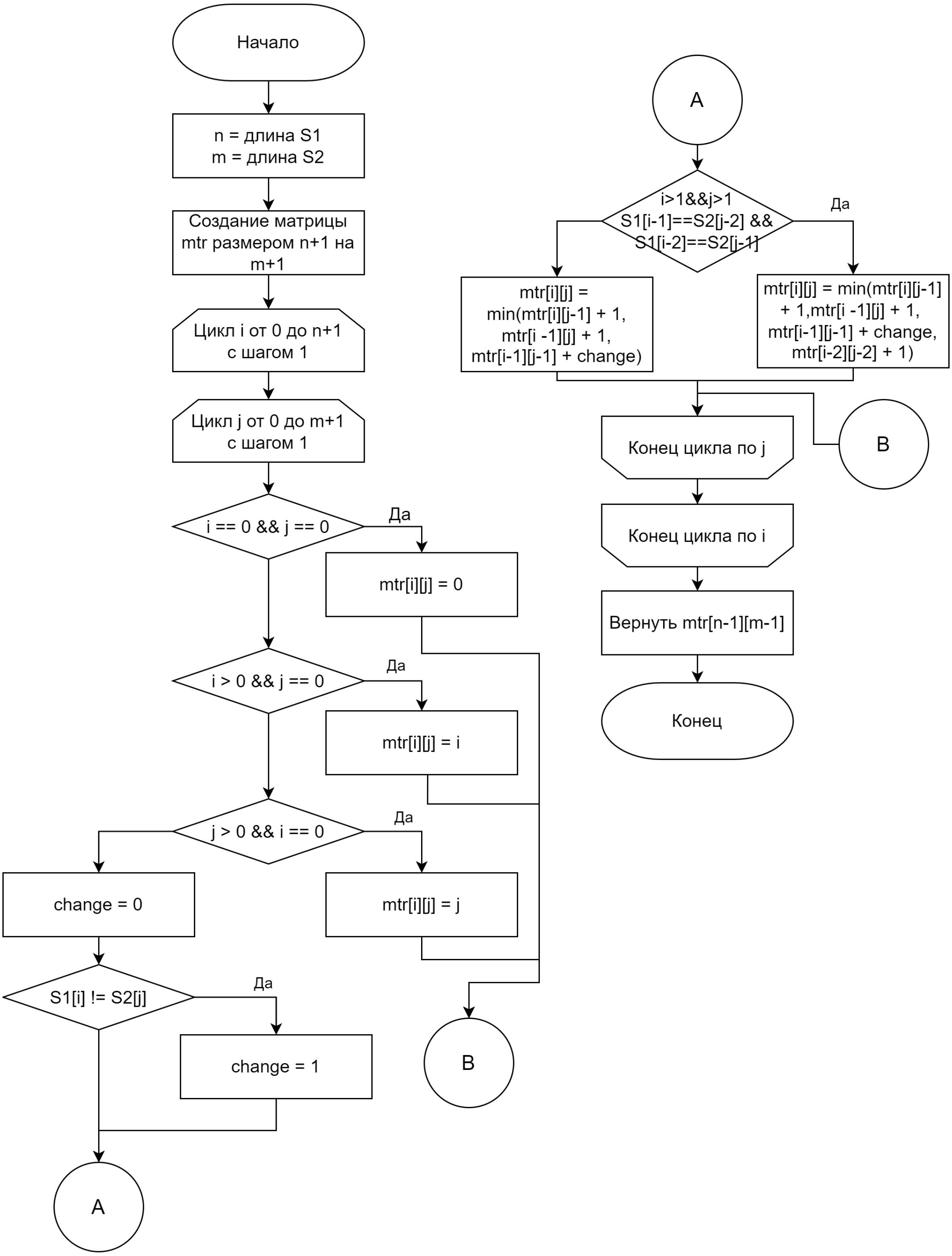

Рисунок 2.2 – Схема нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

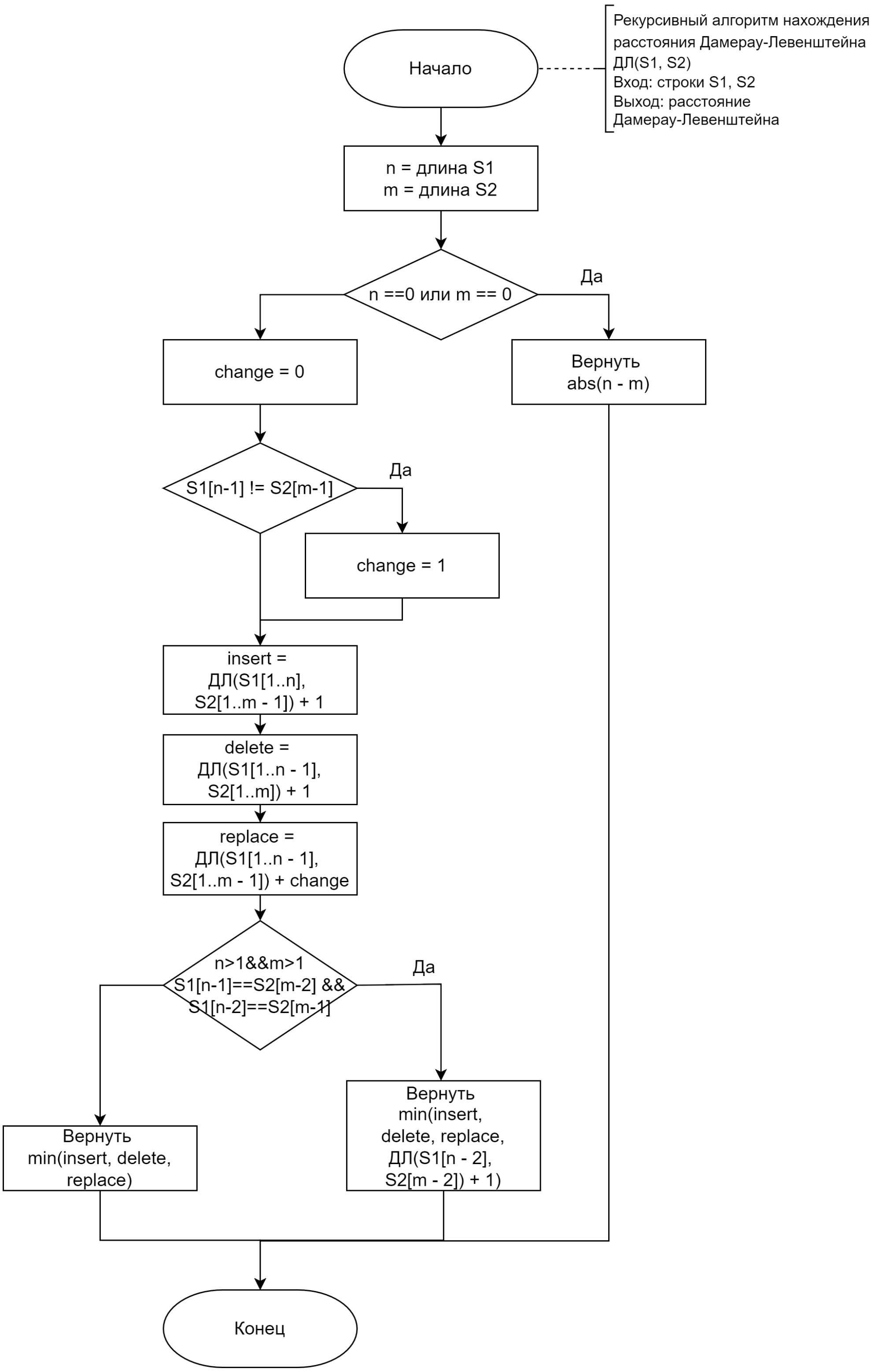

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

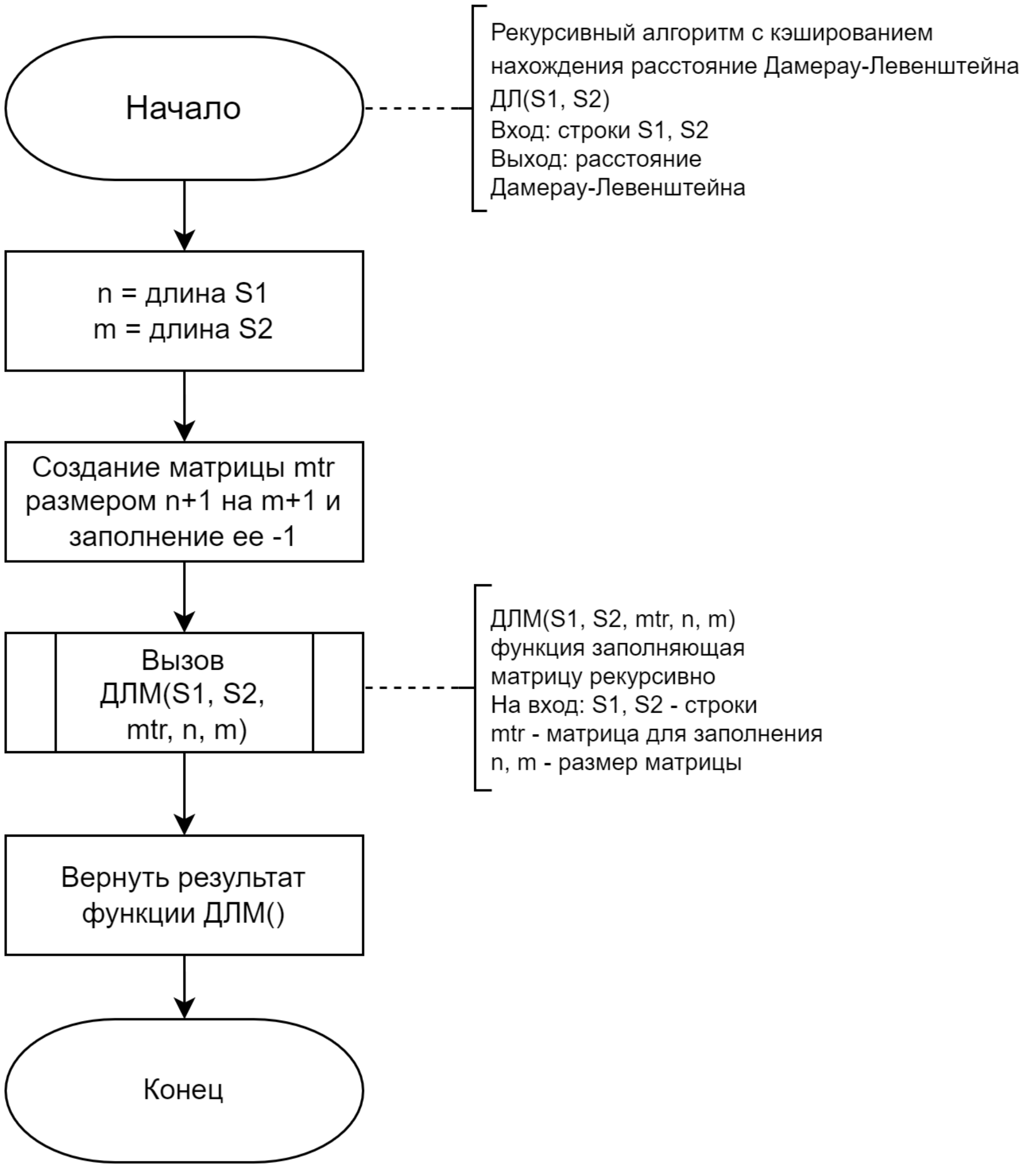

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кешированием

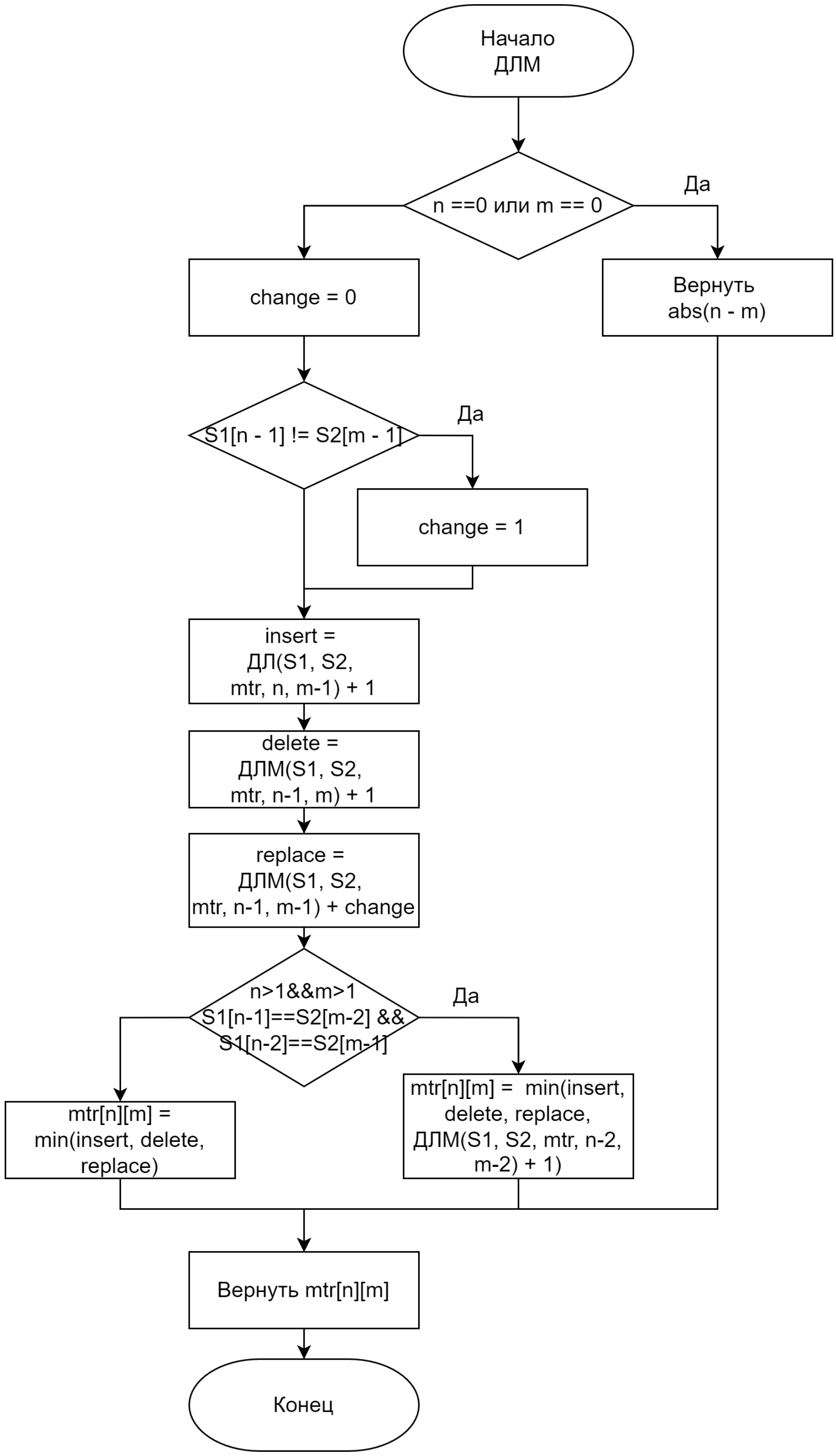

Рисунок 2.5 – Схема алгоритма рекурсивного заполнения матрицы путем поиска расстояния Дамерау-Левенштейна

2.3 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

- строка — массив типа $wchar$ размером длины строки;
- длина строки — целое число типа int;
- 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 Требования к программному обеспечению

К программе предъявлен ряд требований:

— 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-Левенштейна.

3.2 Средства реализации

Для реализации данной лабораторной работы был выбран язык $C++$ [3]. Данный выбор обусловлен наличием у языка встроенного модуля *ctime* измерения процессорного времени и типа данных, позволяющего хранить как кириллические символы, так и латинские — $std :: wstring$, что позволит удовлетворить третьему требованию из п.3.1 соответствуют выдвинутым техническим требованиям.

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

3.3 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.
- `algorithms.cpp` — файл содержит функции поиска расстояния Левенштейна и Дамерау-Левенштейна.
- `allocate.cpp` — файл содержит функции динамического выделения и очищения памяти для матрицы.
- `print_mtr_lev.cpp` — файл содержит функцию вывода матрицы для итеративных алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна, включая строки.
- `cpu_time.cpp` — файл содержит функции, замеряющее процессорное время алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна.
- `memory.cpp` — файл содержит функции, замеряющее память итеративного и рекурсивного алгоритмов поиска расстояния Левенштейна.

В листингах 3.1 – 3.4 приведены реализации алгоритмов поиска расстояний Левенштейна (только нерекурсивный алгоритм) и Дамерау-Левенштейна (нерекурсивный, рекурсивный и рекурсивный с кешированием). В листингах 3.5 – 3.6 приведены реализации алгоритмов выделения памяти под матрицу и вывод матрицы.

Листинг 3.1 – Функция нахождения расстояния Левенштейна с использованием матрицы

```
int lev_mtr(wstring &str1, wstring &str2, bool print) {
    size_t n = str1.length();
    size_t m = str2.length();
    int **mtr = malloc_mtr(n + 1, m + 1);
    int res = 0;
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= m; j++)
            if (i == 0 && j == 0)
                mtr[i][j] = 0;
            else if (i > 0 && j == 0)
                mtr[i][j] = i;
            else if (j > 0 && i == 0)
                mtr[i][j] = j;
    else {
        int change = 0;
        if (str1[i - 1] != str2[j - 1])
            change = 1;

        mtr[i][j] = std::min(mtr[i][j - 1] + 1,
                        std::min(mtr[i - 1][j] + 1,
                                 mtr[i - 1][j - 1] + change));
    }
    if (print)
        print_mtr_lev(str1, str2, mtr, n, m);
    res = mtr[n][m];
    free_mtr(mtr, n);
    return res;
}
```

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

```
int dameray_lev_mtr(wstring &str1, wstring &str2, bool print)
{
    size_t n = str1.length();
    size_t m = str2.length();
    int **mtr = malloc_mtr(n + 1, m + 1);
    int res = 0;

    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= m; j++) {
            if (i == 0 && j == 0)
                mtr[i][j] = 0;
            else if (i > 0 && j == 0)
                mtr[i][j] = i;
            else if (j > 0 && i == 0)
                mtr[i][j] = j;
            else {
                int change = 0;
                if (str1[i - 1] != str2[j - 1])
                    change = 1;

                mtr[i][j] = min(mtr[i][j - 1] + 1,
                            min(mtr[i - 1][j] + 1,
                                mtr[i - 1][j - 1] + change));

                if (i > 1 && j > 1 &&
                str1[i - 1] == str2[j - 2] &&
                str1[i - 2] == str2[j - 1])
                    mtr[i][j] = min(mtr[i][j], mtr[i - 2][j - 2] +
                       1);
            }
    }

    if (print)
        print_mtr_lev(str1, str2, mtr, n, m);
    res = mtr[n][m];
    free_mtr(mtr, n);

    return res;
}
```

Листинг 3.3 – Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
int dameray_lev_rec_t(wstring &str1, wstring &str2, size_t n,
    size_t m) {
    if (n == 0)
        return m;
    if (m == 0)
        return n;

    int change = 0;
    int res = 0;
    if (str1[n - 1] != str2[m - 1])
        change = 1;

    res = min(dameray_lev_rec_t(str1, str2, n, m - 1) + 1,
          min(dameray_lev_rec_t(str1, str2, n - 1, m) + 1,
              dameray_lev_rec_t(str1, str2, n - 1, m - 1) +
                change));

    if (n > 1 && m > 1 &&
    str1[n - 1] == str2[m - 2] &&
    str1[n - 2] == str2[m - 1])
        res = std::min(res, dameray_lev_rec_t(str1, str2, n - 2, m
            - 2) + 1);
    return res;
}

int dameray_lev_rec(wstring &str1, wstring &str2)
{
    return dameray_lev_rec_t(str1, str2, str1.length(),
        str2.length());
}
```

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

```
int dameray_lev_rec_hash_t(wstring &str1, wstring &str2, int **mtr, size_t n, size_t m)
{
    if (n == 0)
        return mtr[n][m] = m;
    if (m == 0)
        return mtr[n][m] = n;
    int change = 0;
    if (str1[n - 1] != str2[m - 1])
        change = 1;
    mtr[n][m] = min(dameray_lev_rec_hash_t(str1, str2, mtr, n, m - 1) + 1,
                min(dameray_lev_rec_hash_t(str1, str2, mtr, n - 1, m) + 1,
                    dameray_lev_rec_hash_t(str1, str2, mtr, n - 1, m - 1) + change));
    if (n > 1 && m > 1 &&
    str1[n - 1] == str2[m - 2] &&
    str1[n - 2] == str2[m - 1])
        mtr[n][m] = min(mtr[n][m], dameray_lev_rec_hash_t(str1, str2, mtr, n - 2, m - 2) + 1);
    return mtr[n][m];
}

int dameray_lev_rec_hash(wstring &str1, wstring &str2, bool print)
{
    size_t n = str1.length();
    size_t m = str2.length();
    int **mtr = malloc_mtr(n + 1, m + 1);
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= m; j++) {
            mtr[i][j] = -1;
        }
    int res = dameray_lev_rec_hash_t(str1, str2, mtr, n, m);
    if (print)
        print_mtr_lev(str1, str2, mtr, n, m);
    free_mtr(mtr, n);
    return res;
}
```

Листинг 3.5 – Функции динамического выделения и очищения памяти под матрицу

```
void free_mtr(int **mtr, std::size_t n) {
    if (mtr != nullptr)
    {
        for (std::size_t i = 0; i < n; i++)
        if (mtr[i] != nullptr)
            free(mtr[i]);
        free(mtr);
    }
}

int **malloc_mtr(std::size_t n, std::size_t m)
{
    if (n == 0)
    return nullptr;

    int **mtr = static_cast<int **>(malloc(n * sizeof(int *)));
    if (mtr != nullptr)
    for (std::size_t i = 0; mtr[i] != nullptr && i < n; i++) {
        mtr[i] = static_cast<int *>(malloc(m * sizeof(int)));
        if (mtr[i] == nullptr)
            free_mtr(mtr, n);
    }

    return mtr;
}
```

Листинг 3.6 – Функции вывода матрицы для алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна

```
void print_mtr_lev(std::wstring str1, std::wstring str2,
int **mtr, std::size_t n, std::size_t m)
{
    for(std::size_t i = 0; i <= n + 1; i++)
    {
        for(std::size_t j = 0; j <= m + 1; j++)
        {
            if (i == 0 && j == 0)
                std::wcout << "  ";
            else if (i == 0)
            if (j == 1)
                std::wcout << "- ";
            else
                std::wcout << str2[j - 2] << " ";
            else if (j == 0)
            if (i == 1)
                std::wcout << "- ";
            else
                std::wcout << str1[i - 2] << " ";
            else
                std::wcout << mtr[i - 1][j - 1] << " ";
        }
        std::wcout << std::endl;
    }
}
```

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
кот	скат	2	2	2	2
друзья	рдузия	3	2	2	2
вагон	гонки	4	4	4	4
бар	раб	2	2	2	2
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) i5-10300H CPU 2.50 ГГц [5].
- Оперативная память: 16 ГБайт.
- Операционная система: Windows 10 Pro 64-разрядная система версии 21H2 [6].

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 «друзья» и «рдузия». При этом выводятся матрицы для алгоритмов, использующих матрицы для промежуточных результатов.

```
   Выберите алгоритм поиска растояния:
1 - 1) Нерекурсивный алгоритм Левенштейна;
  - 2) Нерекурсивный алгоритм Дамерау-Левенштейна;
  - 3) Рекурсивный алгоритм Дамерау-Левенштейна без кэша;
  - 4) Рекурсивный алгоритм Дамерау-Левенштейна с кэшом;
2 - Замерить время и память.

Выбор: 1
Введите первое слово: друзья
Введите второе слово: рдузия
Минимальное количество операций:
  - р д у з и я
- 0 1 2 3 4 5 6
д 1 1 1 2 3 4 5
р 2 1 2 2 3 4 5
у 3 2 2 2 3 4 5
з 4 3 3 3 2 3 4
ь 5 4 4 4 3 3 4
я 6 5 5 5 4 4 3
   Нерекурсивный алгоритм Левенштейна: 3
  - р д у з и я
- 0 1 2 3 4 5 6
д 1 1 1 2 3 4 5
р 2 1 1 2 3 4 5
у 3 2 2 1 2 3 4
з 4 3 3 2 1 2 3
ь 5 4 4 3 2 2 3
я 6 5 5 4 3 3 2
   Нерекурсивный алгоритм Дамерау-Левенштейна: 2
   Рекурсивный алгоритм Дамерау-Левенштейна без кэша: 2
  - р д у з и я
- 0 1 2 3 4 5 6
д 1 1 1 2 3 4 5
р 2 1 1 2 3 4 5
у 3 2 2 1 2 3 4
з 4 3 3 2 1 2 3
ь 5 4 4 3 2 2 3
я 6 5 5 4 3 3 2
   Рекурсивный алгоритм Дамерау-Левенштейна с кэшом: 2
```

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 15 будет достаточно долгим, поэтому нет смысла проводить замеры по времени рекурсивных реализаций алгоритмов поиска расстояния.

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

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

Длина (символ)	**Время, нс**			
	Левенштейн	**Дамерау-Левенштейн**		
	Итеративный	**Итеративный**	**Рекурсивный**	
			Без кеша	**С кешом**
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	15.656	15.656	15.656
6	0	0	62.625	62.625
7	15.656	15.656	328.78	391.41
8	0	0	1706.5	2004
9	15.656	15.656	9393.8	11288
10	15.656	15.656	51932	61545
20	15.656	15.656	-	-
30	31.313	15.656	-	-
40	31.313	31.313	-	-
50	31.313	46.969	-	-
60	62.625	78.282	-	-
70	78.282	78.282	-	-
80	93.938	109.59	-	-
90	109.59	140.91	-	-
100	140.91	172.22	-	-
200	673.22	751.5	-	-

Отдельно сравнивается итеративные алгоритмы поиска расстояний Левенштейна и Дамерау–Левенштейна. Сравнение будет производится 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 оказывается быстрее вплоть до полутора раз (при длинах строк равных 200). Это обосновывается тем, что у алгоритма поиска расстояния Дамерау-Левенштейна задействуется дополнительная операция, которая замедляет алгоритм.

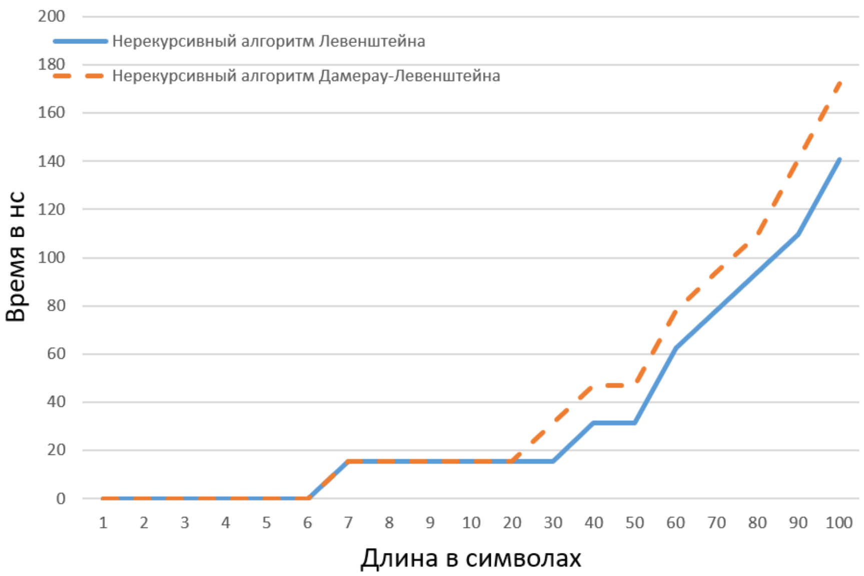

Рисунок 4.2 – Сравнение по времени нерекурсивных реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна

Также сравним рекурсивную и итеративную реализации алгоритма поиска расстояния Дамерау-Левенштейна. Данные представлены в таблице 4.1 и отображены на рисунке 4.3.

На рисунке 4.3 продемонстрировано, что рекурсивный алгоритм становится менее эффективным по времени (вплоть до 21 раз при длине строк равной 7 элементов), чем итеративный.

Кроме того, согласно данным, приведенным в таблице 4.1, рекурсивные алгоритмы при длинах строк более 10 элементов не пригодны к использованию в силу экспоненциально роста затрат процессорного времени, в то время, как затраты итеративных алгоритмов по времени линейны.

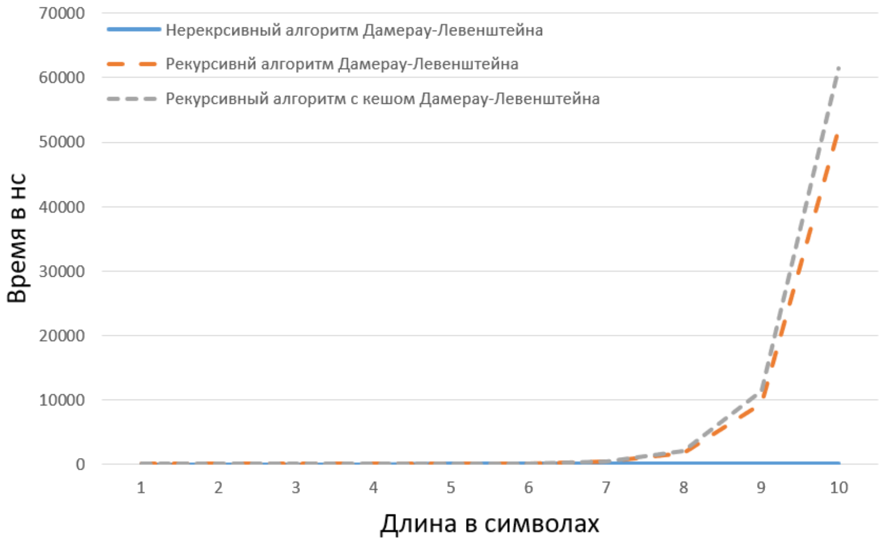

Рисунок 4.3 – Сравнение по времени алгоритмов поиска расстояния Дамерау-Левенштейна

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 равен:

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

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 равен:

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

Использование памяти при итеративной реализации алгоритма поиска расстояния Левенштейна теоретически равно:

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

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 равно:

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

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

Длина (символ)	**Размер в байтах**			
	Левенштейн	**Дамерау-Левенштейн**		
	Итеративный	**Итеративный**	**Рекурсивный**	
			Без кеша	**С кешом**
10	668	672	1840	2420
20	2028	2032	3680	5620
30	4188	4192	5520	9620
40	7148	7152	7360	14420
50	10908	10912	9200	20020
60	15468	15472	11040	26420
70	20828	20832	12880	33620
80	26988	26992	14720	41620
90	33948	33952	16560	50420
100	41708	41712	18400	60020
110	50268	50272	20240	70420
120	59628	59632	22080	81620
130	69788	69792	23920	93620
140	80748	80752	25760	106420
150	92508	92512	27600	120020
160	105068	105072	29440	134420
170	118428	118432	31280	149620
180	132588	132592	33120	165620
190	147548	147552	34960	182420
200	163308	163312	36800	200020

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.

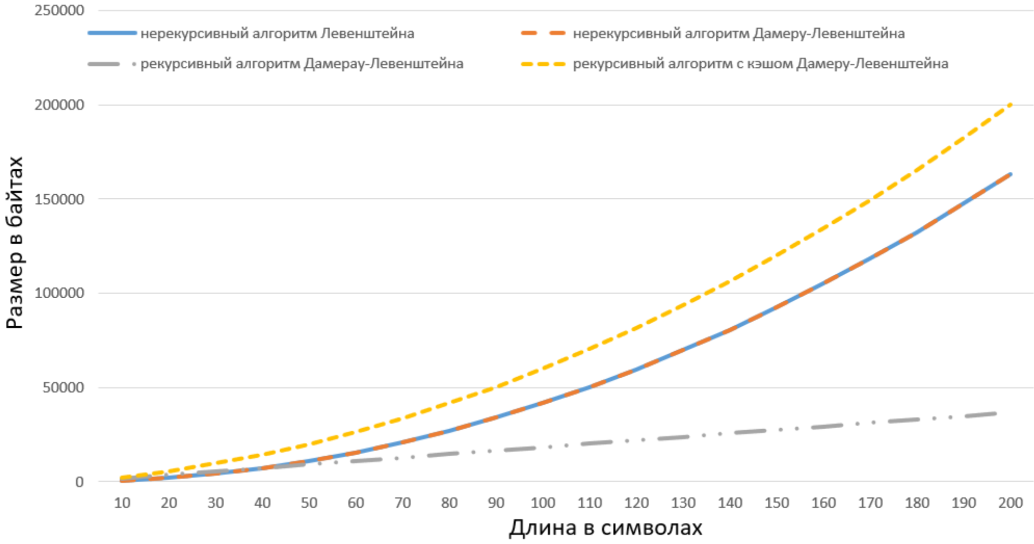

Рисунок 4.4 – Сравнение по памяти алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна — итеративной и рекурсивной реализации

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

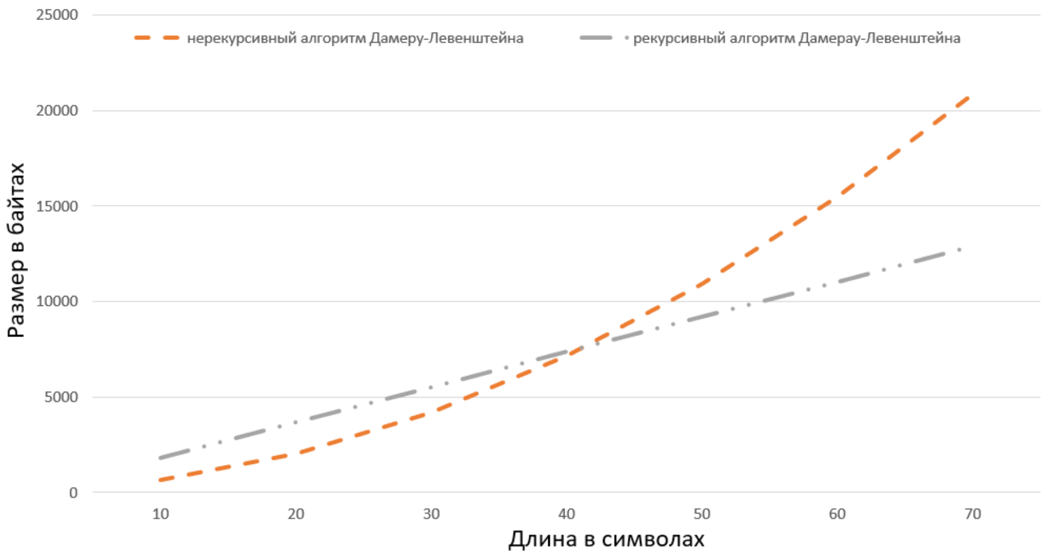

Рисунок 4.5 – Сравнение по памяти алгоритмов поиска расстояния Дамерау-Левенштейна — итеративной и рекурсивной реализации

4.5 Вывод

В данном разделе было произведено сравнение количества затраченного времени и памяти алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна. Наименее затратным по времени оказался итеративный алгоритм нахождения расстояния Левенштейна.

Приведенные характеристики показывают, что рекурсивная реализация алгоритма в 21 раз проигрывает по времени. В связи с этим, рекурсивные алгоритмы следует использовать лишь для малых размерностей строк (1 – 4 символа).

Так как во время печати очень часто возникают ошибки связанные с транспозицией букв [1], алгоритм поиска расстояния Дамерау-Левенштейна является наиболее предпочтительным, не смотря на то, что он проигрывает по времени и памяти алгоритму Левенштейна.

Рекурсивная реализация алгоритма поиска расстояния Дамерау-Левенштейна будет более затратным по времени по сравнению с итеративной реализацией алгоритма поиска расстояния Дамерау-Левенштейна, но менее затратным по памяти по отношению к итеративному алгоритму Дамерау-Левенштейна.

Заключение

В результате исследования было определено, что время алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна растет в геометрической прогрессии при увеличении длин строк. Лучшие показатели по времени дает матричная реализация алгоритма нахождения расстояния Дамерау-Левенштейна и его рекурсивная реализация с кешем, использование которых приводит к 21-кратному превосходству по времени работы уже на длине строки в 4 символа за счет сохранения необходимых промежуточных вычислений. При этом итеративная реализации с использованием матрицы занимают довольно много памяти при большой длине строк.

Цель данной лабораторной работы были достигнуты, а именно описание и исследование особенностей задач динамического программирования на алгоритмах Левенштейна и Дамерау-Левенштейна.

Для достижения поставленной целей были выполнены следующие 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 Дамерау-Левенштейна;
   - рекурсивный метод поиска Дамерау-Левенштейна;
   - рекурсивный с кешированием метод поиска Дамерау-Левенштейна.
3) Выбраны инструменты для замера процессорного времени выполнения реализаций алгоритмов.
4) Проведены анализ затрат работы программы по времени и по памяти, выяснить влияющие на них характеристики.

Список использованных источников

[1] В. Ульянов М. Ресурсно-эффективные компьютерные алгоритмы: учебное пособие. – М.: Издательство «Наука», ФИЗМАТЛИ, 2007. С. 168 – 185.

[2] И. Левенштейн В. Двоичные коды с исправлением выпадений, вставок и замещений символов. – М.: Издательство «Наука», Доклады АН СССР, 1965. Т. 163. С. 845–848.

[3] Документация по Microsoft C++ [Электронный ресурс]. Режим доступа: `https://learn.microsoft.com/ru-ru/cpp/?view=msvc-170&viewFallbackFrom=vs-2017` (дата обращения: 25.09.2022).

[4] C library function clock() [Электронный ресурс]. Режим доступа: `https://www.tutorialspoint.com/c_standard_library/c_function_clock.htm` (дата обращения: 25.09.2022).

[5] Intel [Электронный ресурс]. Режим доступа: `https://ark.intel.com/content/www/ru/ru/ark/products/201839/intel-core-i510300h-processor-8m-cache-up-to-4-50-ghz.html` (дата обращения: 25.09.2022).

[6] Windows 10 Pro 2h21 64-bit [Электронный ресурс]. Режим доступа: `https://www.microsoft.com/ru-ru/software-download/windows10` (дата обращения: 25.09.2022).