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

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №1
по курсу «Анализ алгоритмов»
на тему «Расстояние Левенштейна и Дамерау — Левенштейна»

Студент Завойских Евгения Васильевна

Группа ИУ7-53Б

Оценка (баллы) ________________________________

Преподаватели Волкова Л.Л., Строганов Ю.В.

2022 г.

Оглавление

Введение **3**

1 Аналитическая часть **4**

1.1 Расстояние Левенштейна 4

1.2 Расстояние Дамерау — Левенштейна 4

2 Конструкторская часть **6**

2.1 Разработка алгоритмов 6

3 Технологическая часть **13**

3.1 Средства реализации 13

3.2 Реализации алгоритмов 13

3.3 Описание тестирования 18

4 Исследовательская часть **19**

4.1 Технические характеристики 19

4.2 Постановка эксперимента по замеру времени 19

4.3 Результаты эксперимента 20

4.4 Расчет используемой памяти 22

Заключение **25**

Список использованных источников **26**

Введение

Расстояние Левенштейна, или редакционное расстояние, – это минимальное количество операций замены, вставки и удаления символа, которое нужно выполнить над одной последовательностью символов, чтобы получить другую. Определение расстояния Дамерау — Левенштейна получается добавлением к указанным действиям операции перестановки соседних символов [1].

Расстояние Левенштейна используется для исправления ошибок в словах, поиска дубликатов текстов, сравнения геномов и прочих операций с символьными последовательностями.

Целью данной работы является изучение и исследование особенностей задач динамического программирования на материале алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1) изучить расстояния Левенштейна и Дамерау — Левенштейна;
2) разработать алгоритмы нерекурсивного метода поиска расстояния Левенштейна, нерекурсивного метода поиска расстояния Дамерау — Левенштейна, рекурсивного метода поиска расстояния Дамерау — Левенштейна, рекурсивного с кешированием метода поиска расстояния Дамерау — Левенштейна;
3) реализовать разработанные алгоритмы;
4) выбрать инструменты для замера процессорного времени выполнения реализаций алгоритмов;
5) сравнить алгоритмы по процессорному времени работы реализаций;
6) сравнить алгоритмы по используемой памяти.

1 Аналитическая часть

1.1 Расстояние Левенштейна

Расстояние Левенштейна – это минимальное количество операций вставки, удаления и замены символа, необходимое для преобразования одной символьной последовательности в другую.

Для решения задачи поиска расстояния Левенштейна нужно найти последовательность операций с минимальной суммарной ценой. Цена каждой операции:

- $w(a, b) = 1$ – для замены одного символа на другой;
- $w(a, \lambda) = 1$ – для удаления одного символа;
- $w(\lambda, b) = 1$ – для вставки одного символа;
- $w(a, a) = 0$ – для двух совпавших символов.

Пусть s_1 и s_2 – две строки над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по следующей рекуррентной формуле (1.1)

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ \min\{ & \\ \qquad D(i, j-1) + 1, & \\ \qquad D(i-1, j) + 1, & \\ \qquad D(i-1, j-1) + & \\ \qquad \begin{cases} 0, & \text{если } s_1[i] = s_2[j] \\ 1, & \text{иначе} \end{cases} & \\ \}, & i > 0, j > 0 \end{cases} \tag{1.1}$$

1.2 Расстояние Дамерау — Левенштейна

Расстояние Дамерау — Левенштейна – это минимальное количество операций вставки, удаления, замены и перестановки соседних символов, необходимое для преобразования одной символьной последовательности в другую.

Алгоритм поиска расстояния Дамерау — Левенштейна аналогичен алгоритму поиска расстояния Левенштейна. Дополнительно необходимо учесть операцию перестановки соседних символов, цена которой равна 1. Тогда расстояние Дамерау — Левенштейна можно подсчитать по формуле (1.2)

$$D(i,j) = \begin{cases} 0, \quad i = 0, j = 0 \\ i, \quad j = 0, i > 0 \\ j, \quad i = 0, j > 0 \\ \min\{ \\ \qquad D(i, j-1) + 1, \\ \qquad D(i-1, j) + 1, \\ \qquad D(i-1, j-1) + \\ \qquad \begin{cases} 0, \quad \text{если } s_1[i] = s_2[j] \\ 1, \quad \text{иначе} \end{cases} \\ \qquad , \\ \qquad \begin{cases} D(i-2, j-2) + 1, \\ \text{если } i, j > 1, s_1[i] = s_2[j-1], s_2[j] = s_1[i-1] \\ \infty, \quad \text{иначе} \end{cases} \\ \}, \quad i > 0, j > 0 \end{cases} \tag{1.2}$$

Вывод из аналитической части

В данном разделе были рассмотрены алгоритмы вычисления расстояний Левенштейна и Дамерау — Левенштейна. Формулы вычисления расстояний являются рекуррентными, поэтому каждый из алгоритмов может быть реализован как с использованием рекурсии, так и без.

2 Конструкторская часть

В данном разделе разрабатываются следующие алгоритмы:

– нерекурсивный алгоритм поиска расстояния Левенштейна;
– нерекурсивный алгоритм поиска расстояния Дамерау — Левенштейна;
– рекурсивный алгоритм поиска расстояния Дамерау — Левенштейна без кеширования;
– рекурсивный алгоритм поиска расстояния Дамерау — Левенштейна с кешированием.

В нерекурсивных алгоритмах для сохранения промежуточных результатов вычислений используется матрица, которая построчно заполняется в ходе работы алгоритма. Матрица так же используется в рекурсивном алгоритме поиска с кешированием.

2.1 Разработка алгоритмов

В данном подразделе представлены схемы алгоритмов:

1) на рисунке 2.1 – схема нерекурсивного алгоритма поиска расстояния Левенштейна;

2) на рисунке 2.2 – схема нерекурсивного алгоритма поиска расстояния Дамерау — Левенштейна;

3) на рисунке 2.3 – схема рекурсивного алгоритма поиска расстояния Дамерау — Левенштейна;

4) на рисунке 2.4 – схема рекурсивной подпрограммы алгоритма поиска расстояния Дамерау — Левенштейна;

5) на рисунке 2.5 – схема рекурсивного алгоритма поиска расстояния Дамерау — Левенштейна с кешированием;

6) на рисунке 2.6 – схема рекурсивной подпрограммы алгоритма поиска расстояния Дамерау — Левенштейна с кешированием.

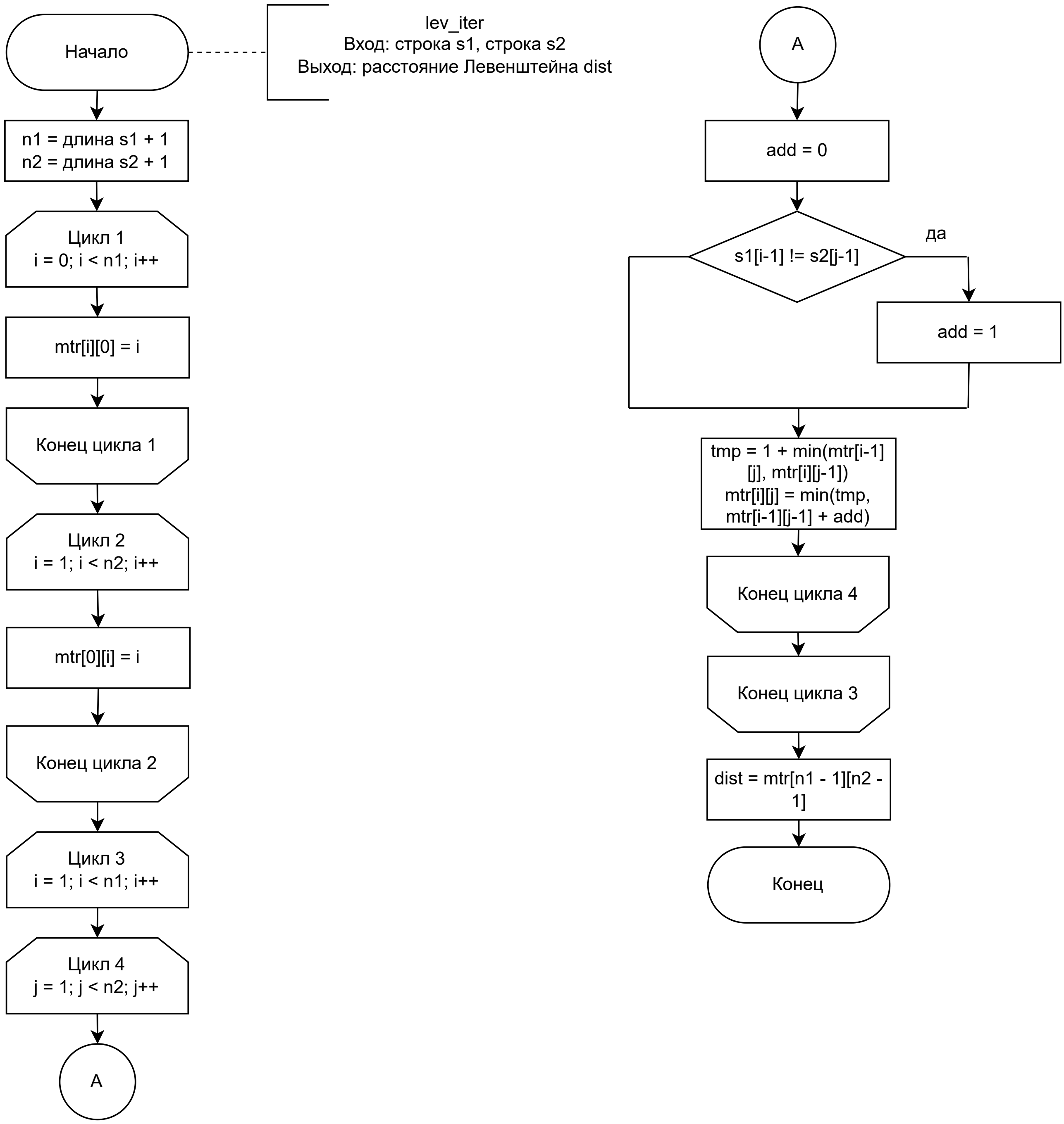

Рисунок 2.1 – Схема нерекурсивного алгоритма поиска расстояния Левенштейна

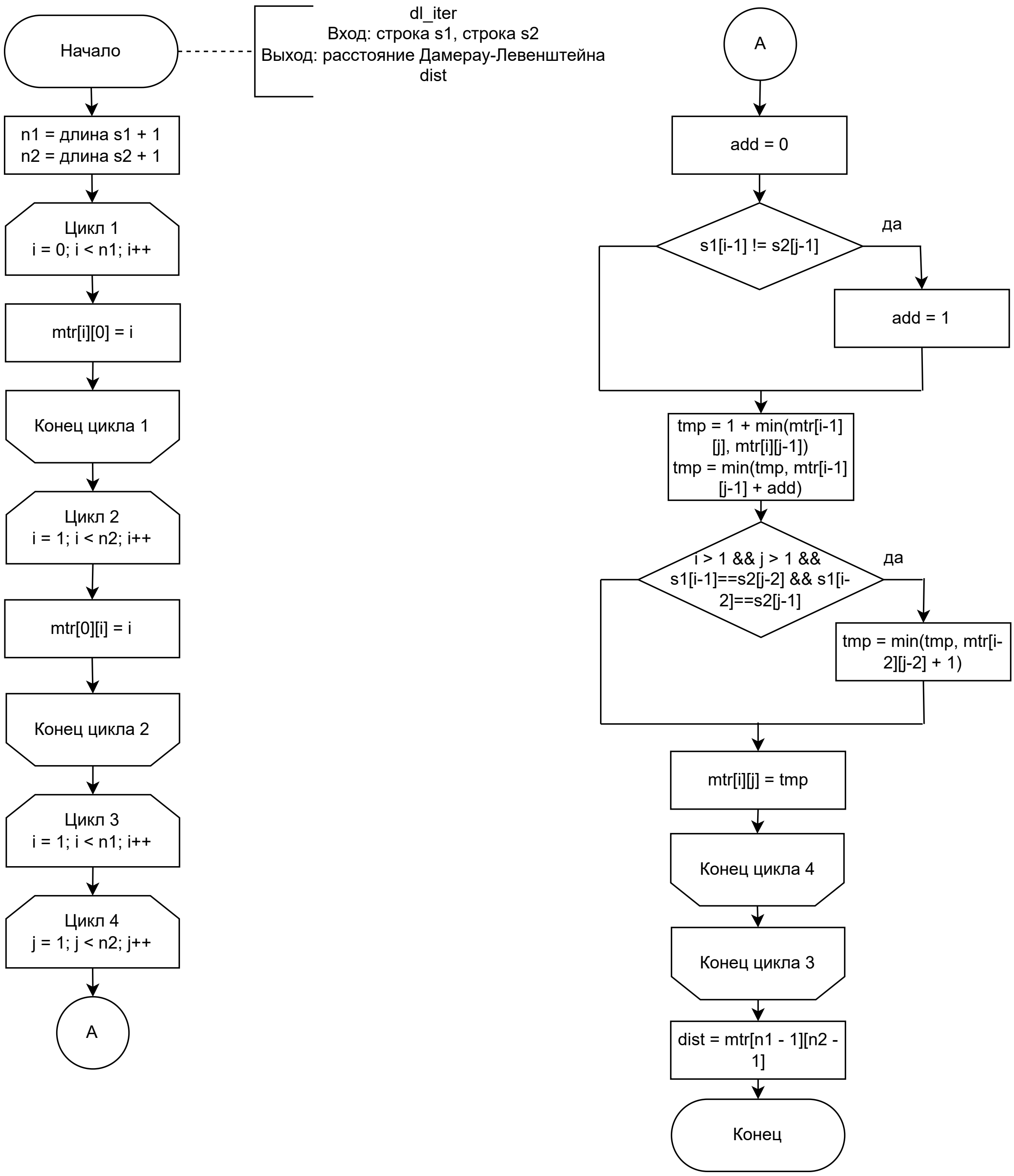

Рисунок 2.2 – Схема нерекурсивного алгоритма поиска расстояния Дамерау — Левенштейна

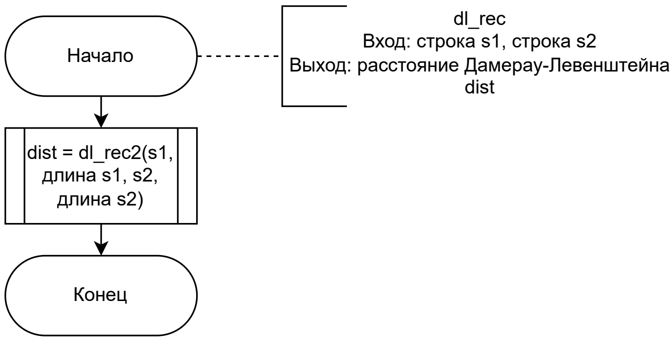

Рисунок 2.3 – Схема рекурсивного алгоритма поиска расстояния Дамерау — Левенштейна

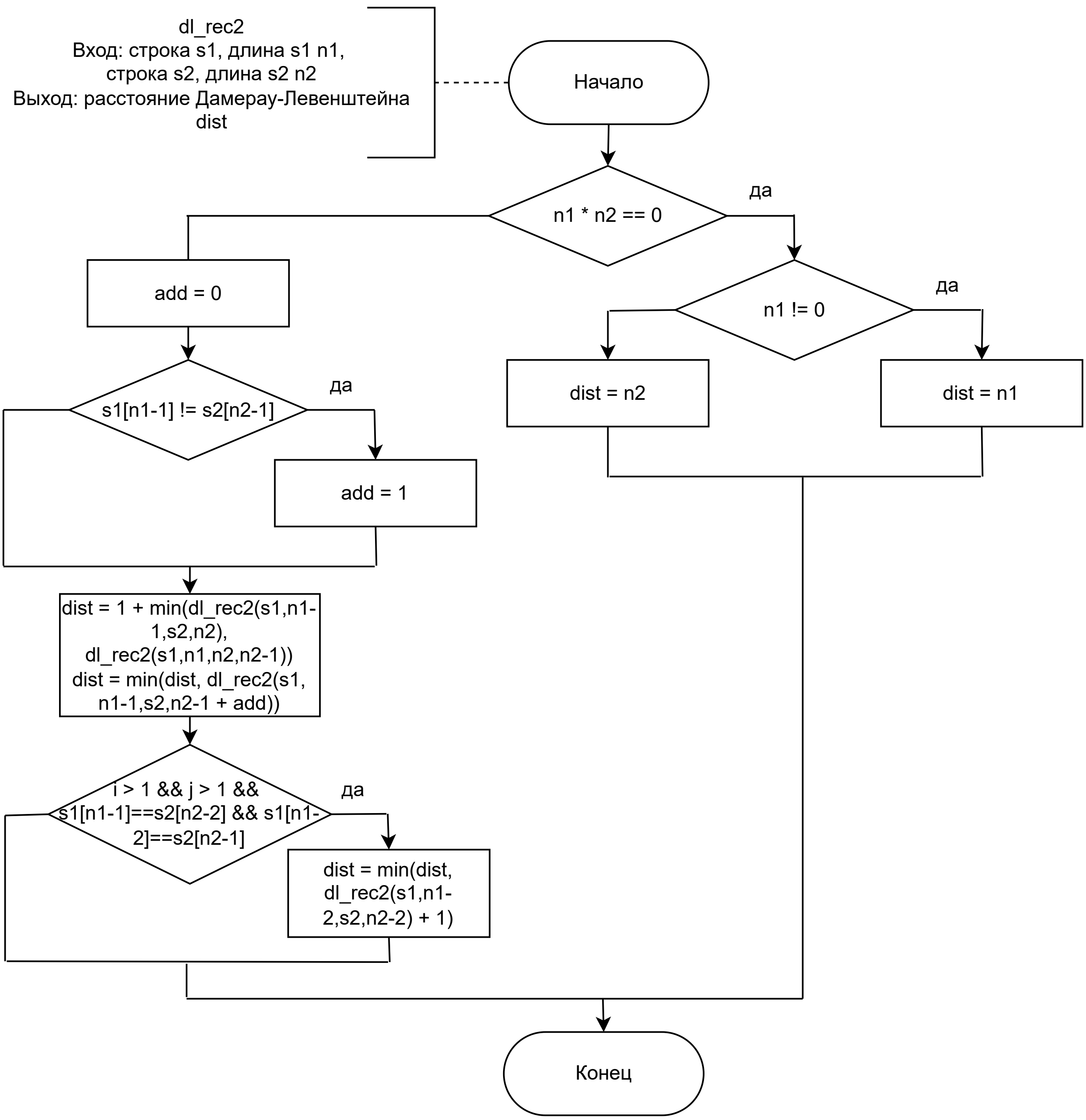

Рисунок 2.4 – Схема рекурсивной подпрограммы алгоритма поиска расстояния Дамерау — Левенштейна

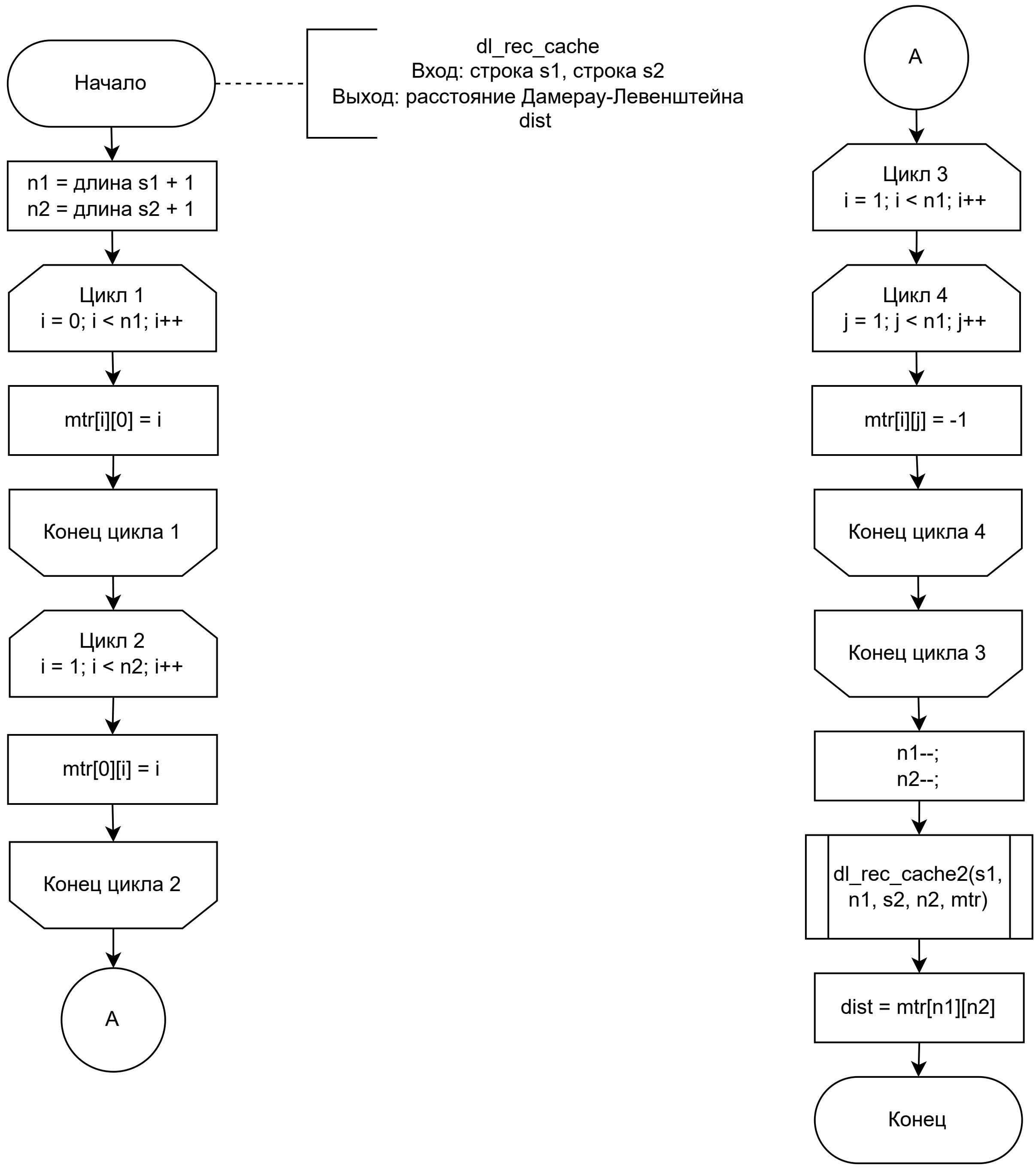

Рисунок 2.5 – Схема рекурсивного алгоритма поиска расстояния Дамерау — Левенштейна с кешированием

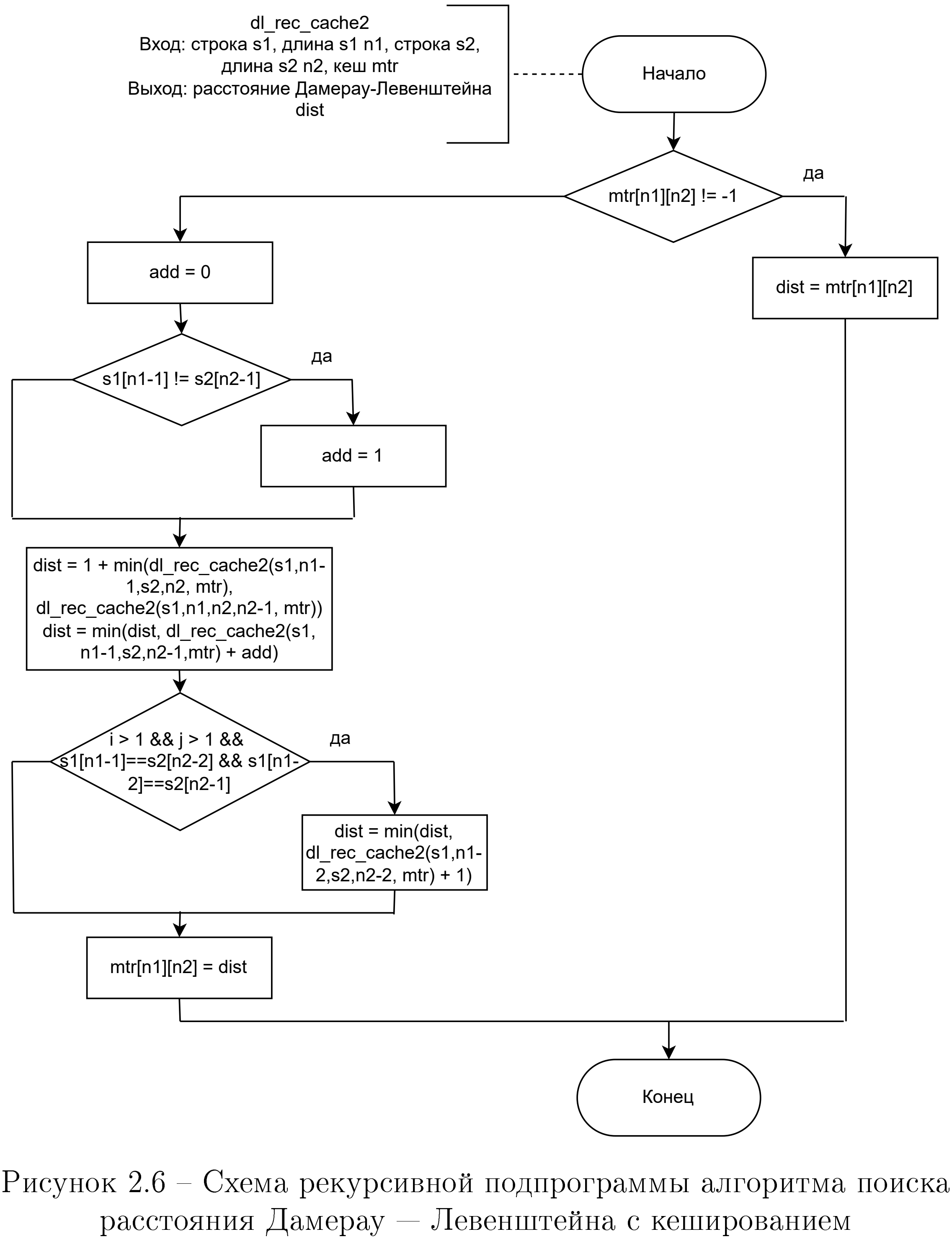

Рисунок 2.6 – Схема рекурсивной подпрограммы алгоритма поиска расстояния Дамерау — Левенштейна с кешированием

Вывод из конструкторской части

В данном разделе были разработаны следующие алгоритмы: нерекурсивный алгоритм поиска расстояния Левентейна, нерекурсивный алгоритм поиска расстояния Дамерау — Левенштейна, рекурсивный алгоритм поиска расстояния Дамерау — Левенштейна с кешированием и без.

3 Технологическая часть

3.1 Средства реализации

Для реализации данной лабораторной работы был выбран язык C++ [2]. В его стандартной библиотеке содержится класс wstring, работающий и с латиницей, и с кириллицей [3]. В качестве среды разработки был выбран текстовый редактор VS Code. Замеры процессорного времени проводились при помощи функции clock из библиотеки time.h [4].

3.2 Реализации алгоритмов

В данном подразделе представлены листинги кода ранее описанных алгоритмов:

– алгоритм нерекурсивного метода поиска расстояния Левенштейна (листинг 3.1);
– алгоритм нерекурсивного метода поиска расстояния Дамерау — Левенштейна (листинг 3.2);
– алгоритм рекурсивного метода поиска расстояния Дамерау — Левенштейна (листинг 3.3);
– алгоритм рекурсивного метода поиска расстояния Дамерау — Левенштейна с кешированием (листинги 3.4).

Листинг 3.1 – Реализация алгоритма нерекурсивного метода поиска расстояния Левенштейна

```
int lev_iter(wstring &s1, wstring &s2)
{
    int n1 = s1.length() + 1, n2 = s2.length() + 1;
    int **mtr = new int * [n1];
    int add, tmp, dist;

    for (int i = 0; i < n1; i++)
    {
        mtr[i] = new int [n2];
        mtr[i][0] = i;
    }

    for (int i = 1; i < n2; i++)
        mtr[0][i] = i;

    for (int i = 1; i < n1; i++)
        for (int j = 1; j < n2; j++)
        {
            add = 0;
            if (s1[i - 1] != s2[j - 1])
                add = 1;

            tmp = 1 + min(mtr[i - 1][j], mtr[i][j - 1]);
            mtr[i][j] = min(tmp, mtr[i - 1][j - 1] + add);
        }

    dist = mtr[n1 - 1][n2 - 1];

    for (int i = 0; i < n1; i++)
        delete[] mtr[i];
    delete[] mtr;

    return dist;
}
```

Листинг 3.2 – Реализация алгоритма нерекурсивного метода поиска расстояния Дамерау — Левенштейна

```
int dl_iter(wstring &s1, wstring &s2)
{
    int n1 = s1.length() + 1, n2 = s2.length() + 1;
    int **mtr = new int * [n1];
    int add, tmp, dist;

    for (int i = 0; i < n1; i++)
    {
        mtr[i] = new int [n2];
        mtr[i][0] = i;
    }

    for (int i = 1; i < n2; i++)
        mtr[0][i] = i;

    for (int i = 1; i < n1; i++)
        for (int j = 1; j < n2; j++)
        {
            add = 0;
            if (s1[i - 1] != s2[j - 1])
                add = 1;

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

            if (i > 1 && j > 1 && s1[i - 1] == s2[j - 2] && s1[i - 2] == s2[
               j - 1])
                tmp = min(tmp, mtr[i - 2][j - 2] + 1);

            mtr[i][j] = tmp;
        }

    dist = mtr[n1 - 1][n2 - 1];

    for (int i = 0; i < n1; i++)
        delete[] mtr[i];
    delete[] mtr;

    return dist;
}
```

Листинг 3.3 – Реализация алгоритма рекурсивного метода поиска расстояния Дамерау — Левенштейна

```
int dl_rec2(wstring &s1, int n1, wstring &s2, int n2)
{
    int dist;
    if (n1 * n2 == 0)
    {
        if (n1 != 0)
            dist = n1;
        else
            dist = n2;
    }
    else
    {
        int add = 0;
        if (s1[n1 - 1] != s2[n2 - 1])
            add = 1;

        dist = 1 + min(dl_rec2(s1, n1 - 1, s2, n2), dl_rec2(s1, n1, s2, n2 - 1));
        dist = min(dist, dl_rec2(s1, n1 - 1, s2, n2 - 1) + add);

        if (n1 > 1 && n2 > 1 && s1[n1 - 1] == s2[n2 - 2] && s1[n1 - 2] == s2[n2 - 1])
            dist = min(dist, dl_rec2(s1, n1 - 2, s2, n2 - 2) + 1);
    }
    return dist;
}

int dl_rec(wstring &s1, wstring &s2)
{
    return dl_rec2(s1, s1.length(), s2, s2.length());
}
```

Листинг 3.4 – Реализация алгоритма рекурсивного с кешированием метода поиска расстояния Дамерау — Левенштейна

```
int dl_rec_cache2(wstring &s1, int n1, wstring &s2, int n2, int **mtr)
{
    int dist;
    if (mtr[n1][n2] != -1)
        dist = mtr[n1][n2];
    else
    {
        int add = 0;
        if (s1[n1 - 1] != s2[n2 - 1])
            add = 1;
        dist = 1 + min(dl_rec_cache2(s1, n1 - 1, s2, n2, mtr), dl_rec_cache2(s1, n1, s2, n2 - 1, mtr));
        dist = min(dist, dl_rec_cache2(s1, n1 - 1, s2, n2 - 1, mtr) + add);
        if (n1 > 1 && n2 > 1 && s1[n1 - 1] == s2[n2 - 2] && s1[n1 - 2] == s2[n2 - 1])
            dist = min(dist, dl_rec_cache2(s1, n1 - 2, s2, n2 - 2, mtr) + 1);
        mtr[n1][n2] = dist;
    }
    return dist;
}
int dl_rec_cache(wstring &s1, wstring &s2)
{
    int n1 = s1.length() + 1, n2 = s2.length() + 1;
    int **mtr = new int * [n1];
    for (int i = 0; i < n1; i++)
    {
        mtr[i] = new int [n2];
        mtr[i][0] = i;
    }
    for (int i = 1; i < n2; i++)
        mtr[0][i] = i;
    for (int i = 1; i < n1; i++)
        for (int j = 1; j < n2; j++)
            mtr[i][j] = -1;
    n1--;
    n2--;
    dl_rec_cache2(s1, n1, s2, n2, mtr);
    int dist = mtr[n1][n2];

    for (int i = 0; i <= n1; i++)
        delete[] mtr[i];
    delete[] mtr;
    return dist;
}
```

3.3 Описание тестирования

В таблице 3.1 приведены тесты для алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна.

Таблица 3.1 – Тесты

Слово №1	**Слово №2**	**Ожидаемый результат**	**Результат**
s	c	1 1 1 1	1 1 1 1
hell	hallo	2 2 2 2	2 2 2 2
corn	cron	2 1 1 1	2 1 1 1
honda	hyundai	3 3 3 3	3 3 3 3
sun	sun	0 0 0 0	0 0 0 0
qwer	wqre	3 2 2 2	3 2 2 2
йцук	цйку	3 2 2 2	3 2 2 2

Вывод из технологической части

В данном разделе был выбран инструмент для замера процессорного времени, были реализованы алгоритмы поиска расстояний Левенштейна и Дамерау — Левенштейна. Также было проведено тестирование реализованных алгоритмов.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры времени:

- операционная система Windows 11 64–bit;
- оперативная память 16 ГБ;
- процессор 2.40 ГГц Intel Core i5–1135G7 [5].

4.2 Постановка эксперимента по замеру времени

Для оценки процессорного времени работы реализаций алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна был проведен эксперимент, в котором определялось влияние длины символьных последовательностей на время работы каждого из алгоритмов. Тестирование проводилось на словах длиной от 5 до 10 символов с шагом 1 и от 20 до 100 с шагом 10. На последовательностях длиной от 20 символов замеры для рекурсивного алгоритма без кеширования не выполнялись из-за быстрого увеличения времени работы. Поскольку методы замера процессорного времени имеют погрешность и возвращают для достаточно коротких задач константу 0, каждый алгоритм запускался по 500 раз, и для полученных 500 значений определялось среднее арифметическое, которое заносилось в таблицу результатов.

Результаты эксперимента были представлены в виде таблицы и графиков, приведенных в следующем подразделе.

4.3 Результаты эксперимента

Таблица 4.1 – Таблица времени работы реализаций алгоритмов (в мкс)

Длина слова	Лев.	Д.-Л. итер.	Д.-Л. рек.	Д.-Л. рек. кеш
5	0	0	0	0
6	0	0	60.80	0
7	0	0	319.94	0
8	0	0	1807.34	0
9	0	0	9831.58	0
10	0	12.57	55262.60	0
20	0	0	–	0
30	30.00	32.21	–	32.80
40	32.04	35.33	–	64.04
50	34.57	41.03	–	96.28
60	62.36	75.76	–	128.47
70	62.58	94.60	–	188.06
80	94.50	126.73	–	282.39
90	126.47	172.12	–	312.43
100	156.01	190.34	–	360.07

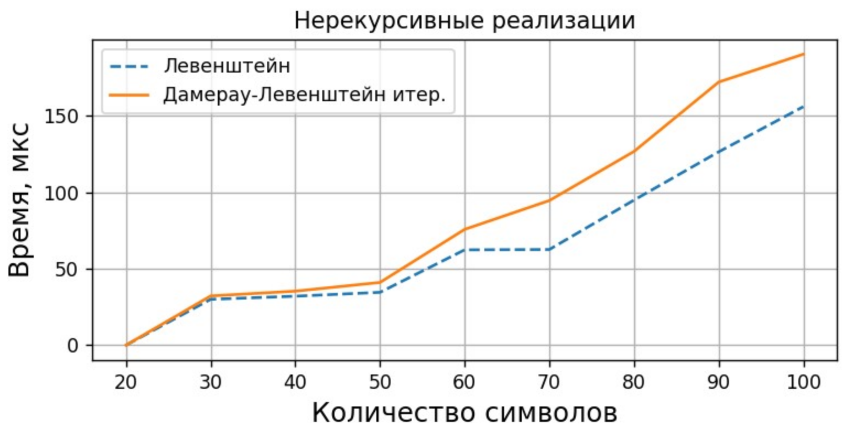

Рисунок 4.1 – Сравнение времени работы реализаций нерекурсивных алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна

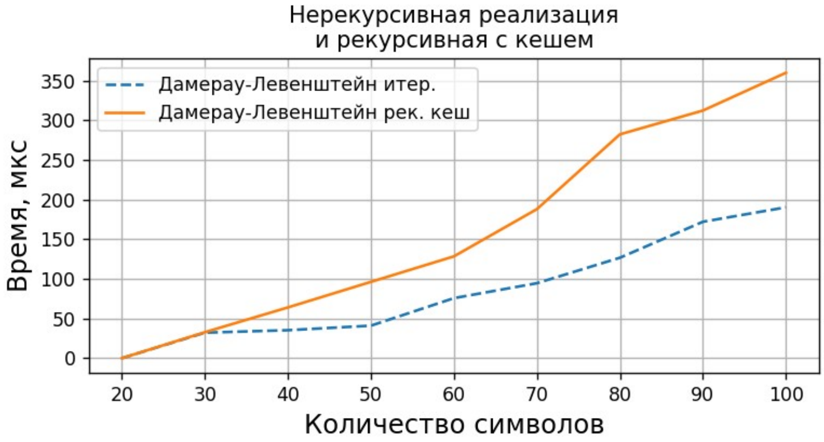

Рисунок 4.2 – Сравнение времени работы реализаций нерекурсивного алгоритма поиска расстояния Дамерау — Левенштейна и рекурсивного алгоритма поиска с кешем

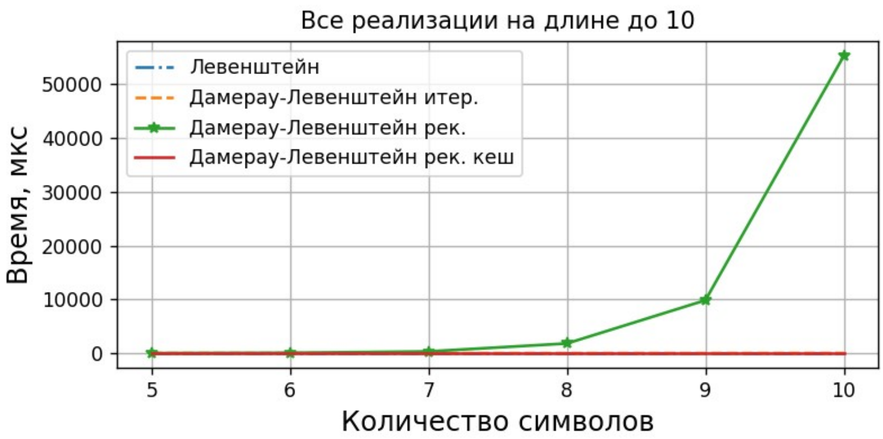

Рисунок 4.3 – Сравнение времени работы реализаций всех алгоритмов на последовательностях малой длины

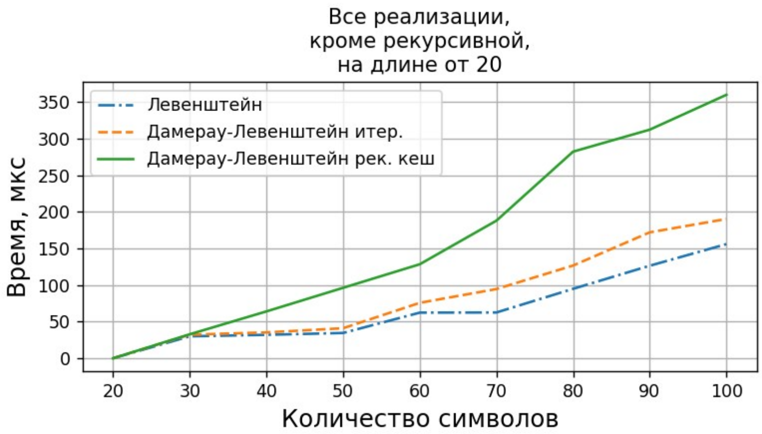

Рисунок 4.4 – Сравнение времени работы реализаций всех алгоритмов, кроме рекурсивного, на последовательностях средней и большой длины

Можно отметить, что на символьных последовательностях длиной до 30 символов реализация нерекурсивного алгоритма поиска расстояния Левенштейна и реализация нерекурсивного алгоритма поиска расстояния Дамерау — Левенштейна отрабатывают приблизительно за одинаковое время. Однако при увеличении длины последовательностей реализация алгоритма поиска расстояния Левенштейна становится более эффективной по времени. Она работает быстрее в 1.2 – 1.3 раза.

Нерекурсивная реализация алгоритма поиска расстояния Дамерау — Левенштейна и рекурсивная реализация с кешем работают в разы быстрее рекурсивной без кеша. Но на той же длине слов нерекурсивная реализация работает примерно в 1.9 раза быстрее рекурсивной реализации с кешем.

4.4 Расчет используемой памяти

Алгоритмы нахождения расстояний Левенштейна и Дамерау — Левенштейна не отличаются друг от друга с точки зрения используемой памяти.

Для каждого вызова рекурсивной реализации алгоритма Дамерау — Левенштейна выделяется память под:

- 2 строки;
- длины строк;
- локальная переменная;
- возвращаемое значение;
- адрес возврата.

Максимальное количество вызовов равно сумме длин двух строк.

Максимальное значение выделяемой памяти выражается формулой (4.1)

$$mem1 = (4 \cdot size(int) + 2 \cdot size(string)) \cdot (|s_1| + |s_2|) \tag{4.1}$$

Для единственного вызова нерекурсивной реализации алгоритма Дамерау — Левенштейна выделяется память под:

- 2 строки;
- длины строк;
- матрицу размерами, равными длинам строк, увеличинным на единицу;
- 4 локальных переменных;
- возвращаемое значение;
- адрес возврата.

Максимальное значение выделяемой памяти выражается формулой (4.2)

$$mem1 = 7 \cdot size(int) + 2 \cdot size(string) + size(int) \cdot (|s_1| + 1) \cdot (|s_2| + 1) \tag{4.2}$$

Память, используемая рекурсивной реализацией, растет пропорционально сумме длин строк, в то время как память, используемая нерекурсивной реализацией, растет пропорционально произведению длин строк, то есть по выделяемой памяти нерекурсивная реализация проигрывает рекурсивной. В рекурсивной реализации с кешем память так же растет пропорционально произведению длин строк.

Вывод из исследовательской части

По времени выполнения нерекурсивная реализация и реализация с кешем гораздо эффективнее рекурсивной без кеша. При этом нерекурсивная реализация работает в 1.9 раза быстрее рекурсивной реализации с кешем. Нерекурсивная реализация алгоритма поиска расстояния Левенштейна работает в 1.2 – 1.3 раза быстрее нерекурсивной реализации алгоритма нахождения расстояния Дамерау — Левенштейна.

По выделяемой памяти реализации, использующие матрицы, проигрывают рекурсивной без кеша: максимальный размер выделяемой памяти в них пропорционален произведению длин строк, в то время как в рекурсивной без кеша – сумме длин строк.

Заключение

В результате выполнения лабораторной работы при исследовании алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна были изучены и отработаны навыки динамического программирования.

В ходе выполения лабораторной работы:

1) были изучены расстояния Левенштейна и Дамерау — Левенштейна;

2) были разработаны алгоритмы нерекурсивного метода поиска расстояния Левенштейна, нерекурсивного метода поиска, рекурсивного метода поиска и рекурсивного с кешированием метода поиска расстояния Дамерау — Левенштейна;

3) был реализован каждый из описанных алгоритмов;

4) были выбраны инструменты для замера процессорного времени выполнения реализаций алгоритмов;

5) было проведено сравнение реализованных алгоритмов по времени работы: выявлено, что реализация рекурсивного алгоритма без кеширования уступает по времени всем другим реализациям, нерекурсивная реализация работает в 1.9 раза быстрее рекурсивной реализации с кешем, нерекурсивная реализация алгоритма поиска расстояния Левенштейна работает в 1.2 – 1.3 раза быстрее нерекурсивной реализации алгоритма нахождения расстояния Дамерау — Левенштейна;

6) было проведено сравнение реализованных алгоритмов по используемой памяти: выявлено, что реализации, использующие матрицу, уступают рекурсивной без кеша.

Таким образом, все поставленные задачи были выполнены, а цель достигнута.

Список использованных источников

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

[2] Справочник по языку C++ – Microsoft Learn [Электронный ресурс]. Режим доступа: https://learn.microsoft.com/ru-ru/cpp/cpp/cpp-language-reference?view=msvc-170l (дата обращения: 18.11.2022).

[3] std::wstring – C++ [Электронный ресурс]. Режим доступа: https://cplusplus.com/reference/string/wstring/ (дата обращения: 15.11.2022).

[4] clock – ctime – C++ [Электронный ресурс]. Режим доступа: https://cplusplus.com/reference/ctime/clock/ (дата обращения: 15.11.2022).

[5] Intel [Электронный ресурс]. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/208658/intel-core-i51135g7-processor-8m-cache-up-to-4-20-ghz.html (дата обращения: 18.11.2022).