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: «Динамическое программирование»

Студент ИУ7-53Б (Группа) _____ (Подпись, дата) В. Марченко (И. О. Фамилия)

Преподаватель _____ (Подпись, дата) Ю. В. Строганов (И. О. Фамилия)

Преподаватель _____ (Подпись, дата) Л. Л. Волкова (И. О. Фамилия)

2022 г.

Оглавление

Введение

Динамическое программирование — метод оптимизации, приспособленный к операциям, в которых процесс принятия решений может быть разбит на отдельные этапы (шаги). Такие операции называются многошаговыми [1].

Алгоритмы поиска и сравнения последовательностей активно используются при работе с неструктурированными данными, обработке больших объемов информации, поисковых запросов и т. д. Возросший объем информации предъявляет все более высокие требования к качеству и скорости поиска [2].

Нечеткий поиск — это поиск информации, при котором выполняется сопоставление информации заданному образцу поиска или близкому к этому образцу значению. Алгоритмы нечеткого поиска используются в большинстве современных поисковых систем (например, для проверки орфографии) [3].

Задачи нечеткого поиска чаще всего возникают при коррекции ошибок, фильтрации нежелательных сообщений, обнаружении плагиата, поиске с учетом форм одного и того же слова и основаны на определении расстояния между строками. Эти методы используются также и в генетике [2].

В данной лабораторной работе динамическое программирование будет изучено на материале алгоритмов вычисления редакционного расстояния. Будут рассмотрены четыре алгоритма: итерационный алгоритм поиска расстояния Левенштейна, итерационный алгоритм поиска расстояния Дамерау-Левенштейна, рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна и рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кэшем.

1 Аналитическая часть

1.1 Цели и задачи

Цель работы: получить навыки динамического программирования на материале алгоритмов вычисления редакционного расстояния.

Задачи текущей лабораторной работы:

1) изучить понятие редакционного расстояния;

2) изучить и реализовать четыре алгоритма вычисления редакционного расстояния — итерационный алгоритм поиска расстояния Левенштейна, итерационный алгоритм поиска расстояния Дамерау-Левенштейна, рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна и рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кэшем;

3) провести сравнительный анализ затрачиваемой памяти всеми алгоритмами на основе теоретических расчетов;

4) провести сравнительный анализ времени работы алгоритмов на основе экспериментальных данных.

1.2 Итерационный алгоритм поиска расстояния Левенштейна

Расстояние Левенштейна между двумя строками в теории информации и компьютерной лингвистике — это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую [4].

Рекуррентная формула, описывающая расстояние Левенштейна:

$$d(S_1, S_2) = D(M, N), \text{ где} \tag{1.1}$$

$$D(i,j) = \begin{cases} 0, & \text{i} = 0, \text{j} = 0, \\ i, & \text{j} = 0, \text{i} > 0, \\ j, & \text{i} = 0, \text{j} > 0, \\ \min\{ & \\ \quad D(i,j-1)+1 & \\ \quad D(i-1,j)+1 & \text{i} > 0, \text{j} > 0. \\ \quad D(i-1,j-1)+m(a[i],b[j]) & \\ \}, & \end{cases} \tag{1.2}$$

Функция $m(a,b)$ определена как:

$$m(a,b) = \begin{cases} 0, & \text{если a} = \text{b}, \\ 1, & \text{иначе.} \end{cases}. \tag{1.3}$$

1.3 Итерационный алгоритм поиска расстояния Дамерау-Левенштейна

В алгоритме поиска расстояния Дамерау-Левенштейна добавляется еще одна операция — транспозиция (перестановка двух соседних символов).

Рекуррентная формула, описывающая расстояние Дамерау-Левенштейна:

$$d(S_1, S_2) = D(M, N), \text{ где} \tag{1.4}$$

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), \text{ если } \min(i,j) = 0, \\ \min\{ \\ \quad d_{a,b}(i,j-1) + 1, \\ \quad d_{a,b}(i-1,j) + 1, \\ \quad d_{a,b}(i-1,j-1) + m(a[i],b[j]), \text{ иначе} \\ \quad \begin{cases} d_{a,b}(i-2,j-2) + 1, \text{ если } i,j > 1, \\ \qquad a[i] = b[j-1], \\ \qquad b[j] = a[i-1], \\ \qquad \infty, \text{ иначе.} \end{cases} \\ \} \end{cases} \tag{1.5}$$

1.4 Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна

Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна отличается от своей итерационной версии тем, что вместо использования матрицы для хранения вычисленных ранее значений, необходимых для подсчета последующих, эти значения вычисляются каждый раз с помощью рекурсивных вызовов.

1.5 Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кэшем

Данная версия алгоритма поиска расстояния Дамерау-Левенштейна является оптимизацией рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна. Оптимизация заключается в использовании кэша, который представляет собой матрицу. В нее записываются значения, вычисленные на различных этапах рекурсии. Таким образом, при необходимости вычисления некоторого нового значения по реккурентной формуле, величины, необходимые для этого, не вычисляются заново каждый раз. Сначала проверяется, было ли

уже вычислено данное значение. В случае, если этого не происходило ранее, выполняются рекурсивные вычисления для получения нового значения.

Вывод из аналитической части

В текущем разделе были рассмотрены четыре алгоритма вычисления редакционного расстояния: итерационный алгоритм поиска расстояния Левенштейна, итерационный алгоритм поиска расстояния Дамерау-Левенштейна, рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна и рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кэшем.

2 Конструкторская часть

2.1 Описание алгоритмов

На рисунках 2.1–2.6 показаны схемы четырех алгоритмов вычисления редакционного расстояния.

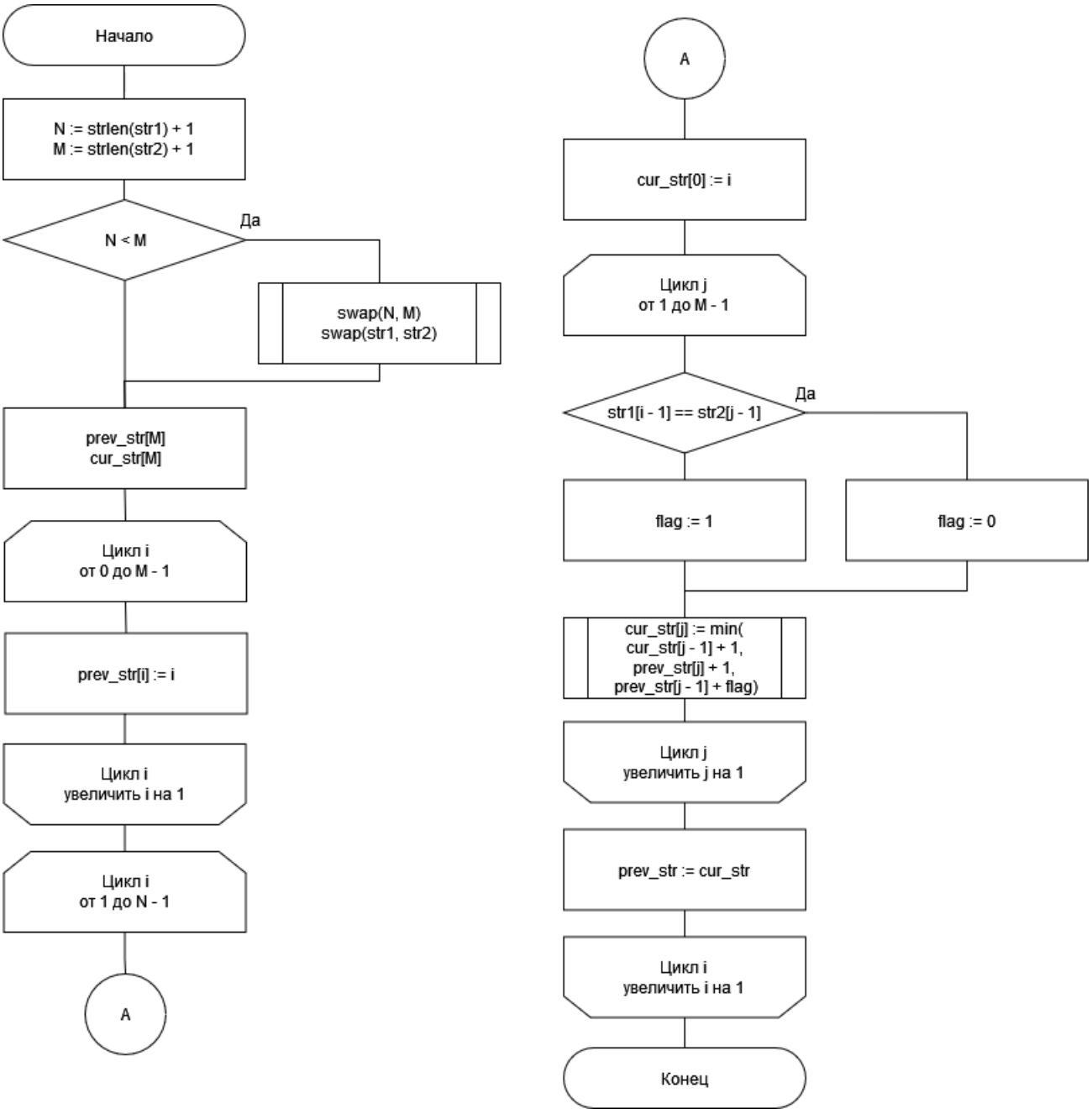

Рисунок 2.1 – Итерационный алгоритм поиска расстояния Левенштейна

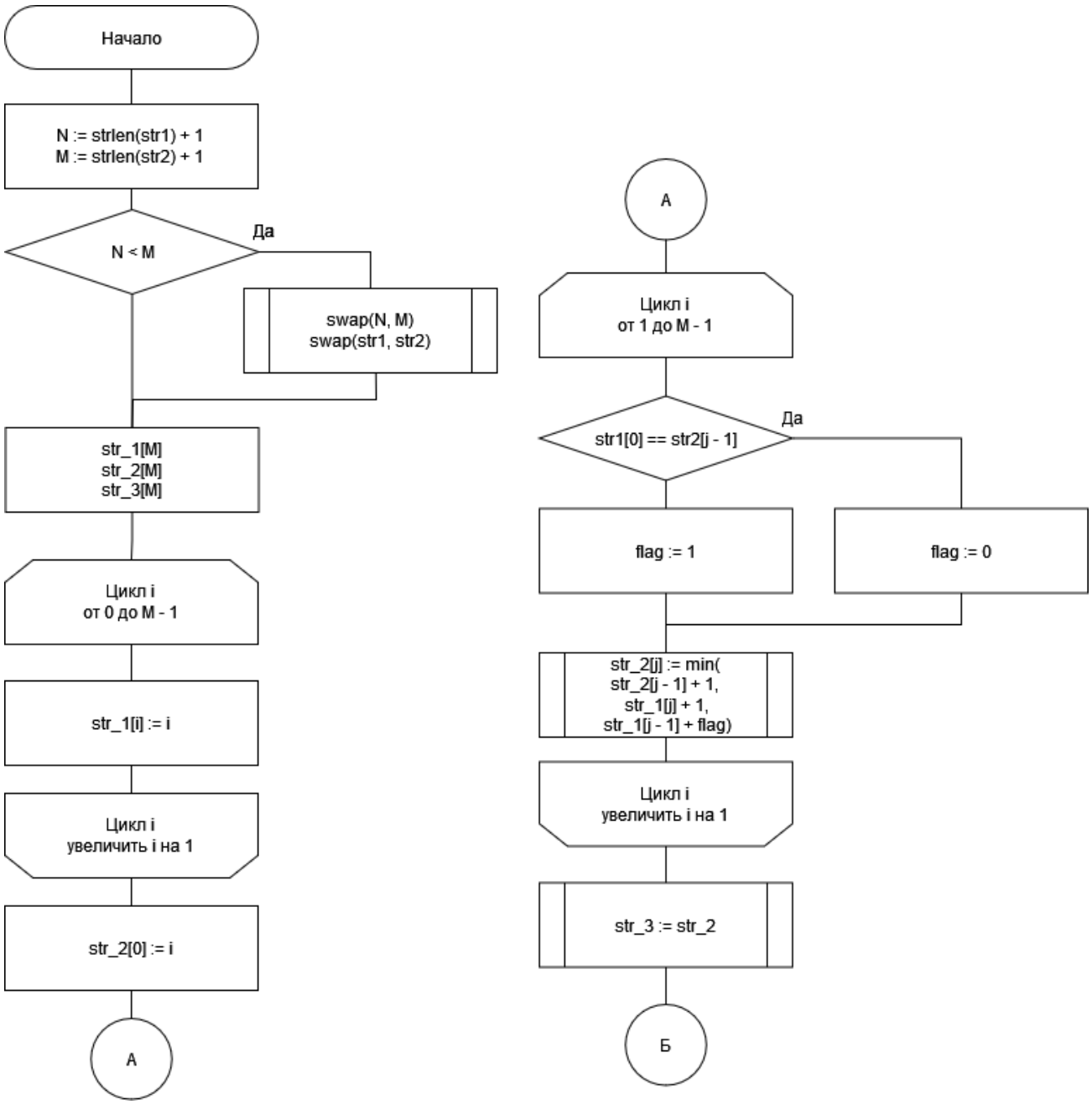

Рисунок 2.2 – Итерационный алгоритм поиска расстояния Дамерау-Левенштейна — ч. 1

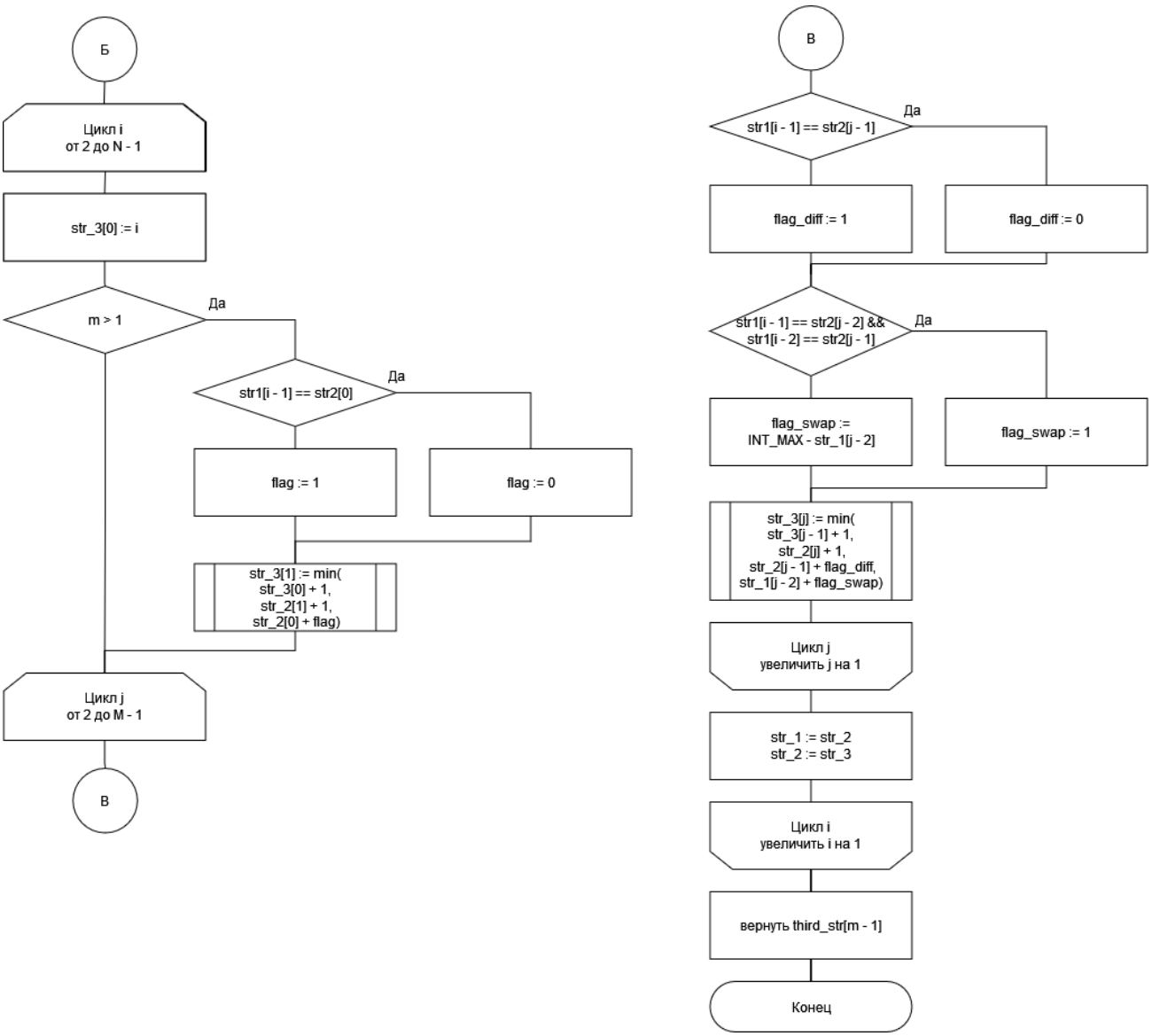

Рисунок 2.3 – Итерационный алгоритм поиска расстояния Дамерау-Левенштейна — ч. 2

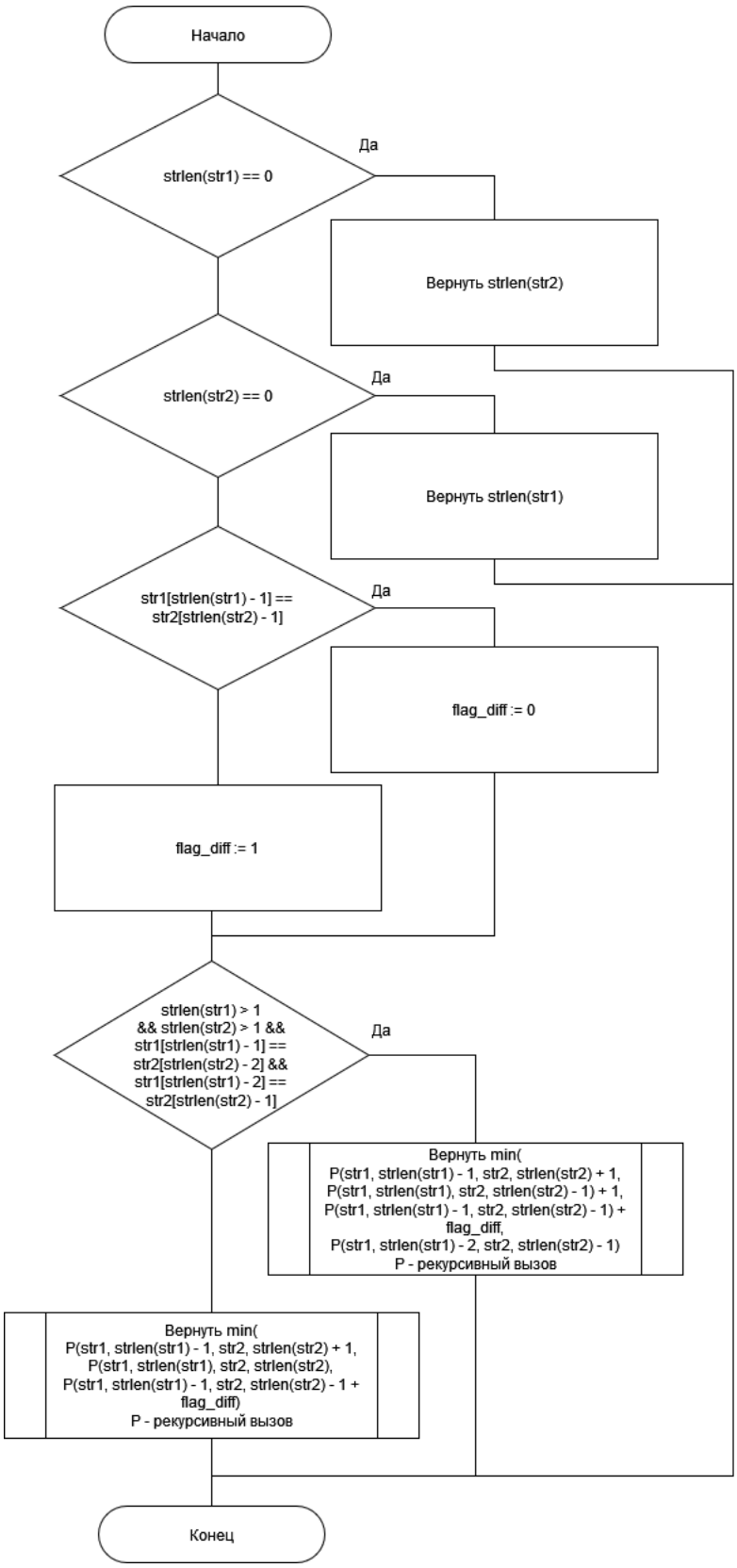

Рисунок 2.4 – Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна

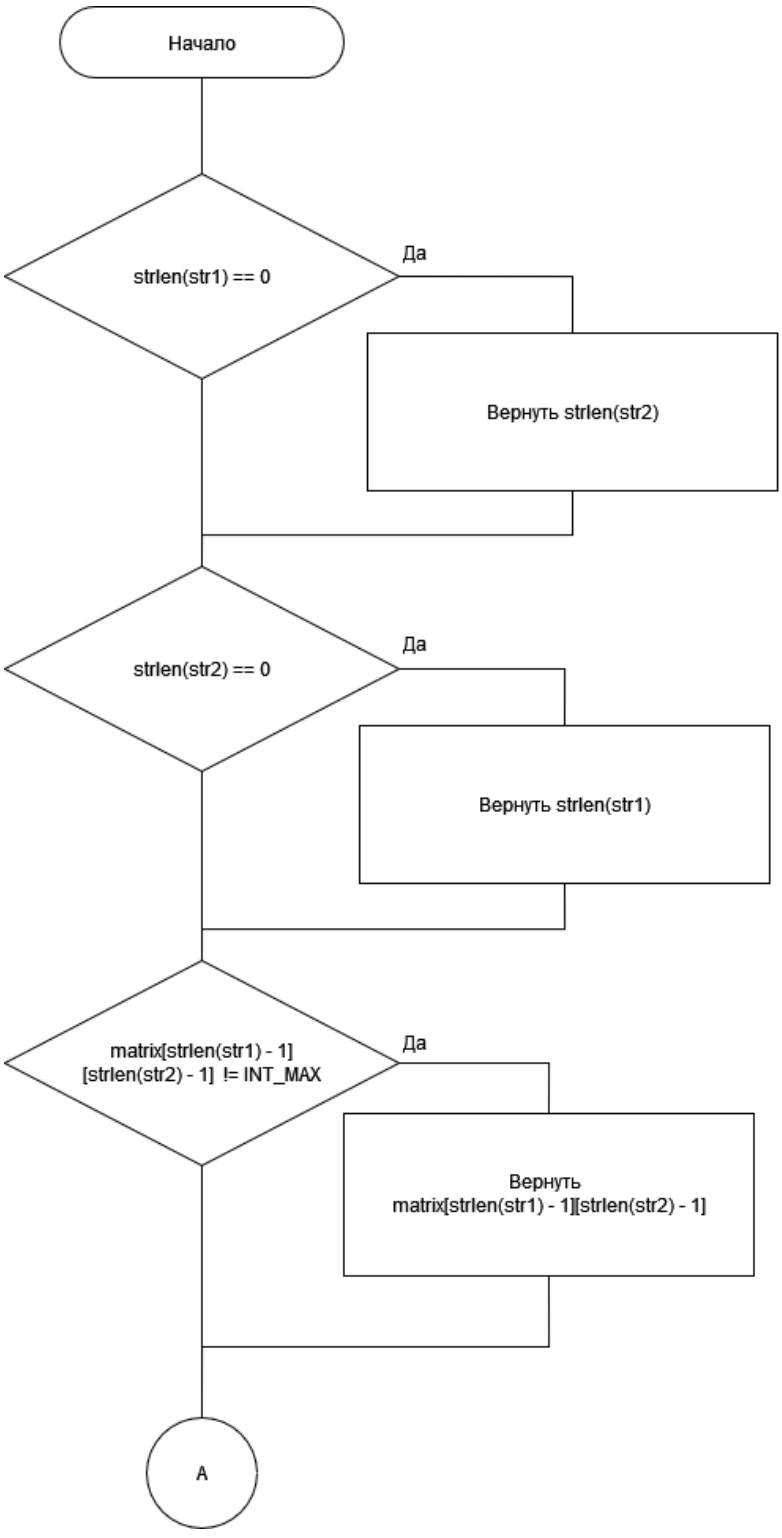

Рисунок 2.5 – Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кэшем — ч. 1

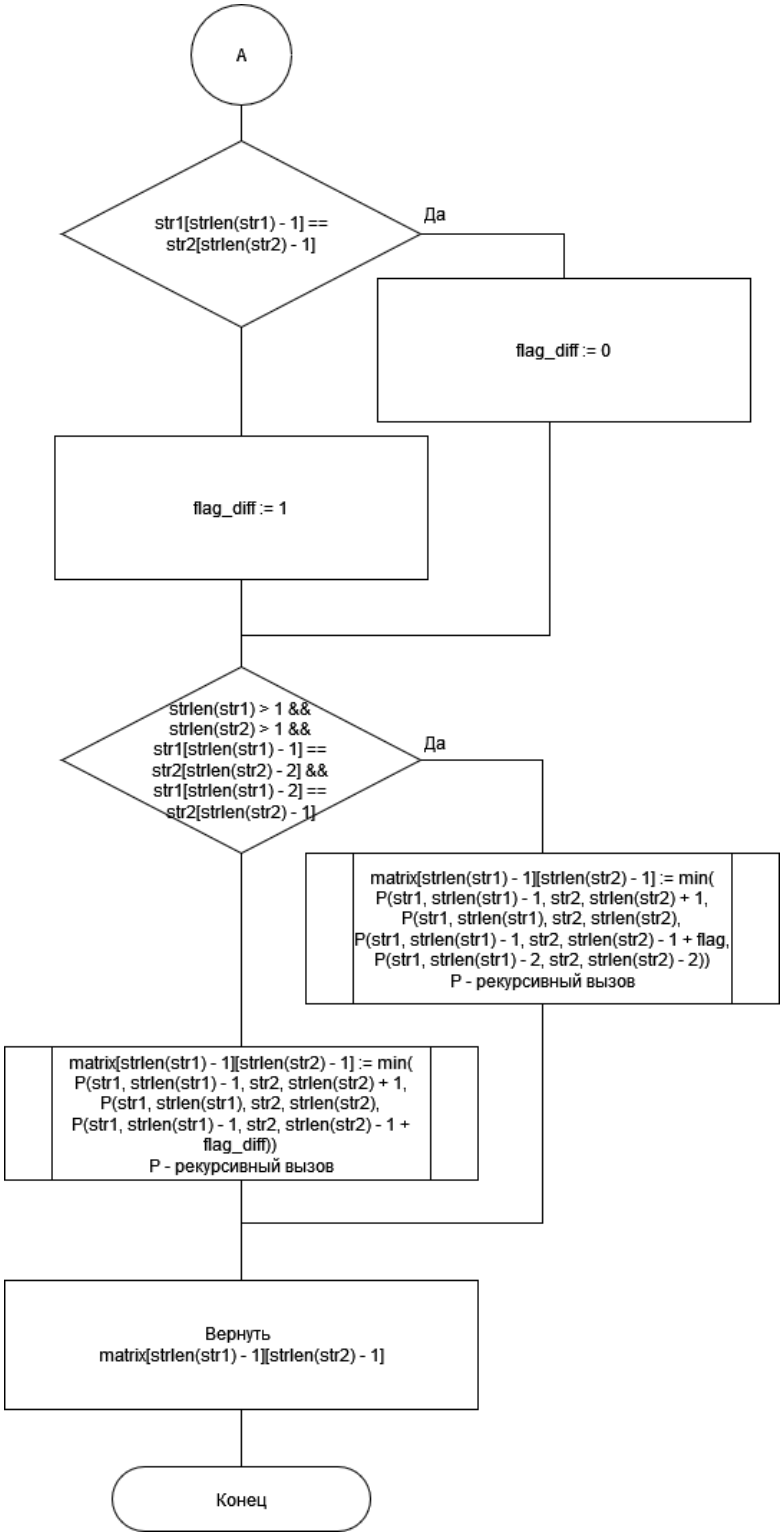

Рисунок 2.6 – Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кэшем — ч. 2

2.2 Описание типов данных

Для реализации алгоритма поиска расстояния Левенштейна и трех модификаций алгоритма поиска расстояния Дамерау-Левенштейна были использованы следующие типы данных:

1) строка — класс **std::string** из стандартной библиотеки языка **C++**;
2) длина строки — переменная типа **std::size_t**;
3) матрица — вектор векторов типа **std::vector**, состоящий из элементов типа **int**.

2.3 Оценка затрат алгоритмов по памяти

Итерационные алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна не отличаются по использованию памяти.

2.3.1 Итерационный алгоритм поиска расстояния Левенштейна (Дамерау-Левенштейна)

Затраты по памяти для итерационного алгоритма поиска расстояния Левенштейна (Дамерау-Левенштейна).

1. Длины строк N и M: $2 \cdot \text{sizeof}(int)$.
2. Строки: $(N + M + 2) \cdot \text{sizeof}(char)$.
3. Матрица: $(N + 1) \cdot (M + 1) \cdot \text{sizeof}(int) + 2 \cdot \text{sizeof}(int)$.
4. Вспомогательные переменные i, j: $2 \cdot \text{sizeof}(int)$.
5. Адрес возврата.

Суммарные затраты по памяти:

$$I = (N + 1) \cdot (M + 1) \cdot \text{sizeof}(int) + 4 \cdot \text{sizeof}(int) + \\ + (N + M + 2) \cdot \text{sizeof}(char). \tag{2.1}$$

2.3.2 Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна

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

1. Длины строк N и M: $2 \cdot \text{sizeof}(int)$.
2. Строки: $(N + M + 2) \cdot \text{sizeof}(char)$.
3. Вспомогательные переменные i, j: $2 \cdot \text{sizeof}(int)$.
4. Адрес возврата.

Пусть R — количество рекурсивных вызовов. Тогда суммарные затраты по памяти:

$$I = R \cdot (4 \cdot \text{sizeof}(int) + (N + M + 2) \cdot \text{sizeof}(char)). \tag{2.2}$$

2.3.3 Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кэшем

Затраты по памяти для рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кэшем для одного вызова.

1. Длины строк N и M: $2 \cdot \text{sizeof}(int)$.
2. Строки: $(N + M + 2) \cdot \text{sizeof}(char)$.
3. Матрица: $(N + 1) \cdot (M + 1) \cdot \text{sizeof}(int) + 2 \cdot \text{sizeof}(int)$.
4. Вспомогательные переменные i, j: $2 \cdot \text{sizeof}(int)$.
5. Адрес возврата.

Пусть R — количество рекурсивных вызовов. Тогда суммарные затраты по памяти:

$$I = R \cdot (6 \cdot \text{sizeof}(int) + (N + M + 2) \cdot \text{sizeof}(char)) + \\ + (N + 1) \cdot (M + 1) \cdot \text{sizeof}(int). \tag{2.3}$$

Вывод из конструкторской части

В текущем разделе на основе теоретических данных, полученных из аналитического раздела, для четырех рассматриваемых в данной лабораторной работе алгоритмов:

1) были построены схемы;
2) были выбраны необходимые для реализации этих алгоритмов типы данных;
3) была проведена оценка затрачиваемого объема памяти.

3 Технологическая часть

В текущем разделе приведены средства реализации алгоритмов поиска редакционного расстояния и листинги кода.

3.1 Требования к программному обеспечению

Программа должна запрашивать у пользователя две строки и порядковый номер алгоритма поиска редакционного расстояния. Затем на экран должен быть выведен результат — вычисленное редакционное расстояние.

3.2 Средства реализации

Для реализации программного обеспечения был выбран язык **C++** ввиду следующих причин:

1) в библиотеке стандартных шаблонов имеется контейнер **std::vector**, который можно использовать для создания матриц;

2) в стандартной библиотеке есть класс **std::string**;

3) для считывания данных и вывода их на экран в стандартной библиотеке существуют соответственно функции **scanf()** и **printf()**.

Таким образом, с помощью языка **C++** можно реализовать программное обеспечение, которое соответствует перечисленным выше требованиям.

3.3 Реализация алгоритмов

3.3.1 Итерационный алгоритм поиска расстояния Левенштейна

В листинге 3.1 показана реализация итерационного алгоритма поиска расстояния Левенштейна.

Листинг 3.1 – Реализация итерационного алгоритма поиска расстояния Левенштейна

```
int levenshtein(std::string str_1, const std::size_t len_1,
   std::string str_2, const std::size_t len_2)
{
    size_t n = len_1 + 1;
    size_t m = len_2 + 1;
    if (n < m)
    {
        std::swap(n, m);
        std::swap(str_1, str_2);
    }
    std::vector<int> prev_str_1(m), curr_str_1(m);
    for (std::size_t i = 0; i < m; i++)
        prev_str_1[i] = i;
    for (std::size_t i = 1; i < n; i++)
    {
        curr_str_1[0] = i;
        for (std::size_t j = 1; j < m; j++)
        {
            bool flag = str_1[i - 1] == str_2[j - 1] ? false :
               true;
            curr_str_1[j] = find_min(3, curr_str_1[j - 1] + 1,
            prev_str_1[j] + 1, prev_str_1[j - 1] + flag);
        }
        prev_str_1 = curr_str_1;
    }
    return curr_str_1[m - 1];
}
```

3.3.2 Итерационный алгоритм поиска расстояния Дамерау-Левенштейна

В листинге 3.2 показана реализация итерационного алгоритма поиска расстояния Дамерау-Левенштейна.

Листинг 3.2 – Реализация итерационного алгоритма поиска расстояния Дамерау-Левенштейна

```
int damerau_levenshtein(std::string str_1, const std::size_t
   len_1, std::string str_2, const std::size_t len_2)
{
    if (len_1 == 0)
        return len_2;
    if (len_2 == 0)
        return len_1;
    size_t n = len_1 + 1;
    size_t m = len_2 + 1;
    if (n < m)
    {
        std::swap(n, m);
        std::swap(str_1, str_2);
    }
    std::vector<int> str_11(m), str_21(m), str_31(m);
    for (std::size_t i = 0; i < m; i++)
        str_11[i] = i;
    str_21[0] = 1;
    for (std::size_t j = 1; j < m; j++)
    {
        bool flag_diff = str_1[0] == str_2[j - 1] ? false : true;
        str_21[j] = find_min(3, str_21[j - 1] + 1, str_11[j] +
           1, str_11[j - 1] + flag_diff);
    }
    str_31 = str_21;
    for (std::size_t i = 2; i < n; i++)
    {
        str_31[0] = i;
        if (1 < m)
        {
```

```
29              bool flag_diff = str_1[i - 1] == str_2[0] ? false :
                    true;
30              str_31[1] = find_min(3, str_31[0] + 1, str_21[1] + 1,
31              str_21[0] + flag_diff);
32          }
33          for (std::size_t j = 2; j < m; j++)
34          {
35              bool flag_diff = str_1[i - 1] == str_2[j - 1] ?
                    false : true;
36              int flag_swap = ((str_1[i - 1] == str_2[j - 2]) &&
                    (str_1[i - 2] == str_2[j - 1])) ? 1 : INT_MAX -
                    str_11[j - 2];
37              str_31[j] = find_min(4, str_31[j - 1] + 1, str_21[j]
                    + 1, str_21[j - 1] + flag_diff, str_11[j - 2] +
                    flag_swap);
38          }
39          str_11 = str_21;
40          str_21 = str_31;
41      }
42      return str_31[m - 1];
43  }
```

3.3.3 Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна

В листинге 3.3 показана реализация рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна.

Листинг 3.3 – Реализация рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна

```
1  int damerau_levenshtein_rec(std::string str_1, const std::size_t
       len_1, std::string str_2, const std::size_t len_2)
2  {
3      if (len_1 == 0)
4          return len_2;
5      if (len_2 == 0)
6          return len_1;
7      bool flag_diff;
```

```
8       if (str_1[len_1 - 1] == str_2[len_2 - 1])
9           flag_diff = false;
10      else
11          flag_diff = true;
12      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]))
13          return find_min(4,
14              (damerau_levenshtein_rec(str_1, len_1 - 1, str_2,
                   len_2) + 1),
15              (damerau_levenshtein_rec(str_1, len_1, str_2, len_2
                   - 1) + 1),
16              (damerau_levenshtein_rec(str_1, len_1 - 1, str_2,
                   len_2 - 1) + flag_diff),
17              (damerau_levenshtein_rec(str_1, len_1 - 2, str_2,
                   len_2 - 2) + 1));
18      else
19          return find_min(3,
20              (damerau_levenshtein_rec(str_1, len_1 - 1, str_2,
                   len_2) + 1),
21              (damerau_levenshtein_rec(str_1, len_1, str_2, len_2
                   - 1) + 1),
22              (damerau_levenshtein_rec(str_1, len_1 - 1, str_2,
                   len_2 - 1) + flag_diff));
23  }
```

3.3.4 Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кэшем

В листинге 3.4 показана реализация рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кэшем.

Листинг 3.4 – Реализация рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кэшем

```
1   static int damerau_levenshtein_cache_function(std::string str_1,
       const std::size_t len_1, std::string str_2, const std::size_t
       len_2, std::vector<std::vector<int>>& matrix)
2   {
```

```
3       if (len_1 == 0)
4           return len_2;
5       if (len_2 == 0)
6           return len_1;
7       if (matrix[len_1 - 1][len_2 - 1] != INT_MAX)
8           return matrix[len_1 - 1][len_2 - 1];
9       bool flag_diff;
10      if (str_1[len_1 - 1] == str_2[len_2 - 1])
11          flag_diff = false;
12      else
13          flag_diff = true;
14      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]))
15          matrix[len_1 - 1][len_2 - 1] = find_min(4,
16              damerau_levenshtein_cache_function(str_1, len_1 - 1,
                    str_2, len_2, matrix) + 1,
17              damerau_levenshtein_cache_function(str_1, len_1,
                    str_2, len_2 - 1, matrix) + 1,
18              damerau_levenshtein_cache_function(str_1, len_1 - 1,
                    str_2, len_2 - 1, matrix) + flag_diff,
19              damerau_levenshtein_cache_function(str_1, len_1 - 2,
                    str_2, len_2 - 2, matrix) + 1);
20      else
21          matrix[len_1 - 1][len_2 - 1] = find_min(3,
22              damerau_levenshtein_cache_function(str_1, len_1 - 1,
                    str_2, len_2, matrix) + 1,
23              damerau_levenshtein_cache_function(str_1, len_1,
                    str_2, len_2 - 1, matrix) + 1,
24              damerau_levenshtein_cache_function(str_1, len_1 - 1,
                    str_2, len_2 - 1, matrix) + flag_diff);
25      return matrix[len_1 - 1][len_2 - 1];
26  }
27
28  int damerau_levenshtein_cache(std::string str_1, const
        std::size_t len_1, std::string str_2, const std::size_t len_2)
29  {
30      std::vector<std::vector<int>> matrix(len_1,
            std::vector<int>(len_2, INT_MAX));
```

```
31      return damerau_levenshtein_cache_function(str_1, len_1,
            str_2, len_2, matrix);
32  }
```

3.4 Тестовые данные

Пусть λ — пустая строка. В таблице 3.1 приведены тестовые данные для четырех функций, реализующих алгоритмы поиска редакционного расстояния. Тесты выполнялись по методологии черного ящика (модульное тестирование). Все тесты пройдены успешно.

Таблица 3.1 – Тестовые данные

Первая строка	**Вторая строка**	**Расстояние Левенштейна**	**Расстояние Дамерау-Левенштейна**
λ	λ	0	0
donut	λ	5	5
λ	bread	5	5
milk	milk	0	0
rabbit	rabqit	1	1
excited	ecxited	2	1
friend	frien	1	1

Вывод из технологической части

В данном разделе был написан исходный код четырех алгоритмов поиска редакционного расстояния. Описаны тесты и приведены результаты тестирования.

4 Исследовательская часть

4.1 Технические характеристики устройства

Технические характеристики устройства, на котором было проведено измерение времени работы алгоритмов:

1) операционная система Windows 11 Pro x64;
2) оперативная память 16 ГБ;
3) процессор Intel® Core™ i7-4790K @ 4.00 ГГц.

4.2 Время работы алгоритмов

Время работы функций замерены с помощью ассемблерной инструкции **rdtsc**, которая читает счетчик Time Stamp Counter и возвращает 64-битное количество тиков с момента последнего сброса процессора.

В таблице 4.1 приведено время работы в тиках четырех функций, реализующих алгоритмы поиска редакционного расстояния при различном количестве символов во входных строках. На рисунке 4.1 изображена зависимость времени работы в тиках алгоритмов поиска редакционного расстояния от количества символов во входных строках.

Таблица 4.1 – Время работы в тиках алгоритмов поиска редакционного расстояния в зависимости от количества символов во входных строках

Кол-во символов	**Алгоритм Левенштейна (тики)**	**Алгоритм Дамерау-Левенштейна (тики)**	**Алгоритм Дамерау-Левенштейна рекурсивный (тики)**	**Алгоритм Дамерау-Левенштейна рекурсивный с кэшем (тики)**
1	6376	7060	2532	12360
2	7296	8572	6200	14452
3	10012	11216	18792	17988
4	10224	12612	24172	28584
5	10384	13668	25272	30320
6	13636	15496	30556	34984
7	136727	171927	575907	36432
8	177248	175408	665208	55660
9	43902	18156	120388	63300
10	45228	18308	239113	114828

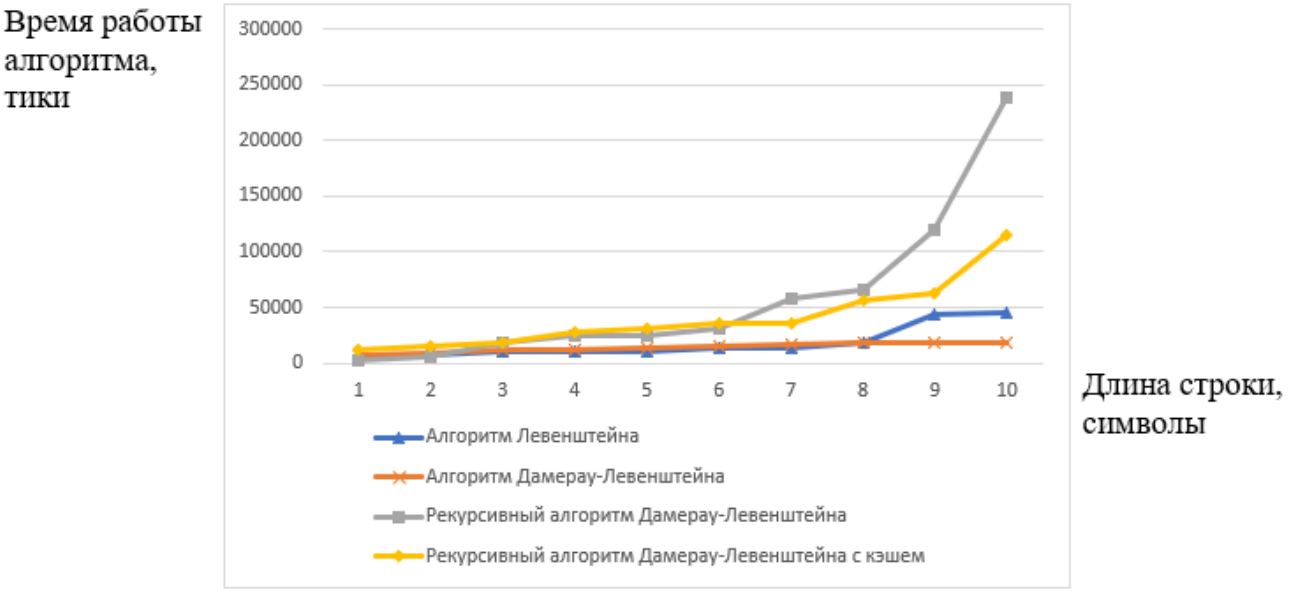

Рисунок 4.1 – Зависимость времени работы в тиках алгоритмов поиска редакционного расстояния в зависимости от количества символов во входных строках

Вывод из исследовательской части

Согласно полученным при проведении эксперимента данным, наиболее эффективным по скорости работы можно считать итерационный алгоритм поиска расстояния Дамерау-Левенштейна. Наименее эффективным оказался рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна. Его оптимизированная версия с кэшем работает быстрее, но все равно не является такой же быстрой, как итерационные алгоритмы.

Заключение

В рамках данной лабораторной работы была достигнута поставленная цель: были получены навыки динамического программирования на материале алгоритмов вычисления редакционного расстояния.

Решены все поставленные задачи:

1) было изучено понятие редакционного расстояния;

2) были изучены и реализованы четыре алгоритма вычисления редакционного расстояния — итерационный алгоритм поиска расстояния Левенштейна, итерационный алгоритм поиска расстояния Дамерау-Левенштейна, рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна и рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кэшем;

3) был проведен сравнительный анализ затрачиваемой памяти всеми алгоритмами на основе теоретических расчетов;

4) был проведен сравнительный анализ времени работы алгоритмов на основе экспериментальных данных.

Согласно полученным при проведении эксперимента данным, наиболее эффективными по скорости работы можно считать итерационные алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна. Менее эффективным является рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кэшем. Медленее всех работает неоптимизированный рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна.

Список использованных источников

1. *Калихман И. Л.*, *Войтенко М. А.* Динамическое программирование в примерах и задачах // Высшая школа. — 1979. — С. 5.
2. *Прытков В. А.* Функция расстояния между строками на основе кусочно-постоянной модели // Белорусский государственный университет информатики и радиоэлектроники. — 2012. — С. 22.
3. *Мосалев П. М.* Обзор методов нечеткого поиска текстовой информации // Вестник МГУП имени Ивана Федорова. — 2013. — С. 87—91.
4. *Лещенко А. В.* Практическое применение алгоритмов нечеткого поиска // Новосибирский государственный технический университет. — 2018. — С. 59—69.