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»

Тема Расстояние Левенштейна и Дамерау-Левенштейна

Студент Пискунов П.

Группа ИУ7-56Б

Преподаватель Волкова Л.Л., Строганов Д.В.

Москва — 2023 г.

Оглавление

Введение

Расстояние Левенштейна, также известное как редакционное расстояние, может быть определено как последовательность операций, необходимых для привращения одной строки в другую с минимальным количеством шагов [1].

В расстоянии Левенштейна используются такие операции как вставка, удаление, замена символов, необходимых для преобразования одной строки в другую.

Расстояние Дамерау - Левенштейна также представляет собой минимальное количество редакционных операций, которое дополнительно включает в себя операцию перестановки двух соседних символов [2] .

Расстояния Левенштейна и Дамерау - Левенштейна находят применение в различных областях, таких как:

- компьютерная лингвистика (автозамена в посиковых запросах, текстовые редакторы);
- биоинформатика (анализ последовательностей белков);
- нечеткий поиск записей в базах (борьба с опечатками).

Целью данной лабораторной работы является изучение алгоритмов поиска редакционных расстояний Левенштейна и Дамерау - Левенштейна. Для успешного выполнения лабораторной работы необходимо выполнить следующие задачи:

1) изучить методы вычисления расстояний Левенштейна и Дамерау - Левенштейна;
2) разработать алгоритмы для вычисления указанных расстояний;
3) реализовать разработанные алгоритмы в виде программного кода;
4) провести анализ затрат реализаций алгоритмов по времени и по памяти;
5) подготовить отчет по лабораторной работе.

1 Аналитическая часть

Расстояния Левенштейна и Дамерау - Левенштейна (редакционное расстояние, дистанция редактирования) – метрика, измеряющая по модулю разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (а именно вставки – I, удаления – D, замены – R), необходимых для превращения одной последовательности символов в другую. Также вводится операция, которая не требует никаких действий – совпадение – M.

В расстоянии Дамерау - Левенштейна, помимо вышеупомянутых операций, также предусматривается возможность перестановки соседних символов, что обозначается как операция X.

Каждой из этих операций можно приписать определенный штраф. Обычно используется сделующий набор штрафов: для операции M (совпадение) штраф равен нулю, а для операций I (вставка), D (удаление), R (замена), X (перестановка) штраф составляет единицу.

Таким образом, задача по вычислению расстояния Дамерау - Левенштейна сводится к нахождению последовательности операций, которая минимизирует общую сумму штрафов. Эту задачу можно решить с помощью использования рекуррентных формул.

1.1 Расстояние Левенштейна

Пусть дано две строки S_1 и S_2. Тогда расстояние Левенштейна можно найти по рекуррентной формуле (1.1):

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

Первые три формулы в системе (1.1) можно рассматривать как базовые и они представляют следующее: первое уравнение описывает ситуацию, когда не требуется никаких действий (символы обеих строк совпадают, так как обе строки пусты). Во втором уравнении рассматривается вставка j символов в пустую строку S_1 для создания строки-копии S_2 длиной j. В третьем уравнении описывается удаление всех i символов из строки S_1 для того, чтобы она соответствовала пустой строке S_2.

Последующие шаги в алгоритме требуют выбора минимального значения из штрафов, которые могут возникнуть в результате следующих операций: вставки символа в S_1 (первое уравнение в группе min), удаление символа из S_1 (второе уравнение в группе min), а также совпадения или замены, в зависимости од того, совпадают ли рассматриваемые символы в данной точке строк (третье уравнение в группе min).

1.2 Расстояние Дамерау - Левенштейна

Для вычисления расстояния Дамерау - Левенштейна между строками S_1 и S_2 используется аналогичная реккурентная формула, как в (1.1). Главное отличие заключается в том, что добавляем четвертую возможную операцию (1.2) в группу min:

$$\begin{cases} D(S_1[1...i-2],S_2[1...j-2]) + 1, \text{ если i, j}>1,\ a_i = b_{j-1}, b_j = a_{i-1} \\ \infty \text{ , иначе} \end{cases} \tag{1.2}$$

Этот сценарий предусматривает перестановку символов, расположенных рядом в строке S_1, только если длины обеих строк больше одного символа, и символы, которые рассматриваем, крест-накрест в строках S_1 и S_2, совпадают. В случае, если хотя бы одно из этих условий не выполняется, данное действие не учитывается при определении минимального расстояния.

Итоговая формула для расчета расстояния Дамерау - Левенштейна выглядит следующим образом (1.3):

$$D(i,j) = \begin{cases} 0, & \text{i} = 0, \text{j} = 0, \\ i, & \text{j} = 0, \text{i} > 0, \\ j, & \text{i} = 0, \text{j} > 0, \\ \min(D(i,j-1)+1, & \\ \quad D(i-1,j)+1, & \\ \quad D(i-1,j-1)+m(S_1[i],S_2[j]), & \\ & \text{если i} > 1, \text{j} > 1, \\ \quad D(i-2,j-2)+1), & S_1[i] = S_2[j-1], \\ & S_1[i-1] = S_2[j], \\ \min(D(i,j-1)+1, & \\ \quad D(i-1,j)+1, & \\ \quad D(i-1,j-1)+m(S_1[i],S_2[j])) & \text{, иначе.} \end{cases} \tag{1.3}$$

Вывод

В данной секции были изучены алгоритмы вычисления расстояния Левенштейна и его модификации – расстояния Дамерау - Левенштейна, которое включает в себя возможность перестановки соседних символов. Формулы для расчета расстояния Левенштейна и Дамерау - Левенштейна между строками задаются через рекурсивные уравнения, что означает, что соответствующие алгоритмы могут быть реализованы как с использованием рекурсии, так и в итеративной форме.

2 Конструкторская часть

Ранее в этой секции были представлены рекуррентные формулы, которые позволяют вычислять расстояние Левенштейна и Дамерау - Левенштейна. Однако, при разработке алгоритмов для решения этих задач, доступны различные подходы, такие как итеративный алгоритм, алгоритм рекурсии с использованием кэширования и алгоритм рекурсии без кэширования. В данной части рассмотрим каждый из этих подходов более подробно. Также в данной части указываются требования к программному обеспечению (ПО).

2.1 Требования к программному обеспечению

Программе передаются две строки в качестве входных данных, и она должна вычислить и вывести искомое расстояние, используя каждый из реализованных алгоритмов: для Левенштейна – итеративный, для Дамерау - Левенштейна – итеративный, рекурсивный без использования кэша и рекурсивный с использованием кэша. Кроме того, необходимо сообщить затраченное каждым алгоритмом процессорное время и память.

В создаваемом приложении пользователю должны быть доступны функции ввода двух строк и выбора желаемого алгоритма. Пользователь должен иметь возможность оценить размер и работу каждого алгоритма.

2.2 Алгоритм поиска расстояния Левенштейна

Для оптимизации нахождения расстояния Левенштейна можно использовать матрицу в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы.

На рисунке 2.1 приведена схема рассматриваемого алгоритма.

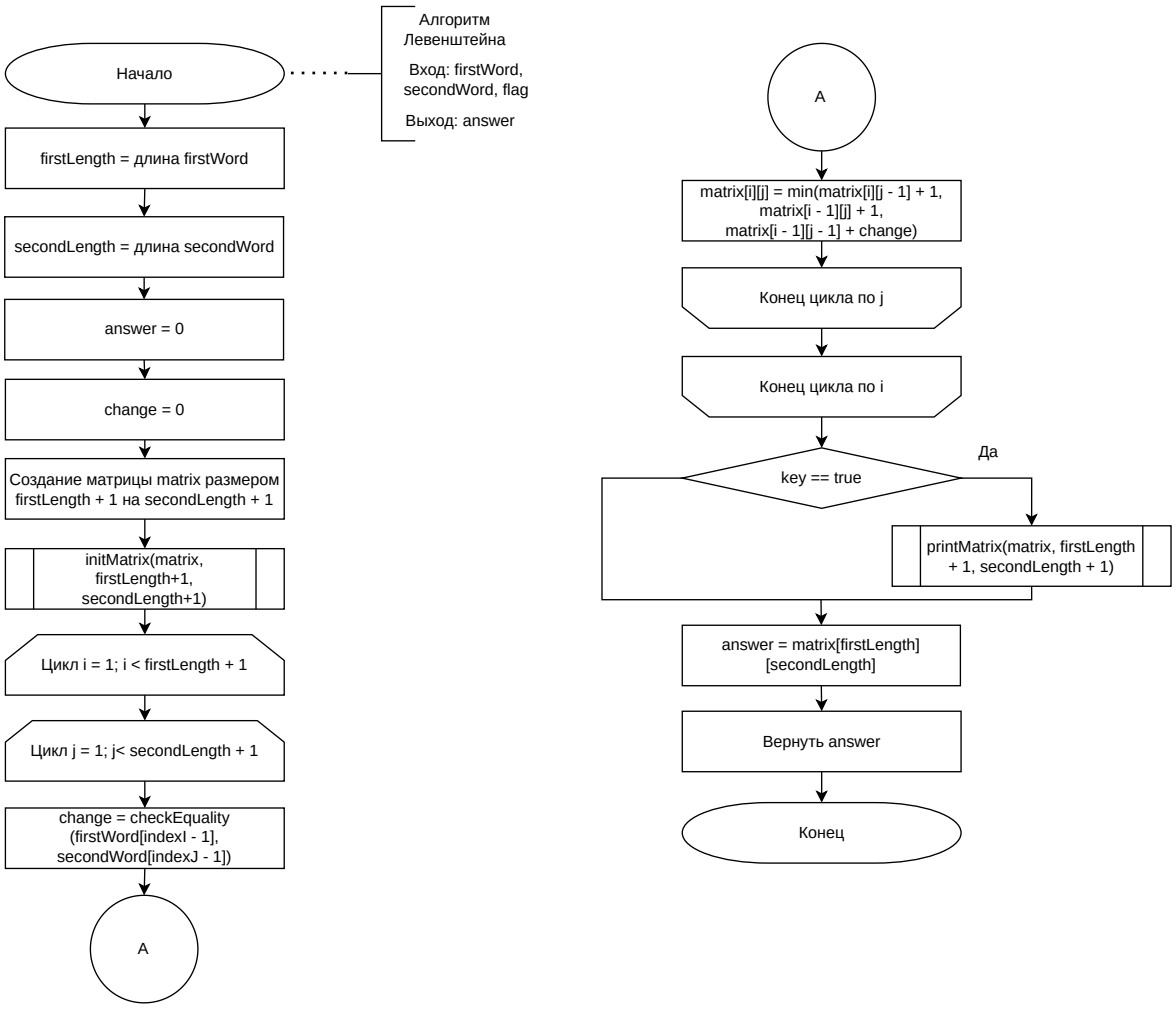

Рисунок 2.1 – Схема итеративного алгоритма поиска расстояния Левенштейна

2.3 Алгоритмы поиска расстояния Дамерау-Левенштейна

Рассматриваются итеративный, рекурсивный без кэширования и рекурсивный с кэшированием алгоритмы поиска расстояния Дамерау - Левенштейна.

На рисунках 2.2 – 2.4 приведены схемы рассматриваемых алгоритмов.

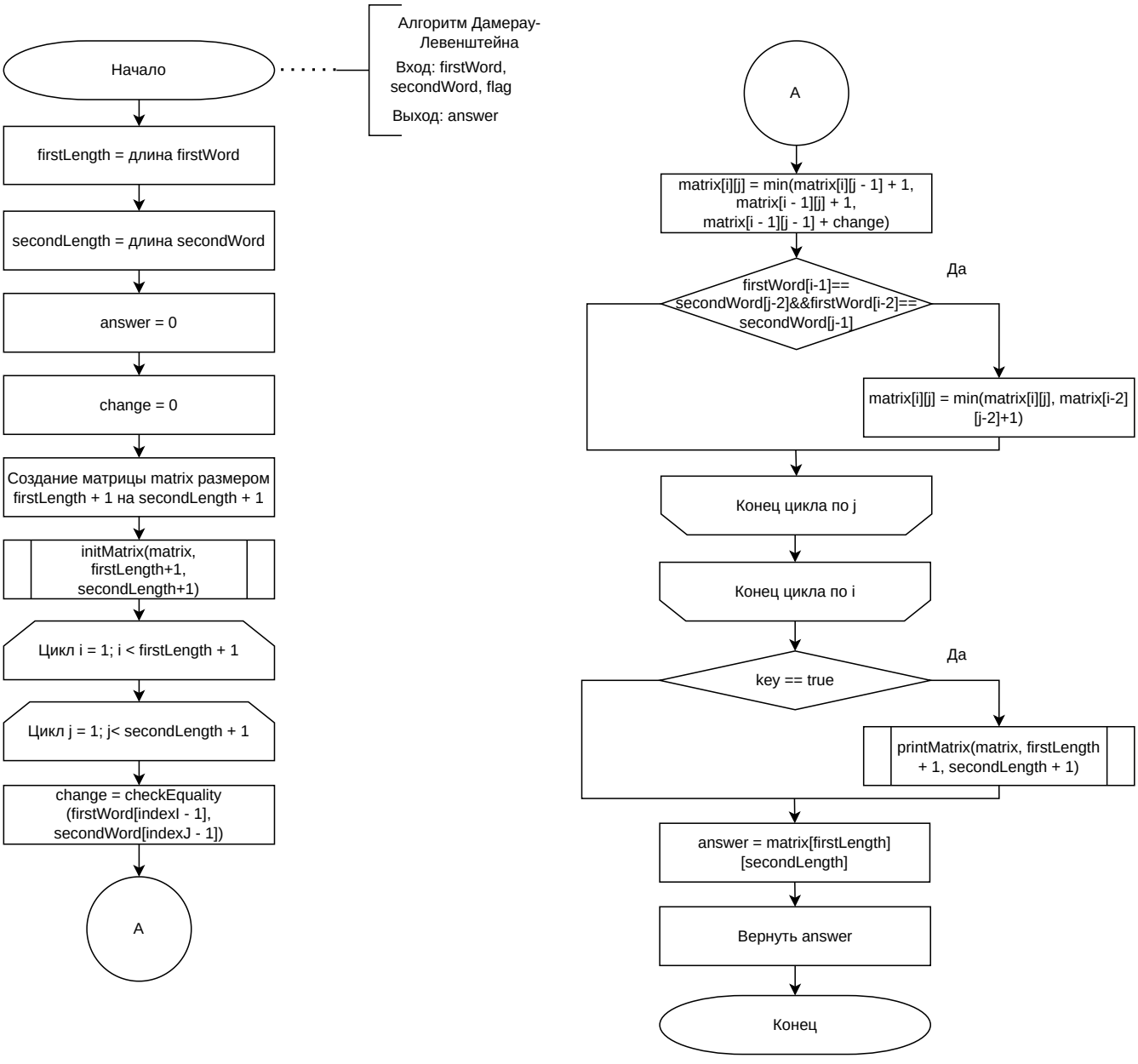

Рисунок 2.2 – Схема нерекурсивного алгоритма поиска расстояния Дамерау-Левенштейна

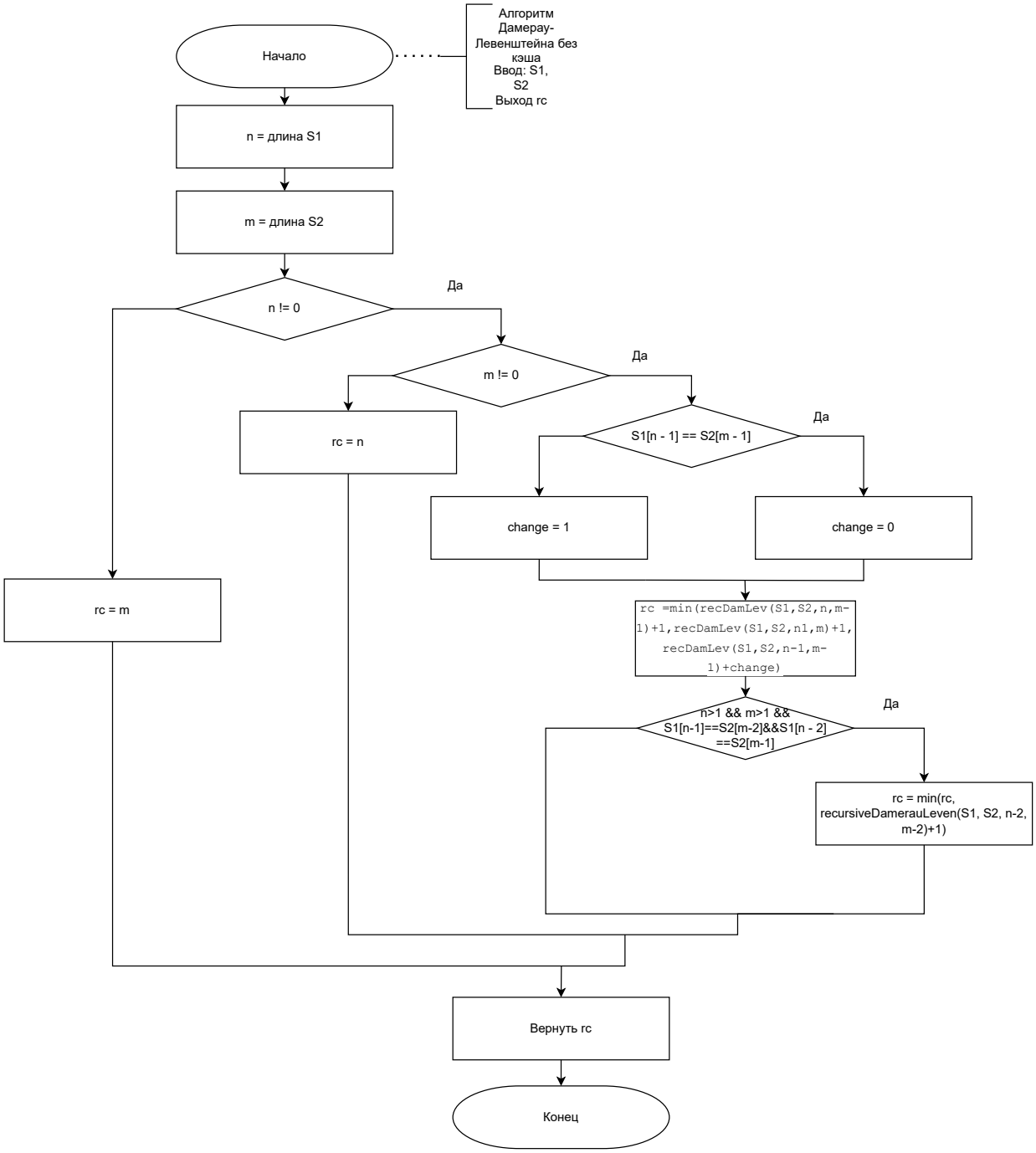

Рисунок 2.3 – Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна без кэширования

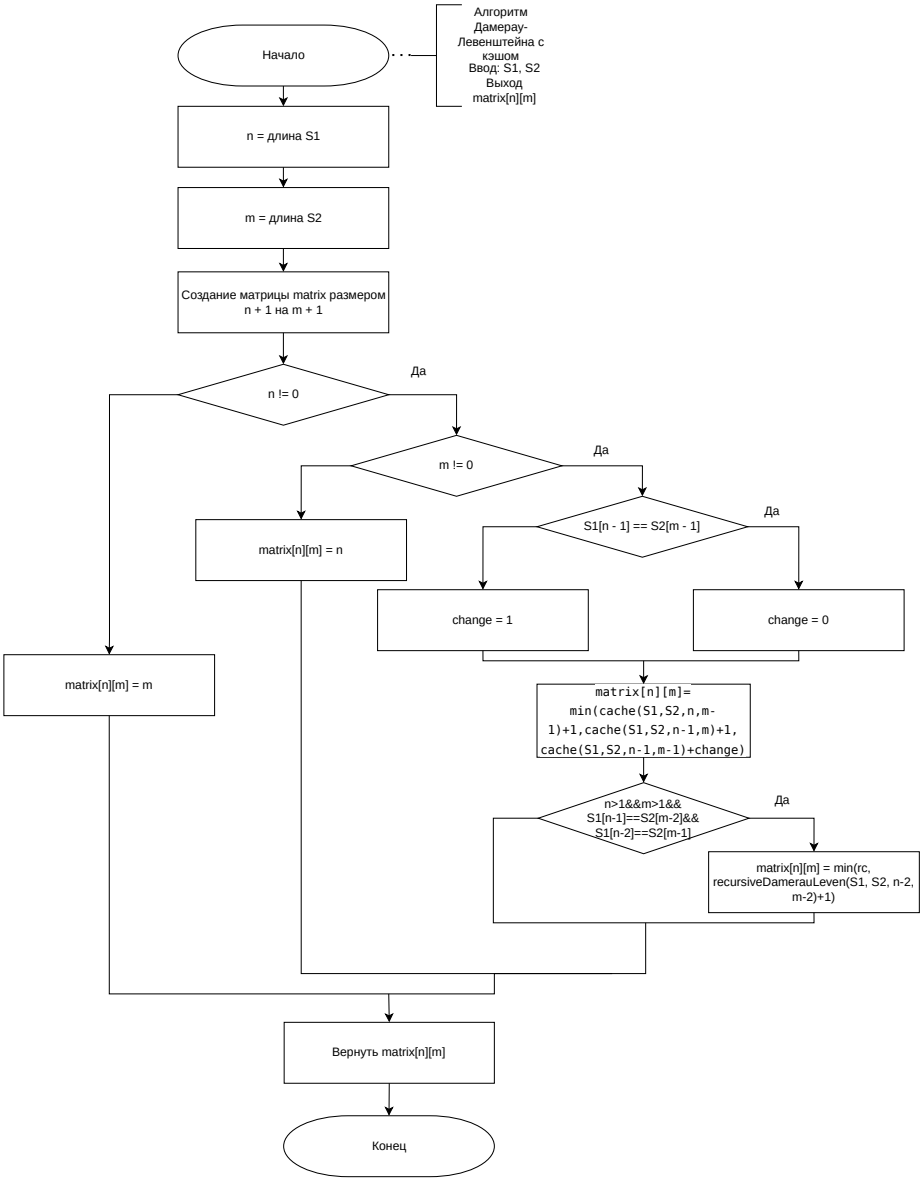

Рисунок 2.4 – Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кэшированием

Вывод

На основе теоретических знаний, полученных в аналитическом разделе, были разработаны схемы алгоритмов, благодаря которым могут быть найдены расстояния Левенштейна и Дамерау - Левенштейна.

3 Технологическая часть

В этом разделе предоставляются реализации выполненных алгоритмов и осуществляется выбор средств реализации.

3.1 Выбор средств реализации

Для выполнения данной лабораторной работы был выбран язык программирования C++. Время измерялось с помощью функции *clock()* из библиотеки *time.h* [3].

3.2 Реализация алгоритмов

В листингах 3.1 – 3.4 представлены реализации рассматриваемых алгоритмов.

В Листинге 3.1 показана реализация нерекурсивного алгоритма нахождения расстояния Левенштейна.

```
1  int normalLevenAlgorithm(const std::wstring &firstWord, const std::wstring
     &secondWord, bool key)
2  {
3    std::size_t firstLength = firstWord.length();
4    std::size_t secondLength = secondWord.length();
5    int answer = 0;
6    int change = 0;
7
8    std::vector<std::vector<int>> matrix(firstLength + 1, std::vector<int>(
   secondLength + 1, 0));
9
10   initMatrix(matrix, firstLength + 1, secondLength + 1);
11
12   for (std::size_t indexI = 1; indexI < firstLength + 1; indexI++)
13     for (std::size_t indexJ = 1; indexJ < secondLength + 1; indexJ++)
14     {
15       change = checkEquality(firstWord[indexI - 1], secondWord[indexJ -
   1]);
16
17       matrix[indexI][indexJ] = std::min(matrix[indexI][indexJ - 1] + 1,
18       std::min(matrix[indexI - 1][indexJ] + 1,
19       matrix[indexI - 1][indexJ - 1] + change));
```

```
20        }
21
22      if (key)
23        printMatrix(matrix, firstLength + 1, secondLength + 1);
24
25      answer = matrix[firstLength][secondLength];
26
27      return answer;
28    }
```

Листинг 3.1 – Нерекурсивный алгоритм поиска расстояния Левенштейна

В Листинге 3.2 показана реализация нерекурсивного алгоритма нахождения расстояния Дамерау - Левенштейна.

```
1  int normalLevenAlgorithm(const std::wstring &firstWord, const std::wstring
     &secondWord, bool key)
2  {
3    std::size_t firstLength = firstWord.length();
4    std::size_t secondLength = secondWord.length();
5    int answer = 0;
6    int change = 0;
7
8    std::vector<std::vector<int>> matrix(firstLength + 1, std::vector<int>(
    secondLength + 1, 0));
9
10   initMatrix(matrix, firstLength + 1, secondLength + 1);
11
12   for (std::size_t indexI = 1; indexI < firstLength + 1; indexI++)
13     for (std::size_t indexJ = 1; indexJ < secondLength + 1; indexJ++)
14     {
15       change = checkEquality(firstWord[indexI - 1], secondWord[indexJ -
    1]);
16
17       matrix[indexI][indexJ] = std::min(matrix[indexI][indexJ - 1] + 1,
18       std::min(matrix[indexI - 1][indexJ] + 1,
19       matrix[indexI - 1][indexJ - 1] + change));
20     }
21
22   if (key)
23     printMatrix(matrix, firstLength + 1, secondLength + 1);
24
25   answer = matrix[firstLength][secondLength];
26
27   return answer;
28 }
```

Листинг 3.2 – Нерекурсивный алгоритм поиска расстояния Дамерау - Левенштейна

В Листинге 3.3 показана реализация рекурсивного алгоритма нахождения расстояния Дамерау - Левенштейна без кэша.

```
1  int recursiveDamerauLeven(const std::wstring firstWord, const std::wstring
       secondWord,
2  std::size_t firstLenght, std::size_t secondLength)
3  {
4    int rc = 0;
5
6    if (firstLenght != 0)
7    {
8      if (secondLength != 0)
9      {
10       int change = checkEquality(firstWord[firstLenght - 1], secondWord[
       secondLength - 1]);
11
12       rc = std::min(recursiveDamerauLeven(firstWord, secondWord, firstLenght
       , secondLength - 1) + 1,
13       std::min(recursiveDamerauLeven(firstWord, secondWord, firstLenght - 1,
        secondLength) + 1,
14       recursiveDamerauLeven(firstWord, secondWord, firstLenght - 1,
       secondLength - 1) + change));
15
16       if (firstLenght > 1 && secondLength > 1 && firstWord[firstLenght - 1]
       == secondWord[secondLength - 2] &&
17       firstWord[firstLenght - 2] == secondWord[secondLength - 1])
18         rc = std::min(rc, recursiveDamerauLeven(firstWord, secondWord,
       firstLenght - 2, secondLength - 2) + 1);
19     }
20     else
21       rc = firstLenght;
22   }
23   else
24     rc = secondLength;
25   return rc;
26   }
```

Листинг 3.3 – Рекурсивный алгоритм поиска расстояния Дамерау - Левенштейна без кэша

В Листинге 3.4 показана реализация рекурсивного алгоритма нахождения расстояния Дамерау - Левенштейна с кэшом.

```
1  int cacheDamerauLeven(std::vector<std::vector<int>> &matrix, const std::
    wstring firstWord, const std::wstring secondWord,
2  const std::size_t &firstLenght, const std::size_t &secondLength)
3  {
4    if (firstLenght != 0)
5    {
6      if (secondLength != 0)
7      {
8        int change = checkEquality(firstWord[firstLenght - 1], secondWord[
    secondLength - 1]);
9
10       matrix[firstLenght][secondLength] = std::min(cacheDamerauLeven(
    matrix,
11       firstWord, secondWord, firstLenght, secondLength - 1) + 1,
12       std::min(cacheDamerauLeven(matrix, firstWord, secondWord,
    firstLenght - 1, secondLength) + 1,
13       cacheDamerauLeven(matrix, firstWord, secondWord, firstLenght - 1,
    secondLength - 1) + change));
14
15       if (firstLenght > 1 && secondLength > 1 && firstWord[firstLenght -
    1] == secondWord[secondLength - 2] &&
16       firstWord[firstLenght - 2] == secondWord[secondLength - 1])
17         matrix[firstLenght][secondLength] = std::min(matrix[firstLenght][
    secondLength],
18         cacheDamerauLeven(matrix, firstWord, secondWord, firstLenght - 2,
    secondLength - 2) + 1);
19     }
20     else
21       matrix[firstLenght][secondLength] = firstLenght;
22   }
23   else
24     matrix[firstLenght][secondLength] = secondLength;
25
26   return matrix[firstLenght][secondLength];
27 }
28 }
```

Листинг 3.4 – Рекурсивный алгоритм поиска расстояния Дамерау - Левенштейна с кэшом

3.3 Тестовые данные

В таблице 3.1 приведены входные данные, на которых было протестированно разработанное ПО.

Таблица 3.1 – Функциональные тесты

Входные данные		**Расстояние и алгоритм**			
		Левенштейна	**Дамерау - Левенштейна**		
Строка 1	**Строка 2**	**Итеративный**	**Итеративный**	**Рекурсивный**	
				Без кеша	**С кешом**
a	b	1	1	1	1
бабушка	бабушка	0	0	0	0
кот	скат	2	2	2	2
56	two	3	3	3	3
exam	example	3	3	3	3
конфета	календула	6	6	6	6
abcde	edcba	4	4	4	4
dasha	arisah	5	4	4	4

Вывод

Были предоставлены листинги кода на выбранном языке программирования и приведена таблица с результатами выполнения программы на заданных тестовых данных.

4 Исследовательская часть

4.1 Интерфейс приложения

На рисунках 4.1 – 4.2 приведено изображение интерфейса главного экрана приложения.

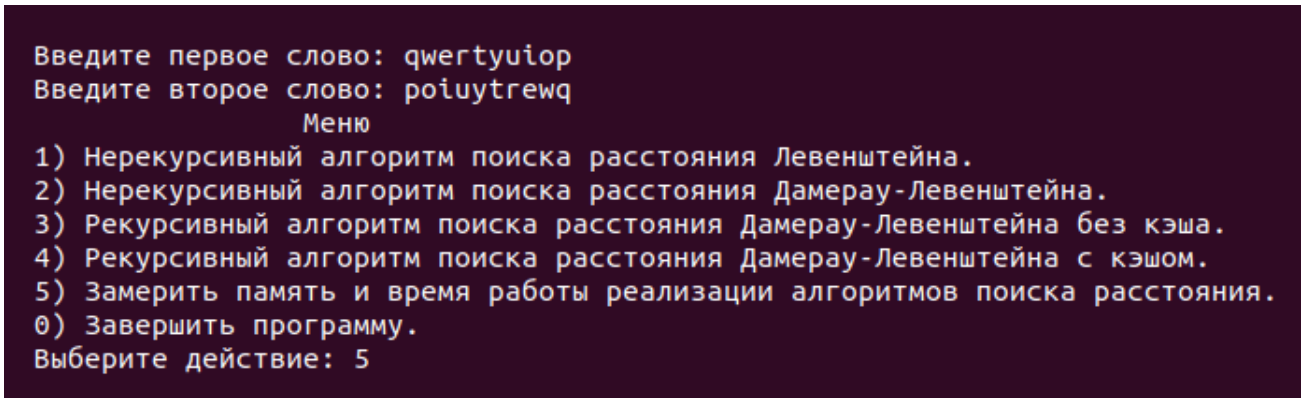

Рисунок 4.1 – Интерфейс

На экране приложения пользователь может ввести два слова, для которых необходимо вычислить расстояние, а также выбрать метод, который будет использован для этого вычисления. В результате выводится полученное расстояние, память и скорость работы алгоритмов, показывающие, как время выполнения операций (измеряемое в секундах) зависит от количества выполненных операций.

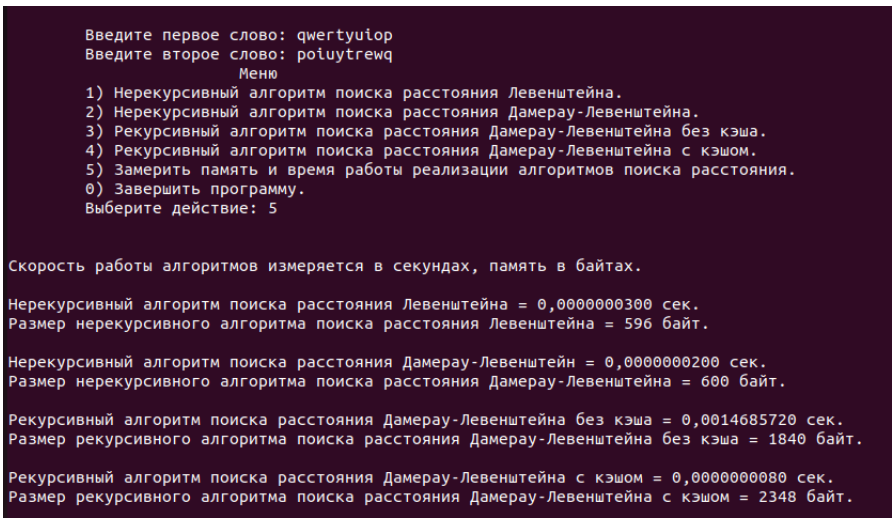

Рисунок 4.2 – Экран с результатом

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система – Ubuntu 22.04.3 LTS;
- оперативная память – 12 Гб;
- процессор – AMD® Athlon silver 3050u with radeon graphics × 2;

4.3 Время выполнения реализаций алгоритмов

На рисунке 4.3 приведено сравнение реализации всех алгоритмов поиска расстояния, у которых строки длиной от 1 до 10.

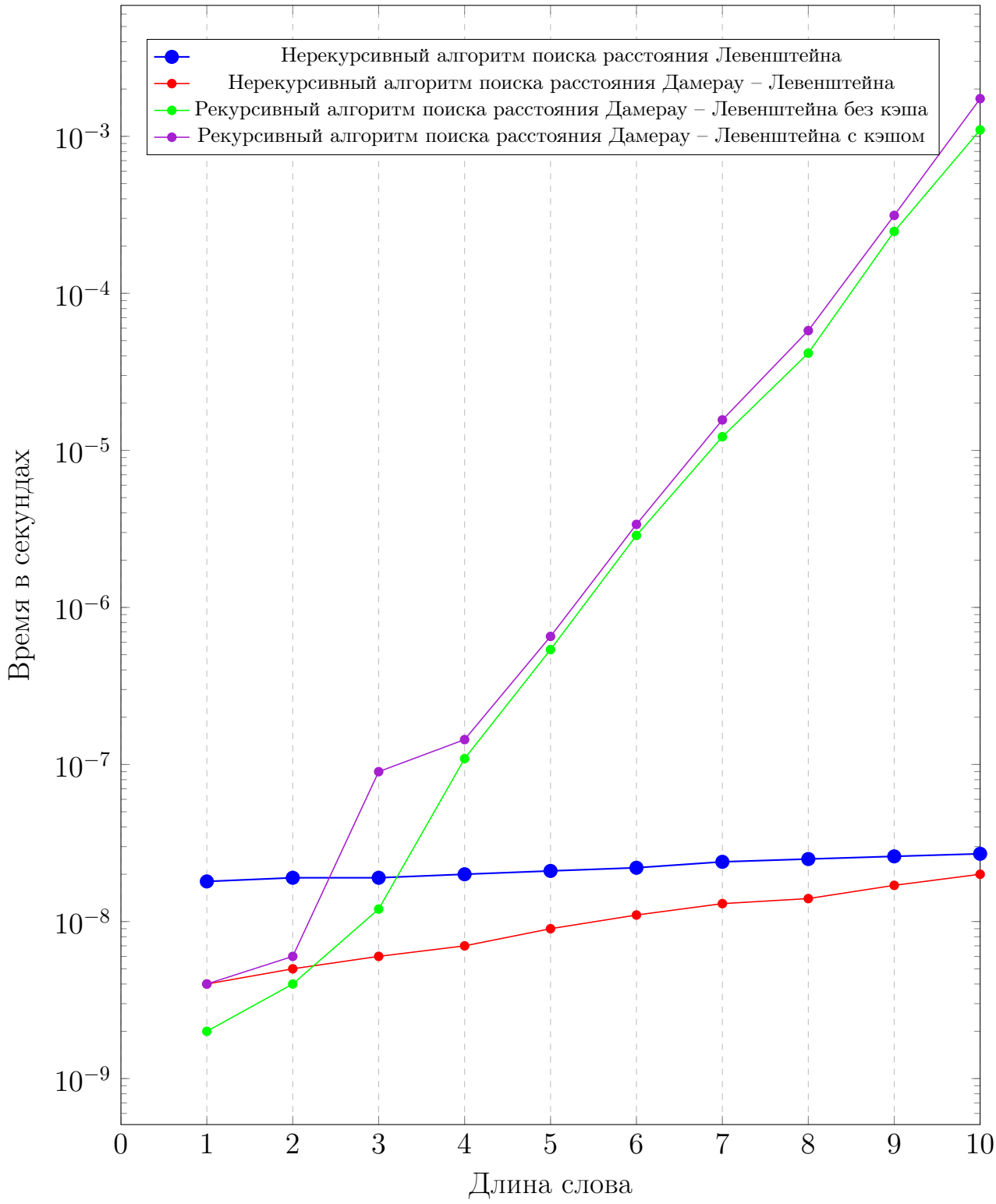

Рисунок 4.3 – Сравнение времени работы реализаций алгоритмов

4.4 Используемая память

Введены 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:

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

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:

$$(n+1)\cdot(m+1)\cdot size(int)+2\cdot size(string)+2\cdot size(size\_t)+ \\ +size(vector<vector<int>>)+(n+1)\cdot size(vecotr<int>)+2\cdot size(int), \tag{4.3}$$

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;
- $size(vector<vector<int>>)+(n+1)\cdot size(vecotr<int>)$ — 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:

$$(n+1)\cdot(m+1)\cdot size(int)+2\cdot size(string)+2\cdot size(size\_t)+ \\ +size(vector<vector<int>>)+(n+1)\cdot size(vecotr<int>)+3\cdot size(int), \tag{4.4}$$

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;
- $size(vector<vector<int>>)+(n+1)\cdot size(vecotr<int>)$ — 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.

Из данных, видно, 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.

Из рисунка 4.4 видно, что рекурсивная реализация алгоритма поиска расстояния Дамерау - Левенштейна эффективная по памяти, чем итеративная.

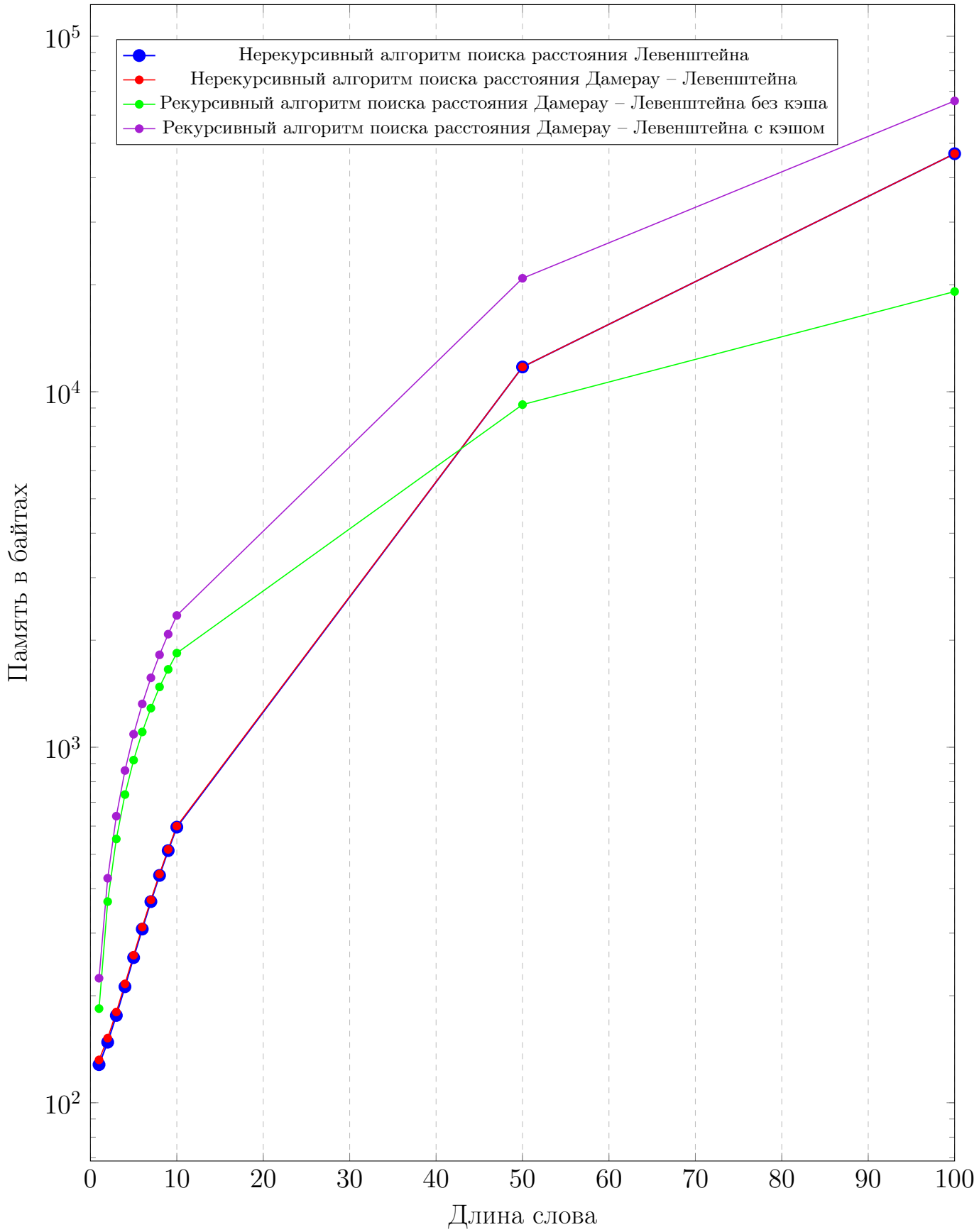

Рисунок 4.4 – Сравнение размеров реализаций алгоритмов в байтах

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

Заключение

По окончании лабораторной работы, при изучении алгоритмов для вычисления расстояния Левенштейна и Дамерау - Левенштейна, успешно применили и усовершенствовали наши навыки в области динамического программирования и разработки программного обеспечения.

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 нерекурсивная реализация алгоритма нахождения расстояния Левенштейна за счет сохранения необходимых промежуточных вычислений. При этом итеративные реализации с использованием матрицы занимают больше память, чем рекурсивнаые реализации, поэтому в этом случае предпочтение отдается рекурсивным алгоритмам нахождения расстояния.

В ходе выполнения лабораторной работы были выполнены следующие задачи:

1) изучены расстояния Левенштейна и Дамерау - Левенштейна;

2) разработаны и реализованы алгоритмы поиска расстояния Левенштейна и Дамерау - Левенштейна;

3) создан программный продукт, позволяющий протестировать реализованные алгоритмы;

4) проведен сравнительный анализ процессорного времени и затрачиваемой алгоритмами памяти;

5) был подготовлен отчет по лабораторной работе.

Список использованных источников

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

[2] В. Ульянов М. Ресурсно-эффективные компьютерные алгоритмы: учебное пособие. — М.: Издательство «Наука», ФИЗМАТЛИ, 2007.

[3] C library function clock() [Электронный ресурс]. — Режим доступа: `https://en.cppreference.com/w/c/chrono/clock_t` (дата обращения: 30.09.2023).