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: «Редакционное расстояние»

Студент ИУ7-56Б _____ Вольняга М. Ю.
(Группа) (Подпись, дата) (И. О. Фамилия)

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

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной лабораторной работы является изучение, реализация и сравнительный анализ алгоритмов поиска редакционных расстояний Левенштейна и Дамерау — Левенштейна.

Расстояние Левенштейна (редакционное расстояние) представляет собой метрику, которая измеряет разницу между двумя строками. Это определяет минимальное количество односимвольных операций (вставка, удаление и замена символа), необходимых для преобразования одной строки в другую [1].

Расстояние Дамерау — Левенштейна представляет собой расширенную версию расстояния Левенштейна, включающую дополнительную операцию — транспозицию. Данная операция позволяет учесть случаи перестановки двух соседних символов в строке [1].

Расстояние Левенштейна и его модификация расстояние Дамерау — Левенштейна имеют широкий спектр применений, например, в лингвистике (сравнение текстовых файлов или исправление ошибок в словах), в биоинформатике (сравнения генов, белков и хромосом) [1; 2].

Задачи лабораторной работы:

1) описать расстояния Левенштейна и Дамерау — Левенштейна;

2) описать алгоритмы поиска расстояний Левенштейна и Дамерау — Левенштейна;

3) разработать программное обеспечение, включающее в себя нерекурсивный алгоритм поиска расстояния Левенштейна и нерекурсивный алгоритм поиска расстояния Дамерау — Левенштейна, а также рекурсивный алгоритм поиска расстояния Дамерау — Левенштейна и рекурсивный с кешированием алгоритм поиска расстояния Дамерау — Левенштейна;

4) провести сравнительный анализ времени выполнения реализаций алгоритмов и занимаемой памяти.

1 Аналитический раздел

1.1 Расстояние Левенштейна

Расстояние Левенштейна между двумя строками — это минимальное количество редакторских операций: вставка (I от англ. insert), замена (R от англ. replace) и удаление (D от англ. delete), необходимых для преобразования одной строки в другую. Стоимость каждой операции может различаться в зависимости от вида операции [3]:

1) $w(a, b) = 1$, $a \neq b$ — цена замены символа a на b;

2) $w(\lambda, b) = 1$ — цена вставки символа b;

3) $w(a, \lambda) = 1$ — цена удаления символа a.

Введем понятие совпадения символов — M (от англ. match). Его стоимость будет равна 0, то есть $w(a, a) = 0$ [1].

Пусть имеется две строки S_1 и S_2 длиной M и N соответственно, тогда редакционное расстояние $d(S_1, S_2)$ можно подсчитать по следующей рекурентной формуле [2]:

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

где сравнение символов строк S_1 и S_2 рассчитывается как

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

1.1.1 Нерекурсивный алгоритм нахождения расстояния Левенштейна

Под нерекурсивным алгоритмом нахождения расстояния Левинштейна понимается итеративная реализация алгоритма.

В качестве структуры данных для хранения промежуточных значений можно использовать матрицу, имеющую размеры $(N + 1) \times (M + 1)$.

Пусть M — длина строки S_1, N — длина строки S_2. $S_1[1...i]$ — подстрока S_1 с длиной i символов, начиная с первого, $S_2[1...j]$ — подстрока S_2 длиной j символов, начиная с первого. Каждая ячейка матрицы, обозначенная как $mtr[i, j]$ содержит значение $D(S1[1...i], S2[1...j])$. Вся матрица заполняется в соответствии с формулой (1.1).

1.2 Расстояние Дамерау — Левенштейна

Расстояние Дамерау — Левенштейна между двумя строками, представляет собой минимальное количество операций, таких как вставка, удаление, замена символа и перестановка соседних символов (транспозиция), чтобы преобразовать одну строку в другую. Это является модифицированной версией расстояния Левенштейна, в которую была добавлена операция перестановки, также известная как транспозиция [1].

Расстояние Дамерау — Левенштейна может быть вычислено по рекурентной формуле [3]:

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

1.2.1 Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна

Рекурсивный алгоритм реализует формулу (1.3), функция D составлена таким образом, что верно следующее.

1) Перевод из пустой строки в пустую не требует действий.

2) Преобразование пустой строки в строку S_1 или наоборот требует $|S_1|$ операций.

3) Преобразование строки S_1 в S_2 включает комбинацию операций: удаление, вставка и, если применимы, замена и транспозиция. Порядок операций не важен.

Пологая, что S_1' — это S_1 без последнего символа и S_1'' — это S_1 без последних двух символов. Для строки S_2 аналогичные обозначения. Тогда, минимальной ценой преобразования строки S_1 в строку S_2, будет минимальное значение из следующих вариантов (при условии, что они применимы).

1) Преобразуем S_1' в S_2, а затем удаляем последний символ из S_1'.

2) Преобразуем S_1 в S_2', затем добавляем последний символ из S_2.

3) Если S_1 и S_2 заканчиваются разными символами, меняем последний символ S_1 на последний символ S_2, после преобразования S_1' в S_2'.

4) Если после перестановки двух последних символов S_1 получаем окончание S_2, то рассматриваем преобразование S_1'' в S_2''.

5) Если S_1 и S_2 заканчиваются одинаковым символом, то рассматриваем преобразование S_1' в S_2'.

1.2.2 Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна с кешированием

Рекурсивный алгоритм с использованием кэша является улучшенной версией рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна. Он использует так же формулу (1.3), но с кешированием. Добавление механизма кеширования, позволяет избежать повторные вычисление для подстрок [1]. Чтобы реализовать механизм кеширования можно использовать матрицу кеша для сохранения промежуточных результатов. Алгоритм последовательно заполняет матрицу $mtr_{M,N}$ значениями $D(i,j)$ расстояний. Размер матрицы кэша равен $(N+1) \times (M+1)$.

1.2.3 Нерекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна

Итеративный алгоритм нахождения расстояния Дамерау — Левенштейна решает проблему с повторными вычислениями у рекурсивного алгоритма [1].

В качестве структуры данных для хранения промежуточных значений можно использовать матрицу, имеющую размеры $(N+1) \times (M+1)$. Каждая ячейка этой матрицы, содержит значение $D(S1[1...i], S2[1...j])$. Вся матрица заполняется в соответствии с формулой (1.3).

Вывод

В представленном разделе описаны: расстояния Левенштейна и Дамерау — Левенштейна, алгоритмы поиска расстояний Левенштейна и Дамерау — Левенштейна. Алгоритмы характеризуются рекуррентными формулами (1.1,

1.3), что позволяет реализовывать два подхода рекурсивный и итеративный [1]. В качестве входных данных алгоритмы обрабатывают строковые последовательности, содержащие символы как кириллицы, так и латиницы, при этом алгоритмическая конструкция предусматривает корректную обработку пустых строковых последовательностей.

2 Конструкторский раздел

В этом разделе представлены: схемы алгоритмов для вычисления расстояний Левенштейна и Дамерау — Левенштейна, описание используемых типов данных и необходимые требования для программного обеспечения.

2.1 Требования к программному обеспечению

К программе предъявляются следующие требования:

– на вход подается две строки;

– на выходе — искомое расстояние;

– интерфейс для выбора операций;

– обработка входящих строк;

– поддержка строк, содержащих символы как в латинском, так и в кириллическом алфавитах;

– возможность произвести замеры процессорного времени работы реализованных алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна.

2.2 Разработка алгоритмов

На вход алгоритмов подаются строки X и Y.

На рисунке 2.1 представлена схема алгоритма поиска расстояния Левенштейна. На рисунках 2.2 – 2.4 представлены схемы алгоритмов поиска расстояния Дамерау — Левенштейна.

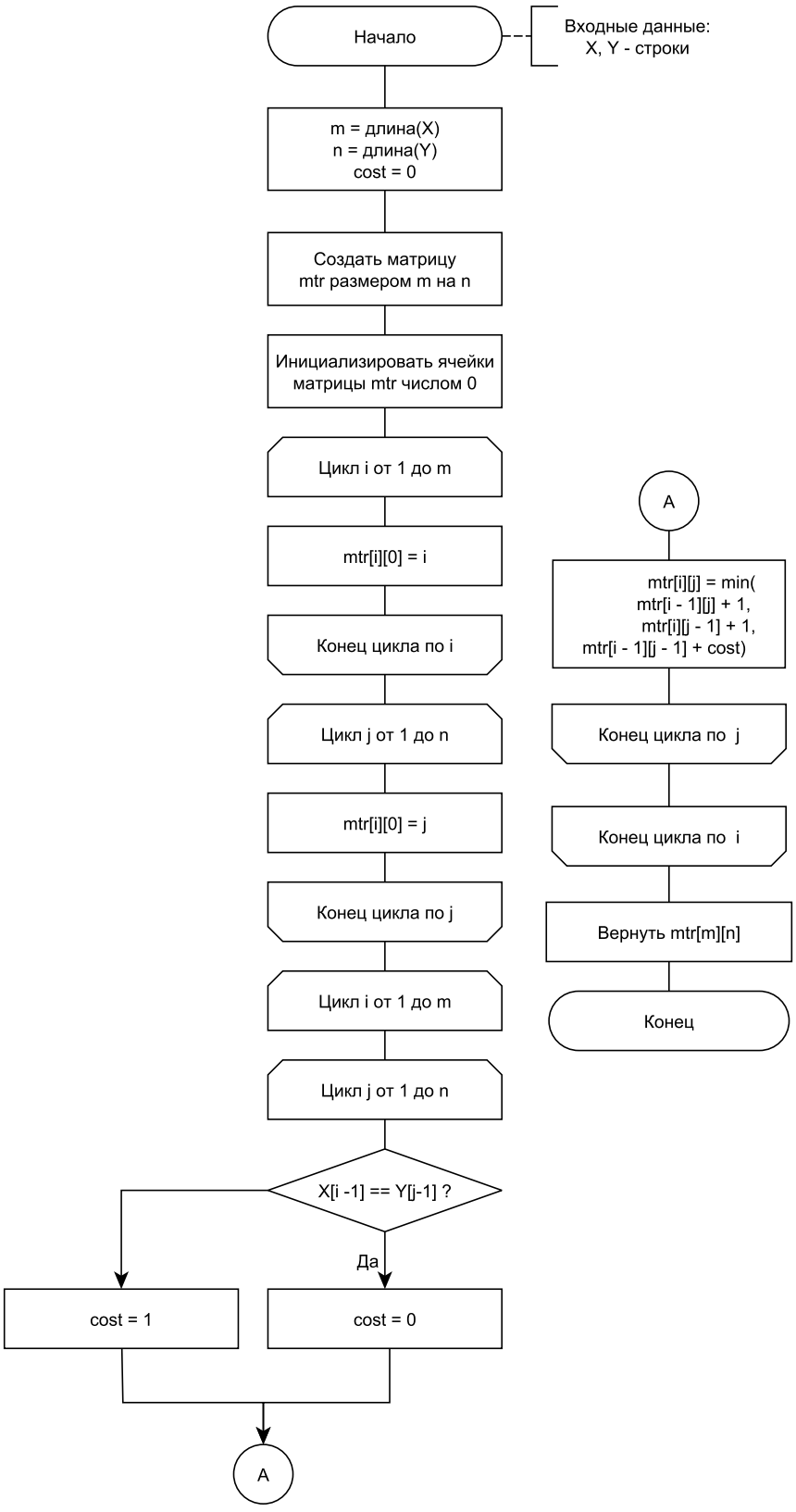

Рисунок 2.1 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

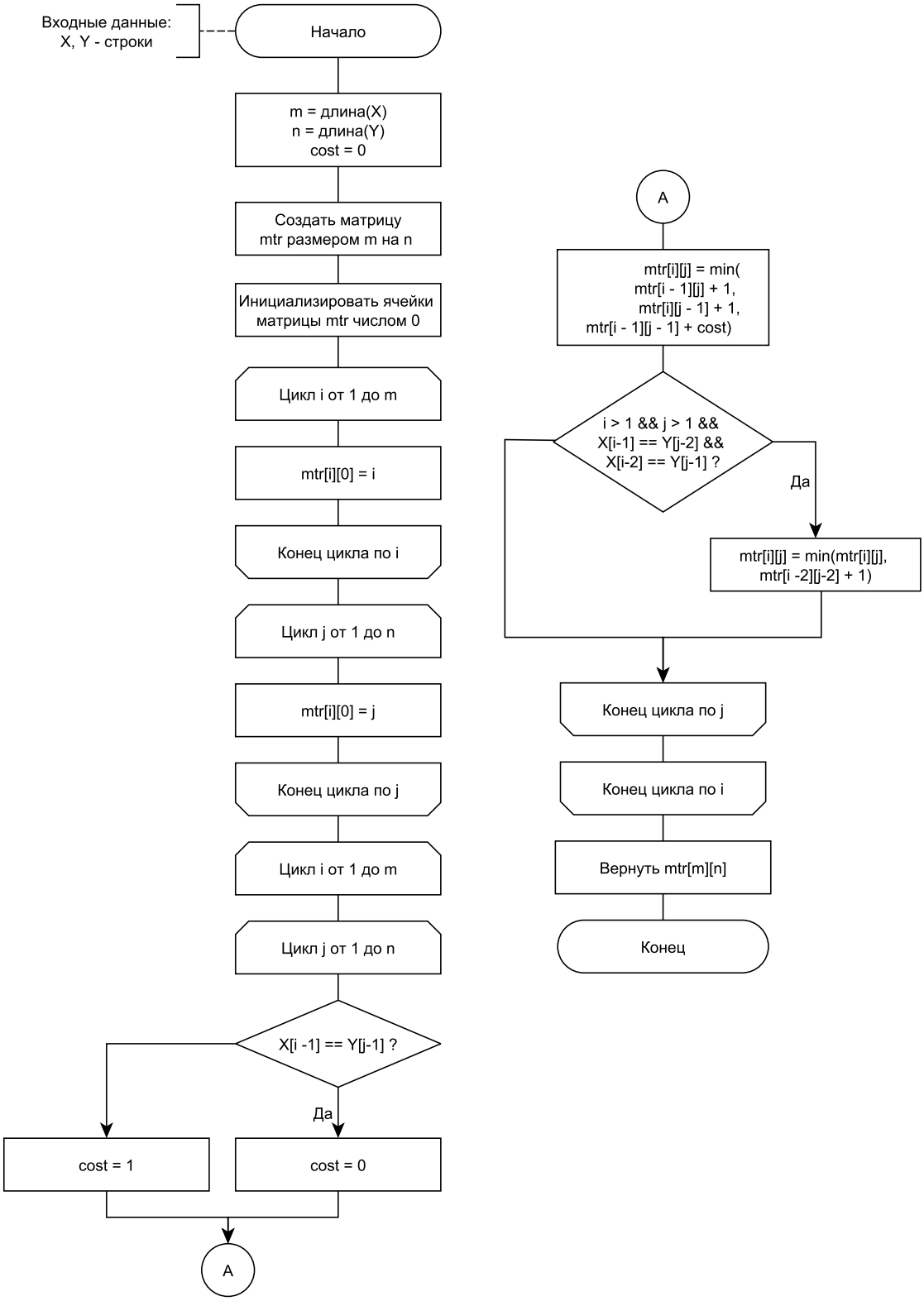

Рисунок 2.2 – Схема нерекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна

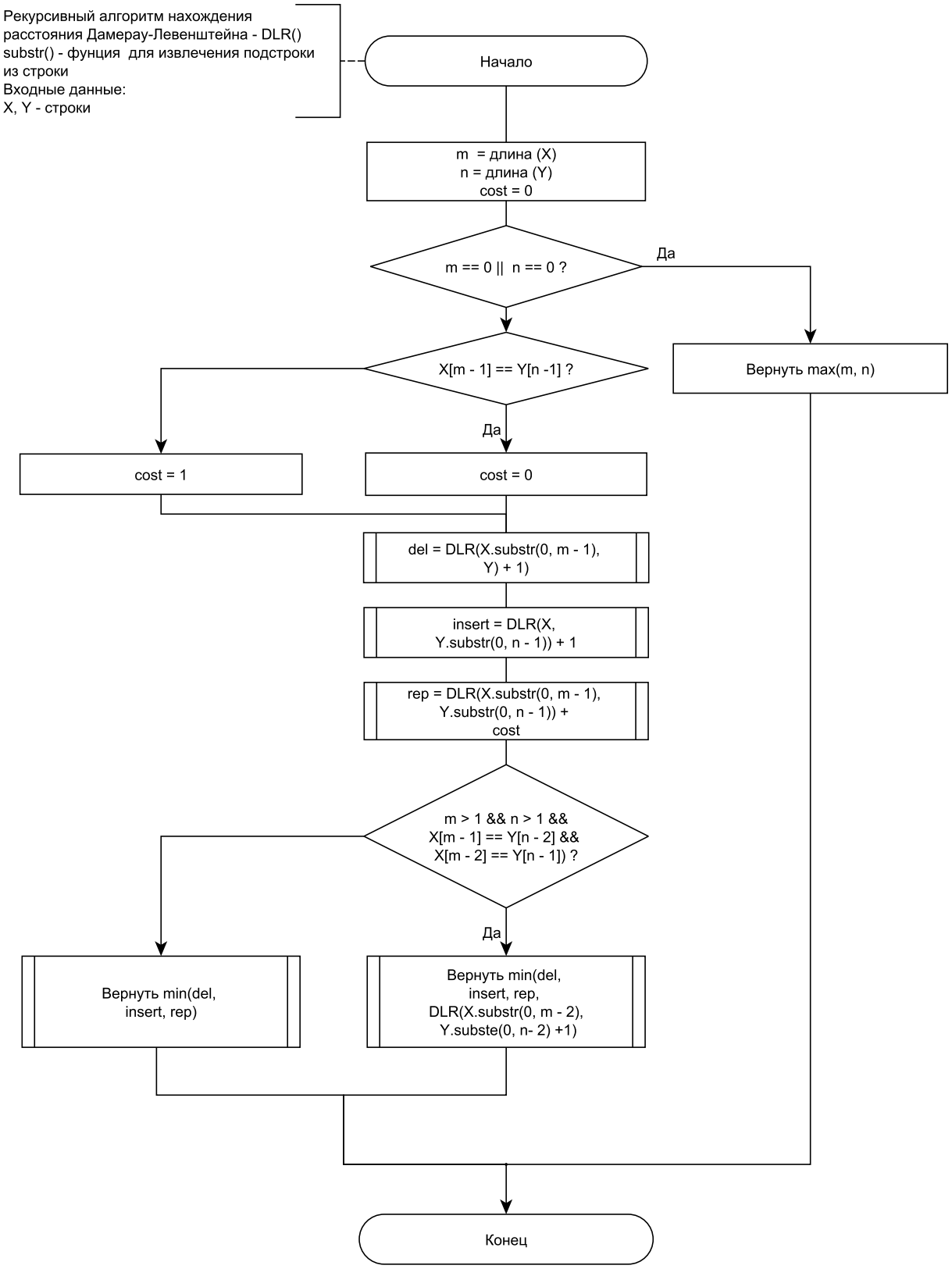

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна

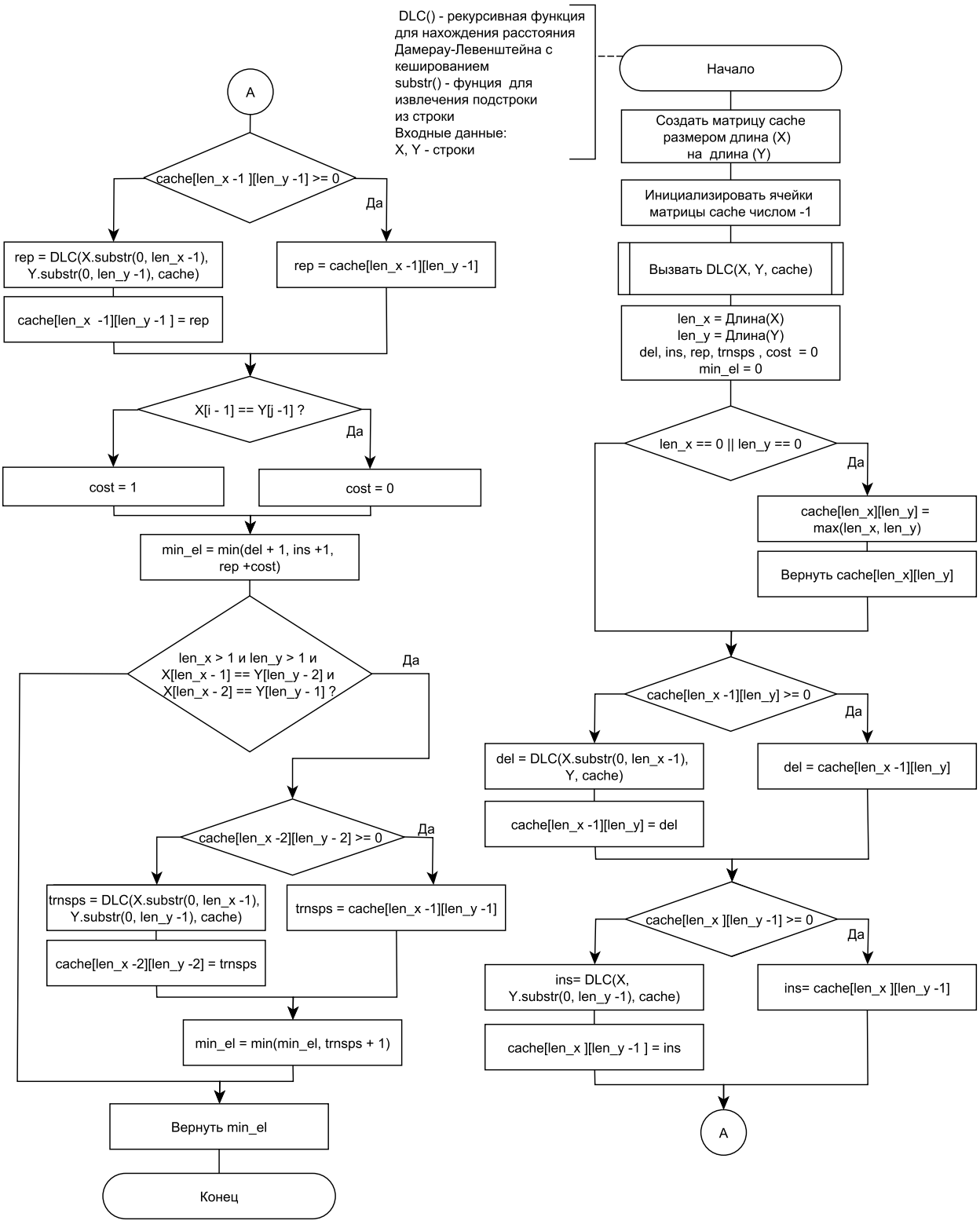

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна с кешированием

2.3 Описание используемых типов данных

При реализации алгоритмов использованы следующие структуры данных:

– строка — символьный массив, размером длины строки;

– матрица — двумерный массив с целыми значениями.

Вывод

В данном разделе на основе аналитической части были построены схемы требуемых алгоритмов, выбраны используемые типы данных.

3 Технологический раздел

В данном разделе будут приведены средства реализации, листинги кода реализации алгоритмов и функциональные тесты.

3.1 Средства реализации

В качестве языка программирования для реализации лабораторной работы был выбран *Python* [4]. Данный выбор обусловлен наличием необходимых библиотек для проведения точного замера времени. Для измерения процессорного времени выполнения была выбрана функция *process_time* из модуля *time* [5].

3.2 Сведения модулей программы

Программа разбита на следующие модули.

- `main.py` — файл, который включает в себя точку входа программы, из которой осуществляется вызов алгоритмов через разработанный интерфейс.
- `algorithms.py` — файл содержит функции поиска расстояния Левенштейна и Дамерау — Левенштейна.
- `measurement.py` — в файле присутствуют функции для измерения процессорного времени выполнения реализаций алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна.

3.3 Реализация алгоритмов

В листингах 3.1 – 3.4 приведены реализации алгоритмов поиска расстояний Левенштейна (только нерекурсивный алгоритм) и Дамерау — Левенштейна (нерекурсивный, рекурсивный и рекурсивный с кешированием).

Листинг 3.1 – Функция нахождения расстояния Левенштейна с использованием матрицы

```
def levenshtein(X, Y):
    m = len(X)
    n = len(Y)

    mtr = [[0 for _ in range(n + 1)] for _ in range(m + 1)]

    for i in range(1, m + 1):
        mtr[i][0] = i
    for j in range(1, n + 1):
        mtr[0][j] = j

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            cost = 0 if X[i - 1] == Y[j - 1] else 1
            mtr[i][j] = min(mtr[i - 1][j] + 1,
                            mtr[i][j - 1] + 1,
                            mtr[i - 1][j - 1] + cost)

    return mtr[m][n]
```

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

```
1
2  def damerauLevenshtein(X, Y):
3      m = len(X)
4      n = len(Y)
5
6      mtr = [[0 for _ in range(n + 1)] for _ in range(m + 1)]
7
8      for i in range(1, m + 1):
9          mtr[i][0] = i
10     for j in range(1, n + 1):
11         mtr[0][j] = j
12
13     for i in range(1, m + 1):
14         for j in range(1, n + 1):
15             cost = 0 if X[i - 1] == Y[j - 1] else 1
16             mtr[i][j] = min(mtr[i - 1][j] + 1, mtr[i][j - 1] +
                   1, mtr[i - 1][j - 1] + cost)
17             if (i > 1 and j > 1 and X[i - 1] == Y[j - 2] and X[i
                   - 2] == Y[j - 1]):
18                 mtr[i][j] = min(mtr[i][j], mtr[i - 2][j - 2] + 1)
19
20     return mtr[m][n]
```

Листинг 3.3 – Рекурсивная функция нахождения расстояния Дамерау — Левенштейна

```
def damerauLevenshteinRecurs(X, Y):
    m = len(X)
    n = len(Y)

    if m == 0 or n == 0:
     return max(m, n)

    cost_del_ins_rep = min(
    [damerauLevenshteinRecurs(X[:-1], Y) + 1,
       damerauLevenshteinRecurs(X, Y[:-1]) + 1,
    damerauLevenshteinRecurs(X[: - 1], Y[: - 1]) + m(X[-1],
       Y[-1])])

    if m > 1 and n > 1 and X[-1] == Y[-2] and X[-2] == Y[-1]:
        cost_del_ins_rep = min(cost_del_ins_rep,
           damerauLevenshteinRecurs(X[:-2], Y[:-2]) + 1)

    return cost_del_ins_rep

def m(a, b):
    return 0 if a == b else 1
```

Листинг 3.4 – Рекурсивная функция нахождения расстояния Дамерау — Левенштейна с «кэшированием»

```
def dLRC(X, Y):
    cache = [[-1] * (len(Y) + 1) for _ in range(len(X) + 1)]
    min_len = max(len(X), len(Y)) + 1
    def dlrc(X, Y):
        len_X = len(X)
        len_Y = len(Y)
        if len_X == 0 or len_Y == 0:
            cache[len_X][len_Y] = max(len_X, len_Y)
            return cache[len_X][len_Y]
        if cache[len_X - 1][len_Y] >= 0:
            del_cost = cache[len_X - 1][len_Y]
        else:
            del_cost = dlrc(X[:-1], Y)
            cache[len_X - 1][len_Y] = del_cost
        if cache[len_X][len_Y - 1] >= 0:
            ins_cost = cache[len_X][len_Y - 1]
        else:
            ins_cost = dlrc(X, Y[:-1])
            cache[len_X][len_Y - 1] = ins_cost
        if cache[len_X - 1][len_Y - 1] >= 0:
            rep_cost = cache[len_X - 1][len_Y - 1]
        else:
            rep_cost = dlrc(X[:-1], Y[:-1])
            cache[len_X - 1][len_Y - 1] = rep_cost
        min_len = min([del_cost + 1,
                       ins_cost + 1,
                       rep_cost + m(X[-1], Y[-1])])
        if len_X > 1 and len_Y > 1 and X[-1] == Y[-2] and X[-2]
           == Y[-1]:
            if cache[len_X - 2][len_Y - 2] >= 0:
                trnsps_cost = cache[len_X - 1][len_Y - 1]
            else:
                trnsps_cost = dlrc(X[:-2], Y[:-2])
                cache[len_X - 2][len_Y - 2] = trnsps_cost
            min_len = min(min_len, trnsps_cost + 1)
        cache[len(X)][len(Y)] = min_len
        return min_len
    return dlrc(X, Y)
```

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**			
		Левенштейна	**Дамерау — Левенштейн**		
Строка 1	**Строка 2**	**Итеративный**	**Итеративный**	**Рекурсивный**	
				Без кеша	**С кешем**
окно	окна	1	1	1	1
mama	mama	0	0	0	0
мама	папа	2	2	2	2
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	стол	2	2	2	2
a	b	1	1	1	1

Вывод

Была создана итеративная реализация алгоритма нахождения расстояния Левенштейна, а также реализованы алгоритмы нахождения расстояния Дамерау — Левенштейна в итеративной, рекурсивной и рекурсивной с применением кеширования формах. Проведено тестирование данных реализаций алгоритмов.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени, представлены далее.

– Процессор: Ryzen 5 4600H, тактовая частота ЦПУ 3.0 ГГц, максимальная частота процессора 4.0 ГГц [6].

– Оперативная память: 16 ГБайт.

– Операционная система: Windows 10 Pro 64-разрядная система [7].

При замерах времени ноутбук был включен в сеть электропитания и был нагружен только системными приложениями.

4.2 Демонстрация работы программы

На рисунке 4.1 представлено визуальное представление работы разработанного программного обеспечения. В частности, продемонстрированы результаты вычислений, выполненных алгоритмами поиска расстояний Левенштейна и Дамерау — Левенштейна для двух строк: «акбачок» и «кабачок». В рамках этой демонстрации представлены матрицы, отражающие промежуточные результаты для соответствующих алгоритмов.

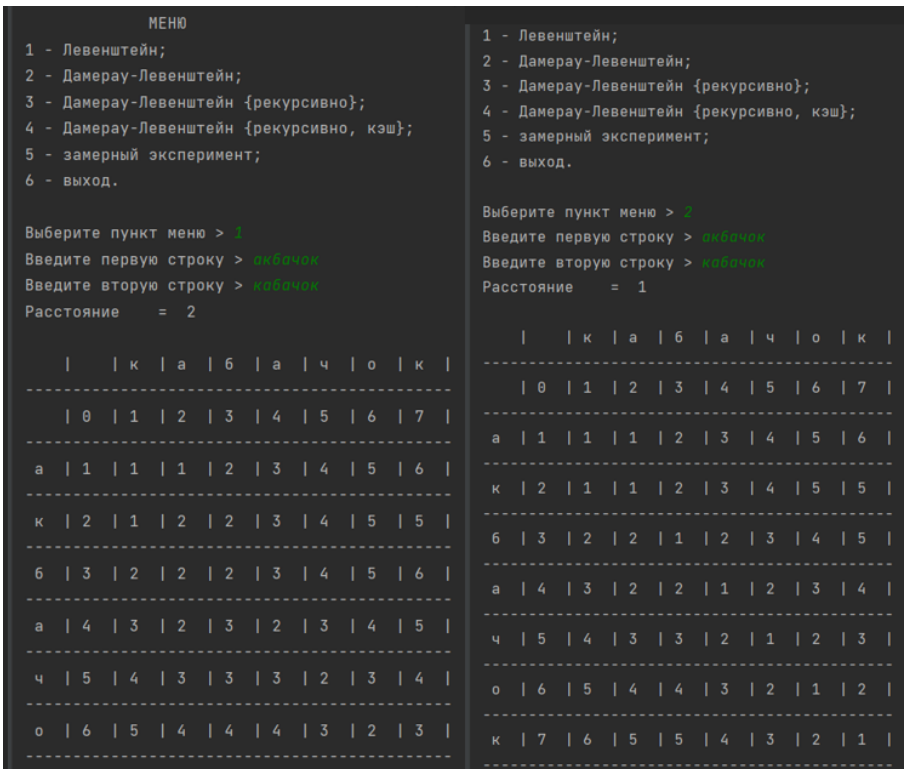

Рисунок 4.1 – Демонстрация работы программы при поиске расстояний Левенштейна и Дамерау — Левенштейна

4.3 Временные характеристики

В таблице 4.1 представлен результат замеров времени реализованных алгоритмов. Замеры проводились для одинаковых длин строк. Строки были сгенерированы путем последовательного добавления символа, в конец строки.

Используя значения из таблицы 4.1, был построен график 4.2 для лучшей визуализации эффективности алгоритмов.

Таблица 4.1 – Результат замеров времени реализованных алгоритмов в тактах процессора

Длина (символ)	**Время, нс**			
	Левенштейн	**Дамерау — Левенштейн**		
	Итеративный	**Итеративный**	**Рекурсивный**	
			Без кеша	**С кешем**
1	1.56e-06	1.56e-06	1.56e-06	3.13e-06
2	3.13e-06	3.13e-06	6.25e-06	6.25e-06
3	6.25e-06	6.25e-06	2.97e-05	1.09e-05
4	9.37e-06	9.37e-06	1.56e-04	1.72e-05
5	1.25e-05	1.25e-05	8.59e-04	2.66e-05
6	1.56e-05	1.72e-05	4.53e-03	3.44e-05
7	2.03e-05	2.50e-05	2.42e-02	4.53e-05
8	2.66e-05	2.97e-05	1.33e-01	5.78e-05
9	2.97e-05	3.91e-05	7.31e-01	7.19e-05
10	3.75e-05	4.53e-05	4.04e+00	8.59e-05
11	4.37e-05	5.47e-05	2.22e+01	1.02e-04

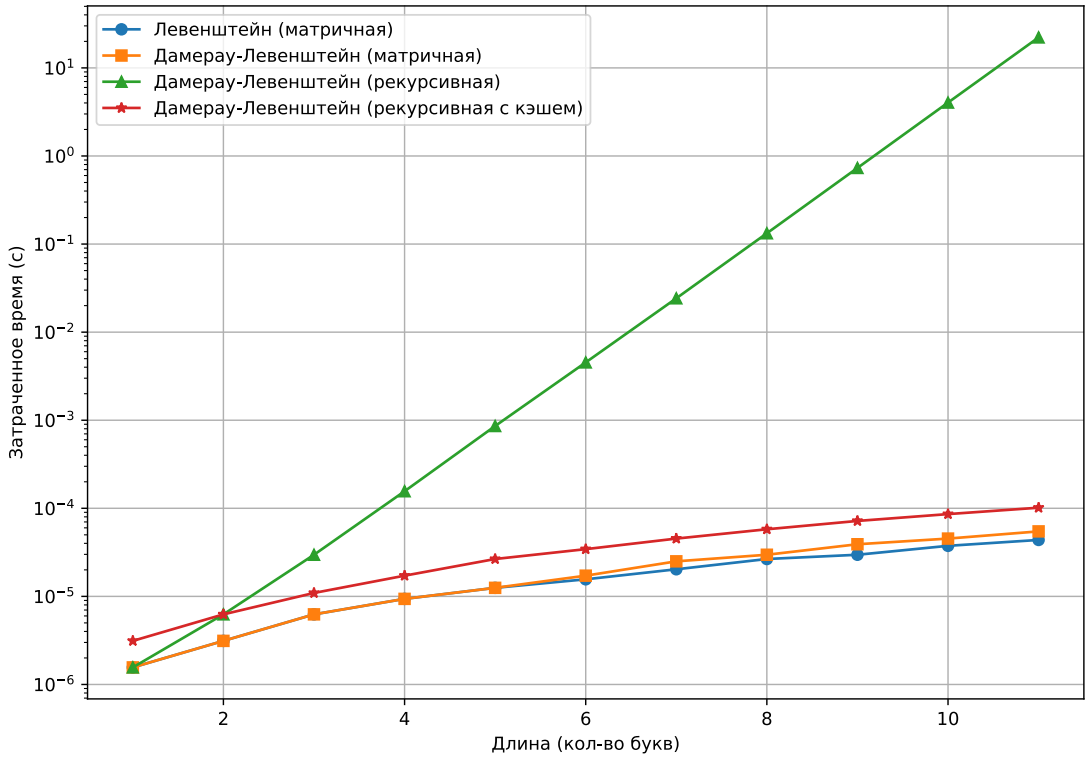

Рисунок 4.2 – Результат замеров времени реализованных алгоритмов

4.4 Характеристики по памяти

Введем следующие обозначения:

– m — длина строки S_1;

– n — длина строки S_2;

– size(v) — функция, вычисляющая размер входного параметра v в байтах;

– $char$ — тип данных, используемый для хранения символа строки;

– int — целочисленный тип данных.

Теоретическая оценка объема используемой памяти для итеративной реализации алгоритма поиска расстояния Левенштейна:

$$M_{LevIter} = m \cdot n \cdot \text{size}(int) + 2 \cdot \text{size}(char*) + 3 \cdot \text{size}(int) + 2 \cdot \text{size}(int) \quad (4.1)$$

где $(m \cdot n) \cdot \text{size}(int)$ — размер матрицы,
$2 \cdot \text{size}(char*)$ — размер двух указателей входных строк,
$2 \cdot \text{size}(int)$ — размер переменных, хранящих длину строк,
$3 \cdot \text{size}(int)$ — размер дополнительных переменных.

Для итеративного алгоритма поиска расстояния Дамерау — Левенштейна теоретическая оценка объема используемой памяти идентична.

Теоретическая оценка объема затраченной памяти для рекурсивных реализаций алгоритма нахождения расстояния Дамерау — Левенштейна.

Расчет объема памяти, используемой каждым вызовом функции поиска расстояния Дамерау — Левенштейна:

$$M_{call} = 2 \cdot \text{size}(char*) + 2 \cdot \text{size}(int) + \text{size}(int) + 8 \quad (4.2)$$

где $2 \cdot \text{size}(char*)$ — двух указателей входных строк,
$2 \cdot \text{size}(int)$ — размер двух входных строк,
size(int) — размер вспомогательной переменной,
8 байт — адрес возврата.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, поэтому максимальный расход памяти равен

$$M_{DLRec} = (m + n) \cdot M_{call} \tag{4.3}$$

где $m + n$ — максимальная глубина стека вызовов,
M_{call} — затраты по памяти для одного рекурсивного вызова.

Рекурсивная реализация алгоритма поиска расстояния Дамерау — Левенштейна с кэшированием для хранения промежуточных значений использует матрицу кэш, размер которой можно рассчитать следующим образом:

$$M_{cache} = (n \cdot m) \cdot \text{size}(int) \tag{4.4}$$

где $(n \cdot m)$ — количество элементов в кэше,

Следовательно, теоретическая оценка объема используемой памяти для рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна с использованием кэша:

$$M_{DLRecCache} = M_{DLRec} + M_{cache} \tag{4.5}$$

В таблице 4.2, приведен расчет затраты памяти в байтах для различных алгоритмов в зависимости от длины строки. Пусть тип *char* равен 1 байту, тип *int* равен 4 байтам, а указатель на строку *char** равен 8 байтам.

Таблица 4.2 – Затраты памяти для различных алгоритмов

Длина (символ)	**Память, байт**			
	Левенштейн	**Дамерау — Левенштейн**		
	Итеративный	**Итеративный**	**Рекурсивный**	
			Без кеша	**С кешем**
1	40	40	72	76
101	40840	40840	7272	48076
201	161640	161640	14472	176076
301	362440	362440	21672	384076
401	643240	643240	28872	672076
501	1004040	1004040	36072	1040076
601	1444840	1444840	43272	1488076
701	1965640	1965640	50472	2016076
801	2566440	2566440	57672	2624076
901	3247240	3247240	64872	3312076

4.5 Вывод

В результате исследования реализуемых алгоритмов по времени выполнения можно сделать следующие выводы:

Итеративные реализации алгоритмов нахождения расстояния Левенштейна и Дамерау — Левенштейна по итогам исследования показали наименьшее время (см. рис. 4.2), но из-за дополнительной операцией в алгоритме Дамерау — Левенштейна, его реализация медленнее в 1.25 раза при длине строк 11 символов.

Реализация рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна значительно проигрывает остальным алгоритмам (см. рис. 4.2). Такая значительная разница в производительности обусловлена проблемой повторных вычислений, с которой сталкиваются рекурсивные алгоритмы [8]. При обработке слов длиной в 11 символов эта проблема становится заметной, так как количество повторных вычислений значительно увеличивается. Внедрение кэширование в алгоритм нахождения расстояния Дамерау — Левенштейна привело к решению проблемы повторных вычислений в рекурсивных алгоритмах.

Полученные результаты могут зависеть от характеристик используемой вычислительной системы. Возможны вариации результатов при выполнении замеров на различных компьютерах.

В результате исследования алгоритмов по затрачиваемой памяти можно сделать следующие выводы:

Итеративные алгоритмы и рекурсивный алгоритм с кэшированием требуют больше памяти по сравнению с рекурсивным. В реализациях, использующих матрицу, максимальный размер используемой памяти увеличивается пропорционально произведению длин строк, в то время как у рекурсивного алгоритма без кэширования объем затрачиваемой памяти увеличивается пропорционально сумме длин строк.

ЗАКЛЮЧЕНИЕ

Цели работы достигнуты: изучены, реализованы и проведен сравнительный анализ алгоритмов поиска редакционных расстояний Левенштейна и Дамерау — Левенштейна.

В ходе выполнения лабораторной работы были решены все задачи:

1) описаны расстояния Левенштейна и Дамерау — Левенштейна;

2) описаны алгоритмы поиска расстояний Левенштейна и Дамерау — Левенштейна;

3) разработано программное обеспечение, включающее в себя нерекурсивный алгоритм поиска расстояния Левенштейна и нерекурсивный алгоритм поиска расстояния Дамерау — Левенштейна, а также рекурсивный алгоритм поиска расстояния Дамерау — Левенштейна и рекурсивный с кешированием алгоритм поиска расстояния Дамерау — Левенштейна;

4) проведен сравнительный анализ времени выполнения реализаций алгоритмов и занимаемой памяти.

   - При обработке строк длиной до 11 символов разница между временем выполнения нерекурсивных реализаций алгоритмов Левенштейна и Дамерау — Левенштейна незначительна, но из-за дополнительной операции в алгоритме Дамерау — Левенштейна, его реализация работает медленнее, а именно в 1.25 раза при длине слов 11 символов.
   - Рекурсивный алгоритм поиска расстояния Дамерау — Левенштейна выполняется на порядок дольше, чем тот же алгоритм, использующий кэширование.
   - Время работы матричного и рекурсивного с кэшированием алгоритмов поиска расстояния Дамерау — Левенштейна приблизительно равно.
   - Итеративные алгоритмы и рекурсивный алгоритм с кэшированием требуют больше памяти по сравнению с рекурсивным. В реализациях, использующих матрицу, максимальный размер используемой

памяти увеличивается пропорционально произведению длин строк, в то время как у рекурсивного алгоритма без кэширования объем затрачиваемой памяти увеличивается пропорционально сумме длин строк.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Погорелов Д. А.*, *Таразанов А. М.* Сравнительный анализ алгоритмов редакционного расстояния Левенштейна и Дамерау-Левенштейна. — 2019. — Режим доступа: https://elibrary.ru/item.asp?id=36907767 (дата обращения: 10.10.2023).

2. *Траулько М. В.* Программная реализация нечеткого поиска текстовой информации в словаре с помощью расстояния Левенштейна. — 2017. — Режим доступа: https://cyberleninka.ru/article/n/programmnaya-realizatsiya-nechetkogo-poiska-tekstovoy-informatsii-v-slovare-s-pomoschyu-rasstoyaniya-levenshteyna (дата обращения: 14.10.2023).

3. *Левенштейн В. И.* Двоичные коды с исправлением выпадений, вставок и замещений символов // М.: Издательство «Наука», Доклады АН СССР. — 1965.

4. Документация по Python 3.11.5 [Электронный ресурс]. — Режим доступа: https://docs.python.org/3/ (дата обращения: 20.09.2023).

5. Python library function process-time() [Электронный ресурс]. — Режим доступа: https://docs.python.org/3/library/time.html (дата обращения: 20.09.2023).

6. Ryzen 4600H Электронный ресурс]. — Режим доступа: https://www.amd.com/en/products/apu/amd-ryzen-5-4600h (дата обращения: 20.09.2023).

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

8. Основные понятия и терминология рекурсии [Электронный ресурс]. — Режим доступа: https://cyberleninka.ru/article/n/osnovnye-ponyatiya-i-terminologiya-rekursi (дата обращения: 26.09.2022).