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 Динь Вьет Ань

Группа ИУ7И-54Б

Оценка (баллы)

Преподаватель Волкова Л.Л., Строганов Ю.В.

Москва — 2023 г.

Оглавление

Введение

Целью данной лабораторной работы является описание, реализация и исследование алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

Расстояние Левенштейна – метрика, измеряющая по модулю разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (вставки, удаления, замены), необходимых для превращения одной последовательности символов в другую.

Расстояния Левенштейна и Дамерау-Левенштейна широко применяются для решения задач компьютерной лингвистики (исправление ошибок в слове, автоматическое распознавание отсканированного текста или речи), биоинформатики (для сравнения генов, хромосом) и других.

Расстояние Дамерау-Левенштейна – модификация расстояния Левештейна. Это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую.

В рамках выполнения лабораторной работы необходимо решить следующие задачи:

— описать расстояния Левенштена и Дамерау-Левенштейна;

— построить схемы алгоритмов следующих методов: нерекурсивный метод поиска расстояния Левенштейна, нерекурсивный метод поиска Дамерау-Левенштейна, рекурсивный метод поиска Дамерау-Левенштейна, рекурсивный с кешированием метод поиска Дамерау-Левенштейна;

— создать ПО, реализующее перечисленные выше алгоритмы;

— сравнить алгоритмы определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);

— описать и обосновать полученные результаты в отчете о выполненной лабораторной работе;

1 Аналитическая часть

В этом разделе будут представлены описания алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна и их практическое применение.

1.1 Нерекурсивный алгоритм нахождения расстояния Левенштейна

Расстояние Левенштейна [1] между двумя строками — это минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую.

Каждая операция имеет свою цену (штраф). Редакционным предписанием называется последовательность действий, необходимых для получения из первой строки второй, и минимизирующих суммарную цену. Суммарная цена есть искомое расстояние Левенштейна.

Введем следующие обозначения операций с указанием соответсвующих штрафов w:

— D (англ. delete) — удаление ($w(a, \lambda) = 1$);

— I (англ. insert) — вставка ($w(\lambda, b) = 1$);

— R (англ. replace) — замена ($w(a, b) = 1,\ a \neq b$);

— M (англ. match) — совпадение ($w(a, a) = 0$).

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по рекуррентной формуле 1.1.

$$D(i,j) = \begin{cases} 0, \text{ если i} = 0, \text{ j} = 0, \\ i, \text{ если j} = 0, \text{ i} > 0, \\ j, \text{ если i} = 0, \text{ j} > 0, \\ \min\{ \\ \quad D(i, j-1) + 1 \\ \quad D(i-1, j) + 1 \\ \quad D(i-1, j-1) + m(a[i], b[j]) \\ \}, \text{ если i} > 0, \text{ j} > 0, \end{cases} \tag{1.1}$$

где m определяется следующим образом:

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

Нерекурсивный алгоритм реализует формулу (1.1). Функция D составлена таким образом, что для перевода из строки a в строку b требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена) в некоторой последовательности.

С ростом i, j прямая рекурсивная реализация формулы (1.1) становится малоэффективной по времени исполнения, так как множество промежуточных значений $D(i,j)$ вычисляются не по одному разу. Для решения этой проблемы можно использовать матрицу для хранения соответствующих промежуточных значений.

Чтобы исключить повторные вычисления, будет использоваться матрица размером $(length(S1) + 1)$x$((length(S2) + 1)$, где $length(S)$ – длина строки S. Значение в ячейке $[i, j]$ равно значению $D(S1[1...i], S2[1...j])$.

Вся таблица (за исключением первого столбца и первой строки) заполняется в соответствии с формулой (1.3).

$$A[i][j] = min \begin{cases} A[i-1][j] + 1 \\ A[i][j-1] + 1 \\ A[i-1][j-1] + m(S1[i], S2[j]) \end{cases}. \tag{1.3}$$

Функция m определена как:

$$m(S1[i], S2[j]) = \begin{cases} 0, & \text{если } S1[i-1] = S2[j-1], \\ 1, & \text{иначе} \end{cases}. \tag{1.4}$$

В результате расстоянием Левенштейна будет ячейка матрицы с индексами $i = length(S1)$ и $j = length(S2)$ при учете, что индексы начинаются с 0. Рекурсивный алгоритм поиска рассточния Левенштейна будет такми образом использовать вспамогательную мптрицу.

1.2 Нерекурсивный алгоритм поиска Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна [2] – это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) соседних символов.

Расстояние Дамерау-Левенштейна может быть найдено по формуле:

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

Формула выводится по тем же соображениям, что и формула (1.1).

1.3 Рекурсивный алгоритм поиска Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна может быть найдено по формуле 1.6.

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), \text{ если } \min(i,j) = 0, \\ \min\{ \\ \quad d_{a,b}(i,j-1) + 1, \\ \quad d_{a,b}(i-1,j) + 1, \\ \quad d_{a,b}(i-1,j-1) + m(a[i],b[j]), \text{ иначе} \\ \quad \begin{cases} d_{a,b}(i-2,j-2) + 1, \text{ если } i,j > 1; \\ \qquad a[i] = b[j-1]; \\ \qquad b[j] = a[i-1] \\ \qquad \infty, \text{ иначе} \end{cases} \\ \} \end{cases}, \tag{1.6}$$

1.4 Рекурсивный с кешированием алгоритм поиска Дамерау-Левенштейна

Рекурсивный алгоритм заполнения можно оптимизировать по времени выполнения с использованием кеша. В качестве кеша используется матрица. Суть данной оптимизации заключается в параллельном заполнении матрицы при выполнении рекурсии. В случае, если рекурсивный алгоритм выполняет прогон для данных, которые еще не были обработаны, результат нахождения расстояния заносится в матрицу. В случае, если обработанные ранее данные встречаются снова, для них расстояние не находится и алгоритм переходит к следующему шагу.

Вывод

В данном разделе были рассмотрены алгоритмы поиска расстояния Левенштейна и расстояния Дамерау-Левенштейна. В частности были приведены рекурентные формулы работы алгоритмов, объяснена разница между расстоянием Левенштейна и расстоянием Дамерау-Левенштейна.

2 Конструкторская часть

В этом разделе будут приведены требования к вводу и программе, а также схемы алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

2.1 Требования к вводу

На вход подаются две строки, причем буквы верхнего и нижнего регистров считаются различными.

2.2 Требования к программе

При вводе двух пустых строк программа не должна аварийно завершатся. Вывод программы - число (расстояние Левенштейна или Дамерау-Левенштейна), для алгоритмов, использующих матрицу для расчёта требуется вывести заполненную матрицу.

2.3 Разработка алгоритма поиска расстояния Левенштейна

На рисунке 2.1 приведена схема нерекурсивного алгоритма нахождения расстояния Левенштейна.

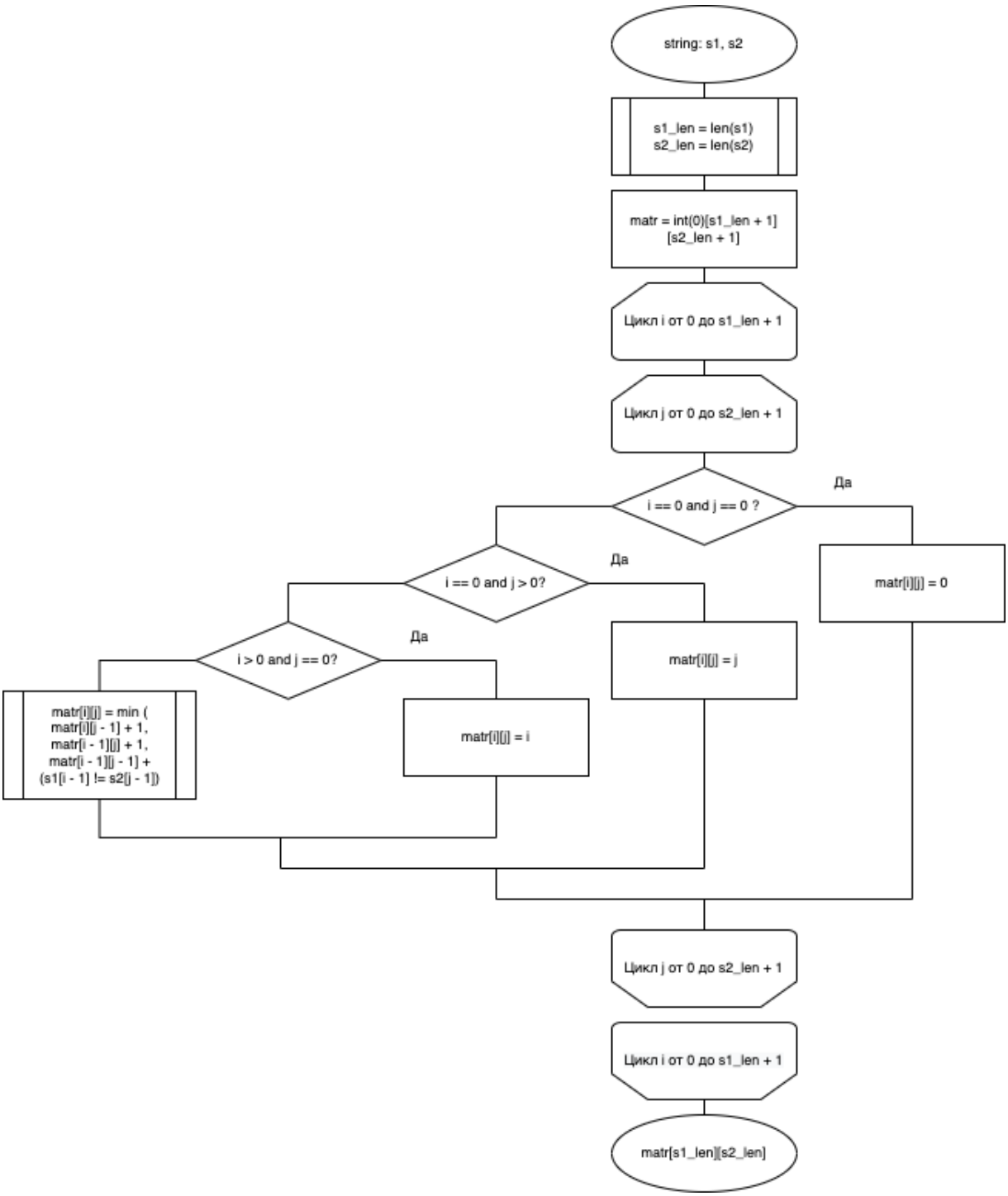

Рисунок 2.1 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

2.4 Разработка алгоритмов поиска расстояния Дамерау-Левенштейна

На рисунке 2.2 приведена схема нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна.

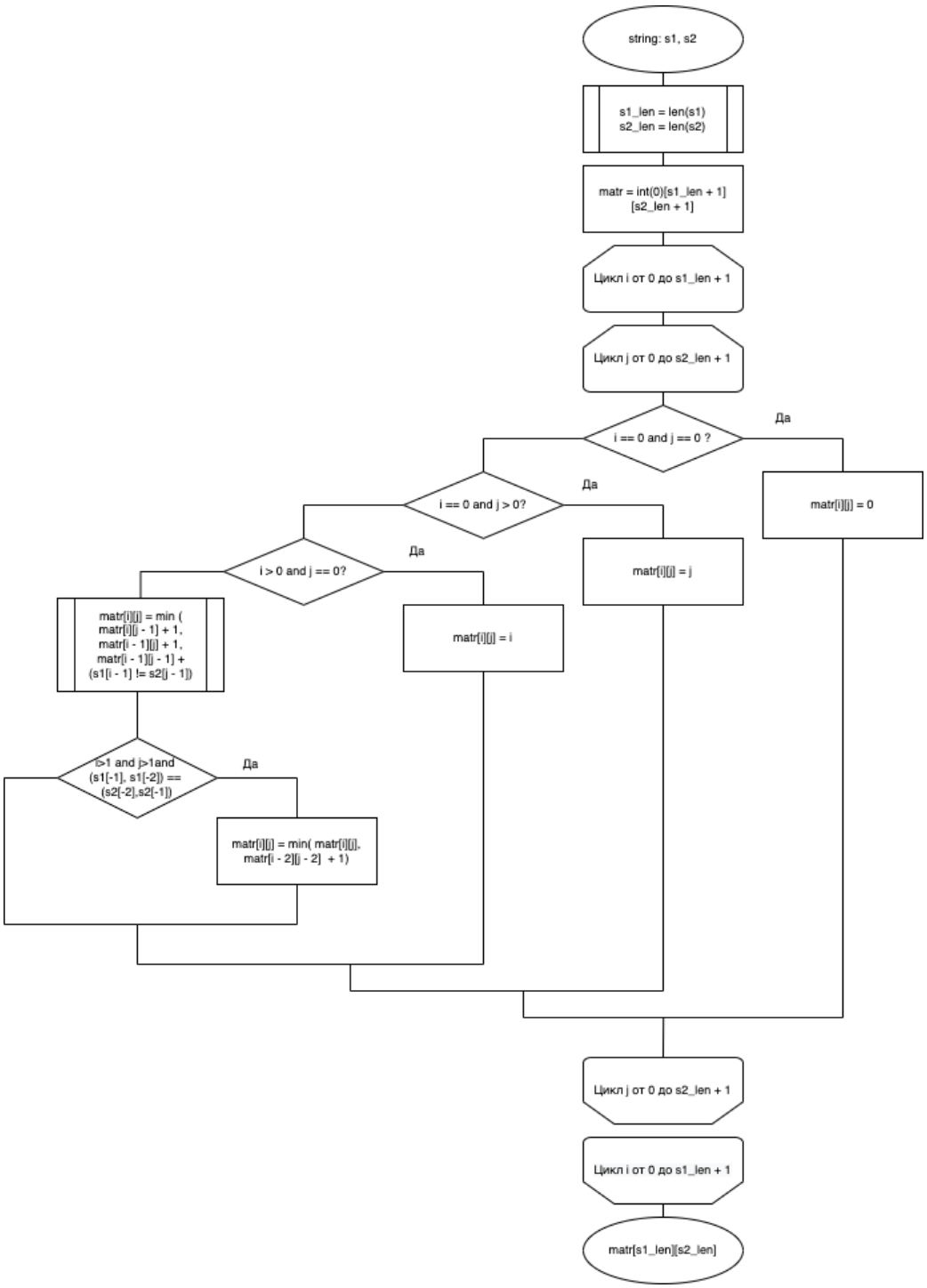

Рисунок 2.2 – Схема нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

На рисунке 2.3 приведена схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна.

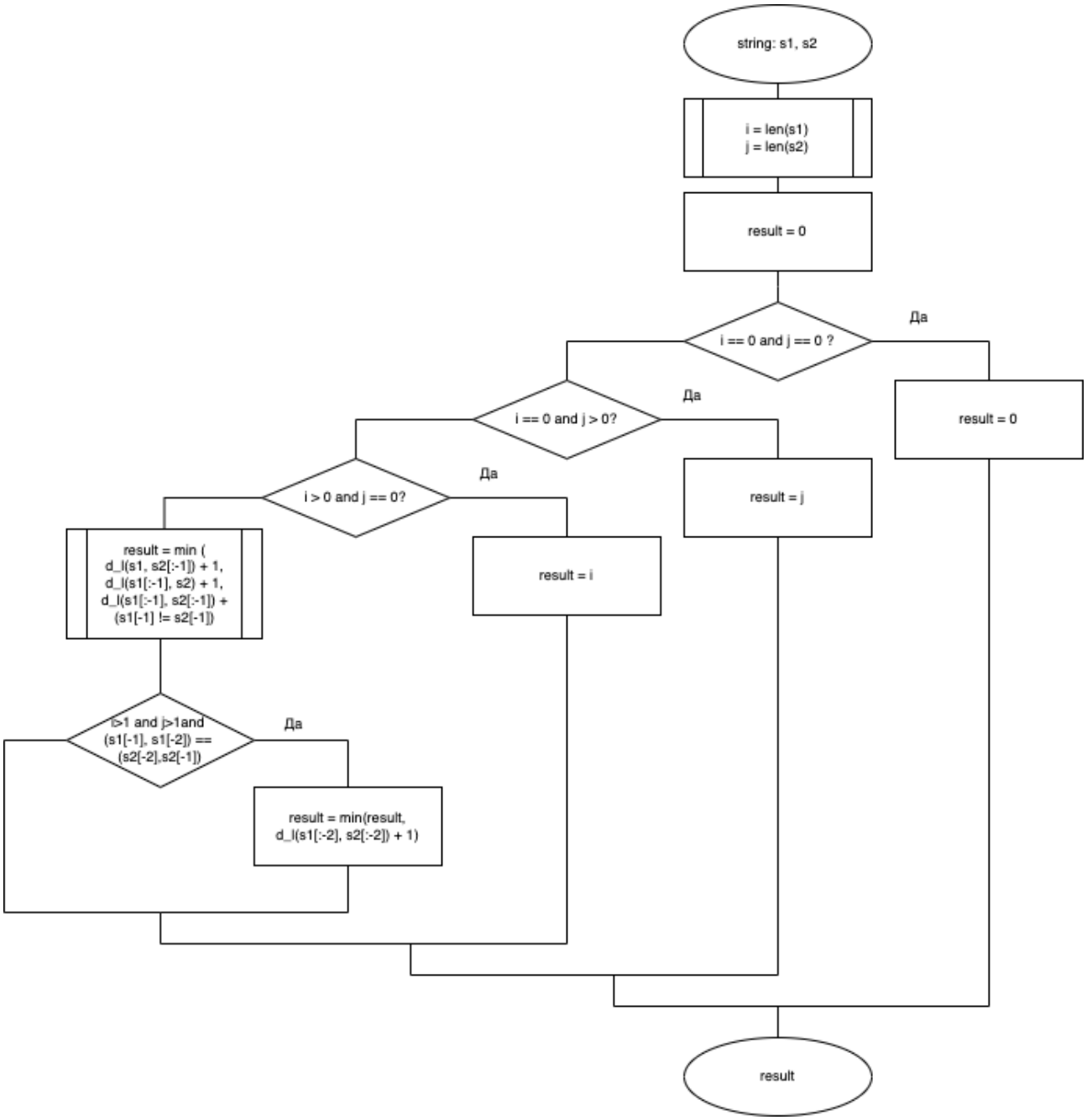

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

На рисунке 2.4 приведена схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с использованием кеша в виде матрицы.

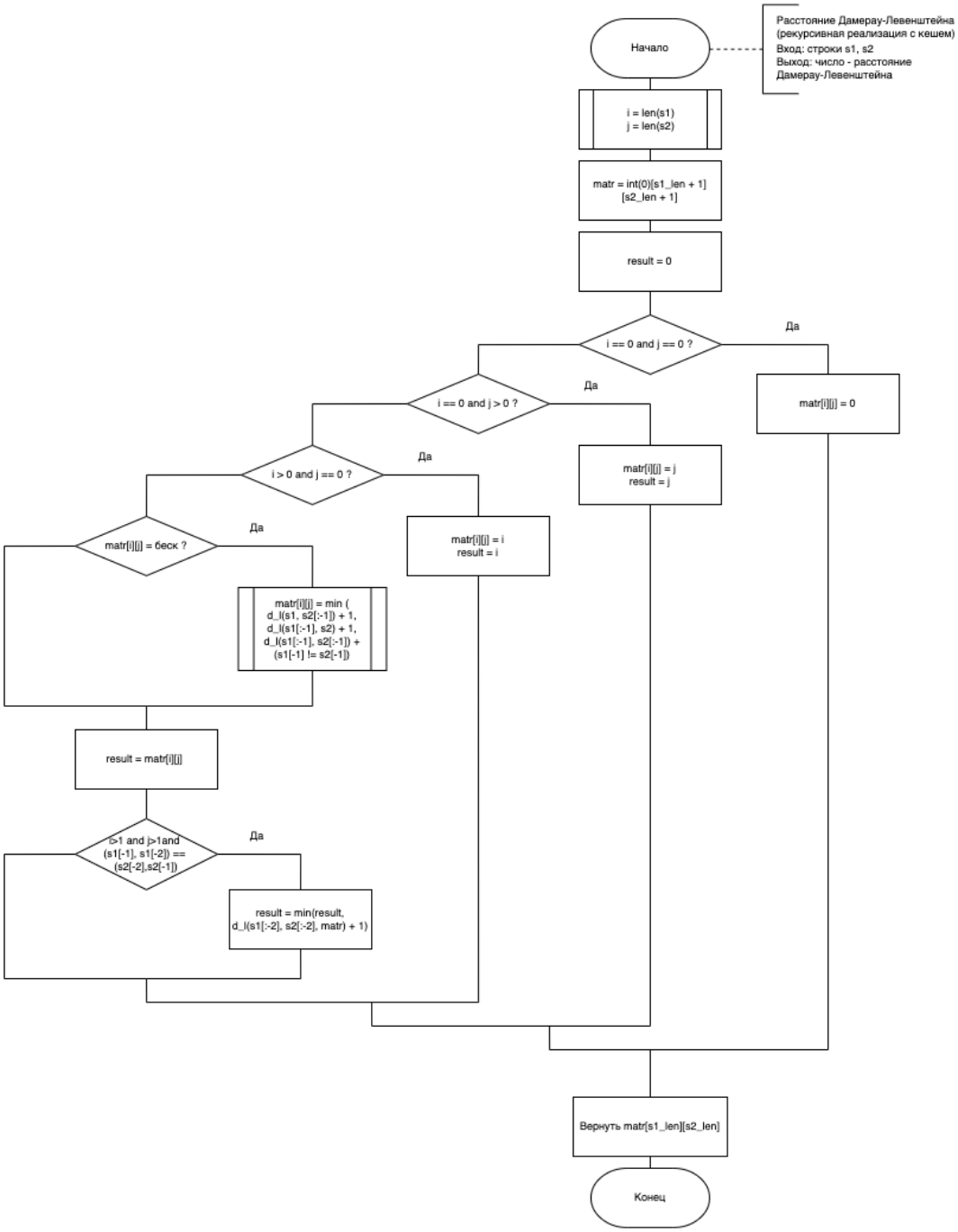

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с использованием кеша в виде матрицы

Вывод

Перечислены требования к вводу и программе, а также на основе теоретических данных, полученных из аналитического раздела, были построены схемы требуемых алгоритмов.

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к ПО

Программа принимает две строки (регистрозависимые). В качестве результата возвращается число, равное редакторскому расстоянию. Необходимо реализовать возможность подсчета процессорного времени и пиковой использованной памяти для каждого из алгоритмов.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран ЯП Python [3].

Данный язык имеет все небходимые инструменты для решения поставленной задачи.

Процессорное время работы реализаций алгоритмов было замерено с помощью функции process_time() из библиотеки time [4].

3.3 Сведения о модулях программы

Программа состоит из трех модулей:

1) main.py — точка входа;

2) algorythms.py — хранит реализацию алгоритмов;

3) compareTime.py — хранит реализацию системы замера времени.

3.4 Листинг кода

В листингах 3.1, 3.2, 3.3, 3.4 приведены реализаций алгоритмов нахождения расстояния Левенштейна и Дамерау–Левенштейна.

Листинг 3.1 – Функция нахождения расстояния Левенштейна нерекурсивным методом

```
def levenshteinDistance(str1, str2, output = True):
    n = len(str1)
    m = len(str2)
    matrix = createMatrix(n + 1, m + 1)

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            add = matrix[i - 1][j] + 1
            delete = matrix[i][j - 1] + 1
            change = matrix[i - 1][j - 1]

            if (str1[i - 1] != str2[j - 1]):
                change += 1

            matrix[i][j] = min(add, delete, change)

    if (output):
        printMatrix(matrix, str1, str2)

    return matrix[n][m]
```

Листинг 3.2 – Функция нахождения расстояния Дамерау–Левенштейна нерекурсивным методом

```
def damerauLevenshteinDistance(str1, str2, output = True):
    n = len(str1)
    m = len(str2)
    matrix = createMatrix(n + 1, m + 1)

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            add = matrix[i - 1][j] + 1
            delete = matrix[i][j - 1] + 1
            change = matrix[i - 1][j - 1]
```

```
11              if (str1[i - 1] != str2[j - 1]):
12                  change += 1
13              swap = n
14              if (i > 1 and j > 1 and
15                  str1[i - 1] == str2[j - 2] and
16                  str1[i - 2] == str2[j - 1]):
17                      swap = matrix[i - 2][j - 2] + 1
18
19              matrix[i][j] = min(add, delete, change, swap)
20
21      if (output):
22          printMatrix(matrix, str1, str2)
23
24      return matrix[n][m]
```

Листинг 3.3 – Функция нахождения расстояния Дамерау–Левенштейна с использованием рекурсии

```
1  def damerauLevenshteinDistanceRecursive(str1, str2, output =
       True):
2      n = len(str1)
3      m = len(str2)
4      flag = 0
5      if ((n == 0) or (m == 0)):
6          return abs(n - m)
7      if (str1[-1] != str2[-1]):
8          flag = 1
9      minDistance = min(
10         damerauLevenshteinDistanceRecursive(str1[:-1], str2) + 1,
11         damerauLevenshteinDistanceRecursive(str1, str2[:-1]) + 1,
12         damerauLevenshteinDistanceRecursive(str1[:-1], str2[:-1])
               + flag
13     )
14     if (n > 1 and m > 1 and str1[-1] == str2[-2] and str1[-2] ==
           str2[-1]):
15         minDistance = min(
16             minDistance,
17             damerauLevenshteinDistanceRecursive(str1[:-2],
                   str2[:-2]) + 1
18         )
19     return minDistance
```

Листинг 3.4 – Функция нахождения расстояния Дамерау–Левенштейна рекурсивным методом с использованием кеша

```
1  def recursiveWithCache(str1, str2, n, m, matrix):
2      if (matrix[n][m] != -1):
3          return matrix[n][m]
4      if (n == 0):
5          matrix[n][m] = m
6          return matrix[n][m]
7      if ((n > 0) and (m == 0)):
8          matrix[n][m] = n
9          return matrix[n][m]
10     add = recursiveWithCache(str1, str2, n - 1, m, matrix) + 1
11     delete = recursiveWithCache(str1, str2, n, m - 1, matrix) + 1
12     change = recursiveWithCache(str1, str2, n - 1, m - 1, matrix)
13     if (str1[n - 1] != str2[m - 1]):
14         change += 1 # flag
15     matrix[n][m] = min(add, delete, change)
16     if (n > 1 and m > 1 and
17         str1[n - 1] == str2[m - 2] and
18         str1[n - 2] == str2[m - 1]):
19
20         matrix[n][m] = min(
21             matrix[n][m],
22             recursiveWithCache(str1, str2, n - 2, m - 2, matrix)
                   + 1
23         )
24     return matrix[n][m]
25
26 def damerauLevenshteinDistanceRecurciveCache(str1, str2, output =
       True):
27     n = len(str1)
28     m = len(str2)
29     matrix = createMatrix(n + 1, m + 1)
30     for i in range(n + 1):
31         for j in range(m + 1):
32             matrix[i][j] = -1
33     recursiveWithCache(str1, str2, n, m, matrix)
34     if (output):
35         printMatrix(matrix, str1, str2)
36     return matrix[n][m]
```

3.5 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна (в таблице столбец подписан "Левенштейн") и Дамерау-Левенштейна (в таблице - "Дамерау-Л."). Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

	Входные данные		Ожидаемый результат	
№	Строка 1	Строка 2	Левенштейн	Дамерау-Л.
1	"пустая строка"	"пустая строка"	0	0
2	"пустая строка"	a	1	1
3	b	"пустая строка"	1	1
4	asdfv	"пустая строка"	5	5
5	"пустая строка"	sdfvs	5	5
8	lll	lll	0	0
9	qwem	qwem	0	0
10	aa	cg	2	2
11	kot	sobaka	5	5
12	stroka	sobaka	3	3
13	kot	kod	1	1
14	cat	caaat	2	2
15	cat	catty	2	2
16	cat	tac	2	2
17	recur	norecur	2	2
18	1234	2143	3	2
19	mriak	mriakmriak	5	5
20	aaaaa	aa	3	3
21	abcdef	acbdef	2	1

Вывод

Были разработаны и протестированы алгоритмы нахождения расстояния Левенштейна нерекурсивно, нахождения расстояния Дамерау – Левенштейна нерекурсивно, рекурсивно, а также рекурсивно с кешированием.

4 Исследовательская часть

В данном разделе приводятся результаты замеров затрат реализаций алгоритмов по пиковой памяти и процессорному времени.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

— Операционная система Window 10 Home Single Language;

— Память 8 Гб;

— Процессор 11th Gen Intel(R) Core(TM) i5-1135G7 2.42 ГГц, 4 ядра.

Во время тестирования устройство было подключено к сети электропитания, нагружено приложениями окружения и самой системой тестирования.

4.2 Время выполнения реализаций алгоритмов

Процессорное время реализаций алгоритмов замерялось при помощи функции process_time() из библиотеки time языка Python. Данная функция возвращает количество секунд, прошедших с начала эпохи, типа float.

Контрольная точка возвращаемого значения не определна, поэтому допустима только разница между результатами последовательных вызовов.

Замеры времени для каждой длины слов проводились 300 раз. В качестве результата взято среднее время работы алгоритма на данной длине слова. При каждом запуске алгоритма, на вход подавались случайно сгенерированные строки, днины строк (Length) совпадаютю.

Результаты замеров приведены на рисунке 4.1 (в микросекундах).

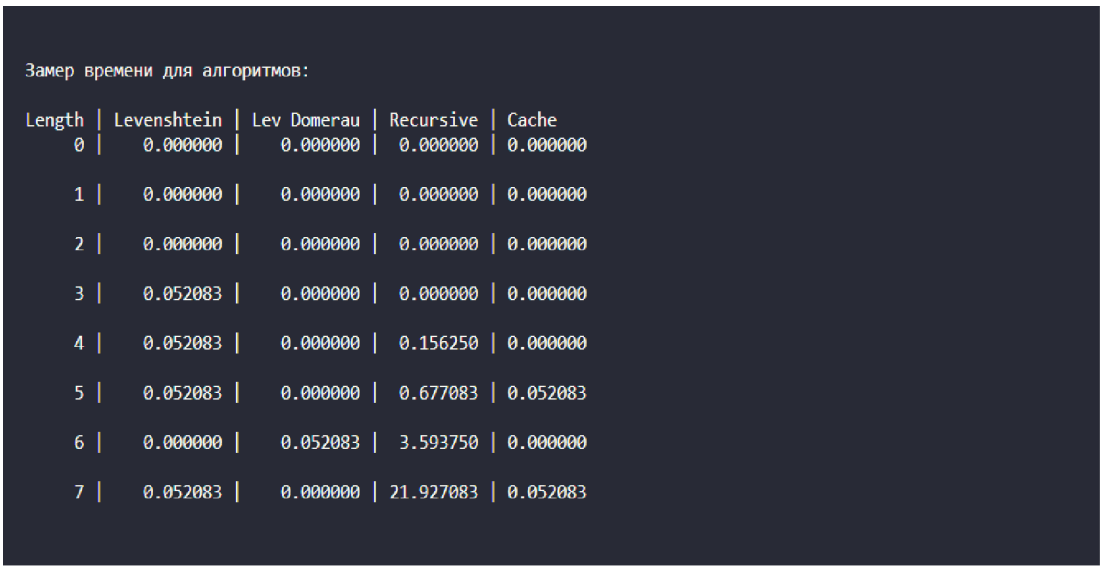

Рисунок 4.1 – Результаты замеров времени (в микросекундах)

На рис. 1.1 приняты следующие обозначения алгоритмов:

— Levenshtein — нерекурсивный алгоритм нахождения расстояния Левенштейна,

— Lev Domerau — нерекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна,

— Recursive — рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна,

— Cache — рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с использованием кеша.

Вывод

В результате замеров можно прийти к выводу, что матричная реализация алгоритмов нахождения расстояний заметно выигрывает по времени при росте строк.

4.3 Затраты памяти выполнения реализаций алгоритмов

Алгоритмы нахождения расстояний Левенштейна и Дамерау — Левенштейна не отличаются друг от друга с точки зрения использования памяти, поэтому достаточно рассмотреть лишь разницу рекурсивной и матричной реализаций данных алгоритмов.

Пусть длина строки S1 — n, длина строки S2 — m, тогда затраты памяти на приведенные алгоритмы будут следующими.

Матричный алгоритм поиска расстояния Левенштейна:

— строки S1, S2 — $(m + n) \cdot sizeof(char)$,

— матрица — $((m + 1) \cdot (n + 1)) \cdot sizeof(int)$,

— текущая строка матрицы — $(n + 1) \cdot sizeof(int)$,

— длины строк — $2 \cdot sizeof(int)$,

— вспомогательные переменные — $3 \cdot sizeof(int)$.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк.

Рекурсивный алгоритм поиска расстояния Дамерау – Левенштейна (для каждого вызова):

— строки S1, S2 — $(m + n) \cdot sizeof(char)$,

— длины строк — $2 \cdot sizeof(int)$,

— вспомогательные переменные — $2 \cdot sizeof(int)$,

— размер аргументов функции — $2 \cdot 24$,

— размер адреса возврата — 4.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк. Таким образом, общая затраченная память

в рекурсивном алгоритме будет равна $m+n+(2\cdot4+2\cdot4+4+2\cdot24)\cdot(n+m) = 85 \cdot m + 85 \cdot n$.

Рекурсивный алгоритм поиска расстояния Дамерау – Левенштейна с использованием кеша (для каждого вызова):

— Для всех вызовов память для хранения самой матрицы — ((m + 1) · (n + 1)) · sizeof(int),

— строки S1, S2 — (m + n) · sizeof(char),

— длины строк — 2 · sizeof(int),

— вспомогательные переменные — 1 · sizeof(int),

— размер аргументов функции — 2 · 24,

— ссылка на матрицу — 8 байт,

— размер адреса возврата — 4.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк. Таким образом, общая затраченная память в рекурсивном алгоритме будет равна $m + n + 4 \cdot (m + 1) \cdot (n + 1) + (2 \cdot 4 + 4 + 4 + 2 \cdot 24) \cdot (n + m) = 4 \cdot m \cdot n + 69 \cdot m + 69 \cdot n + 4$.

Матричный алгоритм поиска расстояния Дамерау – Левенштейна:

— строки S1, S2 — (m + n) · sizeof(char),

— матрица — ((m + 1) · (n + 1)) · sizeof(int),

— длины строк — 2 · sizeof(int),

— вспомогательные переменные — 3 · sizeof(int).

Таким образом, общая затраченная память в рекурсивном алгоритме будет равна $m + n + 4 \cdot (m + 1) \cdot (n + 1) + 6 \cdot 4 = 4 \cdot m \cdot n + 5 \cdot m + 5 \cdot n + 28$.

Результаты замеров приведены на рисунке 4.2.

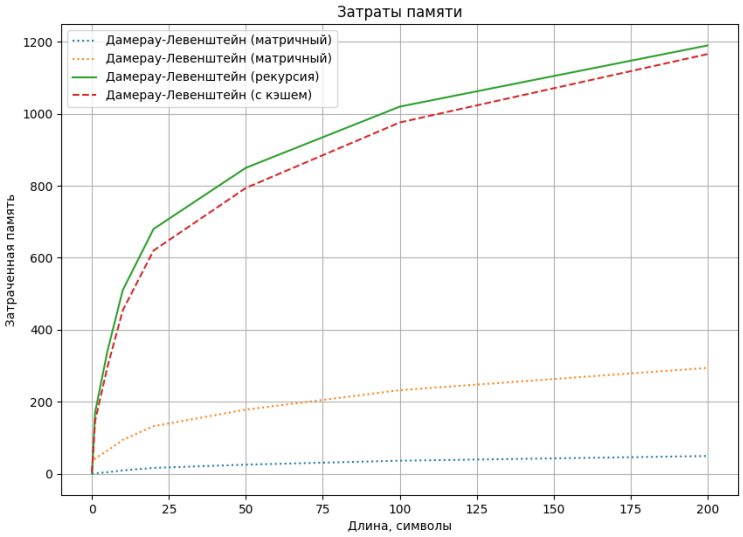

Рисунок 4.2 – Результаты замеров памяти

Вывод

В результате можно прийти к выводу, что матричная реализация алгоритмов нахождения расстояний заметно выигрывает по памяти при росте строк.

Заключение

В ходе выполнения лабораторной работы были решены все задачи:

— описаны алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна;

— реализованы алгоритмы поиска расстояния Левенштейна и расстояния Дамерау-Левенштейна без рекурсии;

— реализованы рекурсивные алгоритмы поиска расстояния Дамерау-Левенштейна с и без матрицы-кеша;

— проведен сравнительный анализ линейной и рекурсивной реализаций алгоритмов определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);

— подготовлен отчет о лабораторной работе.

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализаций для различных длин строк. Было получено, что рекурсивная реализация алгоритмов без кеширования в 3-4 раза проигрывает по памяти нерекурсивной реализации. Однако ее можно улучшить, добавив кеширование, и получить небольшой (в 1.5 раза) выигрыш по памяти по сравнению с нерекурсивными алгоритмами. Анализ временных затрат показал, что для длинных строк (10 символов и больше) рекурсивная реализация алгоритмов работает в 1.5 раза дольше, чем нерекурсивная.

Список использованных источников

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

2. Черненький В. М. Гапанюк Ю. Е. Методика идентификации пассажира по установочным данным. – М.: Вестник МГТУ им. Н.Э. Баумана. Сер. “Приборостроение”, 2012. Т. 163. С. 30–34.

3. Welcome to Python [Электронный ресурс]. Режим доступа: `https://www.python.org` (дата обращения: 01.02.2023).

4. time — Time access and conversions [Электронный ресурс]. Режим доступа: `https://docs.python.org/3/library/time.html#functions` (дата обращения: 01.02.2023).