**Министерство науки и высшего образования Российской Федерации**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

**Факультет безопасности информационных технологий**

**Дисциплина:**

«Алгоритмы и структуры данных»

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 4**

«Интегрирование методом Гаусса-Лежандра с использованием сортировки расчёской и очереди на основе массива с сохранением результата в красно-черное дерево»

**Выполнил:**

Гачко Г. Д., студент группы N3246

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

(подпись)

**Проверил:**

Ерофеев С. А.

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

(подпись)

Санкт-Петербург

2024 г.

Содержание

Введение

1. Описание функционала программы
2. Используемые переменные
3. Анализ алгоритмической сложности
4. Блок-схема
5. Код программы
6. Результаты тестирования

Заключение

Введение

Задача работы – разработать программу интегрирования методом Гаусса-Лежандра с помощью сортировки расчёской. Использовать обычную очередь на базе массива. Записать результат в красно-черное дерево и оценить сложность.

Для выполнения задачи требуется реализовать ввод пар координат из файла, выполнение сортировки, и вычисление интеграла, а также реализовать следующие структуры данных – связный список и кольцевая очередь.

Для реализации был выбран язык программирования Python версии 3.13.0.

# Описание функционала программы

Программа принимает на вход из файла пары координат и записывает их в кольцевую очередь. Далее пары координат сортируются в ней расческой. После этого программа вычисляет интеграл по заданным координатам методом Гаусса-Лежандра, и выводит ответ в stdout.

# Используемые переменные

self.data, data – пары координат типа float, хранящиеся в узлах связного списка.

self.next, self.tail, new\_node, current, node\_i, node\_j, next\_node – переменные типа Node, представляющие узлы связного списка.

gap, self.size, operation\_count, sort\_operations, integration\_operations, i, n, \_ – различные служебные переменные типа int.

x, y, x0, y0, x1, y1, xi, yi, m, total\_integral, integral, dx, xm, shrink – различные служебные переменные типа float (переменная типа float занимает 4 байта).

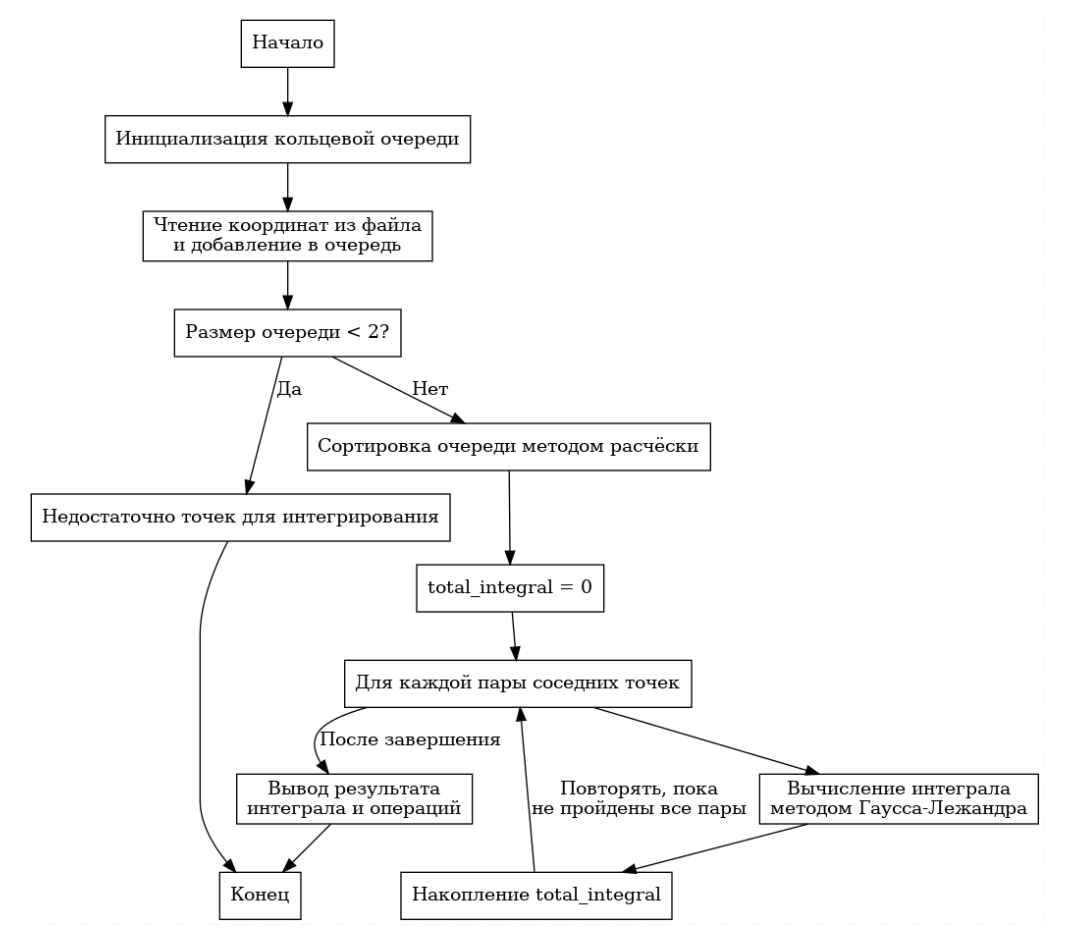
sorted – переменная типа bool.

nodes, weights – списки (List) вещественных чисел типа float.

file, line, x\_str, y\_str – переменные для работы с файлами и строками типа str и объект файла.

queue, coordinate\_queue – переменные типа CircularQueue.

# Блок-схема



# Анализ алгоритмической сложности

Оценка алгоритмической сложности путем подсчета элементарных операций

Примем размер входного массива данных за N. Тогда, количество сдвигов gap == N / 1.3 == 10/13 \* N

1. Сортировка расческой
2. Инициализация переменной gap (размер шага), 1 операция
3. Инициализация переменной shrink (коэффициент уменьшения размера шага), 1 операция
4. Инициализация переменной sorted (флаг отсортированности), 1 операция
5. Начало цикла while, 1 операция
   1. Деление, 1 операция
   2. Присваивание, 1 операция
   3. Проверка на gap < 1, 1 операция
   4. Если gap < 1, 2 операции. Случается раз за сдвиг gap, т.е. 10/13\*N\*2
   5. Установление переменной current, 3 операции
   6. Инициализация индекса i, 1 операция
   7. Начало цикла while, 3 операции. Цикл проходит 10/13\*N раз. Суммарно выходит (1+1+1+1+10/13N+1+3+1)\*10/13N, каждый цикл 10/13N внутри цикла уменьшается на 1.
      1. Инициализация node\_i, 1 операция
      2. Инициализация node\_j, 1 операция
      3. Цикл for, 1 операция, каждый цикл 10/13N уменьшается на 1.
         1. Сдвиг node\_j gap раз, gap\*2 операций
      4. Проверка на доступность перестановки, 5 операций
      5. Две перестановки, 6 операций
      6. Флаг sorted в False, 1 операция
   8. Итерация переменной current, 2 операция
   9. Итерация индекса, 1 операция
   10. Проверка gap, 1 операция
       1. Присвоение, 4 операции
       2. Цикл for, 2 операции для инициализации
          1. Проверка цикла, 2 операции
          2. Присвоение, 2 операции
          3. Сравнение, 5 операций
          4. Две перестановки, 10 операций
          5. Флаг sorted, 1 операция
          6. Присвоение, 1 операция

**Итого**, (3 + (1+1+1+1+2+(10/13\*N\*2)+3+1) + (2\*10/13N\*10/13N)/2 + 2) \* K, где K – коэффициент неотсортированности.

1. Расчет интеграла

За каждые 2 значения в массиве выполняются следующие операции:

1. Инициализация n, 1 операция
2. Инициализация nodes, 4 операции
3. Инициализация weights, 2 операции
4. Вычисление dx, 3 операции
5. Вычисление xm, 3 операции
6. Инициализация integral, 1 операция
7. Инициализация цикла for
8. Цикл for, 2 операции, повторяется 2 раза
   1. Вычисление xi, 4 операции
   2. Проверка, 1 операция
   3. Интерполяция, 1+8 операций
   4. Добавление к интегралу, 4 операции
9. Домножаем на dx, 1 операция

**Итого**, n/2 \* (16 + 34) операции

1. Чтение данных из файла и заполнение очереди
2. Открыть файл, 1 операция
3. За каждую строку, (n):
   1. Считать пару координат, 2 операции
   2. Преобразовать к float, 2 операции
   3. Добавить в очередь, 2 операции

**Итого**, 1 + 6n операций

# Код программы

import math

# Определяем класс узла для связного списка

class Node:

    def \_\_init\_\_(self, data):

        # Данные, хранящиеся в узле (пара координат)

        self.data = data

        # Ссылка на следующий узел

        self.next = None

# Определяем класс кольцевой очереди на базе связного списка

class CircularQueue:

    def \_\_init\_\_(self):

        # Указатель на последний узел в очереди

        self.tail = None

        # Количество элементов в очереди

        self.size = 0

    def enqueue(self, data):

        # Создаем новый узел с данными

        new\_node = Node(data)

        # Если очередь пуста

        if self.tail is None:

            # Новый узел указывает на себя, образуя кольцо

            new\_node.next = new\_node

            # Устанавливаем tail на новый узел

            self.tail = new\_node

        else:

            # Вставляем новый узел после tail и обновляем tail

            new\_node.next = self.tail.next

            self.tail.next = new\_node

            self.tail = new\_node

        # Увеличиваем размер очереди

        self.size += 1

    def is\_empty(self):

        # Проверяем, пуста ли очередь

        return self.tail is None

    def clear(self):

        # Очищаем очередь

        self.tail = None

        self.size = 0

    def \_\_len\_\_(self):

        # Возвращаем размер очереди

        return self.size

# Функция сортировки расчёской над кольцевой очередью

def comb\_sort(queue):

    # Инициализируем размер шага

    gap = queue.size

    # Инициализируем коэффициент уменьшения шага

    shrink = 1.3

    # Флаг для отслеживания состояния сортировки

    sorted = False

    # Цикл продолжается, пока список не будет отсортирован

    while not sorted:

        # Обновляем размер шага

        gap = int(gap / shrink)

        # Если шаг меньше 1, устанавливаем его равным 1

        if gap <= 1:

            gap = 1

            sorted = True

        # Инициализируем текущий узел

        current = queue.tail.next  # Начало очереди

        # Инициализируем индекс

        i = 0

        # Выполняем проход по очереди с текущим шагом

        while i + gap < queue.size:

            # Инициализируем узлы для сравнения

            node\_i = current

            node\_j = current

            # Перемещаемся на gap позиций от текущего узла

            for \_ in range(gap):

                node\_j = node\_j.next

            # Сравниваем данные узлов по x-координате

            if node\_i.data[0] > node\_j.data[0]:

                # Обмениваем данные узлов

                node\_i.data, node\_j.data = node\_j.data, node\_i.data

                # Так как произошел обмен, устанавливаем sorted в False

                sorted = False

            # Переходим к следующему узлу

            current = current.next

            # Увеличиваем индекс

            i += 1

        # Дополнительная проверка для случая, когда gap равен 1

        # и последний элемент может быть не на своем месте

        if gap == 1:

            # Проходим оставшиеся элементы

            current = queue.tail.next

            for \_ in range(queue.size - 1):

                next\_node = current.next

                # Сравниваем текущий и следующий узлы

                if current.data[0] > next\_node.data[0]:

                    # Обмениваем данные узлов

                    current.data, next\_node.data = next\_node.data, current.data

                    # Устанавливаем sorted в False

                    sorted = False

                # Переходим к следующему узлу

                current = next\_node

# Функция линейной интерполяции

def interpolate(x0, y0, x1, y1, x):

    # Вычисляем наклон прямой

    m = (y1 - y0) / (x1 - x0) if x1 != x0 else 0.0

    # Вычисляем значение y в точке x

    y = y0 + m \* (x - x0)

    return y

# Функция интегрирования методом Гаусса-Лежандра

def gauss\_legendre\_integration(x0, y0, x1, y1):

    # Число точек для квадратуры

    n = 2

    # Узлы и веса для 2-точечной квадратуры на [-1, 1]

    nodes = [-1.0 / math.sqrt(3), 1.0 / math.sqrt(3)]

    weights = [1.0, 1.0]

    # Половина длины интервала

    dx = (x1 - x0) / 2.0

    # Середина интервала

    xm = (x1 + x0) / 2.0

    # Инициализируем интеграл

    integral = 0.0

    # Проходим по узлам квадратуры

    for i in range(n):

        # Преобразуем узел к интервалу [x0, x1]

        xi = xm + dx \* nodes[i]

        # Если x1 == x0, устанавливаем xi = x0, чтобы избежать деления на ноль

        if x1 == x0:

            xi = x0

        # Вычисляем yi с помощью интерполяции

        yi = interpolate(x0, y0, x1, y1, xi)

        # Добавляем вклад в интеграл

        integral += weights[i] \* yi

    # Умножаем на dx для получения итогового интеграла

    integral \*= dx

    return integral

# Инициализируем кольцевую очередь

coordinate\_queue = CircularQueue()

# Счетчик элементарных операций

operation\_count = 0

# param for uniq X check

uniq\_X = []

# Открываем файл для чтения

with open('/home/every/dev/AISD\_4th\_sem\_labs/ЛР 3/test.txt', 'r') as file:

    # Проходим по каждой строке файла

    for line in file:

        # Разбиваем строку на x и y

        try:

            x\_str, y\_str = line.strip().split()

            # Преобразуем строки в числа

            x = float(x\_str)

            uniq\_X.append(x)

            y = float(y\_str)

        except:

            print("Входные данные не соответствуют формату")

            exit()

        # Добавляем пару (x, y) в очередь

        coordinate\_queue.enqueue((x, y))

set\_uniq\_X = set(uniq\_X)

if len(uniq\_X) != len(set\_uniq\_X):

    print("В вашем массиве координат есть некорректные значения")

    exit()

# Выполняем сортировку расчёской

comb\_sort(coordinate\_queue)

# Инициализируем общий интеграл

total\_integral = 0.0

# Проверяем, что в очереди достаточно элементов для интегрирования

if coordinate\_queue.size < 2:

    print("Недостаточно точек для интегрирования.")

else:

    # Начинаем с головы очереди

    current = coordinate\_queue.tail.next

    # Проходим по всем узлам

    for \_ in range(coordinate\_queue.size - 1):

        # Получаем текущие точки

        x0, y0 = current.data

        x1, y1 = current.next.data

        # Вычисляем интеграл на текущем интервале

        integral = gauss\_legendre\_integration(x0, y0, x1, y1)

        # Добавляем к общему интегралу

        total\_integral += integral

        # Переходим к следующему узлу

        current = current.next

# Выводим общий интеграл

print("Интеграл равен:", total\_integral)

# Результаты тестирования

Тестирование производилось с помощью интерпретатора Python 3.13.0 и среды разработки [ms-python.python](https://marketplace.visualstudio.com/items?itemName=ms-python.python) версии 2024.3.10791010

Для заданных координат интеграл по методу Гаусса-Лежандра вычислен корректно.

Тест 1:

Входные данные:

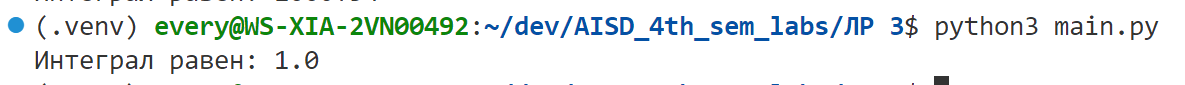
**0 0**

**0 1**

**1 1**

**1 0**

Результат:



Тест 2:

Входные данные:

**0 1**

**0 2**

**0 3**

Результат:

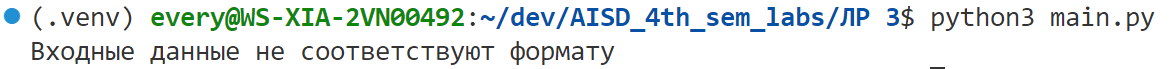


Тест 3:

Входные данные:

**London is the capital of Great Britain!**

Результат:



# Заключение

В ходе выполнения данной лабораторной работы была достигнута поставленная задача. А именно, был составлен и реализован алгоритм по сортировке пар координат расчёской, и подсчётом интеграла по заданным парам координат методом Гаусса-Лежандра с использованием структуры данных кольцевая очередь.

В ходе решения данной задачи были реализованы классы «Node» и «CircularQueue», реализующие структуру данных кольцевой очереди.

Также были реализованы функции «comb\_sort», «interpolate», «gauss\_legendre\_integration», соответственно реализующие алгоритм сортировки расчёской, интерполяцию значения y в точке x, и собственно интегрирование методом Гаусса-Лежандра.

Было выполнено тестирование программы на языке Python версии 3.13.0 с помощью среды разработки [ms-python.python](https://marketplace.visualstudio.com/items?itemName=ms-python.python) версии 2024.3.10791010, в ходе которого все реализованные выводы оказались корректно работающими.