**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 красно-черное дерево.

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 очередь. Далее пары координат сортируются в ней расческой. Результат сортировки записывается в красно-черное дерево. 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).

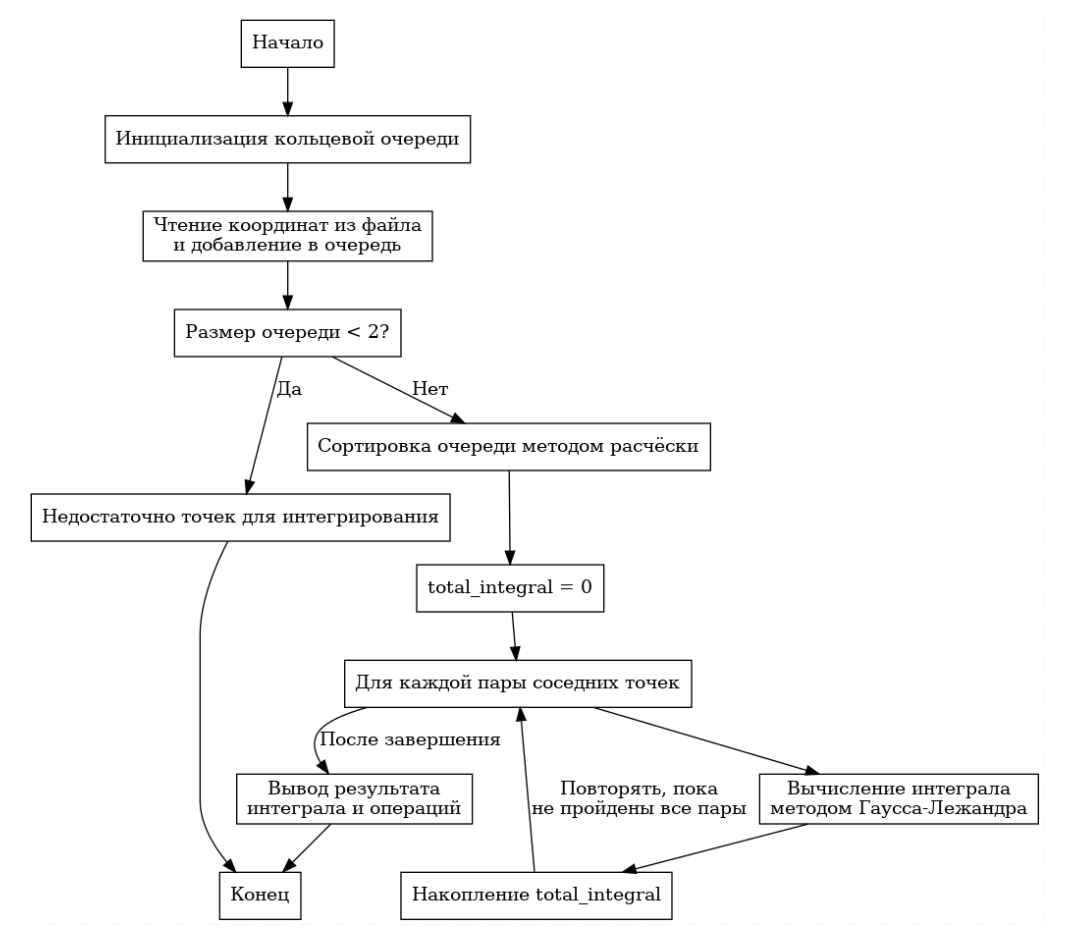
sorted – переменная типа bool.

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 Queue.

# 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 очереди на базе массива

class Queue:

    def \_\_init\_\_(self):

        self.items = []

    def enqueue(self, item):

        # Добавляем элемент в конец очереди

        self.items.append(item)

    def is\_empty(self):

        # Проверяем, пуста ли очередь

        return len(self.items) == 0

    def clear(self):

        # Очищаем очередь

        self.items = []

    def \_\_len\_\_(self):

        # Возвращаем размер очереди

        return len(self.items)

# Функция сортировки расчёской

def comb\_sort(array):

    gap = len(array)

    shrink = 1.3

    sorted = False

    while not sorted:

        gap = int(gap / shrink)

        if gap <= 1:

            gap = 1

            sorted = True

        i = 0

        while i + gap < len(array):

            if array[i][0] > array[i + gap][0]:

                array[i], array[i + gap] = array[i + gap], array[i]

                sorted = False

            i += 1

# 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

# Класс узла для красно-черного дерева

class RBNode:

    def \_\_init\_\_(self, key, value=None, color='RED', left=None, right=None, parent=None):

        self.key = key

        self.value = value  # Сохраняем значение y или пару (x, y)

        self.color = color  # 'RED' или 'BLACK'

        self.left = left if left else None

        self.right = right if right else None

        self.parent = parent

# Класс красно-черного дерева

class RedBlackTree:

    def \_\_init\_\_(self):

        self.NIL = RBNode(None, color='BLACK')

        self.root = self.NIL

    def insert(self, key, value=None):

        # Создаем новый узел

        node = RBNode(key, value)

        node.left = self.NIL

        node.right = self.NIL

        # Обычная вставка в бинарное дерево поиска

        y = None

        x = self.root

        while x != self.NIL and x.key is not None:

            y = x

            if node.key < x.key:

                x = x.left

            else:

                x = x.right

        node.parent = y

        if y is None or y == self.NIL:

            self.root = node  # Дерево было пустым

        elif node.key < y.key:

            y.left = node

        else:

            y.right = node

        # Выполняем фиксацию для поддержания свойств красно-черного дерева

        self.insert\_fixup(node)

    def insert\_fixup(self, z):

        while z.parent != None and z.parent.color == 'RED':

            if z.parent == z.parent.parent.left:

                y = z.parent.parent.right

                if y.color == 'RED':

                    # Случай 1

                    z.parent.color = 'BLACK'

                    y.color = 'BLACK'

                    z.parent.parent.color = 'RED'

                    z = z.parent.parent

                else:

                    if z == z.parent.right:

                        # Случай 2

                        z = z.parent

                        self.left\_rotate(z)

                    # Случай 3

                    z.parent.color = 'BLACK'

                    z.parent.parent.color = 'RED'

                    self.right\_rotate(z.parent.parent)

            else:

                y = z.parent.parent.left

                if y.color == 'RED':

                    # Случай 1

                    z.parent.color = 'BLACK'

                    y.color = 'BLACK'

                    z.parent.parent.color = 'RED'

                    z = z.parent.parent

                else:

                    if z == z.parent.left:

                        # Случай 2

                        z = z.parent

                        self.right\_rotate(z)

                    # Случай 3

                    z.parent.color = 'BLACK'

                    z.parent.parent.color = 'RED'

                    self.left\_rotate(z.parent.parent)

        self.root.color = 'BLACK'

    def left\_rotate(self, x):

        y = x.right

        x.right = y.left

        if y.left != self.NIL:

            y.left.parent = x

        y.parent = x.parent

        if x.parent == None or x.parent == self.NIL:

            self.root = y

        elif x == x.parent.left:

            x.parent.left = y

        else:

            x.parent.right = y

        y.left = x

        x.parent = y

    def right\_rotate(self, y):

        x = y.left

        y.left = x.right

        if x.right != self.NIL:

            x.right.parent = y

        x.parent = y.parent

        if y.parent == None or y.parent == self.NIL:

            self.root = x

        elif y == y.parent.right:

            y.parent.right = x

        else:

            y.parent.left = x

        x.right = y

        y.parent = x

    # Метод для обхода дерева в порядке (in-order)

    def inorder\_helper(self, node):

        if node != self.NIL and node.key is not None:

            self.inorder\_helper(node.left)

            print(f'Ключ: {node.key}, Значение: {node.value}, Цвет: {node.color}')

            self.inorder\_helper(node.right)

    def print\_tree(self):

        self.inorder\_helper(self.root)

    # Функция для визуализации дерева

    def visualize\_tree(self):

        lines, \*\_ = self.\_display\_aux(self.root)

        for line in lines:

            print(line)

    def \_display\_aux(self, node):

        """Возвращает список строк, высоту и ширину поддерева."""

        if node == self.NIL or node.key is None:

            return ["Н", 1, 1, 0]  # Нил узел

        line\_color = 'R' if node.color == 'RED' else 'B'

        line = f'{node.key}({line\_color})'

        # Если нет дочерних узлов

        if node.left == self.NIL and node.right == self.NIL:

            width = len(line)

            height = 1

            middle = width // 2

            return [line], height, width, middle

        # Если только правый дочерний узел

        if node.left == self.NIL:

            lines, n, p, x = self.\_display\_aux(node.right)

            s = line

            u = len(s)

            first\_line = s + ' ' \* (p)

            second\_line = ' ' \* (u) + '\\' + ' ' \* (p - 1)

            shifted\_lines = [ ' ' \* u + line for line in lines]

            return [first\_line, second\_line] + shifted\_lines, n + 2, u + p, u // 2

        # Если только левый дочерний узел

        if node.right == self.NIL:

            lines, n, p, x = self.\_display\_aux(node.left)

            s = line

            u = len(s)

            first\_line = ' ' \* (p) + s

            second\_line = ' ' \* (p - 1) + '/' + ' ' \* (u)

            shifted\_lines = [line + ' ' \* u for line in lines]

            return [first\_line, second\_line] + shifted\_lines, n + 2, p + u, p // 2

        # Если есть оба дочерних узла

        left, n, p, x = self.\_display\_aux(node.left)

        right, m, q, y = self.\_display\_aux(node.right)

        s = line

        u = len(s)

        first\_line = ' ' \* (x + 1) + ' ' \* (p - x - 1) + s + ' ' \* y + ' ' \* (q - y)

        second\_line = ' ' \* x + '/' + ' ' \* (p - x -1 + u + y) + '\\' + ' ' \* (q - y -1)

        if n < m:

            left += [' ' \* p] \* (m - n)

        elif m < n:

            right += [' ' \* q] \* (n - m)

        zipped\_lines = zip(left, right)

        lines = [first\_line, second\_line] + [a + ' ' \* u + b for a, b in zipped\_lines]

        return lines, max(n, m) + 2, p + q + u, p + u // 2

# Инициализируем очередь

coordinate\_queue = Queue()

# Параметры для проверки уникальности X

uniq\_X = []

# Открываем файл для чтения

try:

    with open('./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)

                y = float(y\_str)

                # Проверяем на уникальность X

                if x in uniq\_X:

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

                    exit()

                uniq\_X.append(x)

                # Добавляем пару (x, y) в очередь

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

            except:

                print("Входные данные не соответствуют формату")

                exit()

except FileNotFoundError:

    print("Файл 'test.txt' не найден.")

    exit()

# Выполняем сортировку расчёской

comb\_sort(coordinate\_queue.items)

# Инициализируем красно-черное дерево

rb\_tree = RedBlackTree()

# Записываем результат сортировки в красно-черное дерево

for item in coordinate\_queue.items:

    x, y = item

    rb\_tree.insert(x, y)

# Инициализируем общий интеграл

total\_integral = 0.0

# Проверяем, что в очереди достаточно элементов для интегрирования

if len(coordinate\_queue) < 2:

    print("Недостаточно точек для интегрирования.")

else:

    # Проходим по всем узлам

    for i in range(len(coordinate\_queue.items) - 1):

        # Получаем текущие точки

        x0, y0 = coordinate\_queue.items[i]

        x1, y1 = coordinate\_queue.items[i + 1]

        # Вычисляем интеграл на текущем интервале

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

        # Добавляем к общему интегралу

        total\_integral += integral

# Выводим общий интеграл

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

# Выводим содержимое красно-черного дерева

print("\nКрасно-черное дерево содержит следующие координаты (ключи):")

rb\_tree.print\_tree()

# Визуализируем дерево

print("\nВизуализация красно-черного дерева:")

rb\_tree.visualize\_tree()

# 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:

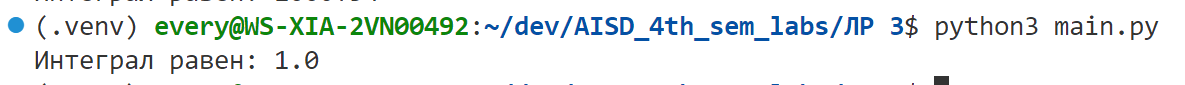
**0 0**

**0 1**

**1 1**

**1 0**

Результат:



Тест 2:

Входные данные:

**0 1**

**0 2**

**0 3**

Результат:

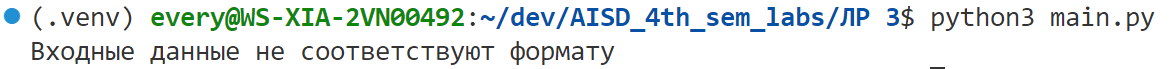


Тест 3:

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 структур данных очередь и красно-черное дерево.

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