**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).

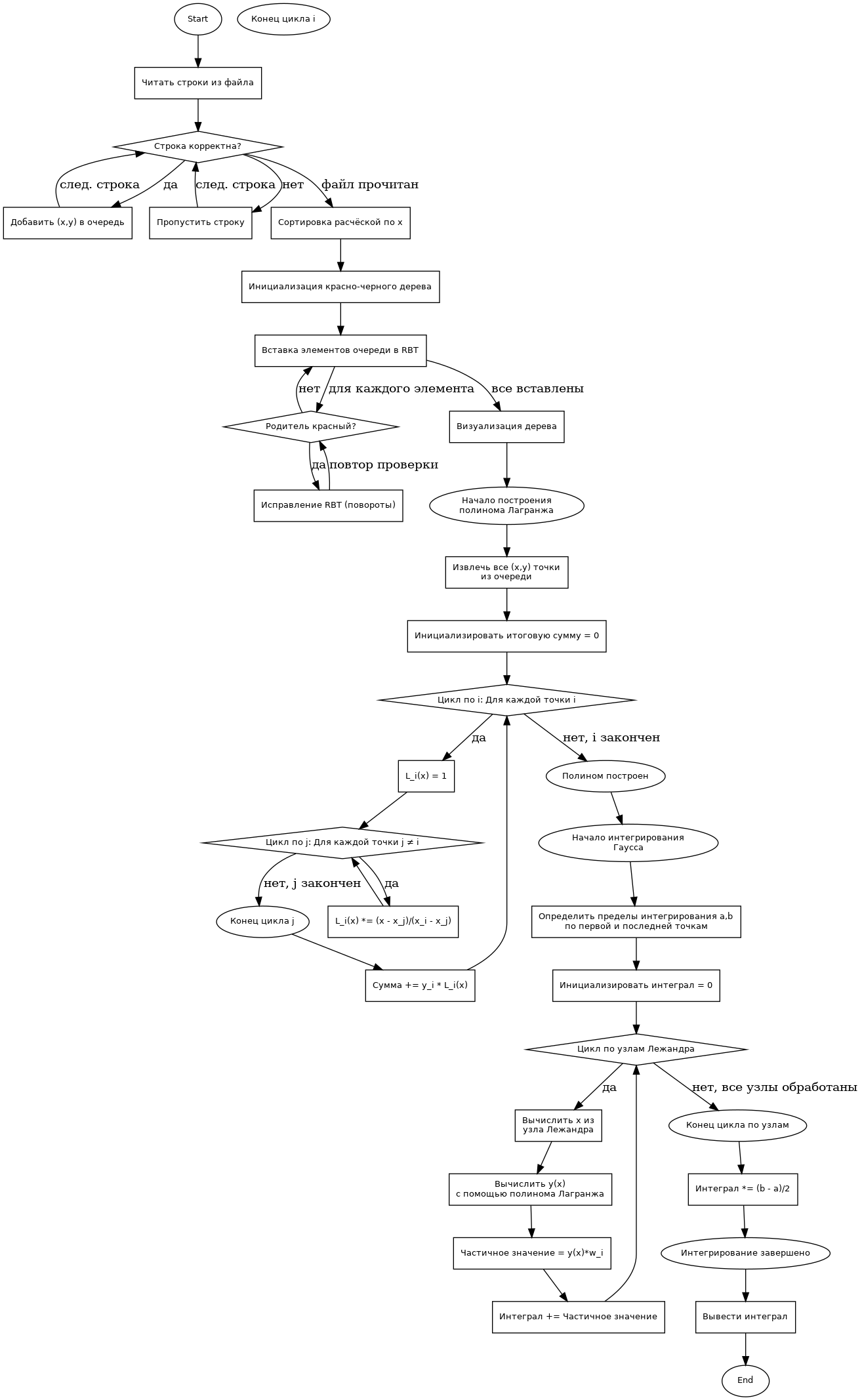
sorted – переменная типа bool.

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

Вывод функции:

1. n^2 \* (5 операций + 8 операций + 3 операции)
2. + 6n операций
3. Итого 16n^2 + 6n операций, где n – число заданных точек.

Расчет интеграла:

1. 11 операций на подсчет пределов интегрирования
2. 1 инициализация переменной Integral
3. За каждую точку Лежандра – 8 + 2
4. Обновление Integral = 4
5. Итого, для интегрирования по 2 точкам, 36 операций
6. Чтение данных из файла и заполнение очереди
7. Открыть файл, 1 операция
8. 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

from graphviz import Digraph

class Queue:

    def \_\_init\_\_(self):

        self.items = []

    def is\_empty(self):

        return len(self.items) == 0

    def push(self, item):

        self.items.append(item)  # Добавляем элемент в конец очереди

    def pop(self):

        if self.is\_empty():

            raise IndexError("Очередь пуста")

        return self.items.pop(0)  # Удаляем и возвращаем первый элемент очереди

    def size(self):

        return len(self.items)

    # Метод для отладки

    def show(self):

        for item in self.items:

            print(item)

    # Метод сортировки расчёской

    def comb\_sort(self, key\_index=0):

        gap = len(self.items)

        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(self.items):

                if self.items[i][key\_index] > self.items[i + gap][key\_index]:

                    self.items[i], self.items[i + gap] = self.items[i + gap], self.items[i]

                    sorted = False

                i += 1

# Определение узла красно-черного дерева

class RBTreeNode:

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

        self.key = key

        self.value = value

        self.color = color  # 'red' или 'black'

        self.left = left

        self.right = right

        self.parent = parent

# Красно-черное дерево

class RedBlackTree:

    def \_\_init\_\_(self):

        self.nil = RBTreeNode(key=None, value=None, color='black')  # Нил-узел (sentinel)

        self.root = self.nil

    def insert(self, key, value):

        # Создаем новый узел

        node = RBTreeNode(key, value)

        node.left = self.nil

        node.right = self.nil

        node.parent = None

        parent = None

        current = self.root

        # Поиск места для вставки

        while current != self.nil:

            parent = current

            if node.key < current.key:

                current = current.left

            else:

                current = current.right

        node.parent = parent

        if parent is None:

            self.root = node  # Дерево было пустым

        elif node.key < parent.key:

            parent.left = node

        else:

            parent.right = node

        node.color = 'red'

        self.insert\_fixup(node)

    def insert\_fixup(self, node):

        while node != self.root and node.parent.color == 'red':

            if node.parent == node.parent.parent.left:

                uncle = node.parent.parent.right

                if uncle.color == 'red':

                    # Случай 1

                    node.parent.color = 'black'

                    uncle.color = 'black'

                    node.parent.parent.color = 'red'

                    node = node.parent.parent

                else:

                    if node == node.parent.right:

                        # Случай 2

                        node = node.parent

                        self.left\_rotate(node)

                    # Случай 3

                    node.parent.color = 'black'

                    node.parent.parent.color = 'red'

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

            else:

                uncle = node.parent.parent.left

                if uncle.color == 'red':

                    # Случай 1

                    node.parent.color = 'black'

                    uncle.color = 'black'

                    node.parent.parent.color = 'red'

                    node = node.parent.parent

                else:

                    if node == node.parent.left:

                        # Случай 2

                        node = node.parent

                        self.right\_rotate(node)

                    # Случай 3

                    node.parent.color = 'black'

                    node.parent.parent.color = 'red'

                    self.left\_rotate(node.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 is None:

            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 is None:

            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 визуализации дерева с помощью Graphviz

    def visualize(self, filename='rbtree'):

        dot = Digraph()

        dot.node(name='nil', label='NIL', shape='circle', color='black')

        def add\_nodes\_edges(node):

            if node != self.nil:

                node\_name = str(id(node))

                # Создаем узел

                dot.node(name=node\_name, label=str(node.key), shape='circle',

                         style='filled', fillcolor=('red' if node.color == 'red' else 'black'),

                         fontcolor='white' if node.color == 'black' else 'black')

                # Левый ребенок

                if node.left != self.nil:

                    left\_name = str(id(node.left))

                    add\_nodes\_edges(node.left)

                else:

                    left\_name = 'nil'

                dot.edge(node\_name, left\_name, label='L')

                # Правый ребенок

                if node.right != self.nil:

                    right\_name = str(id(node.right))

                    add\_nodes\_edges(node.right)

                else:

                    right\_name = 'nil'

                dot.edge(node\_name, right\_name, label='R')

        add\_nodes\_edges(self.root)

        dot.render(filename, view=True, format='png')

# Основная часть программы

# Инициализация очереди

my\_coords = Queue()

# Считываем данные из файла и добавляем в очередь

with open('test.txt', 'r') as file:

    for line in file:

        line = line.strip()

        if not line:

            continue  # Пропускаем пустые строки

        numbers = line.split()

        if len(numbers) != 2:

            print(f"Строка '{line}' не содержит ровно два числа. Пропускаем.")

            continue

        try:

            x, y = map(int, numbers)

            my\_coords.push([x, y])

        except ValueError:

            print(f"Не удалось преобразовать строку '{line}' в числа. Пропускаем.")

LEJANDRE\_COORDINATES = {

    'n = 1': [0],

    'n = 2': [-0.5773503, 0.5773503],

    'n = 3': [-0.7745967, 0, 0.7745967],

    'n = 4': [-0.8611363, -0.3399810, 0.3399810, 0.8611363],

    'n = 5': [-0.9061798, -0.5384693, 0, 0.5384693, 0.9061798],

    'n = 6': [-0.9324700, -0.6612094, -0.2386142, 0.2386142, 0.6612094, 0.9324700]

}

LEJANDRE\_WEIGHTS = {

    'n = 1': [2],

    'n = 2': [1, 1],

    'n = 3': [0.5555556, 0.8888889, 0.5555556],

    'n = 4': [0.3478548, 0.6521451, 0.6521451, 0.3478548],

    'n = 5': [0.4786287, 0.2369269, 0.5688888, 0.2369269, 0.4786287],

    'n = 6': [0.1713245, 0.3607616, 0.4679140, 0.4679140, 0.3607616, 0.1713245]

}

# Считаем интеграл между по всей функции

def gauss(coordinates: Queue, Lejandre\_Number: str = 'n = 2') -> float:

    lower\_x\_limit : float = coordinates.items[0][0] # lower X

    higher\_x\_limit : float = coordinates.items[coordinates.size() - 1][0]  # higher X

    # print(lower\_x\_limit)

    # print(higher\_x\_limit)

    Integral : float = 0

    for lejandre\_coordinate in range(len(LEJANDRE\_COORDINATES[Lejandre\_Number])):

        interpolated\_x\_coordinate = lower\_x\_limit + ((higher\_x\_limit - lower\_x\_limit) \* (LEJANDRE\_COORDINATES[Lejandre\_Number][lejandre\_coordinate] + 1)) / 2

        interpolated\_y\_coordinate = interp\_polynomial(interpolated\_x\_coordinate)

        value : float = interpolated\_y\_coordinate \* LEJANDRE\_WEIGHTS[Lejandre\_Number][lejandre\_coordinate]

        Integral += value

    Integral \*= (higher\_x\_limit - lower\_x\_limit)/2

    print(f"\nЗначение интеграла = {Integral}")

    return Integral

def lagrange\_interpolation(coordinates : Queue):

    """

    Создаёт функцию интерполяционного полинома Лагранжа на основе заданных точек.

    Параметры:

    coordinates : Queue

        Список координат x,ys заданных точек.

    Возвращает:

    polynomial : function

        Функция, которая принимает значение x и возвращает значение интерполяционного полинома в этой точке.

    """

    # Число заданных точек

    n = coordinates.size()

    def polynomial(x):

        """

        Вычисляет значение интерполяционного полинома в точке x.

        Параметры:

        x : float

            Точка, в которой нужно вычислить полином.

        Возвращает:

        y : float

            Значение полинома в точке x.

        """

        total = 0  # Инициализация суммы полинома

        # Проходим по каждому базисному полиному L\_i(x)

        for i in range(n):

            # Инициализация базисного полинома L\_i(x)

            L\_i = 1

            # Вычисляем произведение для L\_i(x)

            for j in range(n):

                if i != j:

                    # Умножаем на каждое (x - x\_j) / (x\_i - x\_j)

                    numerator = x - coordinates.items[j][0]

                    denominator = coordinates.items[i][0] - coordinates.items[j][0]

                    L\_i \*= numerator / denominator

            # Добавляем вклад i-го базисного полинома в общую сумму

            total += coordinates.items[i][1] \* L\_i

        return total

    return polynomial

# Сортируем очередь

my\_coords.comb\_sort(key\_index=0)

# Создаем красно-черное дерево и добавляем элементы из очереди

rb\_tree = RedBlackTree()

for coord in my\_coords.items:

    rb\_tree.insert(coord[0], coord[1])

# Визуализируем дерево

rb\_tree.visualize('red\_black\_tree')

# Создаем функцию для вычисления значений y для интегрирования методом Гаусса

interp\_polynomial = lagrange\_interpolation(my\_coords)

# Задаем количество узлов

gauss(my\_coords, Lejandre\_Number = 'n = 2')

# 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:

**1 1**

**2 4**

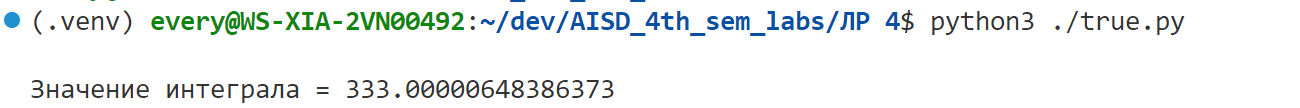
**4 16**

**5 25**

**6 36**

**10 100**

Результат:



Тест 2:

Входные данные:

**0 1**

**0 2**

**0 3**

Результат:

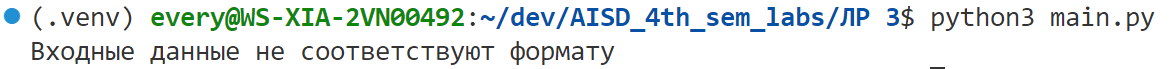


Тест 3:

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.