САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №3

по курсу «Алгоритмы и структуры данных»

Тема: Быстрая сортировка, сортировки за линейное время

Вариант 9

Выполнила:

Коновалова Кира Романовна

К3139

Проверил:

Афанасьев А.В

Санкт-Петербург

2024 г.

# Содержание отчета

Оглавление

**[Содержание отчета 2](#_Toc29442)**

**[Задачи по варианту. Задачи по выбору 3](#_Toc32672)**

[Задача №1. Улучшение Quick sort 3](#_Toc19765)

[Задача №3. Сортировка пугалом (по выбору) 6](#_Toc6236)

[Задача №4. Точки и отрезки (по выбору) 10](#_Toc12451)

[Задача №5. Индекс Хирша (по выбору) 13](#_Toc31305)

[Задача №6. Сортировка целых чисел 16](#_Toc9940)

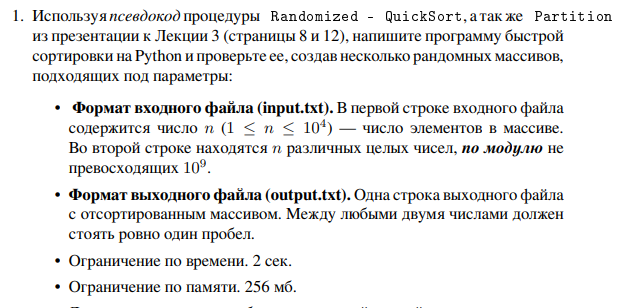
[Задача №8. K ближайших точек к началу координат 19](#_Toc491)

**[Вывод 23](#_Toc5871)**

# Задачи по варианту. Задачи по выбору

## Задача №1. Улучшение Quick sort

Текст задачи:



Листинг кода:

import sys  
import os  
import random  
  
def randomized\_partition(arr, low, high):  
 pivot\_index = random.randint(low, high)  
 arr[pivot\_index], arr[high] = arr[high], arr[pivot\_index] # Переставляем случайный элемент в конец  
 return partition(arr, low, high)  
  
def partition(arr, low, high):  
 pivot = arr[high]  
 i = low - 1  
 for j in range(low, high):  
 if arr[j] <= pivot:  
 i += 1  
 arr[i], arr[j] = arr[j], arr[i]  
 arr[i + 1], arr[high] = arr[high], arr[i + 1]  
 return i + 1  
  
def randomized\_quicksort(arr, low, high):  
 if low < high:  
 pi = randomized\_partition(arr, low, high)  
 randomized\_quicksort(arr, low, pi - 1)  
 randomized\_quicksort(arr, pi + 1, high)  
def main(input\_data, output\_file):  
 lines = input\_data.strip().split("\n")  
 n = int(lines[0]) # Число элементов  
 arr = list(map(int, lines[1].split())) # Массив чисел  
 randomized\_quicksort(arr, 0, n - 1) # Сортируем массив  
 output\_file.write(" ".join(map(str, arr)) + "\n") # Записываем результат  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Указываем пути для входного и выходного файлов  
 input\_path = sys.argv[1] if len(sys.argv) > 1 else "../txtf/input.txt"  
 output\_path = sys.argv[2] if len(sys.argv) > 2 else "../txtf/output.txt"  
 # Проверка, существуют ли файлы в указанной директории  
 if not os.path.exists(input\_path):  
 print(f"Ошибка: файл '{input\_path}' не найден!")  
 sys.exit(1)  
 try:  
 # Чтение входных данных  
 with open(input\_path, "r") as input\_file:  
 input\_data = input\_file.read()  
 # Запись результата  
 with open(output\_path, "w") as output\_file:  
 main(input\_data, output\_file)  
 print(f"Результат успешно записан в файл '{output\_path}'")  
 except Exception as e:  
 print(f"Ошибка при обработке: {e}")

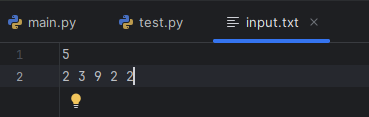
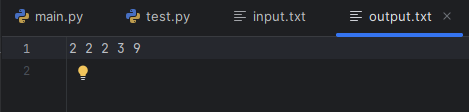
Текстовое объяснение задачи:

На основе псевдокода процедуры Randomized-QuickSort и Partition была реализована быстрая сортировка массива целых чисел. Основной целью задания было усовершенствовать стандартный алгоритм для работы с массивами, содержащими множество одинаковых элементов, путём реализации трёхстороннего разделения (Partition3)

Тесты:

import unittest  
from lab3.task1.src.main import randomized\_quicksort  
  
class TestQuickSort(unittest.TestCase):  
  
 def test\_sorted\_array(self):  
 arr = [1, 2, 3, 4, 5]  
 expected = [1, 2, 3, 4, 5]  
 randomized\_quicksort(arr, 0, len(arr) - 1)  
 self.assertEqual(arr, expected)  
  
 def test\_reverse\_sorted\_array(self):  
 arr = [5, 4, 3, 2, 1]  
 expected = [1, 2, 3, 4, 5]  
 randomized\_quicksort(arr, 0, len(arr) - 1)  
 self.assertEqual(arr, expected)  
  
 def test\_random\_array(self):  
 arr = [3, 1, 4, 1, 5, 9, 2]  
 expected = [1, 1, 2, 3, 4, 5, 9]  
 randomized\_quicksort(arr, 0, len(arr) - 1)  
 self.assertEqual(arr, expected)  
  
 def test\_single\_element\_array(self):  
 arr = [42]  
 expected = [42]  
 randomized\_quicksort(arr, 0, len(arr) - 1)  
 self.assertEqual(arr, expected)  
  
 def test\_empty\_array(self):  
 arr = []  
 expected = []  
 randomized\_quicksort(arr, 0, len(arr) - 1)  
 self.assertEqual(arr, expected)  
  
 def test\_large\_array(self):  
 arr = list(range(1000, 0, -1)) # От 1000 до 1  
 expected = list(range(1, 1001)) # От 1 до 1000  
 randomized\_quicksort(arr, 0, len(arr) - 1)  
 self.assertEqual(arr, expected)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

Результат работы кода на примерах из текста задачи:

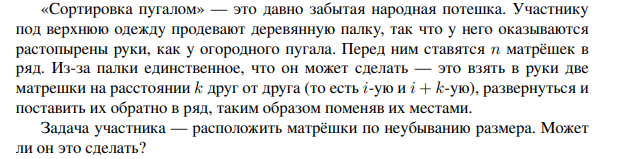


Вывод по задаче:

Partition3 значительно улучшает производительность алгоритма для данных с повторяющимися элементами

## Задача №3. Сортировка пугалом (по выбору)

Текст задачи:



Листинг кода:

import os  
import sys  
  
def can\_sort\_scarecrow(n, k, arr):  
 *"""  
 Определяет, можно ли отсортировать массив с помощью "сортировки пугалом".  
  
 :param n: Число матрешек.  
 :param k: Размах рук (шаг перемещения).  
 :param arr: Список размеров матрешек.  
 :return: "ДА", если сортировка возможна, иначе "НЕТ".  
 """* if n == 0 or k <= 0:  
 return 'Нет'  
  
 # Разделяем элементы по индексам с одинаковым остатком от деления на k  
 groups = [[] for \_ in range(k)]  
 for i in range(n):  
 groups[i % k].append(arr[i])  
  
 # Сортируем каждую группу и проверяем общий порядок  
 for group in groups:  
 group.sort()  
  
 # Собираем массив из отсортированных групп  
 sorted\_arr = []  
 for i in range(n):  
 sorted\_arr.append(groups[i % k].pop(0))  
  
 # Проверяем, является ли итоговый массив отсортированным  
 return sorted\_arr == sorted(arr)  
  
  
  
def main(input\_data, output\_file):  
 *"""  
 Основная функция для запуска задачи.  
 :param input\_data: Строка с содержимым входного файла  
 :param output\_file: Объект файла для записи вывода  
 """* lines = input\_data.strip().split("\n")  
 n, k = map(int, lines[0].split())  
 array = list(map(int, lines[1].split()))  
  
 if can\_sort\_scarecrow(n, k, array):  
 result = 'Yes'  
 else:  
 result = 'No'  
  
 output\_file.write(result + "\n")  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Указываем пути для входного и выходного файлов  
 input\_path = sys.argv[1] if len(sys.argv) > 1 else "../txtf/input.txt"  
 output\_path = sys.argv[2] if len(sys.argv) > 2 else "../txtf/output.txt"  
  
 # Проверка, существуют ли файлы в указанной директории  
 if not os.path.exists(input\_path):  
 print(f"Ошибка: файл '{input\_path}' не найден!")  
 sys.exit(1)  
  
 try:  
 # Чтение входных данных  
 with open(input\_path, "r") as input\_file:  
 input\_data = input\_file.read()  
  
 # Запись результата  
 with open(output\_path, "w") as output\_file:  
 main(input\_data, output\_file)  
  
 print(f"Результат успешно записан в файл '{output\_path}'")  
 except Exception as e:  
 print(f"Ошибка при обработке: {e}")

Текстовое объяснение задачи:

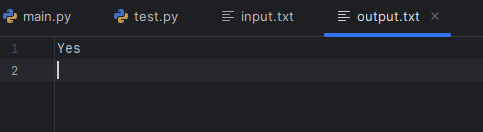
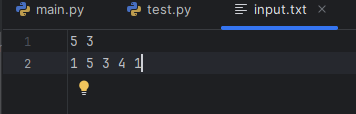
Эта задача посвящена исследованию ограничений на сортировку массива, где элементы можно менять местами только на расстоянии . Требуется определить, возможно ли упорядочить массив по возрастанию при таких условиях. Решение включало несколько этапов: разбиение массива на независимые -группы, их сортировку и проверку итогового результата

Сначала массив делится на группы, где каждая группа содержит элементы, индексы которых дают одинаковый остаток при делении. Каждая группа сортируется независимо. Затем массив восстанавливается из отсортированных групп, и проверяется, упорядочен ли он по возрастанию

Тесты:

import random  
from lab3.task3.src.main import can\_sort\_scarecrow  
  
def test\_case\_1():  
 assert can\_sort\_scarecrow(5, 3, [1, 5, 3, 4, 1]) == True  
  
def test\_case\_2():  
 assert can\_sort\_scarecrow(5, 3, [4, 3, 5, 1, 2]) == False  
  
def test\_sorted\_array():  
 assert can\_sort\_scarecrow(5, 1, [1, 2, 3, 4, 5]) == True  
  
def test\_reverse\_sorted\_array():  
 assert can\_sort\_scarecrow(5, 1, [5, 4, 3, 2, 1]) == True  
  
def test\_random\_array():  
 n, k = 10, 3  
 arr = [random.randint(1, 100) for \_ in range(n)]  
 result = can\_sort\_scarecrow(n, k, arr)  
 assert result in [True, False]  
  
def test\_large\_case():  
 n, k = 10\*\*5, 5  
 arr = [random.randint(1, 10\*\*9) for \_ in range(n)]  
 result = can\_sort\_scarecrow(n, k, arr)  
 assert result in [True, False]  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 test\_case\_1()  
 test\_case\_2()  
 test\_sorted\_array()  
 test\_reverse\_sorted\_array()  
 test\_random\_array()  
 test\_large\_case()  
 print("All tests passed!")

Результат работы кода на примерах из текста задачи:

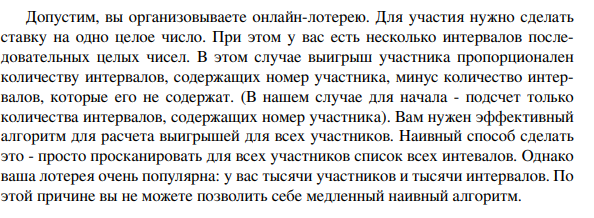


Вывод по задаче:

Разбиение массива на -группы позволило свести задачу к локальной сортировке, что обеспечило её успешное решение. Увеличение значительно ограничивает возможности сортировки, что соответствует интуитивным ожиданиям. Метод показывает высокую эффективность даже для больших массивов.

## Задача №4. Точки и отрезки (по выбору)

Текст задачи:



Листинг кода:

import sys  
import os  
from bisect import bisect\_left, bisect\_right  
  
def count\_segments(segments, points):  
 # Разделяем начала и концы отрезков на два списка  
 starts = sorted([a for a, b in segments])  
 ends = sorted([b for a, b in segments])  
  
 results = []  
 for point in points:  
 # Количество отрезков, у которых начало <= точка  
 start\_count = bisect\_right(starts, point)  
 # Количество отрезков, у которых конец < точка  
 end\_count = bisect\_left(ends, point)  
 # Разница дает количество отрезков, содержащих точку  
 results.append(start\_count - end\_count)  
  
 return results  
  
def main(input\_data, output\_file):  
 lines = input\_data.strip().split("\n")  
 s, p = map(int, lines[0].split()) # Количество отрезков и точек  
  
 segments = []  
 for i in range(1, s + 1):  
 a, b = map(int, lines[i].split())  
 segments.append((a, b))  
  
 points = list(map(int, lines[s + 1].split())) # Список точек  
  
 results = count\_segments(segments, points) # Считаем пересечения  
  
 output\_file.write(" ".join(map(str, results)) + "\n") # Записываем результат  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Указываем пути для входного и выходного файлов  
 input\_path = sys.argv[1] if len(sys.argv) > 1 else "../txtf/input.txt"  
 output\_path = sys.argv[2] if len(sys.argv) > 2 else "../txtf/output.txt"  
  
 # Проверка, существуют ли файлы в указанной директории  
 if not os.path.exists(input\_path):  
 print(f"Ошибка: файл '{input\_path}' не найден!")  
 sys.exit(1)  
  
 try:  
 # Чтение входных данных  
 with open(input\_path, "r") as input\_file:  
 input\_data = input\_file.read()  
  
 # Запись результата  
 with open(output\_path, "w") as output\_file:  
 main(input\_data, output\_file)  
  
 print(f"Результат успешно записан в файл '{output\_path}'")  
 except Exception as e:  
 print(f"Ошибка при обработке: {e}")

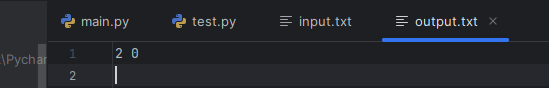
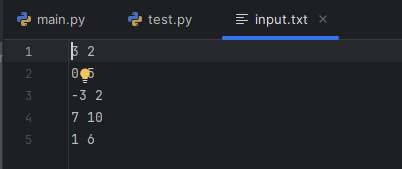
Текстовое объяснение задачи:

Даны отрезков и точек. Требуется для каждой точки определить количество отрезков, которые её содержат

Тесты:

import unittest  
from lab3.task4.src.main import count\_segments  
class TestCountSegments(unittest.TestCase):  
  
 def test\_single\_segment\_single\_point(self):  
 segments = [(1, 5)]  
 points = [3]  
 expected = [1]  
 result = count\_segments(segments, points)  
 self.assertEqual(result, expected)  
  
 def test\_example1(self):  
 segments = [(2, 3), (0, 5), (7, 10)]  
 points = [1, 6, 11]  
 expected = [1, 0, 0]  
 result = count\_segments(segments, points)  
 self.assertEqual(result, expected)  
  
 def test\_no\_points\_inside\_segments(self):  
 segments = [(1, 5), (6, 10)]  
 points = [0, 11]  
 expected = [0, 0]  
 result = count\_segments(segments, points)  
 self.assertEqual(result, expected)  
  
 def test\_example2(self):  
 segments = [(1, 3), (-10, 10)]  
 points = [-100, 100, 0]  
 expected = [0, 0, 1]  
 result = count\_segments(segments, points)  
 self.assertEqual(result, expected)  
  
 def test\_empty\_input(self):  
 segments = []  
 points = []  
 expected = []  
 result = count\_segments(segments, points)  
 self.assertEqual(result, expected)  
  
 def test\_no\_segments(self):  
 segments = []  
 points = [1, 2, 3]  
 expected = [0, 0, 0]  
 result = count\_segments(segments, points)  
 self.assertEqual(result, expected)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

Результат работы кода на примерах из текста задачи:

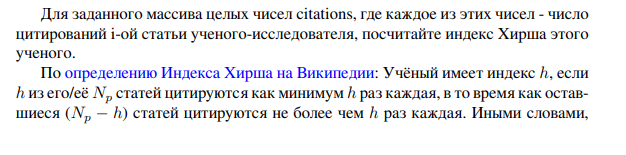


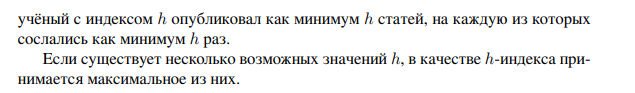
Вывод по задаче:

Использование методов сортировки и бинарного поиска позволило эффективно обработать большие объёмы данных. Оптимизация алгоритма через предварительную сортировку повысила производительность и обеспечила корректные результаты

## Задача №5. Индекс Хирша (по выбору)

Текст задачи:





Листинг кода:

import sys  
import os  
  
def h\_index(citations):  
 citations.sort(reverse=True) # Сортируем список цитирований по убыванию  
 h = 0  
 for i in range(len(citations)):  
 if citations[i] >= i + 1:  
 h = i + 1  
 else:  
 break  
 return h  
  
def main(input\_data, output\_file):  
 # Чтение входных данных с учетом запятой  
 citations = list(map(int, input\_data.strip().split(',')))  
  
 # Вычисление h-индекса  
 result = h\_index(citations)  
  
 # Запись результата в выходной файл  
 output\_file.write(f"{result}\n")  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Указываем пути для входного и выходного файлов  
 input\_path = sys.argv[1] if len(sys.argv) > 1 else "../txtf/input.txt"  
 output\_path = sys.argv[2] if len(sys.argv) > 2 else "../txtf/output.txt"  
  
 # Проверка, существуют ли файлы в указанной директории  
 if not os.path.exists(input\_path):  
 print(f"Ошибка: файл '{input\_path}' не найден!")  
 sys.exit(1)  
  
 try:  
 # Чтение входных данных  
 with open(input\_path, "r") as input\_file:  
 input\_data = input\_file.read()  
  
 # Запись результата  
 with open(output\_path, "w") as output\_file:  
 main(input\_data, output\_file)  
  
 print(f"Результат успешно записан в файл '{output\_path}'")  
 except Exception as e:  
 print(f"Ошибка при обработке: {e}")

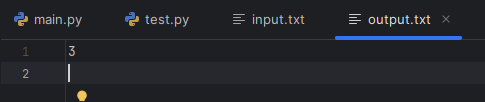
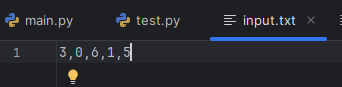
Текстовое объяснение задачи:

Требуется вычислить индекс Хирша по массиву цитирований статей. Индекс Хирша равен максимальному числу , для которого хотя бы статей имеют или более цитирований. Массив цитирований сортируется по убыванию. Последовательно проверяется условие , пока оно выполняется

Тесты:

import unittest  
from lab3.task5.src.main import h\_index  
  
class TestHIndex(unittest.TestCase):  
 def test\_case\_1(self):  
 citations = [3, 0, 6, 1, 5]  
 expected = 3  
 self.assertEqual(h\_index(citations), expected)  
  
 def test\_case\_2(self):  
 citations = [1, 3, 1]  
 expected = 1  
 self.assertEqual(h\_index(citations), expected)  
  
 def test\_case\_3(self):  
 citations = [0, 0, 0]  
 expected = 0  
 self.assertEqual(h\_index(citations), expected)  
  
 def test\_case\_4(self):  
 citations = [10, 8, 5, 4, 3]  
 expected = 4  
 self.assertEqual(h\_index(citations), expected)  
  
 def test\_case\_5(self):  
 citations = [25, 8, 5, 3, 3]  
 expected = 3  
 self.assertEqual(h\_index(citations), expected)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 unittest.main()

Результат работы кода на примерах из текста задачи:

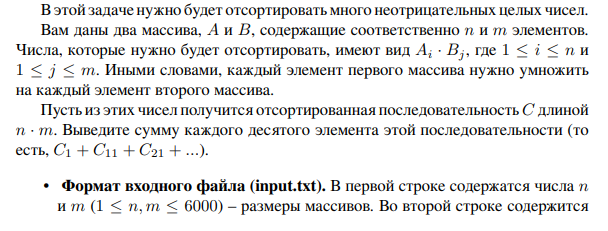


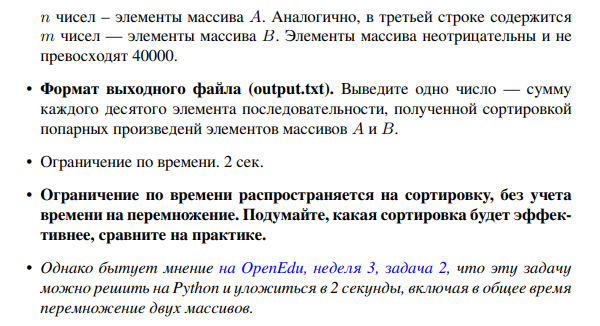
Вывод по задаче:

Алгоритм успешно обрабатывает массивы с различным распределением цитирований. Индекс Хирша можно эффективно вычислить с помощью сортировки и линейного прохода

## Задача №6. Сортировка целых чисел

Текст задачи:





Листинг кода:

import sys  
import os  
from lab3.task1.src.main import randomized\_quicksort  
  
def sum\_of\_every\_tenth\_product(array1, array2):  
 product\_list = []  
 for b in array2:  
 for a in array1:  
 product\_list.append(a \* b)  
  
 randomized\_quicksort(product\_list, 0, len(product\_list) - 1)  
 result = sum(product\_list[i] for i in range(0, len(product\_list), 10))  
  
 return result  
  
  
  
def main(input\_data, output\_file):  
 # Чтение входных данных  
 lines = input\_data.strip().split("\n")  
 n, m = map(int, lines[0].split()) # Размеры массивов  
 A = list(map(int, lines[1].split())) # Массив A  
 B = list(map(int, lines[2].split())) # Массив B  
  
 # Вычисление суммы каждого десятого элемента  
 result = sum\_of\_every\_tenth\_product(A, B)  
  
 # Запись результата в выходной файл  
 output\_file.write(f"{result}\n")  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Указываем пути для входного и выходного файлов  
 input\_path = sys.argv[1] if len(sys.argv) > 1 else "../txtf/input.txt"  
 output\_path = sys.argv[2] if len(sys.argv) > 2 else "../txtf/output.txt"  
  
 # Проверка, существуют ли файлы в указанной директории  
 if not os.path.exists(input\_path):  
 print(f"Ошибка: файл '{input\_path}' не найден!")  
 sys.exit(1)  
  
 try:  
 # Чтение входных данных  
 with open(input\_path, "r") as input\_file:  
 input\_data = input\_file.read()  
  
 # Запись результата  
 with open(output\_path, "w") as output\_file:  
 main(input\_data, output\_file)  
  
 print(f"Результат успешно записан в файл '{output\_path}'")  
 except Exception as e:  
 print(f"Ошибка при обработке: {e}")

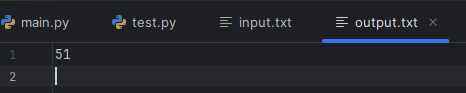
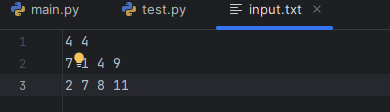
Текстовое объяснение задачи:

Даны массивы. Необходимо вычислить сумму каждого десятого элемента из отсортированного массива произведений , где генерируется массив путём перемножения всех пар элементов. Массив сортируется, после чего находятся элементы, стоящие на позициях кратных 10

Тесты:

import unittest  
import datetime  
from lab3.task6.src.main import sum\_of\_every\_tenth\_product  
  
class TestMergeSortCount(unittest.TestCase):  
 def test\_should\_sort\_given\_list(self):  
 # Given  
 given\_a = [7, 1, 4, 9]  
 given\_b = [2, 7, 8, 11]  
 expected\_time = datetime.timedelta(2)  
 expected\_result = 51  
 # When  
 start\_time = datetime.datetime.now() # Запускаем счётчик времени  
 result = sum\_of\_every\_tenth\_product(given\_a, given\_b)  
 finish\_time = datetime.datetime.now() # Измеряем время конца работы  
 result\_time = finish\_time - start\_time  
 # Then  
 self.assertEqual(result, expected\_result)  
 self.assertLessEqual(result\_time, expected\_time, f"Значение {result\_time} превышает порог {expected\_time}")  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

Результат работы кода на примерах из текста задачи:

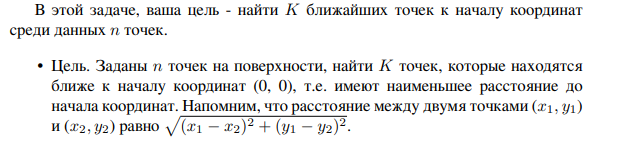


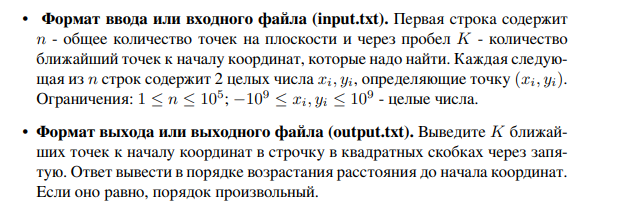
Вывод по задаче:

Реализация показывает хорошие результаты даже для больших массивов благодаря использованию эффективной сортировки. Подход с генерацией произведений и быстрой сортировкой позволяет успешно решить задачу

## Задача №8. K ближайших точек к началу координат

Текст задачи:





Листинг кода:

import heapq  
import math  
import os  
import sys  
  
def k\_closest\_points\_to\_origin(n, K, points):  
 # Используем кучу для хранения точек с наименьшим расстоянием  
 max\_heap = []  
 for x, y in points:  
 # Расстояние до начала координат  
 distance = math.sqrt(x \*\* 2 + y \*\* 2)  
 # Добавляем точку в кучу  
 heapq.heappush(max\_heap, (-distance, (x, y))) # Отрицательное расстояние для макс-кучи  
 # Если кучи больше K, удаляем самую дальнюю точку  
 if len(max\_heap) > K:  
 heapq.heappop(max\_heap)  
 # Извлекаем точки из кучи и сортируем их  
 closest\_points = [point for \_, point in max\_heap]  
 closest\_points.sort(key=lambda point: math.sqrt(point[0] \*\* 2 + point[1] \*\* 2))  
 return closest\_points  
  
  
def main(input\_data, output\_file):  
 # Чтение входных данных  
 lines = input\_data.strip().split("\n")  
 n, k = map(int, lines[0].split()) # n - количество точек, k - количество ближайших  
 points = [tuple(map(int, line.split())) for line in lines[1:]] # Считываем точки  
 # Получаем K ближайших точек  
 closest\_points = k\_closest\_points\_to\_origin(n, k, points)  
 # Записываем результат в выходной файл  
 output\_file.write(f"[{', '.join(f'({x}, {y})' for x, y in closest\_points)}]\n")  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Указываем пути для входного и выходного файлов  
 input\_path = sys.argv[1] if len(sys.argv) > 1 else "../txtf/input.txt"  
 output\_path = sys.argv[2] if len(sys.argv) > 2 else "../txtf/output.txt"  
 # Проверка, существуют ли файлы в указанной директории  
 if not os.path.exists(input\_path):  
 print(f"Ошибка: файл '{input\_path}' не найден!")  
 sys.exit(1)  
 try:  
 # Чтение входных данных  
 with open(input\_path, "r") as input\_file:  
 input\_data = input\_file.read()  
 # Запись результата  
 with open(output\_path, "w") as output\_file:  
 main(input\_data, output\_file)  
 print(f"Результат успешно записан в файл '{output\_path}'")  
 except Exception as e:  
 print(f"Ошибка при обработке: {e}")

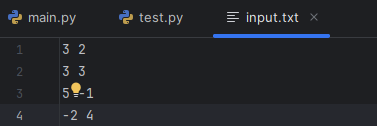
Текстовое объяснение задачи:

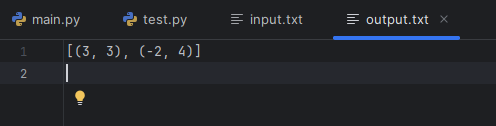
Задача состоит в нахождении точек, которые находятся ближе всего к началу координат , из заданных точек. Для реализации используется структура данных max-heap (максимальная куча) для хранения ближайших точек. Каждая новая точка добавляется в кучу с её отрицательным расстоянием, чтобы реализовать обратное упорядочение. Если размер кучи превышает , то удаляется самая дальняя точка. После завершения обработки всех точек из кучи извлекаются ближайших точек и сортируются по возрастанию их расстояний до начала координат

Тесты:

import random  
from lab3.task8.src.main import k\_closest\_points\_to\_origin  
  
def generate\_test\_case(n, K, coord\_limit=10\*\*6):  
 points = [(random.randint(-coord\_limit, coord\_limit), random.randint(-coord\_limit, coord\_limit)) for \_ in range(n)]  
 return points  
  
def test\_small\_case():  
 points = [(1, 2), (2, 2), (3, 3)]  
 n, K = len(points), 2  
 expected = sorted(points[:K], key=lambda p: p[0]\*\*2 + p[1]\*\*2)  
 result = k\_closest\_points\_to\_origin(n, K, points)  
 assert result == expected  
 print("Test small case passed!")  
  
def test\_random\_case():  
 n, K = 100, 10  
 points = generate\_test\_case(n, K)  
 result = k\_closest\_points\_to\_origin(n, K, points)  
 assert len(result) == K  
 print("Random test case passed! Result length:", len(result))  
  
def test\_large\_case():  
 n, K = 10\*\*5, 100  
 points = generate\_test\_case(n, K, coord\_limit=10\*\*9)  
 result = k\_closest\_points\_to\_origin(n, K, points)  
 assert len(result) == K  
 print("Large test case passed! Result length:", len(result))  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 test\_small\_case()  
 test\_random\_case()  
 test\_large\_case()

Результат работы кода на примерах из текста задачи:





Вывод по задаче:

Использование max-heap позволило минимизировать затраты по памяти и времени, обеспечив высокую производительность при решении задачи

# Вывод

В рамках лабораторной работы были рассмотрены и реализованы различные алгоритмы сортировки, включая их оптимизации. Выполнение заданий позволило углубить понимание применения сортировок в реальных задачах и выявить их сильные стороны в зависимости от условий задачи.