Lab 4

Problem 1

You are given the following nearly sorted list of integers:

```
[10, 9, 8, 7, 6, 4, 5, 2, 1]
```

Your task is to **track the number of recursion calls** made during the partitioning steps of **Quick Sort** using different **pivot selection strategies** (first element, last element, and median-of-three). Compare how each pivot selection strategy impacts the resulting subarrays, the number of partition steps, and explain why one strategy might lead to better or worse performance.

*Note: The **median-of-three** method selects the pivot as the **middle value** of the first, middle, and last elements of the array*

Problem 2

Imagine you're working for an e-commerce company, and you need to identify the **2nd lowest price** of products from an unsorted list of product prices. To achieve this, you can use the **partition step** of Quick Sort, which is often used in selecting the k-th smallest element. In this case, we'll apply the **median-of-three pivot selection method** to find the **2nd smallest price** in the list.

You are given the following unsorted list of product prices in dollars:

[12, 3, 5, 7, 19, 1, 10, 15]

- Explain how the partitioning process of Quick Sort helps identify the k-th smallest element by using the pivot's position to narrow down the search space.
- Apply the median-of-three pivot selection method to partition the array and progressively narrow down the subarray to find the 2nd smallest element.

Note: Why Use Median-of-Three Instead of Other Methods?

- Picking the first, middle, and last gives a good approximation of the true median. Only **3 comparisons** are needed, avoiding extra overhead.
- **Median-of-Four/Five** – More comparisons **(5-7)** with **minimal improvement**.
- The **average may not exist** in the array, requiring extra operations.

Problem 3

In a class, students' marks are recorded in a list, but many marks are repeated. Standard sorting algorithms like Quick Sort can become inefficient when dealing with numerous duplicates, as they result in excessive comparisons and swaps, leading to **O(n²)** time complexity in the worst case.

To address this, you are tasked with partitioning the list of marks efficiently using the **Three-Way Quick Sort** partitioning method, which reduces unnecessary comparisons by grouping duplicate marks together.

Input:

- **arr = [10, 20, 10, 5, 30, 20, 15, 5, 25, 30, 20]**

Note on Three-Way Quick Sort:

*Unlike the standard Quick Sort, which divides the list into two partitions, the **Three-Way Quick Sort** method partitions the list into three sections:*

- ***Marks less than the pivot** in the first section.*
- ***Marks equal to the pivot** in the second section.*
- ***Marks greater than the pivot** in the third section.*

Task:

1. **Explain step-by-step** how the Three-Way Quick Sort partitioning method works on the input list. **Write the final list** after applying the method.
2. How can **Three-Way Quick Sort** improve the time complexity in this case compared to **standard Quick Sort**?
3. Based on your analysis, compare how **Three-Way Quick Sort** handles duplicates in comparison to other sorting algorithms like **Merge Sort** and **Heap Sort**. Which method handles duplicates more efficiently, and why?

Problem 4

You are tasked with implementing a sorting algorithm for a **database system** that manages large numbers of **customer records** (each containing customer information such as names, ages, and addresses). The records are stored in an array, and you need to sort the records based on **customer age in ascending order**.

Scenario 1: Sorting User Records for a Fast Real-Time Application

- The database contains a **large dataset of user records**, and the system is a **real-time application** (e.g., an e-commerce platform). The data is **frequently updated**, and sorting must occur efficiently. The primary goal is to ensure the **sorting operation is as fast as possible**, with **minimal memory overhead**.

Scenario 2: Sorting Large External Files for Storage

- The database contains **very large datasets** (e.g., **terabytes of data**) stored across **external storage devices** (e.g., hard drives, cloud storage). The data is **too large to fit into memory all at once** and needs to be sorted in **chunks**. **Stability** is important, as the relative order of records with the same age must be preserved, especially when multiple sorting operations (e.g., sorting by name after sorting by age) are required.

Which algorithm (Merge Sort or Quick Sort) would you choose for sorting the records for each scenario, and why?

Problem 5

```
def heapify(arr, n, i):
    swaps = 0
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        swaps += 1
        swaps += heapify(arr, n, largest)
    return swaps
```

Manually simulate the call `heapify(arr, 5, 0)` for the array `[4, 10, 3, 5, 1]`.

- List the sequence of comparisons and swaps that occur.
- How many swaps occur in total during this call?

Problem 6

A company runs a **real-time task scheduling system** where multiple tasks are processed each minute. Each task has an associated **priority**, and the system must ensure that tasks are processed based on their urgency. New tasks arrive constantly, and the system must be able to **add**, **remove**, and **process tasks dynamically**. The system needs to decide how to efficiently manage the task queue to ensure that tasks are processed in the correct order, respecting their priorities.

Your goal is to design a system where:

1. Guarantees that each task operation (insertion, removal) happens in **O(log n)** time
2. Tasks with **higher priority** are processed first.

Tasks:

1. **Describe how you would implement an efficient task scheduling system** that processes tasks based on their priority satisfying the above system goal.
2. **Simulate a scheduling process** with the following list of tasks, ensuring tasks are processed based on their priority:
    - tasks = [(3, 'Task A'), (5, 'Task B'), (2, 'Task C'), (4, 'Task D'), (1, 'Task E')]

Hint:

*Consider using a **priority queue** implemented with a **max-heap**, where tasks are stored so that the highest-priority task is always easily accessible and processed first. This structure allows efficient addition, removal, and retrieval of tasks based on priority.*