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*

<u>Answer</u>

Partition Steps for Each Pivot Selection Method

1. First Element as Pivot:

- **Partition 1**: Pivot = 10, Subarrays = `[9, 8, 7, 6, 4, 5, 2, 1]` (left), `[]` (right)
- **Partition 2**: Pivot = 9, Subarrays = `[8, 7, 6, 4, 5, 2, 1]` (left), `[]` (right)
- **Partition 3**: Pivot = 8, Subarrays = `[7, 6, 4, 5, 2, 1]` (left), `[]` (right)
- **Partition 4**: Pivot = 7, Subarrays = `[6, 4, 5, 2, 1]` (left), `[]` (right)
- **Partition 5**: Pivot = 6, Subarrays = `[4, 5, 2, 1]` (left), `[]` (right)
- **Partition 6**: Pivot = 4, Subarrays = `[2, 1]` (left), `[5]` (right)
- **Partition 7**: Pivot = 2, Subarrays = `[1]` (left), `[]` (right)
- **Partition 8**: Pivot = 5, Subarrays = `[]` (left), `[]` (right)
- **Total Partitions**: 8

2. Last Element as Pivot:

- **Partition 1**: Pivot = 1, Subarrays = `[]` (left), `[10, 9, 8, 7, 6, 4, 5, 2]` (right)
- **Partition 2**: Pivot = 2, Subarrays = `[]` (left), `[10, 9, 8, 7, 6, 4, 5]` (right)
- **Partition 3**: Pivot = 5, Subarrays = `[4]` (left), `[10, 9, 8, 7, 6]` (right)
- **Partition 4**: Pivot = 4, Subarrays = `[]` (left), `[]` (right)
- **Partition 5**: Pivot = 6, Subarrays = `[]` (left), `[10, 9, 8, 7]` (right)
- **Partition 6**: Pivot = 7, Subarrays = `[]` (left), `[10, 9, 8]` (right)
- **Partition 7**: Pivot = 8, Subarrays = `[]` (left), `[10, 9]` (right)

- **Partition 8**: Pivot = 9, Subarrays = `[]` (left), `[10]` (right)
- **Total Partitions**: 8

3. Median-of-Three as Pivot:

- **Partition 1**: Pivot = 6 (median of 10, 6, 1), Subarrays = `[4, 5, 2, 1]` (left), `[7, 8, 9, 10]` (right)
- **Partition 2**: **Left Subarray (Pivot = 2, from median of 4, 2, 1)**, Subarrays = `[1]` (left), `[4, 5]` (right)
- **Partition 3**: **Right Subarray** (Pivot = 9, from median of 7, 9, 10), Subarrays = `[7, 8]` (left), `[10]` (right)
- **Partition 4**: Pivot = 5, Subarrays = `[4]` (left), `[]` (right)
- **Partition 5**: Pivot = 8, Subarrays = `[7]` (left), `[]` (right)
- **Total Partitions**: 5

Analysis and Comparison

1. First Element as Pivot

- **Unbalanced partition**, with almost all elements placed in the left subarray.
- The right subarray is empty, meaning Quick Sort will continue working on a large, nearly unsorted left portion.
- This leads to inefficient recursion and increases the number of comparisons, resulting in **O(n^2)** worst-case complexity when the list is already sorted or reverse-sorted.

2. Last Element as Pivot

Again, an **unbalanced partition**, but in the opposite direction—this time, all elements go to the right subarray.

- Just like the first strategy, this leads to excessive recursion depth and poor performance.
- The partitioning does not effectively reduce the problem size, resulting in **O(n^2)** worst-case complexity.

3. Median-of-Three as Pivot

- **Balanced partition**, with elements distributed relatively evenly between left and right subarrays.
- Since Quick Sort works best when partitions are balanced, this method reduces recursion depth and keeps the algorithm closer to **O(n log n)** complexity.
- This is the most efficient pivot selection strategy in this case, preventing extreme unbalanced partitions.

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.

Answer

How the Partitioning Process Helps Find the k-th Smallest Element

1. **Partitioning organizes elements into two groups**:
    - Left: Elements **smaller** than the pivot.
    - Right: Elements **greater** than the pivot.
    - The pivot itself is in its **final sorted position**.
2. **Instead of sorting the entire list, we reduce the problem size**:
    - If the pivot index = `k - 1`, we **found the k-th smallest element**.
    - If `k - 1` is in the left subarray, we **recur on the left**.
    - If `k - 1` is in the right subarray, we **recur on the right**.

Step to find 2nd smallest price element

Step 1. Selecting the Median-of-Three Pivot

The **median-of-three method** picks the pivot as the **median of the first, middle, and last elements**.

- **First element** = 12

- **Middle element** (index `len(arr) // 2 = 8 // 2 = 4`) = `19`
- **Last element** = `15`
- **Median of (12, 19, 15) = 15 → Pivot = 15**

Step 2. Partitioning Around Pivot (15)

We rearrange elements so that:

- Elements **less than** `15` go to the left.
- Elements **greater than** `15` go to the right.

Partitioning Process:

1. Compare `12` (left) → **keep (less than 15)**.
2. Compare `3` (left) → **keep (less than 15)**.
3. Compare `5` (left) → **keep (less than 15)**.
4. Compare `7` (left) → **keep (less than 15)**.
5. Compare `19` (right) → **move right (greater than 15)**.
6. Compare `1` (left) → **keep (less than 15)**.
7. Compare `10` (left) → **keep (less than 15)**.

Resulting Partitioned List: `[12, 3, 5, 7, 1, 10, 15, 19]`

Pivot `15` is placed at index 6.

Step 3. Identify the 2nd Smallest Element

- The left subarray: `[12, 3, 5, 7, 1, 10]` (elements less than `15`)
- The right subarray: `[19]` (elements greater than `15`)
- Since we need the **2nd smallest element**, we continue partitioning the **left subarray** `[12, 3, 5, 7, 1, 10]`.

Step 4. Recursive Partitioning on Left Subarray

New subarray: `[12, 3, 5, 7, 1, 10]`

Selecting the New Pivot (Median-of-Three)

- **First element** = `12`
- **Middle element** = `7`
- **Last element** = `10`
- **Median of (12, 7, 10) = 10 → New Pivot = 10**

Partitioning Around Pivot (10)

Rearrange elements so that:

- **Less than 10** → `[3, 5, 7, 1]`
- **Greater than 10** → `[12]`

Resulting Partitioned List: `[3, 5, 7, 1, 10, 12]`

Pivot `10` is placed at index 4.

Step 5. Identify the 2nd Smallest Element

- The left subarray: `[3, 5, 7, 1]` (elements less than `10`)
- Since we need the **2nd smallest element**, we continue partitioning **this subarray**.

New Pivot Selection (Median-of-Three)

- **First element** = `3`
- **Middle element** = `7`
- **Last element** = `1`
- **Median of (3, 7, 1) = 3 → Pivot = 3**

Partitioning Around Pivot (3)

Rearrange elements so that:

- **Less than 3** → `[1]`
- **Greater than 3** → `[5, 7]`

Resulting Partitioned List: `[1, 3, 5, 7]`

Pivot `3` is placed at index 1.

Step 6. Find the 2nd Smallest

- **The 2nd smallest element is at index 1 → "3".**

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(n^2)** 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?

<u>Answer</u>

Step-by-Step Explanation of Three-Way Quick Sort Partitioning:

Input list:
`arr = [10, 20, 10, 5, 30, 20, 15, 5, 25, 30, 20]`

1. **Choose the pivot**:
   We select **20** as the pivot. The goal is to partition the list into three sections:
   - Marks less than the pivot.
   - Marks equal to the pivot.
   - Marks greater than the pivot.
2. **Partitioning Process**:
   - We initialize three pointers:
     - `low` (start of the list).
     - `mid` (traverses through the list).
     - `high` (end of the list).
   - **Step-by-step partitioning**:

        - `arr[mid] = 10` is less than 20. We swap `arr[mid]` and `arr[low]`, then increment both `low` and `mid`.
        - `arr[mid] = 20` is equal to 20. We just increment `mid`.
        - `arr[mid] = 10` is less than 20. We swap `arr[mid]` and `arr[low]`, then increment both `low` and `mid`.
        - `arr[mid] = 5` is less than 20. We swap `arr[mid]` and `arr[low]`, then increment both `low` and `mid`.
        - `arr[mid] = 30` is greater than 20. We swap `arr[mid]` and `arr[high]`, then decrement `high`.
        - `arr[mid] = 20` is equal to 20. We just increment `mid`.
        - This process continues until the array is fully partitioned, with elements less than 20 in the first section, equal to 20 in the second, and greater than 20 in the third.

3. **Final Partitioned List**: After partitioning, the final array will look like this:

    - **Marks less than 20**: `[10, 10, 5, 5, 15]`
    - **Marks equal to 20**: `[20, 20, 20]`
    - **Marks greater than 20**: `[30, 30, 25]`
4. **Partitioned list**: `[10, 10, 5, 5, 15, 20, 20, 20, 30, 30, 25]`
5. Now, Quick Sort will recursively sort the **lower part** (elements less than 20) and the **higher part** (elements greater than 20).
6. Sorting the Lower Part `[10, 10, 5, 5, 15]` with pivot 10 will result in `[5, 5, 10, 10, 15]`
7. Sorting the Higher Part `[30, 30, 25]` with pivot 30 will result in `[25, 30, 30]`
8. **Final sorted list:** `[5, 5, 10, 10, 15, 20, 20, 20, 25, 30, 30]`

Time Complexity of Three-Way Quick Sort:

- In this case, **standard Quick Sort** can degrade to **O(n²)** when many duplicates exist, as it repeatedly partitions the same elements without meaningful reductions. This happens because standard Quick Sort partitions into only two sections, leading to excessive swaps and comparisons.
- **Three-Way Quick Sort**, however, efficiently handles duplicates by grouping all equal elements together in one pass. This reduces the number of recursive calls, leading to improved performance, ensuring the average time complexity remains at **O(n log n)**

Comparison with Merge Sort and Heap Sort:

1. **Merge Sort**:

    - **Time Complexity**: Merge Sort has a time complexity of **O(n log n)**, even in the worst case, as it divides the array into two halves and recursively merges them. It does not handle duplicates any differently than other elements, but the algorithm will still perform **O(n log n)** comparisons and merge steps, even when duplicates are present.

    - **Handling Duplicates**: Merge Sort does not optimize for duplicates. It will still perform comparisons for each duplicate element during the merge phase, making it less efficient when there are many duplicates compared to Three-Way Quick Sort.
2. **Heap Sort**:

    - **Time Complexity**: Heap Sort also has a time complexity of **O(n log n)** in all cases. It builds a heap and then repeatedly extracts the maximum (or minimum) element.
    - **Handling Duplicates**: Heap Sort does not have a specific optimization for handling duplicates. While it performs **O(log n)** operations for each element during the heapification process, it does not minimize the comparisons for duplicate elements as Three-Way Quick Sort does.
3. **Three-Way Quick Sort**:

    - Three-Way Quick Sort excels in scenarios with many duplicate values, such as sorting student exam scores, product ratings. In large datasets, many students may have the same score, numerous products may share identical ratings. By grouping duplicates in a single pass, this method reduces redundant comparisons and swaps, minimizing recursive calls. This can achieve **O(n)** in the best case. Compared to Merge Sort and Heap Sort, it handles duplicates more efficiently, making it ideal for datasets with frequent repeated values.

    Conclusion: **Three-Way Quick Sort** is the most efficient in handling duplicates, especially when compared to **Merge Sort** and **Heap Sort**, as it specifically optimizes for duplicates by grouping equal elements together, reducing unnecessary work and comparisons.

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?

Answer

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

- For a real-time application that requires fast sorting with minimal memory overhead, **Quick Sort** is the preferred algorithm. Quick Sort has an average-case time complexity of **O(n log n)** and is generally faster than Merge Sort for in-memory sorting due to lower memory usage. Since the data is frequently updated, Quick Sort's in-place nature is beneficial, reducing memory overhead compared to Merge Sort, which requires additional space for merging.

Scenario 2: Sorting Large External Files for Storage

- For large datasets that cannot fit into memory and require external sorting, **Merge Sort** is the better choice. Merge Sort has a consistent **O(n log n)** time complexity and is a **stable sorting algorithm**, preserving the relative order of records with the same key (age). This is particularly useful when secondary sorting operations, such as sorting by name after sorting by age, are required.
- Since Merge Sort processes data in **chunks (divide-and-conquer approach)**, it is well-suited for **external sorting**, where data is read from and written to storage efficiently. The **External Merge Sort** variant is commonly used in database systems to handle massive datasets, as it allows sorting large files efficiently by breaking them into smaller sorted segments and merging them iteratively.

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?

Answer

Step-by-Step Execution of `heapify(arr, 5, 0)`

1. **Initial Setup**
   - **Root (`i = 0`) = 4**
   - **Left child (`left = 2 * 0 + 1 = 1`) = 10**
   - **Right child (`right = 2 * 0 + 2 = 2`) = 3**
2. **First Comparison**
   - Compare `arr[left] = 10` with `arr[largest] = 4`
   - **10 > 4**, so `largest = 1`
3. **Second Comparison**
   - Compare `arr[right] = 3` with `arr[largest] = 10`
   - **3 < 10**, so `largest` remains 1
4. **First Swap**
   - `arr[i]` (4) **swaps with** `arr[largest]` (10)
   - Updated array: `[10, 4, 3, 5, 1]`
   - **Total swaps = 1**

Recursive Call: `heapify(arr, 5, 1)`

1. **New Root (`i = 1`) = 4**
   - **Left child (`left = 2 * 1 + 1 = 3`) = 5**
   - **Right child (`right = 2 * 1 + 2 = 4`) = 1**
2. **Comparison**
   - Compare `arr[left] = 5` with `arr[largest] = 4`
   - **5 > 4**, so `largest = 3`
3. **Swap**
   - `arr[i]` (4) **swaps with** `arr[largest]` (5)
   - Updated array: `[10, 5, 3, 4, 1]`
   - **Total swaps = 2**
4. **Recursive Call: `heapify(arr, 5, 3)`**
   - `i = 3` has **no children**, so recursion stops.

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.*

Answer

Designing an Efficient Task Scheduling System

To efficiently manage task scheduling where tasks with higher priority are processed first and operations (insertion, removal) occur in **O(log n)** time, we can use a **Priority Queue** implemented with a **Binary Heap (Max-Heap)**. A Max-Heap ensures that the highest-priority task (largest priority value) is always at the top and can be efficiently extracted in **O(log n)** time.

Implementation Details

- In a **max heap**, tasks are organized such that for every parent node, its priority is **greater than or equal** to the priorities of its child nodes.
- The heap structure ensures that the highest-priority task is always at the root.
- Inserting a new task takes **O(log n)** time as the heap maintains its order by bubbling up the new task.
- Removing (processing) the highest-priority task also takes **O(log n)** by swapping it with the last element and re-heapifying.

Heapify-Up (Insertion Process)

When inserting a new task `(priority, task_name)`:

1. **Insert the task at the end** of the heap (to maintain the complete binary tree property).
2. **Compare the task with its parent**: If it has a higher priority, **swap it with the parent**.
3. **Repeat the process** until the task is in its correct position (either at the root or when its parent has a higher priority).
4. **Time Complexity: O(log n)** since at most the height of the heap needs adjustments.

Heapify-Down (Processing/Removal Process)

When removing the highest-priority task (root node):

1. **Replace the root** with the last element in the heap.
2. **Compare the new root with its children**: Swap it with the highest-priority child if necessary.
3. **Repeat until the heap property is restored**, meaning the parent is always greater than both children.
4. **Time Complexity: O(log n)** since the tree is restructured at most along its height.

Simulating Task Scheduling: Given tasks:
`tasks = [(3, 'Task A'), (5, 'Task B'), (2, 'Task C'), (4, 'Task D'), (1, 'Task E')]`

We insert them into a **Max-Heap**, where each task is represented as `(priority, task_name)`. The heap maintains the order dynamically.

Heap Construction (Insertion Process)

1. Insert **(3, 'Task A')** → Heap: `[(3, 'Task A')]`
2. Insert **(5, 'Task B')** → Heap: `[(5, 'Task B'), (3, 'Task A')]`
3. Insert **(2, 'Task C')** → Heap: `[(5, 'Task B'), (3, 'Task A'), (2, 'Task C')]`
4. Insert **(4, 'Task D')** → Heap: `[(5, 'Task B'), (4, 'Task D'), (2, 'Task C'), (3, 'Task A')]`
5. Insert **(1, 'Task E')** → Heap: `[(5, 'Task B'), (4, 'Task D'), (2, 'Task C'), (3, 'Task A'), (1, 'Task E')]`

Processing Tasks in Priority Order

Using **heap extraction**, we process tasks in the order of highest priority:

1. **Extract (5, 'Task B')** → Heap after removal: `[(4, 'Task D'), (3, 'Task A'), (2, 'Task C'), (1, 'Task E')]`
2. **Extract (4, 'Task D')** → Heap after removal: `[(3, 'Task A'), (1, 'Task E'), (2, 'Task C')]`

3. **Extract (3, 'Task A')** → Heap after removal: `[(2, 'Task C'), (1, 'Task E')]`
4. **Extract (2, 'Task C')** → Heap after removal: `[(1, 'Task E')]`
5. **Extract (1, 'Task E')** → Heap is now empty.

Final Processing Order: Task B → Task D → Task A → Task C → Task E