Merge Sort, Quick Sort, and Heap Sort Case Study in Python

1. Introduction

Sorting algorithms are essential in computer science for organizing data efficiently. Three widely used sorting algorithms are Merge Sort, Quick Sort, and Heap Sort. This document provides an in-depth study of these algorithms, including their implementations in Python and case studies.

---

2. Merge Sort

Concept:
- Merge Sort is a divide-and-conquer algorithm.
- It splits the array into two halves, recursively sorts them, and merges the sorted halves.

Python Implementation:

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
```

Case Study:

Scenario: Sorting student scores.
- Input: `[85, 42, 67, 90, 32, 76]`
- Output: `[32, 42, 67, 76, 85, 90]`

---

3. Quick Sort
Concept:
- Quick Sort is a divide-and-conquer algorithm.
- It selects a pivot element, partitions the array, and sorts the partitions recursively.

Python Implementation:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        left = [x for x in arr[1:] if x <= pivot]
        right = [x for x in arr[1:] if x > pivot]
        return quick_sort(left) + [pivot] + quick_sort(right)
```

Case Study:
Scenario: Sorting book prices.
- Input: `[200, 150, 180, 120, 90, 300]`
- Output: `[90, 120, 150, 180, 200, 300]`

---

4. Heap Sort
Concept:
- Heap Sort uses a binary heap data structure.
- It first builds a max heap and then extracts the maximum element repeatedly.

Python Implementation:

```
def heapify(arr, n, i):
    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]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

Case Study:
Scenario:Sorting employee salaries.

- Input: `[55000, 75000, 62000, 48000, 90000]`
- Output: `[48000, 55000, 62000, 75000, 90000]`

---

5. Conclusion
Each sorting algorithm has its own advantages:
Merge Sort: Best for linked lists and large datasets.
Quick Sort: Faster in practice but has worst-case $O(n^2)$ complexity.
Heap Sort: Useful for priority queues but slightly slower in practice.