Advanced Sorting Algorithms – Implementation and Performance Analysis (Quick, Merge & Heap Sort)

👷 **Submitted by:**

AARUSH C S
ATHUL K KOSHY
MOHAMMED RIZWAN PP
MOIDEEN NIHAL

---

1. Introduction

Sorting is a core concept in computer science, essential for data processing, searching, and optimization. While basic sorting algorithms work for small datasets, more efficient algorithms are needed for large-scale applications. This project explores three efficient and widely used sorting algorithms: Quick Sort, Merge Sort, and Heap Sort. These are comparison-based, divide-and-conquer or tree-based algorithms used in real-world applications due to their optimal performance.

---

2. Problem Statement

Handling large volumes of data requires sorting algorithms that are fast and resource-efficient. Basic algorithms like Bubble or Insertion Sort become impractical. The challenge is to implement and analyze more efficient sorting algorithms that scale better with input size, especially for real-world applications like databases, search engines, and system schedulers.

---

3. Objectives

- **Implement Quick Sort, Merge Sort, and Heap Sort in Python.**
- **Compare their time and space complexity on different dataset sizes.**
- **Visualize and analyze their step-by-step operations (optional).**
- **Determine which algorithm is most efficient under different conditions.**
- **Understand real-world use cases for each algorithm.**

---

4. Literature Review

- **Quick Sort: A divide-and-conquer algorithm that uses partitioning. Very fast on average but not stable. Worst case is $O(n^2)$, average is $O(n \log n)$.**
- **Merge Sort: A stable, divide-and-conquer algorithm with consistent $O(n \log n)$ performance. It uses additional space for merging.**

- **Heap Sort: A tree-based algorithm that turns the array into a heap structure and extracts elements. Time complexity is O(n log n), and it's in-place but not stable.**
  These algorithms are foundational to many modern libraries and frameworks (e.g., Python's Timsort is a hybrid using merge sort concepts).

---

5. Methodology

1. **Implement Quick Sort, Merge Sort, and Heap Sort in Python.**
2. **Generate random datasets of varying sizes (e.g., 10, 100, 1000, 10,000).**
3. **Measure performance using time taken for sorting.**
4. **Optionally visualize how each algorithm sorts the data.**
5. **Analyze results and record observations.**

---

6. Tools and Technologies

- **Language: Python**
- **Libraries: random, time, matplotlib (for visualizations)**

---

7. Expected Results

- **Quick Sort will be fastest on average for unsorted random data.**
- **Merge Sort will perform consistently across all data types.**
- **Heap Sort will use less memory but may be slightly slower than quick sort.**
- **Each algorithm will be suitable for different real-world scenarios.**

---

8. Challenges and Solutions

Challenge	Solution
Understanding recursive logic	**Use diagrams and dry-run examples**
Managing large datasets	**Use system resources wisely and avoid memory-intensive visualizations**
Comparing time/space accurately	**Run multiple tests and average results**

---

9. Conclusion

This project provides a deeper understanding of advanced sorting algorithms and their practical applications. By implementing and comparing Quick, Merge, and Heap Sort, students learn not only how to code efficiently but also how to choose the right algorithm based on the situation and constraints.

---

10. References

1. **"Introduction to Algorithms" by Cormen, Leiserson, Rivest, and Stein**
2. **GeeksforGeeks – Quick Sort, Merge Sort, Heap Sort Tutorials**
3. **Python Documentation – Sorting Techniques**
4. **HackerRank/LeetCode – Sorting Problems and Challenges**
5. **W3Schools – Sorting in Python**

CODE:-

Quick Sort

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    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)
```

Merge Sort

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
```

```
    return merge(left, right)


def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Heap Sort

```
def heapify(arr, n, i):
    largest = i
    l = 2*i + 1
    r = 2*i + 2

    if l < n and arr[l] > arr[largest]:
        largest = l

    if r < n and arr[r] > arr[largest]:
        largest = r
```

```
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)

    # Build max-heap
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extract elements one by one
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # swap
        heapify(arr, i, 0)
```

Performance Test

```
import random
import time

data = random.sample(range(10000), 1000)

# Quick Sort Test
start = time.time()
qs = quick_sort(data.copy())
print("Quick Sort Time:", round(time.time() - start, 6), "seconds")

# Merge Sort Test
start = time.time()
```

```
ms = merge_sort(data.copy())
print("Merge Sort Time:", round(time.time() - start, 6), "seconds")

# Heap Sort Test
hs = data.copy()
start = time.time()
heap_sort(hs)
print("Heap Sort Time:", round(time.time() - start, 6), "seconds")
```