Merge Sort and Quick Sort Complexity Analysis

# 1. Merge Sort

## Step 1: Recurrence Relation

Merge Sort divides the array into 2 halves and merges them.  
So, the time to sort n elements is:  
T(n) = 2 \* T(n/2) + O(n)  
 - 2 \* T(n/2): Sorting the two halves.  
 - O(n): Merging the two halves.

## Step 2: Solve the Recurrence

Expand:  
T(n) = 2T(n/2) + n  
 = 2[2T(n/4) + n/2] + n  
 = 4T(n/4) + 2n  
 = 8T(n/8) + 3n  
 = ...

## Step 3: Generalize

After k levels:  
T(n) = 2^k \* T(n/2^k) + k \* n  
When n/2^k = 1:  
=> 2^k = n  
=> k = log₂n  
So, T(n) = n \* T(1) + n log₂n = O(n log n)

## Conclusion for Merge Sort

Time Complexity:  
 - Best: O(n log n)  
 - Average: O(n log n)  
 - Worst: O(n log n)  
  
Space Complexity: O(n) (due to temporary arrays)

# 2. Quick Sort

## Step 1: Recurrence Relation

Quick Sort partitions the array and recursively sorts left and right parts.  
If the pivot splits the array into two parts:  
T(n) = T(k) + T(n - k - 1) + O(n)  
where k is the number of elements on the left of pivot.  
Partitioning takes O(n).

## Step 2: Best / Average Case

If the pivot splits perfectly in the middle:  
T(n) = 2T(n/2) + O(n)  
Same as Merge Sort.  
So, T(n) = O(n log n).

## Step 3: Worst Case

If the pivot is always the smallest or largest element:  
One part has 0 elements, other has (n - 1) elements.  
So, T(n) = T(n - 1) + O(n)  
Expand:  
T(n) = T(n - 1) + n  
 = T(n - 2) + (n - 1) + n  
 = ...  
 = 1 + 2 + 3 + ... + n = O(n²)

## Conclusion for Quick Sort

Time Complexity:  
 - Best: O(n log n)  
 - Average: O(n log n)  
 - Worst: O(n²)  
  
Space Complexity:  
 - Average: O(log n) recursion stack  
 - Worst: O(n) recursion stack

## Final Summary

Algorithm | Best Case | Average Case | Worst Case | Space  
------------|-----------|---------------|-------------|-------  
Merge Sort | O(n log n) | O(n log n) | O(n log n) | O(n)  
Quick Sort | O(n log n) | O(n log n) | O(n²) | O(log n) avg, O(n) worst