# Sorting Algorithms: Explanation, Code Snippets, and Complexity Analysis

Jad Elwahy

May 11, 2025

## 1 Introduction

In this document, we will explore three common sorting algorithms: Insertion Sort, Merge Sort, and Quick Sort. We will provide the implementation of each algorithm in Python, followed by an analysis of their time and space complexities.

## 2 Insertion Sort

### 2.1 Code Implementation

```
def insertion_sort(array, size):
    """
    Time Complexity: O(n  )
    Space Complexity: O(1)
    """
    for j in range(1, size):
        key = array[j]
        i = j-1
        while i >= 0 and array[i] > key:
            array[i + 1] = array[i]
            i = i - 1
        array[i + 1] = key
    return array
```

### 2.2 Explanation

Insertion Sort is a simple sorting algorithm that builds the final sorted array one item at a time. It works similarly to the way you might sort playing cards in your hands. It iterates through the input list, growing the sorted portion of the list one element at a time.

### 2.3 Time and Space Complexity

**Best Case:** $O(n)$ when the array is already sorted. In this case, each element is compared only once and no shifting occurs.

**Worst Case:** $O(n^2)$, which occurs when the array is sorted in reverse order. In this case, each element has to be compared with all other elements.

**Space Complexity:** $O(1)$, as no extra space is required except for the input array.

## 3 Merge Sort

### 3.1 Code Implementation

```
def merge_sort(arr):
    """
    Time Complexity:  O(n log n)
    Space Complexity: O(n)
    """
    n = len(arr)
    if n <= 1:
        return arr[:]  # make a shallow copy

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

    # merge the two sorted halves
    merged = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1

    # append any remaining tail
    if i < len(left):
        merged.extend(left[i:])
    if j < len(right):
        merged.extend(right[j:])

    return merged
```

### 3.2 Explanation

Merge Sort is a divide-and-conquer algorithm. It divides the array into halves, recursively sorts each half, and then merges the two sorted halves to produce the sorted result.

### 3.3 Time and Space Complexity

**Best Case:** $O(n \log n)$, which occurs when the array is already sorted or nearly sorted. However, the time complexity is still $O(n \log n)$ due to the merge steps.

**Worst Case:** $O(n \log n)$, which occurs in all cases since the array is always split into two halves and merged back together.

**Space Complexity:** $O(n)$, since we need additional space to store the two halves and the merged array.

## 4 Quick Sort

### 4.1 Code Implementation

```
def quick_sort(array):
    """
    Complexity:    (n  )
    """
    pivot = 0
    i = 1
    j = len(array) - 1
    while(j > i):
        while array[i] <= array[pivot]:
            i += 1
        while array[j] > array[pivot]:
            j -= 1
        if i < j:
            temp = array[i]
            array[i] = array[j]
            array[j] = temp
    temp2 = array[j]
    array[j] = array[pivot]
    array[pivot] = temp2
    pivot = j
    return array
```

### 4.2 Explanation

Quick Sort is a divide-and-conquer algorithm. It selects a 'pivot' element, partitions the array around the pivot, and recursively sorts the subarrays. The

partitioning step is what makes Quick Sort efficient.

### 4.3 Time and Space Complexity

**Best Case:** $O(n \log n)$, which occurs when the pivot divides the array into two nearly equal halves.

**Worst Case:** $O(n^2)$, which occurs when the pivot is the smallest or largest element in the array (e.g., when the array is already sorted or reverse sorted).

**Space Complexity:** $O(\log n)$ for the recursion stack in the best and average case, but can reach $O(n)$ in the worst case.

## 5 Conclusion

We have discussed three popular sorting algorithms: Insertion Sort, Merge Sort, and Quick Sort. Each algorithm has its own advantages and trade-offs depending on the input data. Insertion Sort is simple but inefficient for large datasets, Merge Sort is efficient but requires extra space, and Quick Sort is usually the fastest but can degrade to quadratic time in the worst case.