Q3. Understand Sorting Algorithms:

Before we implement, let's briefly explain the sorting algorithms:

* Bubble Sort: Repeatedly steps through the list, compares adjacent elements and swaps them if they're in the wrong order. O(n^2) time complexity.
* Insertion Sort: Builds the final sorted array one item at a time. O(n^2) time complexity.
* Quick Sort: Uses a divide-and-conquer strategy. Picks a 'pivot' element and partitions the other elements into two sub-arrays. Average case O(n log n) time complexity.
* Merge Sort: Also uses a divide-and-conquer approach. Divides the array into two halves, sorts them, and then merges the two sorted halves. O(n log n) time complexity.

Analysis:

* Time complexity comparison:
  + Bubble Sort: O(n^2) in the worst and average case, O(n) in the best case (when the array is already sorted).
  + Quick Sort: O(n log n) in the average and best case, O(n^2) in the worst case (rare, occurs when the pivot is always the smallest or largest element).

Quick Sort is generally preferred over Bubble Sort for several reasons:

* Better average-case performance: Quick Sort's O(n log n) is significantly faster than Bubble Sort's O(n^2) for large datasets.
* In-place sorting: Quick Sort typically sorts in-place, requiring only O(log n) additional space for recursion. Bubble Sort also sorts in-place.
* Cache efficiency: Quick Sort's partitioning approach tends to be more cache-friendly, which can lead to better performance on modern hardware.
* Adaptability: Quick Sort can be easily modified to improve performance for partially sorted arrays or arrays with many duplicate elements.
* Practical performance: Even though Quick Sort has a worst-case time complexity of O(n^2), this scenario is rare and can be mitigated with techniques like choosing a good pivot.