**Bubble Sort:** Repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until no more swaps are needed.

* **Time Complexity:**
  + Best Case: O(n) (when the list is already sorted)
  + Average Case: O(n^2)
  + Worst Case: O(n^2)

**Insertion Sort:** Builds the final sorted array one item at a time by repeatedly picking the next item and inserting it into its correct position.

* **Time Complexity:**
  + Best Case: O(n) (when the list is already sorted)
  + Average Case: O(n^2)
  + Worst Case: O(n^2)

**Quick Sort:** Uses a divide-and-conquer approach. It selects a 'pivot' element and partitions the array into elements less than the pivot and elements greater than the pivot, then recursively sorts the sub-arrays.

* **Time Complexity:**
  + Best Case: O(nlogn)
  + Average Case: O(nlogn)
  + Worst Case: O(n^2) (occurs when the smallest or largest element is always chosen as the pivot)

**Merge Sort:** Uses a divide-and-conquer approach similar to Quick Sort but divides the array into halves, sorts each half, and then merges the sorted halves.

* **Time Complexity:**
  + Best Case: O(nlogn)
  + Average Case: O(nlogn)
  + Worst Case: O(nlogn)

1. **Time Complexity:**

* **Bubble Sort:** O(n^2) in the worst and average cases.
* **Quick Sort:** O(nlogn) on average, but O(n^2) in the worst case.

1. **Why Quick Sort is Preferred:**

* **Efficiency:** Quick Sort is typically faster for large datasets due to its O(nlogn) average time complexity.
* **In-Place Sorting:** Quick Sort sorts in place, meaning it does not require additional memory proportional.
* **Adaptive:** With good pivot selection strategies (like using the median-of-three), Quick Sort can be very efficient and often outperforms other O(nlogn) algorithms.