**1.Understand Sorting Algorithms:**

**Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort):**

**Bubble Sort**

**Concept**:  
Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The largest element "bubbles" to the end in each pass.

**Example**:  
For the list [5, 3, 8, 4], Bubble Sort compares 5 & 3 → swaps → continues until sorted.

**Time Complexity**:

* Best Case: O(n) (already sorted)
* Worst Case: O(n²)
* Stable: Yes
* Space: O(1)

**Use Case**:  
Very simple, used for teaching purposes or small datasets.

**Insertion Sort**

**Concept**:  
Insertion Sort builds the sorted array one element at a time. It picks an element and inserts it into its correct position among the already sorted elements.

**Example**:  
Given [7, 4, 5], it places 4 before 7, and then places 5 between 4 and 7 → Result: [4, 5, 7].

**Time Complexity**:

* Best Case: O(n) (nearly sorted)
* Worst Case: O(n²)
* Stable: Yes
* Space: O(1)

**Use Case**:  
Efficient for small, nearly sorted datasets.

**Quick Sort**

**Concept**:  
Quick Sort uses the divide-and-conquer technique. It selects a pivot element, partitions the array such that all elements less than the pivot come before it and greater come after it, and recursively sorts the partitions.

**Steps**:

1. Choose a pivot.
2. Partition the array into left (less than pivot) and right (greater than pivot).
3. Recursively sort both sides.

**Time Complexity**:

* Best & Average: O(n log n)
* Worst Case: O(n²) (if pivot is poorly chosen)
* Stable:No
* Space: O(log n) (due to recursion)

**Use Case**:  
Fastest in practice; often used in real-world applications.

**Merge Sort**

**Concept**:  
Merge Sort is another divide-and-conquer algorithm. It divides the array into halves, recursively sorts each half, and then merges the sorted halves into a single sorted array.

**Steps**:

1. Divide array until each sub-array has one element.
2. Merge sub-arrays in sorted order.

**Time Complexity**:

* Best, Worst & Average: O(n log n)
* Stable: Yes
* Space: O(n) (needs extra array for merging)

**Use Case**:  
Best for large datasets where stable sorting is needed.

**4**.**Analyses:**

**Compare the performance (time complexity) of Bubble Sort and Quick Sort:**

|  |  |  |
| --- | --- | --- |
| **Cases** | **Bubble Sort** | **Quick Sort** |
| **Best** | O(n) | O(n log n) |
| **Average** | O(n²) | O(n log n) |
| **Worst** | O(n²) | |  | | --- | |  |  |  | | --- | | O(n²) (rare, poor pivot) | |
| **Space** | O(1) (in-place) | |  | | --- | |  |  |  | | --- | | O(log n) (due to recursion) | |
| **Stable** | Yes | No |

**Discuss why Quick Sort is generally preferred over Bubble Sort:**

**Faster Time Complexity:**

* **Quick Sort** has an average time complexity of **O(n log n)**
* **Bubble Sort** has **O(n²)** in both average and worst case  
   Quick Sort is significantly faster for large datasets.

**Efficient for Large Data:**

* Quick Sort handles **thousands or millions of elements** efficiently.
* Bubble Sort becomes **very slow** as the input size grows.

**Fewer Comparisons and Swaps:**

* Quick Sort minimizes the number of swaps and comparisons.
* Bubble Sort repeatedly compares **every adjacent pair**, leading to many unnecessary swaps.

**Better Memory Use:**

* Quick Sort is **in-place** and uses only **O(log n)** space (due to recursion).
* Bubble Sort is also in-place, but the **slow performance** makes it unsuitable for big data.

**Practical Usage:**

* Most real-world libraries and frameworks (Java, Python, C++) use Quick Sort or its variants.
* Bubble Sort is **mainly used for teaching and understanding sorting logic**.