## **1. Bubble Sort**

## Repeatedly compare adjacent elements and swap them if they’re in the wrong order.

## Like "bubbling" the largest element to the end in each pass.

### Example:

## Given [5, 3, 8, 4], after first pass → [3, 5, 4, 8]

### Time Complexity:

## Best case (already sorted): O(n)

## Average/Worst case: O(n²)

### Not suitable for large data. Only useful for educational purposes or very small lists.

## **2. Insertion Sort**

### Idea:

## Like sorting a hand of playing cards.

## Take each element and insert it into its correct position in the already sorted part.

### Example:

## For [5, 3, 8, 4]:

## Start with 5 (sorted)

## Insert 3 before 5 → [3, 5]

## Then 8 → [3, 5, 8]

## Then 4 between 3 and 5 → [3, 4, 5, 8]

### Time Complexity:

## Best case (nearly sorted): O(n)

## Worst case: O(n²)

### -> Good for small or nearly sorted arrays.

## **3. Quick Sort (Divide and Conquer)**

### Idea:

## Pick a pivot.

## Partition the array: elements < pivot to the left, elements > pivot to the right.

## Recursively sort left and right parts.

### Example:

## For [5, 3, 8, 4], pivot 5:

## Left: [3, 4], Right: [8]

## Recursively sort them → [3, 4, 5, 8]

### Time Complexity:

## Best/Average case: O(n log n)

## Worst case (bad pivot): O(n²)

### Very fast in practice, used in many real-world systems.

## **4. Merge Sort (Divide and Conquer)**

### Idea:

## Divide array into halves until you get single-element arrays

## Merge them back in sorted order

### Example:

## For [5, 3, 8, 4]:

## Split → [5, 3] and [8, 4]

## Sort both → [3, 5] and [4, 8]

## Merge → [3, 4, 5, 8]

### Time Complexity:

## Always: O(n log n)

### Stable sort, good for linked lists or external sorting.

## **Comparison of Bubble Sort vs Quick Sort**

| **Feature** | **Bubble Sort** | **Quick Sort** |
| --- | --- | --- |
| Best Case | O(n) *(already sorted)* | O(n log n) |
| Average Case | O(n²) | O(n log n) |
| Worst Case | O(n²) | O(n²) *(bad pivot)* |
| Space Complexity | O(1) | O(log n) *(recursive stack)* |
| Stability | Stable | Not stable (by default) |
| Real-world Use | Rarely used | Widely used (Java's Arrays.sort uses it) |

## **Why Quick Sort is Preferred Over Bubble Sort**

### 1. Much Faster for Large Inputs

## Bubble Sort compares every pair multiple times, making it inefficient (especially for arrays >100 elements).

## Quick Sort uses a divide-and-conquer strategy, drastically reducing the number of comparisons and swaps.

### 2. Efficient Use of Memory

## Both are in-place (don’t need extra arrays), but Quick Sort only uses a small recursive stack (O(log n)), which is acceptable.

### 3. Practically Efficient

## Even though Quick Sort has a worst case of O(n²), with good pivot strategies (like random or median-of-three), this is rarely hit.

## It performs extremely well in practice, which is why it's used in many standard libraries.

### 4. Scalability

## Quick Sort easily handles millions of records, while Bubble Sort slows down drastically even with a few thousand.

## 