If you have 1 lakh (100,000) records, how sorting behaves?

Bubble Sort

Let's be blunt: **it struggles badly.**

- Time Complexity = **$O(n^2)$**
- So operations ≈ **100,000 × 100,000 = 10,000,000,000 (10 billion comparisons)**

That's huge.
Even if one operation takes very little time, this can take **minutes or worse**.

What happens internally?

- Keeps scanning the array again and again
- Swaps elements repeatedly
- CPU usage goes high
- Very inefficient

Conclusion: Practically unusable for 1 lakh records ❌

Merge Sort

Now this is where things get interesting.

- Time Complexity = **$O(n \log n)$**
- For 100,000:
    - $\log_2(100000) \approx 17$

- Total operations ≈ **100,000 × 17 = ~1.7 million**

Compare:

- Bubble Sort → **10 billion**
- Merge Sort → **1.7 million**

That's a **massive difference**.

How Merge Sort "manages" large data

It uses **Divide & Conquer**:

1. Split data into smaller chunks
   (100000 → 50000 → 25000 → ...)
2. Sort small pieces (easy & fast)
3. Merge them efficiently

Key point:

- Works on **smaller parts**, not whole data at once
- That's why performance is stable and fast

Memory Handling

- Merge Sort uses **extra memory (O(n))**
- For 1 lakh integers:

- ~400 KB (approx) → totally fine in modern systems

Real-world answer (Interview style)

"If we have 1 lakh records, Bubble Sort becomes inefficient due to O(n²) complexity, leading to billions of comparisons. Merge Sort handles it efficiently using divide and conquer with O(n log n), reducing operations drastically and making it suitable for large datasets."

Final Take

- Small data → anything works
- Large data (like 1 lakh) → **use Merge Sort (or even better: Quick Sort / TimSort in real apps)**

Bubble Sort vs Merge Sort

Bubble Sort

Idea:
Repeatedly compares adjacent elements and swaps them if they are in the wrong order.

Example flow:
`[5, 3, 2] → [3, 5, 2] → [3, 2, 5] → [2, 3, 5]`

Time Complexity:

- Best Case (already sorted): **O(n)**
- Average Case: **O(n²)**
- Worst Case: **O(n²)**

Space Complexity:

- **O(1)** (in-place)

Pros:

- Very simple to understand
- No extra memory needed

Cons:

- Extremely slow for large data
- Not used in real-world applications

Merge Sort

Idea:
Divide the array into halves, sort each half, then merge them back.

Example flow:
`[5, 3, 2, 8] → [5,3] [2,8] → [3,5] [2,8] → [2,3,5,8]`

Time Complexity:

- Best Case: **O(n log n)**
- Average Case: **O(n log n)**
- Worst Case: **O(n log n)**

Space Complexity:

- **O(n)** (extra memory required)

Pros:

- Very efficient for large datasets
- Stable sorting algorithm
- Predictable performance

Cons:

- Requires extra memory
- Slightly complex compared to bubble sort

Quick Comparison Table

Feature	Bubble Sort	Merge Sort
Time Complexity	$O(n^2)$	O(n log n)
Space Complexity	O(1)	O(n)
Stability	Stable	Stable
Speed	Slow ❌	Fast ✅
Usage	Learning only	Real-world apps

Interview Answer (Short & Perfect)

Bubble Sort vs Merge Sort:

- **Bubble Sort** is a simple comparison-based algorithm where adjacent elements are swapped repeatedly.
    - Time Complexity: **$O(n^2)$** (worst & average)
    - Space Complexity: **O(1)**
    - Best for: small or nearly sorted data (learning purpose)
- **Merge Sort** is a divide-and-conquer algorithm that splits the array, sorts, and merges.
    - Time Complexity: **O(n log n)** (all cases)
    - Space Complexity: **O(n)**

- Best for: large datasets, efficient and stable

Conclusion: Merge Sort is much faster and preferred in real-world applications.

Java Code

Bubble Sort

```
public class BubbleSort {
  public static void bubbleSort(int[] arr) {
    int n = arr.length;

    for (int i = 0; i < n - 1; i++) {
      boolean swapped = false;

      for (int j = 0; j < n - i - 1; j++) {
        if (arr[j] > arr[j + 1]) {
          // swap
          int temp = arr[j];
          arr[j] = arr[j + 1];
          arr[j + 1] = temp;
          swapped = true;
        }
      }

      // Optimization: stop if already sorted
      if (!swapped) break;
    }
  }

  public static void main(String[] args) {
    int[] arr = {5, 3, 2, 8};

    bubbleSort(arr);

    for (int num : arr) {
      System.out.print(num + " ");
```

```
    }
  }
}
```

Merge Sort

```
public class MergeSort {

  public static void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
      int mid = (left + right) / 2;

      mergeSort(arr, left, mid);
      mergeSort(arr, mid + 1, right);

      merge(arr, left, mid, right);
    }
  }

  public static void merge(int[] arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int[] leftArr = new int[n1];
    int[] rightArr = new int[n2];

    for (int i = 0; i < n1; i++)
      leftArr[i] = arr[left + i];

    for (int j = 0; j < n2; j++)
      rightArr[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
      if (leftArr[i] <= rightArr[j]) {
        arr[k++] = leftArr[i++];
      } else {
        arr[k++] = rightArr[j++];
```

```
        }
    }

    while (i < n1) {
        arr[k++] = leftArr[i++];
    }

    while (j < n2) {
        arr[k++] = rightArr[j++];
    }
  }

  public static void main(String[] args) {
    int[] arr = {5, 3, 2, 8};

    mergeSort(arr, 0, arr.length - 1);

    for (int num : arr) {
        System.out.print(num + " ");
    }
  }
}
```