DSA Sorting Algorithms – Quick Revision Notes

✅ 1. Bubble Sort

Approach:
Compare adjacent elements and swap them if they are in the wrong order. Repeat this until the array is sorted.

Key Properties:

- Type: Comparison-based
- In-place: ✅ Yes
- Stable: ✅ Yes
- Adaptive: ✅ Yes (optimized version)
- Recursive: ❌ No

Time Complexity:

- Best: O(n) (optimized)
- Average: $O(n^2)$
- Worst: $O(n^2)$

Space Complexity: O(1)

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

✅ 2. Selection Sort

Approach:
Find the minimum element and place it at the beginning. Repeat for each position in the array.

Key Properties:

- Type: Comparison-based
- In-place: ✅ Yes

- Stable: ❌ No
- Adaptive: ❌ No
- Recursive: ❌ No

Time Complexity:

- Best / Avg / Worst: $O(n^2)$

Space Complexity: O(1)

Use Case: Simple to understand but not efficient; used for small datasets.

✅ 3. Insertion Sort

Approach:
Build the sorted array one element at a time by inserting current element into its correct position in the already sorted part.

Key Properties:

- Type: Comparison-based
- In-place: ✅ Yes
- Stable: ✅ Yes
- Adaptive: ✅ Yes
- Recursive: ❌ No

Time Complexity:

- Best: O(n)
- Average: $O(n^2)$
- Worst: $O(n^2)$

Space Complexity: O(1)

Use Case: Efficient for small or nearly sorted datasets.

✅ 4. Merge Sort

Approach:
Divide the array into two halves, sort them recursively, and then merge the sorted halves.

Key Properties:

- Type: Divide and Conquer
- In-place: ❌ No
- Stable: ✅ Yes
- Adaptive: ❌ No
- Recursive: ✅ Yes

Time Complexity:

- Best / Avg / Worst: O(n log n)

Space Complexity: O(n)

Use Case: Works well with linked lists and large datasets; used when stability is required.

✅ 5. Quick Sort

Approach:
Choose a pivot, partition the array around the pivot, and recursively sort the left and right parts.

Key Properties:

- Type: Divide and Conquer
- In-place: ✅ Yes
- Stable: ❌ No
- Adaptive: ❌ No
- Recursive: ✅ Yes

Time Complexity:

- Best / Average: O(n log n)
- Worst: $O(n^2)$ (if pivot is smallest/largest repeatedly)

Space Complexity: O(log n) (recursive stack)

Use Case: Fastest average-case sorting for arrays; used in many standard libraries.

✅ 6. Heap Sort

Approach:
Convert array into a max heap, then repeatedly swap the first element with the last unsorted element and heapify.

Key Properties:

- Type: Comparison-based
- In-place: ✅ Yes
- Stable: ❌ No
- Adaptive: ❌ No
- Recursive: ✅ Yes (heapify)

Time Complexity:

- Best / Avg / Worst: O(n log n)

Space Complexity: O(1)

Use Case: Space-efficient sorting when stability is not needed.

📊 Summary Comparison Table

Algorithm	Time (Best)	Time (Avg)	Time (Worst)	Space	Stable	In-place
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	✅	✅
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	❌	✅
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	✅	✅
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	✅	❌
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	❌	✅
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	❌	✅

✅ Tips for MCQs

- Quick Sort is fastest on average, but **not stable**.
- Merge Sort is **stable** and best for **linked lists**.
- Insertion Sort is **adaptive**, **stable**, and great for **nearly sorted data**.
- Heap Sort is **in-place** and has consistent O(n log n) performance.
- Bubble and Selection Sort are simple, but inefficient for large data.