**1. Bubble Sort**

Bubble sort is a simple comparison-based sorting algorithm where adjacent elements are repeatedly swapped if they are in the wrong order.

| **Criterion** | **Bubble Sort** |
| --- | --- |
| **Time Complexity** | Best: O(n) (when already sorted), Worst: O(n²), Average: O(n²) |
| **Space Complexity** | O(1) (in-place) |
| **Stability** | Stable |
| **In-Place** | Yes |
| **Adaptivity** | Adaptive (can stop early if no swaps are made) |
| **Best Use Case** | Educational purposes, small data sets, simple applications |

**Pros**:

* Very simple to understand and implement.
* Adaptive (can be optimized to stop early if the list is already sorted).

**Cons**:

* Slow on large datasets (O(n²) time complexity).
* Inefficient for sorting large arrays or lists.

**2. Selection Sort**

Selection sort works by repeatedly finding the minimum (or maximum) element from the unsorted portion of the array and swapping it with the first unsorted element.

| **Criterion** | **Selection Sort** |
| --- | --- |
| **Time Complexity** | Best: O(n²), Worst: O(n²), Average: O(n²) |
| **Space Complexity** | O(1) (in-place) |
| **Stability** | Not stable |
| **In-Place** | Yes |
| **Best Use Case** | Small data sets, educational purposes, when memory usage is a concern |

**Pros**:

* Simple and easy to implement.
* In-place sorting with O(1) space complexity.
* Does not require additional memory.

**Cons**:

* Inefficient on large data sets (O(n²) time complexity).
* Not stable, meaning it can change the order of equal elements.

**3. Insertion Sort**

Insertion sort builds the sorted array one item at a time by inserting each element into its proper position in the already sorted portion of the array.

| **Criterion** | **Insertion Sort** |
| --- | --- |
| **Time Complexity** | Best: O(n) (when already sorted), Worst: O(n²), Average: O(n²) |
| **Space Complexity** | O(1) (in-place) |
| **Stability** | Stable |
| **In-Place** | Yes |
| **Adaptivity** | Adaptive (best-case O(n) for nearly sorted arrays) |
| **Best Use Case** | Small data sets, nearly sorted arrays, applications where stability is important |

**Pros**:

* Very efficient for small or nearly sorted datasets.
* Stable (preserves relative order of equal elements).
* Adaptive (can perform better on nearly sorted data).

**Cons**:

* Inefficient for large datasets (O(n²) time complexity).
* Slower than other algorithms like quicksort or merge sort on large datasets.

**4. Shell Sort**

Shell sort is an extension of insertion sort that allows the exchange of items that are far apart, thus reducing the number of inversions in the array.

| **Criterion** | **Shell Sort** |
| --- | --- |
| **Time Complexity** | Best: O(n log n), Worst: O(n²), Average: O(n^3/2) |
| **Space Complexity** | O(1) (in-place) |
| **Stability** | Not stable |
| **In-Place** | Yes |
| **Adaptivity** | Partially adaptive (depends on the gap sequence used) |
| **Best Use Case** | Medium-sized datasets, educational purposes, when performance is important but simplicity is a priority |

**Pros**:

* Faster than bubble, selection, and insertion sort on medium-sized arrays.
* Can be adaptive based on the gap sequence.

**Cons**:

* Not stable.
* Its worst-case time complexity is still O(n²) in some cases.
* Gap sequence heavily influences performance (different gap sequences can lead to very different performance).

**5. Quick Sort**

Quick sort is a divide-and-conquer algorithm that divides the array into subarrays based on a pivot element, sorting each subarray recursively.

| **Criterion** | **Quick Sort** |
| --- | --- |
| **Time Complexity** | Best: O(n log n), Worst: O(n²) (when pivot is poorly chosen), Average: O(n log n) |
| **Space Complexity** | O(log n) (in-place, recursive stack space) |
| **Stability** | Not stable |
| **In-Place** | Yes |
| **Best Use Case** | Large data sets, general-purpose sorting, divide-and-conquer applications |

**Pros**:

* Very fast for large data sets (average O(n log n) time complexity).
* In-place sorting (uses less memory).
* Often faster than merge sort and heap sort in practice, despite the same average time complexity.

**Cons**:

* Not stable (can change the relative order of equal elements).
* Worst-case performance can degrade to O(n²) (though this is rare with good pivot selection strategies like randomized pivot or median-of-three).
* Recursive implementation leads to stack overhead.

**6. Merge Sort**

Merge sort is a divide-and-conquer algorithm that divides the array into two halves, recursively sorts them, and then merges the sorted halves.

| **Criterion** | **Merge Sort** |
| --- | --- |
| **Time Complexity** | Best: O(n log n), Worst: O(n log n), Average: O(n log n) |
| **Space Complexity** | O(n) (due to the additional space needed for merging) |
| **Stability** | Stable |
| **In-Place** | No |
| **Best Use Case** | Large data sets, external sorting, when stable sorting is required |

**Pros**:

* Always O(n log n) time complexity, even in the worst case.
* Stable (preserves relative order of equal elements).
* Good for external sorting (sorting large data sets that don't fit in memory).

**Cons**:

* Not in-place (requires additional memory proportional to the array size).
* Slower than quick sort in practice due to additional overhead for merging.
* Requires extra memory, which can be a drawback for large arrays.

**Summary Comparison Table**

| **Algorithm** | **Time Complexity (Best)** | **Time Complexity (Worst)** | **Space Complexity** | **Stability** | **In-Place** | **Adaptivity** | **Best Use Case** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Bubble Sort** | O(n) | O(n²) | O(1) | Stable | Yes | Adaptive | Small, nearly sorted data |
| **Selection Sort** | O(n²) | O(n²) | O(1) | Not Stable | Yes | Non-Adaptive | Small, memory-constrained |
| **Insertion Sort** | O(n) | O(n²) | O(1) | Stable | Yes | Adaptive | Small, nearly sorted data |
| **Shell Sort** | O(n log n) | O(n²) | O(1) | Not Stable | Yes | Partially Adaptive | Medium-sized data sets |
| **Quick Sort** | O(n log n) | O(n²) | O(log n) | Not Stable | Yes | Non-Adaptive | Large data sets, general-purpose |
| **Merge Sort** | O(n log n) | O(n log n) | O(n) | Stable | No | Non-Adaptive | Large data sets, stability needed |

**Key Takeaways**

* **Bubble, Selection, and Insertion Sorts** are simple but inefficient for large datasets, with O(n²) time complexity.
* **Shell Sort** is an improvement over Insertion Sort, but its performance heavily depends on the choice of gap sequence.
* **Quick Sort** is generally the fastest for large datasets (average O(n log n)), but it can degrade to O(n²) in the worst case.
* **Merge Sort** guarantees O(n log n) performance but requires O(n) extra space and is slower than Quick Sort in practice for most applications. It is particularly useful for stable sorting or when working with large data that can't fit in memory.

Top of Form

Bottom of Form