**Comparison Table**

| **Feature** | **Bubble Sort** | **Insertion Sort** | **Selection Sort** |
| --- | --- | --- | --- |
| **Pointer Use** | Yes | Yes | Yes |
| **Stability** | ✅ Yes (if implemented properly) | ✅ Yes | ❌ No |
| **Best Time** | O(n) | O(n) | O(n²) |
| **Average Time** | O(n²) | O(n²) | O(n²) |
| **Worst Time** | O(n²) | O(n²) | O(n²) |
| **Space (Auxiliary)** | O(1) | O(1) | O(1) |
| **Swaps** | Many (every comparison may swap) | Few (only when inserting) | Few (one per iteration) |
| **Use Case** | Nearly sorted arrays | Small/nearly sorted arrays | Consistent but inefficient |
| **Optimization Present** | ✅ Early exit with swap flag | ❌ Not typical | ❌ None |

* All three algorithms are **O(n²)** in average and worst-case performance, making them **unsuitable for large datasets**.
* **Insertion Sort** performs best among the three for **nearly sorted arrays**, thanks to fewer comparisons and shifts.
* **Bubble Sort**, with the **swap flag optimization**, slightly improves performance on sorted data but still makes unnecessary comparisons.
* **Selection Sort** is the **most predictable** but offers no performance advantage and makes unnecessary comparisons regardless of data order.
* All are **in-place** (O(1) auxiliary space), which is efficient in terms of memory.
* Using **pointers** instead of array indexing doesn't improve complexity but can be a good **learning exercise** for understanding memory access and pointer manipulation in C++.