**Exercise 3: Sorting Customer Orders**

**Bubble Sort:**

* **Description:** A simple comparison-based algorithm where each pair of adjacent elements is compared, and the elements are swapped if they are in the wrong order. This process is repeated until the list is sorted.
* **Time Complexity:** O(n^2) for average and worst cases, O(n) for the best case (when the list is already sorted).
* **Space Complexity:** O(1) (in-place sorting).
* **Pros and Cons:** Easy to understand and implement, but inefficient for large datasets.

**Insertion Sort:**

* **Description:** Builds the sorted array one item at a time by repeatedly picking the next item and inserting it into the sorted portion of the array.
* **Time Complexity:** O(n^2) for average and worst cases, O(n) for the best case (when the list is already sorted).
* **Space Complexity:** O(1) (in-place sorting).
* **Pros and Cons:** More efficient than bubble sort for small datasets or nearly sorted arrays, but still inefficient for large datasets.

**Quick Sort:**

* **Description:** A divide-and-conquer algorithm that selects a 'pivot' element and partitions the array into two sub-arrays, one with elements less than the pivot and one with elements greater than the pivot. The process is recursively applied to the sub-arrays.
* **Time Complexity:** O(n log n) for average cases, O(n^2) for worst cases (can be avoided with good pivot selection strategies like random pivot).
* **Space Complexity:** O(log n) for the average case (due to recursion).
* **Pros and Cons:** Very efficient for large datasets, especially with good pivot selection. However, it is more complex to implement than simpler algorithms.

**Merge Sort:**

* **Description:** A stable, comparison-based divide-and-conquer algorithm that divides the array into halves, recursively sorts them, and then merges the sorted halves.
* **Time Complexity:** O(n log n) for all cases.
* **Space Complexity:** O(n) (due to the auxiliary space needed for merging).
* **Pros and Cons:** Consistently efficient and stable but requires additional space.

**Performance Comparison:**

* **Bubble Sort:**
  + **Best Case:** O(n). When the array is already sorted.
  + **Average Case:** O(n^2)
  + **Worst Case:** O(n^2)
  + **Space Complexity:** O(1)
* **Quick Sort:**
  + **Best Case:** O(n log n)
  + **Average Case:** O(n log n)
  + **Worst Case:** O(n^2). When the pivot selection is poor, e.g., always picking the smallest or largest element.
  + **Space Complexity:** O(log n). Due to recursion.

**Why Quick Sort is Generally Preferred Over Bubble Sort:**

* **Efficiency:** Quick Sort has an average-case time complexity of O(n log n), making it much faster for large datasets compared to Bubble Sort's O(n^2).
* **Scalability:** Quick Sort handles larger datasets more efficiently due to its divide-and-conquer approach.
* **Practical Performance:** Despite its worst-case time complexity, Quick Sort typically outperforms other O(n log n) algorithms like Merge Sort in practice due to lower constant factors and better cache performance.