**Exercise 3: Sorting Customer Orders**

**1. Understand Sorting Algorithms**

* **Bubble Sort:**
  + **Description**: A simple comparison-based algorithm. Repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order.
  + Traverse from left and compare adjacent elements and the higher one is placed at right side.
  + In this way, the largest element is moved to the rightmost end at first.
  + This process is then continued to find the second largest and place it and so on until the data is sorted.
  + This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.
  + **Time Complexity:**
    - Best Case: O(n) (when the array is already sorted)
    - Average Case: O(n^2)
    - Worst Case: O(n^2)
  + Space Complexity: O(1) (in-place sorting)
* **Insertion Sort:**
  + **Description**: Builds the final sorted array one item at a time, picking the next item and inserting it into its correct position.

**Algorithm:**

* + To achieve insertion sort, follow these steps:
  + We have to start with second element of the array as first element in the array is assumed to be sorted.
  + Compare second element with the first element and check if the second element is smaller then swap them.
  + Move to the third element and compare it with the second element, then the first element and swap as necessary to put it in the correct position among the first three elements.
  + Continue this process, comparing each element with the ones before it and swapping as needed to place it in the correct position among the sorted elements.
  + Repeat until the entire array is sorted.
  + **Time Complexity:**
    - Best Case: O(n) (when the array is already sorted)
    - Average Case: O(n^2)
    - Worst Case: O(n^2)
  + Space Complexity: O(1) (in-place sorting)
* **Quick Sort:**
  + **Description**: A divide-and-conquer algorithm. Picks an element as a pivot and partitions the array around the pivot. Recursively applies the same process to the sub-arrays.
  + The key process in quickSort is a partition() . The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.
  + Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.
  + **Partition Algorithm:**

The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal) elements as i . While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise, we ignore the current element.

* + **Time Complexity:**
    - 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., the smallest or largest element in a sorted array)
  + Space Complexity: O(log n) (due to recursive calls)
* **Merge Sort:**
  + **Description**: A divide-and-conquer algorithm. Divides the array into halves, recursively sorts them, and then merges the sorted halves.

**Here’s a step-by-step explanation of how merge sort works**:

* + **Divide:**Divide the list or array recursively into two halves until it can no more be divided.
  + **Conquer:**Each subarray is sorted individually using the merge sort algorithm.
  + **Merge:**The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.
  + **Time Complexity:**
    - Best Case: O(n log n)
    - Average Case: O(n log n)
    - Worst Case: O(n log n)
  + Space Complexity: O(n) (not in-place due to the merging process)

**Analysis**

* **Time Complexity Comparison**:
  + **Bubble Sort**: O(n^2) in the average and worst case. It performs poorly on large datasets due to its quadratic time complexity.
  + **Quick Sort**: O(n log n) in the average and best case, but O(n^2) in the worst case. However, with good pivot selection (e.g., median of three), it performs much better in practice.

**Why Quick Sort is Generally Preferred**:

* Quick Sort is preferred over Bubble Sort due to its significantly better average-case time complexity of O(n log n), making it more efficient for large datasets.
* While Bubble Sort is simple to implement and understand, its inefficiency makes it impractical for real-world use cases involving large data sets.
* Quick Sort's divide-and-conquer approach and in-place sorting make it both space-efficient and faster for most applications, especially with optimizations like choosing a good pivot.