**Exercise - 3:** **Sorting Customer Orders**

**Step1:**

**Understanding the Sorting Algorithms**

**Bubble Sort**

* **Description**: Bubble Sort is 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) in the average and worst case.

**How Bubble Sort Works:**

1. **Compare Adjacent Elements**: Start at the beginning of the list and compare the first two elements.
2. **Swap if Necessary**: If the first element is greater than the second element, swap them.
3. **Move to the Next Pair**: Move to the next pair of elements and repeat the process until the end of the list.
4. **Repeat**: Repeat the entire process for the entire list, excluding the last sorted elements with each pass (since they are already sorted).
5. **Stop**: The algorithm stops when no more swaps are needed, indicating that the list is sorted.

**Example:**

Consider the list: [5, 1, 4, 2, 8]

* **First Pass**:
  + Compare 5 and 1, swap to get [1, 5, 4, 2, 8]
  + Compare 5 and 4, swap to get [1, 4, 5, 2, 8]
  + Compare 5 and 2, swap to get [1, 4, 2, 5, 8]
  + Compare 5 and 8, no swap needed: [1, 4, 2, 5, 8]
* **Second Pass**:
  + Compare 1 and 4, no swap needed: [1, 4, 2, 5, 8]
  + Compare 4 and 2, swap to get [1, 2, 4, 5, 8]
  + Compare 4 and 5, no swap needed: [1, 2, 4, 5, 8]
  + Compare 5 and 8, no swap needed: [1, 2, 4, 5, 8]
* **Third Pass**:
  + Compare 1 and 2, no swap needed: [1, 2, 4, 5, 8]
  + Compare 2 and 4, no swap needed: [1, 2, 4, 5, 8]
  + Compare 4 and 5, no swap needed: [1, 2, 4, 5, 8]
  + Compare 5 and 8, no swap needed: [1, 2, 4, 5, 8]

**Quick Sort**

Quick Sort is an efficient, comparison-based, divide-and-conquer sorting algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively. This process continues until the base case of an empty or single-element sub-array is reached, which is by definition sorted.

**Time Complexity**: O(n log n) on average, but can degrade to O(n^2) in the worst case.

**How Quick Sort Works:**

1. **Choose a Pivot**: Select an element from the array as the pivot. Various strategies can be used to choose the pivot, such as picking the first element, the last element, the middle element, or a random element.
2. **Partitioning**: Rearrange the array so that all elements less than the pivot come before it, and all elements greater than the pivot come after it. This step is called partitioning.
3. **Recursive Sorting**: Recursively apply the above steps to the sub-arrays of elements with smaller and larger values.
4. **Base Case**: When the size of the sub-array is zero or one, it is already sorted, and the recursion ends.

**Example Array**

Consider the array: [10, 80, 30, 90, 40, 50, 70]

**Step-by-Step Process**

1. **Choosing Pivot**:
   * Let's choose the last element, 70, as the pivot.
2. **Partitioning**:
   * We need to rearrange the array so that all elements less than 70 come before it and all elements greater than 70 come after it.

**Initial Array: [10, 80, 30, 90, 40, 50, 70]**

* Start with two pointers: one at the beginning (low) and one at the end (high).
* The pivot element is 70.

**Partition Process:**

* Compare 10 (first element) with 70 (pivot). Since 10 < 70, move the pointer to the next element.
* Compare 80 with 70. Since 80 > 70, leave it and move to the next element.
* Compare 30 with 70. Since 30 < 70, move it to the left side.
* Compare 90 with 70. Since 90 > 70, leave it and move to the next element.
* Compare 40 with 70. Since 40 < 70, move it to the left side.
* Compare 50 with 70. Since 50 < 70, move it to the left side.

After partitioning around pivot 70, the array looks like this: [10, 30, 40, 50, 70, 90, 80]

**Recursive Steps**

1. **Left Sub-array**:
   * The left sub-array is [10, 30, 40, 50].
   * Choose the pivot as the last element, 50.
2. **Partitioning Left Sub-array**:
   * Compare 10 with 50. Since 10 < 50, move to the next element.
   * Compare 30 with 50. Since 30 < 50, move to the next element.
   * Compare 40 with 50. Since 40 < 50, move to the next element.

After partitioning around pivot 50, the left sub-array is [10, 30, 40, 50].

1. **Right Sub-array**:
   * The right sub-array is [90, 80].
   * Choose the pivot as the last element, 80.
2. **Partitioning Right Sub-array**:
   * Compare 90 with 80. Since 90 > 80, move 80 to the left side.

After partitioning around pivot 80, the right sub-array is [80, 90].

**Final Sorted Array:**

After recursively sorting all sub-arrays, we get the final sorted array: [10, 30, 40, 50, 70, 80, 90].

**Merge Sort:**

**Description:** Merge Sort is a divide-and-conquer algorithm that divides the unsorted list into n sublists until each sublist contains one element and then merges those sublists to produce a sorted list.

**Time Complexity**: O(n log n) in all cases (worst, average, best).

**Example Array**

Consider the array: [38, 27, 43, 3, 9, 82, 10]

**Step-by-Step Process**

**Initial Array: [38, 27, 43, 3, 9, 82, 10]**

1. **Divide**:
   * Divide the array into two halves: [38, 27, 43] and [3, 9, 82, 10].
2. **Divide Again**:
   * Divide [38, 27, 43] into [38] and [27, 43].
   * Divide [27, 43] into [27] and [43].
   * Divide [3, 9, 82, 10] into [3, 9] and [82, 10].
   * Divide [3, 9] into [3] and [9].
   * Divide [82, 10] into [82] and [10].
3. **Conquer (Merge)**:
   * Merge [38] and [27, 43]:
     + Merge [27] and [43] to get [27, 43].
     + Merge [38] and [27, 43] to get [27, 38, 43].
   * Merge [3] and [9] to get [3, 9].
   * Merge [82] and [10] to get [10, 82].
   * Merge [3, 9] and [10, 82] to get [3, 9, 10, 82].
   * Finally, merge [27, 38, 43] and [3, 9, 10, 82] to get [3, 9, 10, 27, 38, 43, 82].

**Final Sorted Array:**

After recursively sorting and merging all sub-arrays, we get the final sorted array: [3, 9, 10, 27, 38, 43, 82].