**SORTING CUSTOMER ORDERS**

1.Bubble Sort

- Bubble Sort is a simple comparison-based sorting algorithm.

- It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

Algorithm:

1. Start at the beginning of the array.

2. Compare each pair of adjacent items and swap them if the first item is greater than the second item.

3. Continue this process until the end of the array.

4. Repeat the process for the remaining elements (ignoring the last sorted elements) until no more swaps are needed.

Time Complexity:

- Best: O(n) (when the list is already sorted)

- Average: O(n^2)

- Worst: O(n^2)

Advantages:

- Simple to implement.

- Requires no additional memory beyond the original array.

Disadvantages:

- Inefficient on large lists.

- Generally considered impractical for large datasets.

2. Insertion Sort

- Insertion Sort builds the sorted array one item at a time.

- It removes elements from the input data, finds the location they belong in the sorted list, and inserts them there.

Algorithm:

1. Start with the second element of the array.

2. Compare it to the elements before it and insert it into the correct position in the sorted portion of the array.

3. Repeat the process for the next element until the entire array is sorted.

Time Complexity:

- Best: O(n) (when the list is already sorted)

- Average: O(n^2)

- Worst: O(n^2)

Advantages:

- Simple to implement.

- Efficient for small datasets or nearly sorted lists.

- Stable (does not change the relative order of equal elements).

Disadvantages:

- Inefficient for large lists.

- Performance degrades quickly with larger data sets.

3. Quick Sort

- Quick Sort is a divide-and-conquer sorting algorithm.

- It works by selecting a 'pivot' element and partitioning the array into two sub-arrays: elements less than the pivot and elements greater than the pivot.

- It then recursively sorts the sub-arrays.

Algorithm:

1. Choose a pivot element from the array (commonly the middle or the last element).

2. Partition the array into two parts: elements less than the pivot and elements greater than the pivot.

3. Recursively apply the same process to the two sub-arrays.

Time Complexity:

- Best: O(nlog n)

- Average: O(nlog n)

- Worst: O(n^2) (when the smallest or largest element is consistently chosen as the pivot)

Advantages:

- Efficient for large datasets.

- In-place sorting algorithm (requires little additional memory).

Disadvantages:

- The worst-case performance is \(O(n^2)\), though this can be mitigated with techniques such as randomized pivot selection.

- Not stable.

4. Merge Sort

- Merge Sort is another divide-and-conquer algorithm.

- It divides the array into two halves, recursively sorts each half, and then merges the sorted halves to produce a sorted array.

Algorithm:

1. Divide the array into two halves.

2. Recursively sort both halves.

3. Merge the two sorted halves to produce the sorted array.

Time Complexity:

- Best: O(nlog n)

- Average: O(n log n)

- Worst: O(n log n)

Advantages:

- Consistently efficient, regardless of the initial order of elements.

- Stable (maintains the relative order of equal elements).

Disadvantages:

- Requires additional memory proportional to the size of the array (for the temporary arrays used during merging).

- Slightly more complex to implement compared to simpler algorithms like Bubble Sort or Insertion Sort.

Why Quick Sort is Generally Preferred Over Bubble Sort

1. Time Complexity:

Efficiency: Quick Sort has an average-case time complexity of O(n logn), making it much more efficient than Bubble Sort, which has an average and worst-case time complexity of O(n^2). For large datasets, Quick Sort is significantly faster.

2. Performance on Large Datasets:

Scalability: Quick Sort scales better with larger datasets. Bubble Sort’s performance degrades quickly as the size of the data increases due to its quadratic time complexity.

3. Divide-and-Conquer Approach:

Effective Partitioning: Quick Sort’s divide-and-conquer strategy allows it to efficiently sort large arrays by dividing the problem into smaller subproblems and solving them recursively. This results in better average performance compared to the simple pairwise comparison approach used in Bubble Sort.