**EXERCISE 3: SORTING CUSTOMER ORDERS**

**Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).**

Sorting algorithms are used to arrange the values either in ascending or descending order.

**1) Bubble Sort:**

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This process is repeated until the list is sorted.

This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

**Algorithm,**

* 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.

**Time Complexity**

* Best: O(n)
* Average: O(n^2)
* Worst: O(n^2)

**Auxiliary Space:** O(1)

**2) Insertion Sort**

It is a in-place sorting algorithm. Insertion sort works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list.

It is a stable sorting algorithm - elements with equal values maintain their relative order in the sorted output.

**Algorithm,**

* Assume the first element to be sorted
* start with second element
* 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)
* Average case: O(n^2 )
* Worst case: O(n^2)

**Space Complexity**

Auxiliary Space: O(1).

**3) Quick Sort**

Quick sort is based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

There are many different choices for picking pivots.

* Always pick the first element as a pivot .
* Always pick the last element as a pivot (implemented below)
* Pick a random element as a pivot .
* Pick the middle as the pivot.

The key process is a partition. The target of partitions is to place the 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.

**Time Complexity:**

* Best Case : Ω (N log (N))
* Average Case: θ ( N log (N))
* Worst Case: O(N ^ 2)

**Space Complexity:**

O(1), if we don’t consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make O ( N )

**4) Merge Sort**

Merge sort is a sorting algorithm that follows the divide-and-conquer approach.

It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

Three process takes place,

* **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), Additional space is required for the temporary array used during merging.

**Compare the performance (time complexity) of Bubble Sort and Quick Sort.**

|  |  |  |
| --- | --- | --- |
|  | Bubble Sort | Quick Sort |
| Best Case | **O(n) :** Occurs when the array is already sorted. The algorithm only needs one pass through the array to confirm it is sorted (no swaps needed) | **O(n log n):** Occurs when the pivot divides the array into two equal halves consistently. The work done at each level of recursion remains balanced, resulting in the most efficient case. |
| Average Case | **O(n^2):** Bubble Sort requires a quadratic number of comparisons and swaps because it has to compare each element with every other element. | **O(n log n):** Quick Sort divides the array into roughly equal parts with each pivot selection, leading to logarithmic depth in the recursion and linear work per level. |
| Worst Case | **O(n^2):** Occurs when the array is sorted in reverse order. The algorithm has to make the maximum number of swaps to sort the array. | **O(n^2):** Occurs when the pivot selection is poor, such as always selecting the smallest or largest element as the pivot. This causes the algorithm to degenerate into a less efficient sort similar to Bubble Sort. |

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

* Quick Sort has an average time complexity of O(n log n), making it significantly faster for large datasets compared to Bubble Sort's O(n^2) complexity.
* Quick Sort is more cache-efficient due to its in-place nature, which makes better use of the CPU cache compared to Bubble Sort.
* Even though Quick Sort has a worst-case time complexity of O(n^2) this scenario can often be avoided with good pivot selection strategies (e.g., using median-of-three)
* Quick Sort can be optimized further with techniques like tail recursion elimination and iterative versions, making it more adaptable and efficient in various scenarios.