MUKIIZA ANGEL NINA TWINE S22B23/007

I) Describe the working principle of the insertion sort algorithm.

The insertion sort algorithm works by dividing the input list into two parts: a sorted subarray and an unsorted subarray. It iterates through the unsorted subarray, taking one element at a time, and inserts it into its correct position within the sorted subarray. The algorithm continues this process until all elements are part of the sorted subarray, resulting in a fully sorted list.

Here's an an outline of the insertion sort's basic workings :

* Start with a sorted subarray of one element (the first element) and an unsorted subarray containing the rest of the elements.
* Iterate through the unsorted subarray, comparing each element with the elements in the sorted subarray, moving larger elements to the right to make space for the current element.
* Insert the current element into its correct position within the sorted subarray.
* Continue this process until all elements in the unsorted subarray have been moved to the sorted subarray, resulting in a fully sorted list.

Insertion sort is a simple but relatively inefficient sorting algorithm for large datasets. However, it's efficient for small lists or lists that are already partially sorted.

II) Discuss the best-case, average-case, and worst-case time complexities of insertion

sort. Under what conditions does the best-case scenario occur?

* Best-case time complexity:The best-case scenario occurs when the input list is already sorted. In this case, insertion sort has a linear time complexity of O(n). This is because there are no elements that need to be shifted, and the algorithm only needs to make n - 1 comparisons to confirm that each element is in its correct position. It's quite efficient for nearly sorted or small lists.
* Average-case time complexity:The average-case time complexity of insertion sort is O(n^2). This is because, on average, you will need to make roughly n^2/4 comparisons and n^2/4 swaps to sort an input list. The exact number of comparisons and swaps can vary depending on the initial order of the elements in the list. For random or unsorted data, the average case is still quadratic, making it less efficient compared to more advanced sorting algorithms.
* Worst-case time complexity:The worst-case scenario occurs when the input list is sorted in reverse order. In this case, insertion sort has a time complexity of O(n^2) as well. This is because, for each element in the unsorted subarray, you need to compare it to and shift it past every element in the sorted subarray. This leads to a quadratic number of comparisons and swaps.

The best-case scenario for the insertion sort algorithm occurs when the input list is already sorted or nearly sorted. In other words, the best-case conditions are met when the elements in the list are in the correct order, and there is little or no need for swapping or shifting elements during the sorting process.

III) This result is correct, and the list has been sorted correctly. The insertion sort algorithm repeatedly compares each element to the elements to its left and swaps them if they are in the wrong order. This process is repeated for each element in the list, effectively sorting it in ascending order.

The sorted list is in ascending order, starting with the smallest element (1) and ending with the largest element (17), which confirms that the insertion sort algorithm worked as expected.

IV) Insertion sort is more efficient than bubble sort but still has limitations when it comes to larger datasets, making it best suited for small lists or nearly sorted data. Bubble sort is even less efficient and is mainly used for educational purposes due to its simplicity and lack of practicality in real-world applications.

In contrast, merge sort outperforms both insertion and bubble sort in terms of efficiency, consistently providing O(n log n) time complexity, making it a superior choice for large datasets, and it's commonly used in practical applications when performance is critical.

V) Yes, insertion sort would be a good choice for a nearly sorted list with just a few elements out of order. Insertion sort is particularly efficient in such cases because it takes advantage of the existing order, minimizing the number of comparisons and swaps required. The small number of out-of-order elements would require only a few comparisons and swaps, making insertion sort a practical and efficient choice for sorting nearly sorted lists or lists with a limited degree of disorder.