Quick Sort

1. **Overview**: QuickSort is a popular sorting algorithm that uses the divide-and-conquer technique to efficiently sort an array or list.
2. **Divide-and-Conquer**: The algorithm divides the array into smaller sub-arrays, sorts them independently, and then combines the sorted sub-arrays to obtain the final sorted array.
3. **Pivot Selection**: QuickSort selects a pivot element from the array. Common choices for the pivot are the first, last, or middle element.
4. **Partitioning**: The array is rearranged such that all elements less than the pivot are on one side, and all elements greater than the pivot are on the other side. The pivot element is now in its correct sorted position.
5. **Partitioning Process**: QuickSort uses two pointers, one moving from the beginning of the array, and the other moving from the end. They stop when elements are found that need to be swapped, and the process continues until the pointers cross each other.
6. **Recursion**: After partitioning, the algorithm recursively applies QuickSort to the sub-arrays on both sides of the pivot until the entire array is sorted.
7. **Base Case**: The recursion stops when the sub-array size becomes 1 or 0, as single-element arrays are already sorted.
8. **Average Time Complexity**: **QuickSort has an average-case time complexity of O(n log n), making it one of the fastest sorting algorithms for large datasets.**
9. **Worst-case Time Complexity**: **In the worst case, QuickSort may have a time complexity of O(n^2) if the pivot selection leads to highly unbalanced partitions. However, using random or median-of-three pivot selection reduces the likelihood of this scenario.**
10. **Space Complexity: The space complexity of the recursive QuickSort algorithm is O(n), where n is the number of elements in the input array.**
11. **In-place Sorting**: QuickSort is an in-place sorting algorithm, meaning it doesn't require additional memory for sorting.
12. **Unstable Sorting**: QuickSort is an unstable sorting algorithm, which means the relative order of equal elements might not be preserved after sorting.
13. **Real-world Performance**: QuickSort is often faster than other sorting algorithms, such as MergeSort and HeapSort, due to its efficient partitioning and small constant factors.

Step-by-step process of QuickSort on the given array [44, 2, 5, 6, 2, 5, 44, 11, 66, 102, -2] and explain the output:

Step 1: Original Array: [44, 2, 5, 6, 2, 5, 44, 11, 66, 102, -2]

Step 2: Applying QuickSort: The leftmost element, 44, is selected as the pivot.

Step 3: Partitioning Process: The array is partitioned around the pivot 44. Elements smaller than or equal to 44 are moved to the left side, and elements greater than 44 are moved to the right side.

Partitioned Array: [-2, 2, 5, 6, 2, 5, 11, 44, 44, 102, 66]

Step 4: Recursion: The QuickSort function is called recursively on the left sub-array and the right sub-array.

Left Sub-array: [-2, 2, 5, 6, 2, 5, 11]

Right Sub-array: [102, 66]

Step 5: Partitioning and Recursion on Left Sub-array: The left sub-array [-2, 2, 5, 6, 2, 5, 11] is partitioned using the leftmost element (-2) as the pivot.

Partitioned Left Sub-array: [-2, 2, 2, 5, 5, 11, 6]

Recursion on Left Sub-array: The QuickSort function is called recursively on the left sub-array.

Left Sub-array: [-2, 2, 2, 5, 5, 11]

Right Sub-array: [6]

Step 6: Partitioning and Recursion on Left Sub-array: The left sub-array [-2, 2, 2, 5, 5, 11] is partitioned using the leftmost element (-2) as the pivot.

Partitioned Left Sub-array: [-2, 2, 2, 5, 5, 11]

Recursion on Left Sub-array: The QuickSort function is called recursively on the left sub-array.

Left Sub-array: [-2, 2, 2, 5, 5, 11]

Right Sub-array: [] (Empty)

Step 7: Recursion on Right Sub-array: Since the right sub-array [6] has only one element, no further recursion is needed.

Step 8: Final Result: The QuickSort algorithm completes, and the array is sorted in ascending order.

Sorted Array: [-2, 2, 2, 5, 5, 6, 11, 44, 44, 66, 102]

The output displays the sorted array after applying the QuickSort algorithm.

Approach 1: Recursive QuickSort function

1. The **partition** function is responsible for partitioning the array based on a pivot element. It takes the array, start index, and end index as parameters.
2. Inside the **partition** function, the pivot is selected as the first element (**arr[start]**) of the given sub-array.
3. The function counts the number of elements smaller than or equal to the pivot using the **count** variable.
4. The pivot index is calculated as **start + count**, and then the pivot is swapped with the element at **pivotIndex**.
5. The function uses two pointers (**i** and **j**) to move inward from the left and right ends of the sub-array, respectively.
6. The pointers continue moving until they cross each other. If an element is found on the left side that is greater than the pivot, and an element is found on the right side that is lesser than the pivot, they are swapped to maintain the partition.
7. Finally, the **partition** function returns the pivot index.
8. The **quickSort** function is the recursive implementation of the QuickSort algorithm. It takes the array, start index, and end index as parameters.
9. If the start index is greater than or equal to the end index, it means the sub-array has only one element or is empty, so the function returns.
10. Otherwise, it calls the **partition** function to partition the array and get the pivot index.
11. It then recursively calls **quickSort** on the left sub-array (from **start** to **pivotIndex-1**) and the right sub-array (from **pivotIndex+1** to **end**).
12. The **print** function is used to display the elements of an array.
13. In the **main** function, an example array is created (**{44,2,5,6,2,5,44,11,66,102,-2}**) and displayed.
14. The **quickSort** function is called on the array to sort it.
15. Finally, the sorted array is printed using the **print** function.