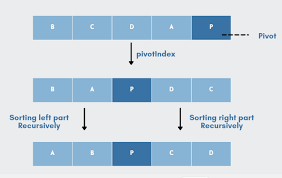
**Quick Sort:**

****

Quick Sort is a widely used sorting algorithm that employs a divide-and-conquer strategy. 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 recursively sorted.

How it works:

1. Choose a pivot element from the array.

2. Partition the array into two sub-arrays - elements less than the pivot and elements greater than the pivot.

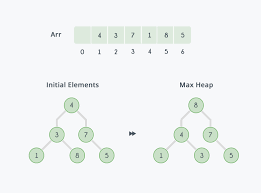
3. Recursively apply Quick Sort to the sub-arrays.

4. Combine the sorted sub-arrays and the pivot to produce the sorted array.

Worst-case Complexity:

The worst-case time complexity of Quick Sort occurs when the chosen pivot is consistently the smallest or largest element in the array, leading to highly unbalanced partitions. In this case, the time complexity is O(n^2). However, on average, Quick Sort exhibits a time complexity of O(n log n), making it efficient for most practical cases.

**Heap Sort:**

****

Heap Sort is a comparison-based sorting algorithm that utilizes a binary heap data structure. It involves building a max-heap (for ascending order) or min-heap (for descending order) from the input array and repeatedly extracting the root element, which is the maximum (or minimum) element in the heap.

How it works:

1. Build a max-heap (for ascending order) from the input array.

2. Swap the root element (the maximum) with the last element and reduce the heap size.

3. Repeat step 2 until the heap is empty, placing the maximum element at the end of the array.

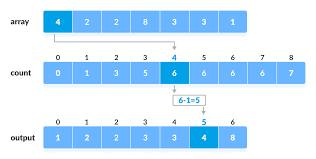
4. The array is now sorted.

Worst-case Complexity:

The worst-case and average-case time complexity of Heap Sort is O(n log n), making it efficient for most scenarios.

**Counting Sort:**

Counting Sort is a non-comparative sorting algorithm that works well for integers within a known, small range. It counts the frequency of each element in the input and reconstructs a sorted array.



How it works:

1. Determine the range of input values.

2. Initialize a count array to store the frequency of each element.

3. Count the occurrences of each element in the input.

4. Compute the cumulative sum of the counts.

5. Place elements in their sorted positions using the cumulative counts.

6. Create the sorted output array.

Worst-case Complexity:

The worst-case and average-case time complexity of Counting Sort is O(n + k), where 'n' is the number of elements to be sorted, and 'k' is the range of input values. It's efficient when the range of values (k) is not significantly larger than the number of elements (n).