Algorithms And Data Structures

Kapilraj Trivedi

Training outline

Module 1

Introduction

Module 2

DATA STRUCTURES

Module 3

COMPLEXITY ANALYSIS

Module 4

SORTING ALGORITHMS

Module 5

NON-LINEAR DATA STRUCTURES

Module 6

ABSTRACT DATA TYPES

Module 7

BINARY SEARCH

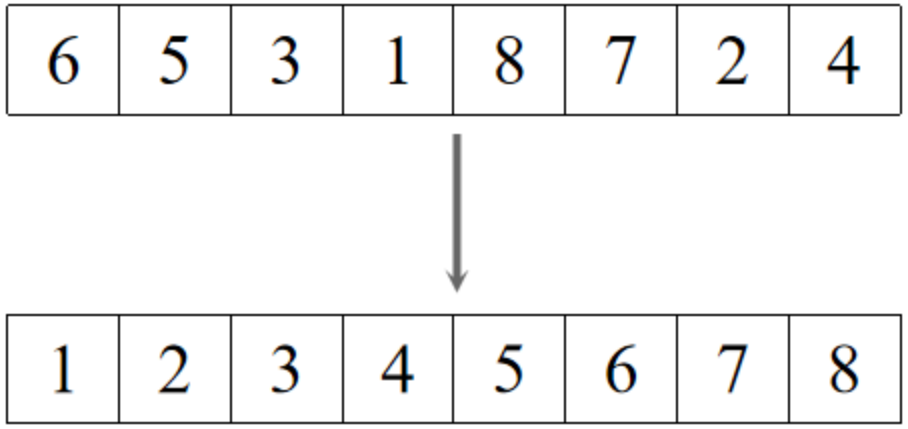
Module 8

DIJKSTRA'S ALGORITHM

Module 4 Sorting Algorithms

Sorting

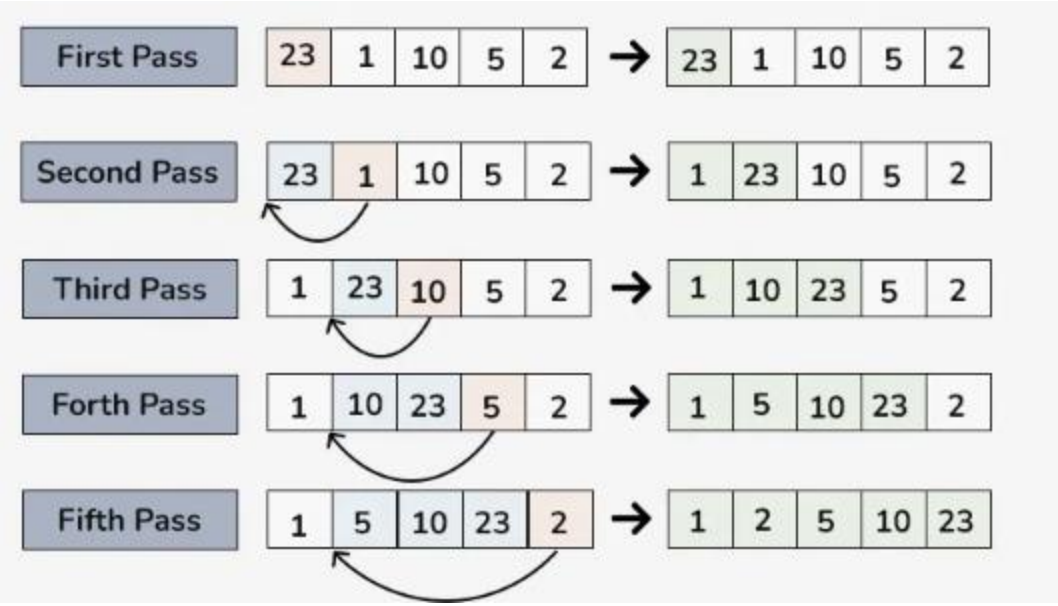
- Suppose we have some unsorted list
- We want to make it sorted



Insertion Sort

- "Naive" sorting algorithm
- One-by-one, take each element and move it
- When all elements have been moved, list is sorted!
- In pseudocode:

```
procedure insertionSort(arr: array of elements)
    n = length(arr)
    for i = 1 to n-1
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key
            arr[j + 1] = arr[j]
            j = j - 1
        arr[j + 1] = key
```



Insertion Sort

- In Psuedocode
  - procedure insertionSort(arr: array of elements)
  - n = length(arr)
  - for i = 1 to n-1
  - key = arr[i]
  - j = i - 1
  - while j >= 0 and arr[j] > key
  - arr[j + 1] = arr[j]
  - j = j - 1
  - arr[j + 1] = key

N steps

N steps

Total: N^2 steps

Time Complexity of Insertion Sort

•**Best case**: O(n), If the list is already sorted, where n is the number of elements in the list.

•**Average case**: $O(n^2)$, If the list is randomly ordered

•**Worst case**: $O(n^2)$, If the list is in reverse order

Insertion Sort

- **Advantages of Insertion Sort:**
  - Simple and easy to implement.
  - Stable sorting algorithm.
  - Efficient for small lists and nearly sorted lists.
  - Space-efficient.
- **Disadvantages of Insertion Sort:**
  - Inefficient for large lists.
  - Not as efficient as other sorting algorithms (e.g., merge sort, quick sort) for most cases.
- **Applications of Insertion Sort:**
  - Insertion sort is commonly used in situations where:
  - The list is small or nearly sorted.
  - Simplicity and stability are important.

Bubble Sort

- Traverse the list, taking pairs of elements
- Swap if order incorrect
- Repeat N times
- Now it's sorted!
- In pseudocode:

```
procedure bubbleSort(arr: array of elements)
    n = length(arr)
    for i = 0 to n-1
        for j = 0 to n-i-1
            if arr[j] > arr[j+1]
                swap arr[j] and arr[j+1]
```

Bubble Sort

- In Psuedocode

```
procedure bubbleSort(arr: array of elements)
    n = length(arr)
    for i = 0 to n-1                          N steps
        for j = 0 to n-i-1                    N steps
            if arr[j] > arr[j+1]
                swap arr[j] and arr[j+1]
```

Total: N^2 steps

Time Complexity of Bubble Sort

- **Best case**: O(n), If the list is already sorted, where n is the number of elements in the list.
- **Average case**: $O(n^2)$
- **Worst case**: $O(n^2)$

Bubble Sort

- **Advantages of Bubble Sort:**
  - Bubble sort is easy to understand and implement.
  - It does not require any additional memory space.
  - It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.
- **Disadvantages of Bubble Sort:**
  - Bubble sort has a time complexity of $O(N^2)$ which makes it very slow for large data sets.
  - Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.

Selection Sort

- The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part.
- This process is repeated for the remaining unsorted portion until the entire list is sorted.
- Now it's sorted!
- In pseudocode:

```
procedure selectionSort(arr: array of elements)
    n = length(arr)
    for i = 0 to n-1
        min_index = i
        for j = i+1 to n
            if arr[j] < arr[min_index]
                min_index = j
        swap arr[i] and arr[min_index]
```

Selection Sort

Time Complexity of Selection Sort

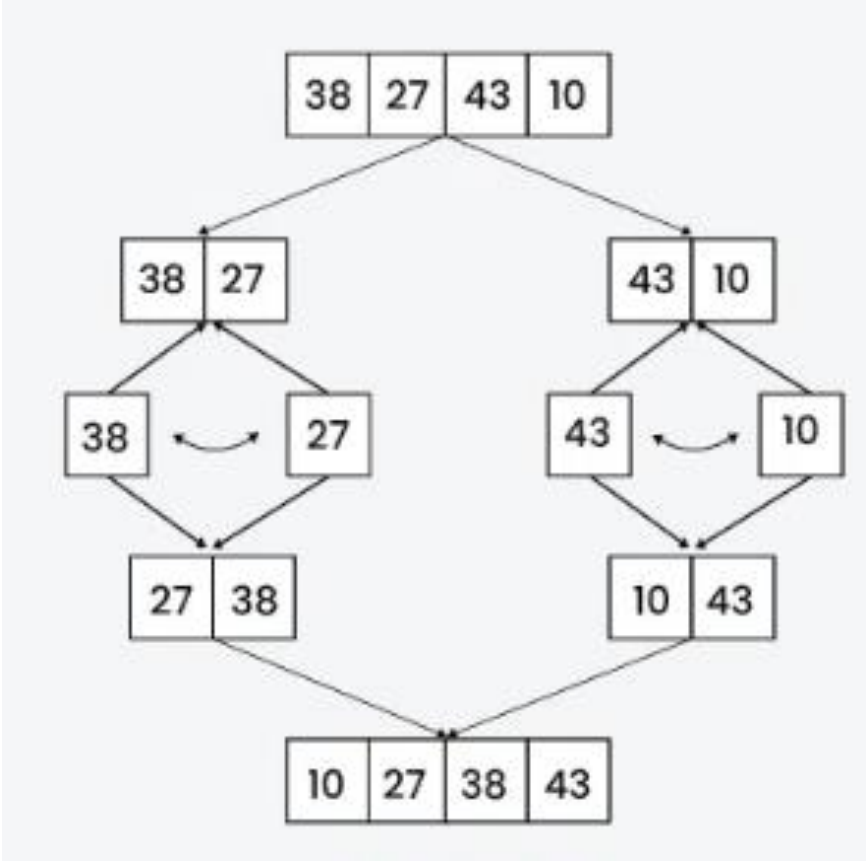
- The time complexity of Selection Sort is $O(N^2)$ as there are two nested loops:
  - One loop to select an element of Array one by one = O(N)
  - Another loop to compare that element with every other Array element = O(N)
- Therefore, overall complexity = O(N) * O(N) = O(N*N) = $O(N^2)$
- **Best case**: O(n2), If the list is already sorted, where n is the number of elements in the list.
- **Average case**: $O(n^2)$
- **Worst case**: $O(n^2)$

Selection Sort

- **Advantages of Selection Sort Algorithm**
  - Simple and easy to understand.
  - Works well with small datasets.
- **Disadvantages of the Selection Sort Algorithm**
  - Selection sort has a time complexity of $O(n^2)$ in the worst and average case.
  - Does not work well on large datasets.
  - Does not preserve the relative order of items with equal keys which means it is not stable.

Divide and Conquer Algorithm

- **Divide and Conquer algorithm** is a problem-solving strategy that involves breaking down a complex problem into smaller, more manageable parts, solving each part individually, and then combining the solutions to solve the original problem.
- It is a widely used algorithmic technique in computer science and mathematics.
- Generic algorithm strategy
  - Divide the problem into smaller parts
  - Solve (conquer) the problem for each part
  - Recombine the parts
- Example: In the Merge Sort algorithm, the “Divide and Conquer” strategy is used to sort a list of elements



Stages of Divide and Conquer Algorithm

- **Divide**:
  - Break down the original problem into smaller subproblems.
  - Each subproblem should represent a part of the overall problem.
  - The goal is to divide the problem until no further division is possible.
- **Conquer**:
  - Solve each of the smaller subproblems individually.
  - If a subproblem is small enough (often referred to as the “base case”), we solve it directly without further recursion.
  - The goal is to find solutions for these subproblems independently.
- **Merge:**
  - Combine the sub-problems to get the final solution of the whole problem.
  - Once the smaller subproblems are solved, we recursively combine their solutions to get the solution of larger problem.
  - The goal is to formulate a solution for the original problem by merging the results from the subproblems.

Applications of Divide and Conquer Algorithm

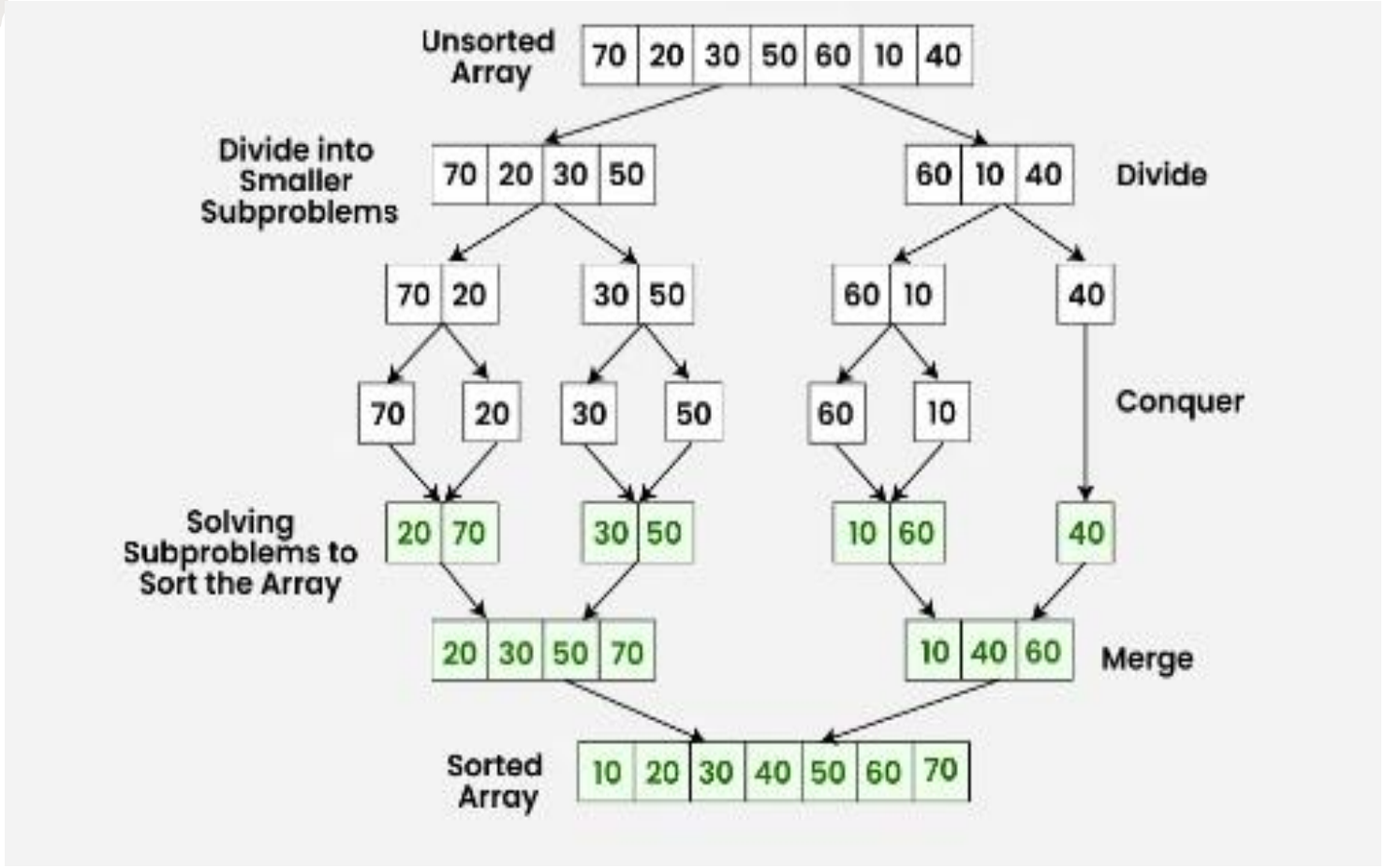
- **Merge Sort:** Merge sort is a classic example of a divide and conquer sorting algorithm. It breaks down the array into smaller subarrays, sorts them individually, and then merges them to obtain the sorted array.
- **Median Finding:** The median of a set of numbers can be found using a divide and conquer approach. By recursively dividing the set into smaller subsets, the median can be determined efficiently.
- **Min and Max finding:** Divide and Conquer algorithm can be used to find both the minimum and maximum elements in an array simultaneously. By splitting the array into halves and comparing the min-max pairs from each half, the overall min and max can be identified in logarithmic time complexity.
- **Matrix Multiplication:** Strassen's algorithm for matrix multiplication is a divide and conquer technique that reduces the number of multiplications required for large matrices by breaking down the matrices into smaller submatrices and combining their products.
- **Closest Pair problem:** The closest pair problem involves finding the two closest points in a set of points in a multidimensional space. A divide and conquer algorithm, such as the "divide and conquer closest pair" algorithm, can efficiently solve this problem by recursively dividing the points and merging the solutions from the subproblems.

Merge Sort

- The algorithm 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.
- The process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together.
- This process is repeated until the entire array is sorted.

Example of Merge Sort

- Let us look at an example:



Merge Sort

- In pseudocode:

```
procedure mergeSort(arr: array of elements)
    if length(arr) <= 1
        return arr
    mid = length(arr) / 2
    left_half = mergeSort(arr[0:mid])
    right_half = mergeSort(arr[mid:])
    return merge(left_half, right_half)

procedure merge(left_half: array of elements, right_half: array of elements)
    result = []
    while left_half is not empty and right_half is not empty
        if left_half[0] <= right_half[0]
            append left_half[0] to result
            remove first element from left_half
        else
            append right_half[0] to result
            remove first element from right_half
    append remaining elements of left_half and right_half to result
    return result
```

Merge Sort

Time Complexity of Quick Sort

- **Best case**: O(n log n),
- **Average case**: O(n log n)
- **Worst case**: O(n log n)

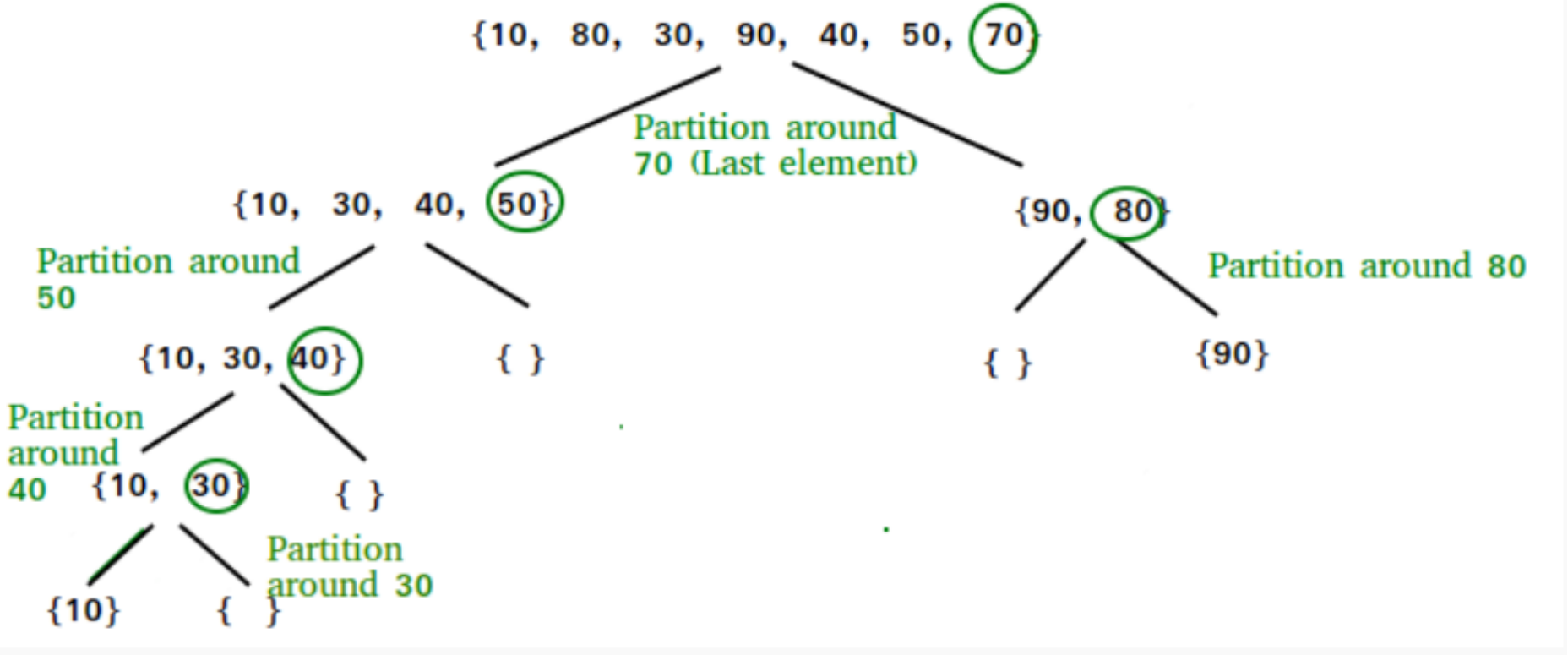
- **Advantages of Merge Sort:**
  - Stability: Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
  - Guaranteed worst-case performance: Merge sort has a worst-case time complexity of O(N logN), which means it performs well even on large datasets.
  - Simple to implement: The divide-and-conquer approach is straightforward.
- **Disadvantage of Merge Sort:**
  - Space complexity: Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
  - Not in-place: Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- **Applications of Merge Sort:**
  - Sorting large datasets
  - External sorting (when the dataset is too large to fit in memory)
  - Inversion counting (counting the number of inversions in an array)
  - Finding the median of an array

Quick Sort

•The algorithm 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.
•The key process in quickSort is a partition().
•The target of partitions is to place the pivot (any element can be chosen to be a 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.

Example of Quick Sort

- Let us look at an example:



Quick Sort

- In pseudocode:

```
procedure quickSort(arr: array of elements, low: integer, high: integer)
    if low < high
        pivot_index = partition(arr, low, high)
        quickSort(arr, low, pivot_index - 1)
        quickSort(arr, pivot_index + 1, high)

procedure partition(arr: array of elements, low: integer, high: integer)
    pivot = arr[high]
    i = low - 1
    for j = low to high - 1
        if arr[j] < pivot
            i = i + 1
            swap arr[i] and arr[j]
    swap arr[i + 1] and arr[high]
    return i + 1
```

Quick Sort

Time Complexity of Quick Sort

- **Best case**: O(n log n) - The best-case scenario for quicksort occur when the pivot chosen at each step divides the array into roughly equal halves.In this case, the algorithm will make balanced partitions, leading to efficient Sorting.
- **Average case**: O(n log n) - Quicksort's average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.
- **Worst case**: O(n^2) - The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element. To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (Randomized Quicksort) to shuffle the element before sorting.

- **Advantages of Quick Sort:**
  - It is a divide-and-conquer algorithm that makes it easier to solve problems.
  - It is efficient on large data sets.
  - It has a low overhead, as it only requires a small amount of memory to function.
- **Disadvantage of Merge Sort:**
  - It has a worst-case time complexity of O(N2), which occurs when the pivot is chosen poorly.
  - It is not a good choice for small data sets.
  - It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

Heap Sort

•Heap sort is a comparison-based sorting technique based on Binary Heap data structure.
•It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning.
•Repeat the same process for the remaining elements.

•We will discuss this topic during the Module – Non-Linear Data Structures

Counting Sort

•**Counting Sort** is a non-comparison-based sorting algorithm that works well when there is limited range of input values.
•It is particularly efficient when the range of input values is small compared to the number of elements to be sorted.
•The basic idea behind Counting Sort is to count the frequency of each distinct element in the input array and use that information to place the elements in their correct sorted positions.

Counting Sort

- In pseudocode:

```
procedure countingSort(arr: array of elements)
    max_val = maximum value in arr
    counts = array of size max_val + 1, initialized to 0
    output = array of size length(arr)

    for num in arr
        counts[num] += 1

    for i = 1 to max_val
        counts[i] += counts[i - 1]

    for i = length(arr) - 1 down to 0
        output[counts[arr[i]] - 1] = arr[i]
        counts[arr[i]] -= 1

    return output
```

Counting Sort

Complexity Analysis of Counting Sort

- **Time Complexity:** O(N+M), where N and M are the size of **inputArray[]** and **countArray[]** respectively.
  - **Worst-case:** O(N+M).
  - **Average-case:** O(N+M).
  - **Best-case:** O(N+M).
- **Auxiliary Space:** O(N+M), where N and M are the space taken by **outputArray[]** and **countArray[]** respectively.

- **Advantages of Counting Sort:**
  - Counting sort generally performs faster than all comparison-based sorting algorithms, such as merge sort and quicksort, if the range of input is of the order of the number of input.
  - Counting sort is easy to code
  - Counting sort is a **stable algorithm**.
- **Disadvantage of Counting Sort:**
  - Counting sort doesn't work on decimal values.
  - Counting sort is inefficient if the range of values to be sorted is very large.
  - Counting sort is not an **In-place sorting algorithm**, It uses extra space for sorting the array elements.

Bucket Sort

- **Bucket sort** is a sorting technique that involves dividing elements into various groups, or buckets.
- These buckets are formed by uniformly distributing the elements.
- Once the elements are divided into buckets, they can be sorted using any other sorting algorithm.
- Finally, the sorted elements are gathered in an ordered fashion.
- Create n empty buckets (Or lists) and do the following for every array element arr[i].
    - Insert arr[i] into bucket[n*array[i]]
    - Sort individual buckets using insertion sort.
    - Concatenate all sorted buckets.

Bucket Sort

- In pseudocode:

```
procedure bucketSort(arr: array of elements)
    max_val = maximum value in arr
    min_val = minimum value in arr
    bucket_range = (max_val - min_val) / length(arr)

    buckets = array of empty lists of size length(arr)

    for num in arr
        index = (num - min_val) / bucket_range
        append num to buckets[index]

    sorted_arr = []
    for bucket in buckets
        sorted_arr.extend(sort(bucket))

    return sorted_arr
```

Complexity Analysis of Bucket Sort

- **Worst-case:** $O(N^2)$.
- **Average-case:** O(N+M).
- **Best-case:** O(N+M).

Thank you