LESSON 1
Topic: Divide and Conquer I (sorting and selection)

1) Divide and Conquer (5 minutes)

The following three elements make up this technique:

Divide: This entails breaking the issue up into smaller issues.
Conquer: Call recursively until the subproblem is resolved.
Combine the subproblems to arrive at the overall problem's definitive answer.

Divide And Conquer Algorithm:

```
divideAndConquer(a, i, j)
{
   if(small(a, i, j))
     return(Solution(a, i, j))
   else
     mid = divide(a, i, j)            // f1(n)
     b = divideAndConquer (a, i, mid)          // T(n/2)
     c = divideAndConquer (a, mid+1, j)        // T(n/2)
     d = combine(b, c)              // f2(n)
  return(d)
}
```

The following are some standard algorithms that follow Divide and Conquer algorithm.

1. Merge Sort
2. Quick Sort
3. Closet Pair of Points
4. Strassen's Algorithm
5. Karatsuba Algorithm for Fast Multiplication

Time Complexity:

The divide and conquer algorithm, which determines the highest and lowest member in an array, has an O(n) time complexity. This is due to the fact that splitting the array in half

each time results in a total of log(n) divisions. We compare two elements in each division to determine the highest and lowest element, which is a constant-time operation. The entire time complexity is therefore O(n*log(n)).

Space Complexity:

The divide and conquer method, which determines the highest and lowest element in an array, has a space complexity of O(log(n)). This is so that we may break the array up into smaller pieces using recursion, which uses up space on the call stack with each call. The number of times we can divide the array in half determines the maximum depth of the recursion tree, which is log(n). As a result, the complexity of space is O(log(n)).

Let's discuss some of the Divide and Conqure Algorithms.

2) Merge Sort (10 minutes)

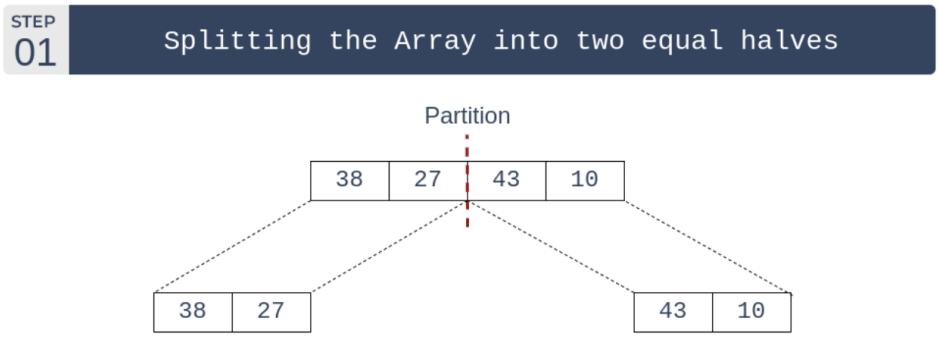
A sorting technique known as merge sort divides a large array into smaller subarrays, sorts each subarray individually, and then merges the sorted subarrays back together to create the final sorted array.

How does the merge sort function?
A recursive technique called merge sort constantly divides an array in half until it can have only one element remaining (an array with one member is always sorted). The sorted subarrays are then combined into a single array.

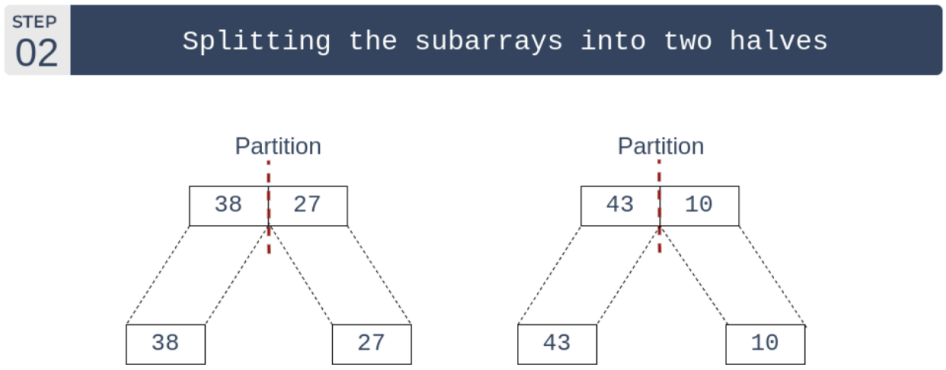
Example:

Lets consider an array arr[] = {38, 27, 43, 10}

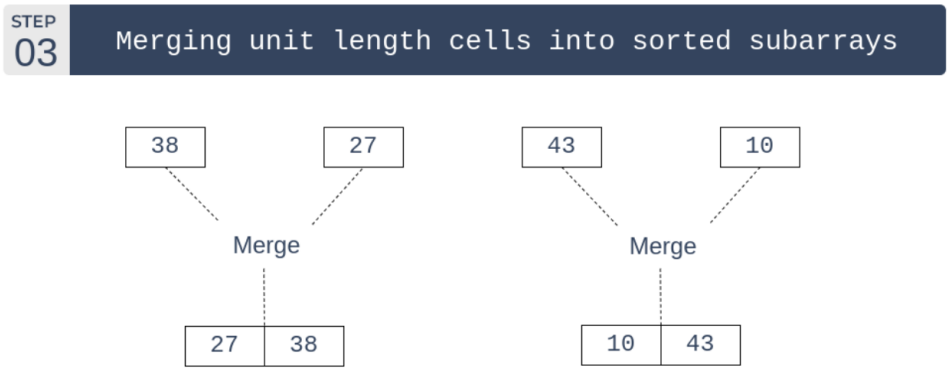
Initially divide the array into two equal halves:



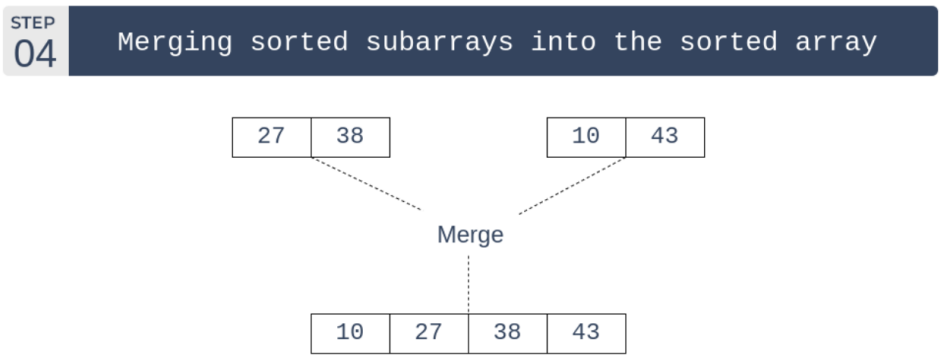
These subarrays are further divided into two halves. Now they become array of unit length that can no longer be divided and array of unit length are always sorted.



We combine these sorted subarrays to create larger sorted subarrays.



This merging procedure is carried out repeatedly until the smaller subarrays are combined to form the sorted array.



Applications of Merge Sort:

1. Sorting Large Datasets: Merge sort's guaranteed worst-case time complexity of O(n log n) makes it especially well-suited for sorting Large Datasets.

2. Merge sort is frequently used in external sorting, which is done when the data that has to be sorted is too big to fit in memory.

3. Merge sort can be customized to handle a variety of input distributions, including partially sorted, almost sorted, and entirely unsorted data.

Issues with Merge Sort:

1. Space complexity: During the sorting process, the combined sub-arrays from the merge sort must be stored in additional memory.

2. Not present: Merge sort takes additional RAM to hold the sorted data because it is not an in-place sorting method. In applications where memory usage is an issue, this could be a drawback.

3. For tiny datasets, it's not necessarily best: Merge sort has a higher time complexity than some other sorting algorithms, such insertion sort, for small datasets. This may cause performance to be slower for very tiny datasets.

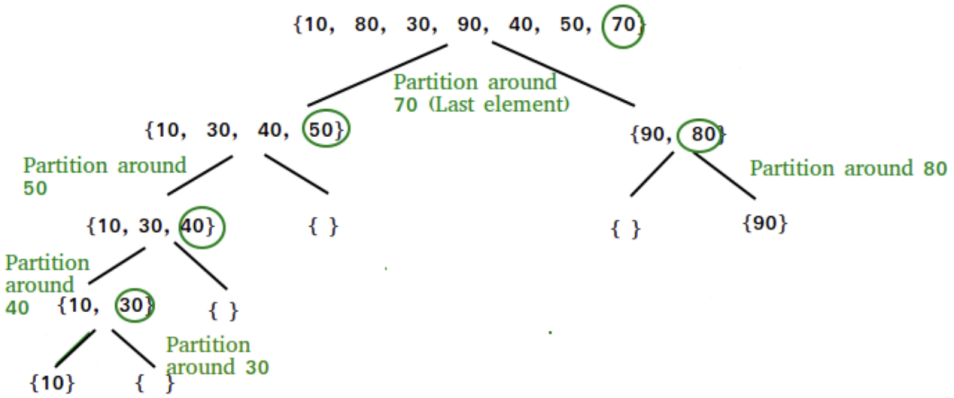
3) Quick Sort (10 minutes)

QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that selects an element as a pivot and partitions the provided array around the picked pivot by positioning the pivot in the sorted array's correct location.

How is QuickSort put to use?

Partition is the main quickSort operation. The goal of partitioning is to arrange all smaller elements to the left of the pivot and all larger elements to the right of the pivot in the sorted array. Any element can be chosen as the pivot.

Once the pivot is at the proper position, partition is carried out recursively on either side of the pivot, and this eventually sorts the array.



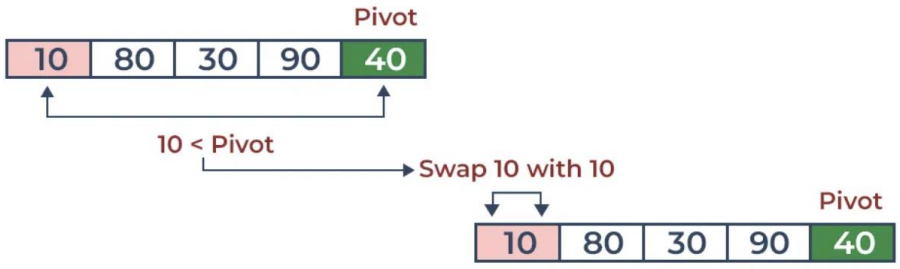
Selection of a Pivot:

There are numerous options available when selecting a pivot.

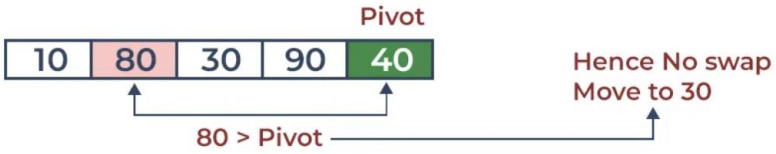
1. As a rule, choose the first component as the pivot.
2. Always choose the last element to serve as a pivot (as shown below)
3. Choose a pivot that is at random.
4. Decide on the center as the pivot.

Example:

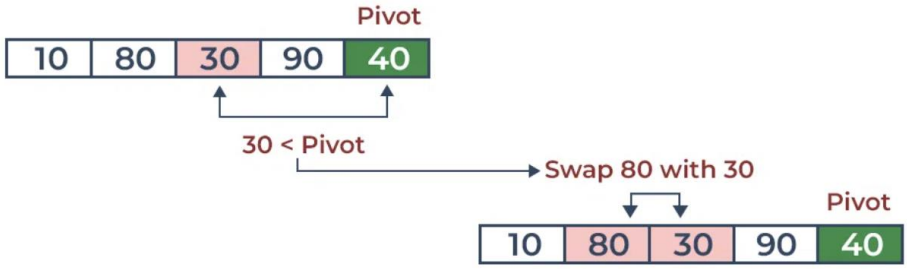
Consider: arr[] = {10, 80, 30, 90, 40}.
Compare 10 to the pivot and arrange it accordingly since it is less than the pivot.



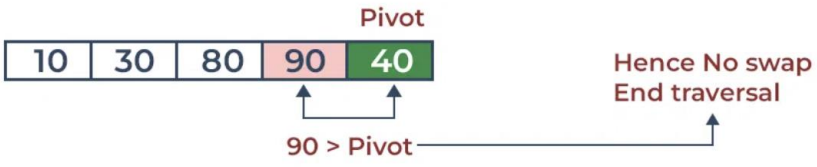
Compare 80 with the pivot. It is greater than pivot.



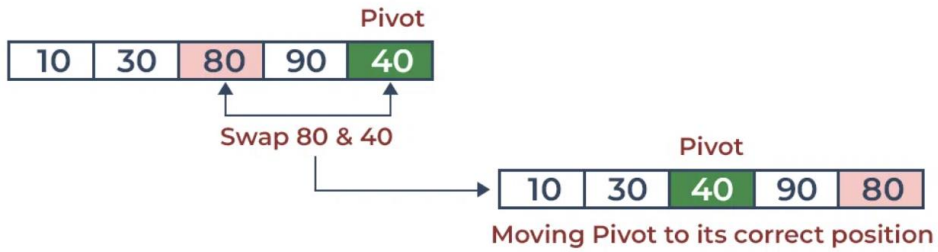
Compare 30 with pivot. It is less than pivot so arrange it accordingly.



Compare 90 with the pivot. It is greater than the pivot



Arrange the pivot in its correct position.



The pivot is continually placed in its actual location in the sorted array as the partition procedure is performed recursively. The array is sorted by repeatedly setting pivots to their actual positions.

Video References (15 minutes)

- Divide and Conquer
https://www.youtube.com/watch?v=Pu2KeAy8BGY&ab_channel=VinsloevAcademy

- Merge Sort
https://www.youtube.com/watch?v=4VqmGXwpLqc&ab_channel=MichaelSambol

- Quick Sort
https://www.youtube.com/watch?v=ZHVk2blR45Q&ab_channel=RobEdwards