# Lecture

**NOTE:** FOR FURTHER DETAILS AND MORE COMPREHENSIVE STUDY, PLEASE SEE RECOMMENDED BOOKS OR INTERNET.

## Sorting

Sorting is nothing but arranging the data in ascending or descending order. The term ***sorting*** came into picture, as humans realized the importance of searching quickly.

## Merge Sort

Merge Sort follows the rule of Divide and Conquer to sort a given set of numbers/elements, recursively, hence consuming less time.

In the last two tutorials, we learned about Selection Sort and Insertion Sort, both of which have a worst-case running time of $O(n^2)$. As the size of input grows, insertion and selection sort can take a long time to run.

Merge sort, on the other hand, runs in $O(n \log n)$ time in all the cases.

Before jumping on to, how merge sort works and its implementation, first let's understand what the rule of ***Divide and Conquer*** is?

### Divide and Conquer

If we can break a single big problem into smaller sub-problems, solve the smaller sub-problems and combine their solutions to find the solution for the original big problem, it becomes easier to solve the whole problem. Let's take an example, Divide and Rule.

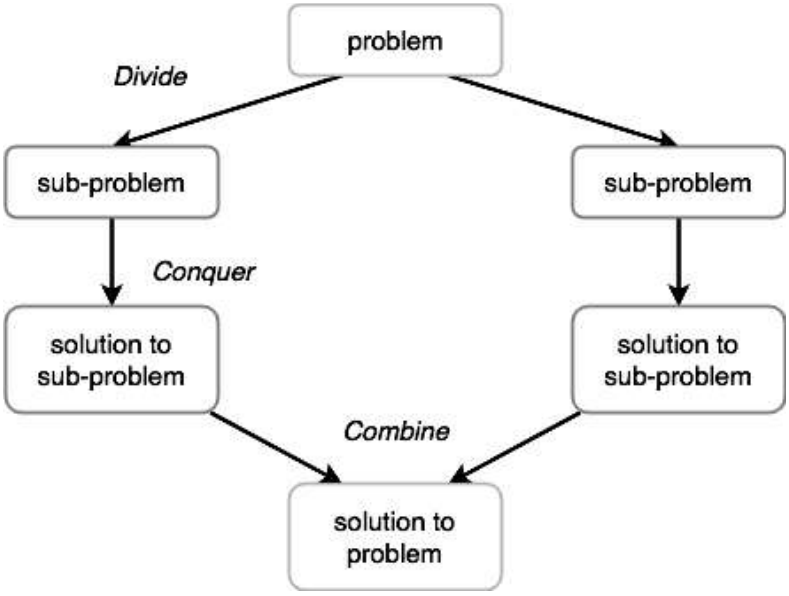
When British rule came to Sub-continent of India, they saw a country with different religions living in harmony, hard working but naive citizens, unity in diversity, and found it difficult to establish their empire. So, they adopted the policy of Divide and Rule. Where the population of region was collectively a one big problem for them, they divided the problem into smaller problems, by instigating rivalries between local kings, making them stand against each other, and this worked very well for them.

Well that was history, and a socio-political policy (***Divide and Rule***), but the idea here is, if we can somehow divide a problem into smaller sub-problems, it becomes easier to eventually solve the whole problem.

In **Merge Sort**, the given unsorted array with ***n*** elements is divided into ***n*** sub-arrays, each having one element, because a single element is always sorted in itself. Then, it repeatedly merges these sub-arrays, to produce new sorted sub-arrays, and in the end, one complete sorted array is produced.

The concept of Divide and Conquer involves three steps:

1. ***Divide*** the problem into multiple small problems.
2. ***Conquer*** the sub-problems by solving them. The idea is to break down the problem into atomic sub-problems, where they are actually solved.
3. ***Merge*** the solutions of the sub-problems to find the solution of the actual problem.




**Example:** Consider the following array of numbers ***red*** is the ***divide part*** and ***green*** is the ***merge part***

| 27 | 10 | 12 | 25 | 34 | 16 | 15 | 31 |
|---|---|---|---|---|---|---|---|

**Divide** it into two sub parts

| 27 | 10 | 12 | 25 |
|---|---|---|---|

| 34 | 16 | 15 | 31 |
|---|---|---|---|

Divide each part into two sub parts

| 27 | 10 |
|---|---|

| 12 | 25 |
|---|---|

| 34 | 16 |
|---|---|

| 15 | 31 |
|---|---|

Divide each part into two sub parts

| 27 |
|---|

| 10 |
|---|

| 12 |
|---|

| 25 |
|---|

| 34 |
|---|

| 16 |
|---|

| 15 |
|---|

| 31 |
|---|

**Merge** parts (smartly)

| 10 | 27 |
|---|---|

| 12 | 25 |
|---|---|

| 16 | 34 |
|---|---|

| 15 | 31 |
|---|---|

Merge parts

| 10 | 12 | 25 | 27 |
|---|---|---|---|

| 15 | 16 | 31 | 34 |
|---|---|---|---|

Merge parts into one

| 10 | 12 | 15 | 16 | 25 | 27 | 31 | 34 |
|---|---|---|---|---|---|---|---|

## Code for Merge Sort:

Below is the code for methods of Merge Sort.

```
public void mergeSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int mid = (low + high) / 2;

        mergeSort(arr, low, mid);
        mergeSort(arr, mid + 1, high);
        merge(arr, low, mid, high);
    }
}

private void merge(int[] arr, int low, int mid, int high)
{
    int[] merged = new int[arr.length];
    int i, j, k;
    k = 0;
    i = low;
    j = mid + 1;
    while (i <= mid && j <= high)
    {
        if (arr[i] < arr[j])
        {
            merged[k] = arr[i];
            k++;
            i++;
        }
        else
        {
            merged[k++] = arr[j++];
        }
    }

    while (i <= mid)
    {
        merged[k++] = arr[i++];
    }

    while (j <= high)
    {
        merged[k++] = arr[j++];
    }

    for (i = high; i >= low; i--)
```

```
    {
        arr[i] = merged[--k];
    }
}
```

## Quick Sort

Quick Sort is also based on the concept of ***Divide and Conquer***, just like merge sort. But in ***quick sort*** all the major work is done while ***dividing the array*** into sub-arrays, while in case of ***merge sort***, all the real work happens during ***merging*** the sub-arrays. In case of quick sort, the combine step does absolutely nothing.

It is also called partition-exchange sort. This algorithm divides the list into three main parts:

1. **Pivot** element (Central element)
2. Elements less than the ***Pivot*** element
3. Elements greater than the ***Pivot*** element

Pivot element can be any element from the array, it can be the first element, the last element or any random element. In this class, we will take the ***middle*** element as ***pivot***.

### How Quick Sorting Works?

Following are the steps involved in quick sort algorithm:

1. Choose a **pivot** value (in this case, choose the value of the **middle element**)
2. Initialize left (***i***) and right (***j***) pointers at both ends.
3. Starting at the left pointer and moving to the right, find the first element which is ***greater than pivot*** value.
4. Similarly, starting at the right pointer and moving to the left, find the first element, which is ***smaller than pivot*** value.
5. **Swap** elements found in 3 and 4.
6. Repeat step 3, 4, 5 until left pointer is greater or equal to right pointer.
7. Repeat the whole thing for the two sub-array, array left of the right pointer (***low to j***) and array right of the left pointer (***i to high***).
8. 

### Complexity Analysis of Quick Sort

For an array, in which partitioning leads to unbalanced subarrays, to an extent where on the left side there are no elements, with all the elements greater than the pivot, hence on the right side.

If keep on getting unbalanced subarrays, then the running time is the worst case, which is $O(n^2)$. Where as if partitioning leads to almost equal subarrays, then the running time is the best, with time complexity as $O(n \log n)$.

**Example:** Consider array of numbers {52, 37, 63, 14, 17, 8, 6, 25}

We color left pointer ***i*** as ***green***, right pointer ***j*** as ***red***, and pivot ***p*** as ***blue***. Combine color shows the location of multiple pointers.

Quick Sort (52, 37, 63, 14, 17, 8, 6, 25)

**52, 37, 63, 14, 17, 8, 6, 25**
**52, 37, 63, 14, 17, 8, 6, 25**
*Swap*
**6, 37, 63, 14, 17, 8, 52, 25**
*Swap*
**6, 8, 63, 14, 17, 37, 52, 25**
**6, 8, 63, 14, 17, 37, 52, 25**
**6, 8, 63, 14, 17, 37, 52, 25**
**6, 8, 63, 14, 17, 37, 52, 25**

***i*** *>* ***j*** *therefore, stop and recursive call*

Quick Sort (6, 8)
Quick Sort (63, 14, 17, 37, 52, 25)

## Code for Quick Sort:

Below is the code for a method of Quick Sort.

```
public void quickSort(int[] arr, int low, int high)
{
    if(low>=high)
    {
        return;
    }
    int mid = (low + high) / 2;
    int pivot = arr[mid];

    int i = low, j = high;
    while (i <= j)
    {
        while (arr[i] < pivot)
        {
            i++;
        }
        while (arr[j] > pivot)
        {
            j--;
        }
        if (i <= j)
        {
            int temp = arr[i];
```

```
            arr[i] = arr[j];
            arr[j] = temp;
            i++;
            j--;
        }
    }
    quickSort(arr, low, j);
    quickSort(arr, i, high);
}
```