DIVIDE & CONQUER ALGORITHMS

* Divide & Conquer algorithms are problem-solving techniques when you break down a problem into smaller, more manageable subproblems, solve each subproblem independently.
* And then combine their solutions to solve the original problem.

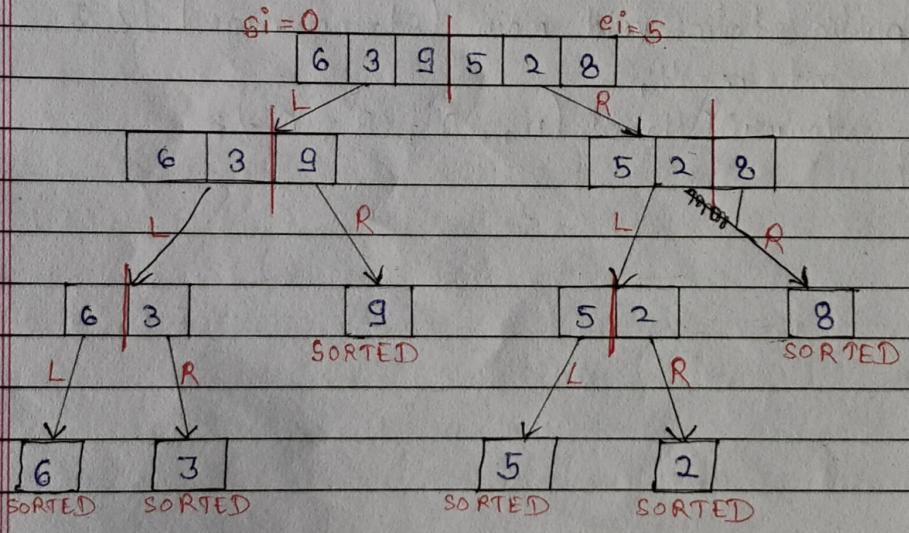
★ MERGE SORT

* Time Complexity, $\Rightarrow O(n \log n)$

Approach: S1. Divide $\rightarrow mid = (start\ index + end\ index)/2$

or

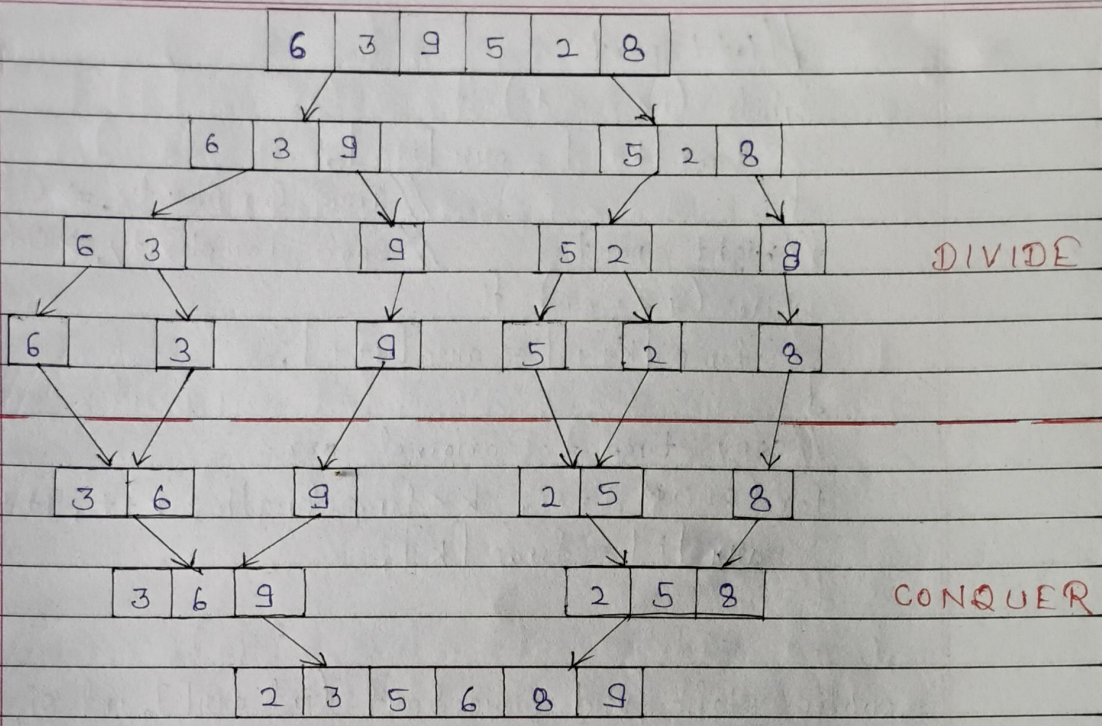
$\rightarrow mid = start\ index + (end\ index - start\ index)/2$



S2. Firstly, mergeSort (left)
then mergeSort (right)

S3: mergesort (left) + mergeSort (right) inside temp[].

2	3	5	6	8	9
0	1	2	3	4	5



⇒
```
public class MergeSort {
    public static void merge (int arr[], int si, int mid, int ei[]
        int temp[] = new int [ei-si+1];
        int i = si; //iterator for left part
        int j = mid+1; //iterator for right part
        int k = 0; //iterator for temp arr
        while (i<=mid && j<=ei) {
            if (arr[i] < arr[j]) {
                temp[k] = arr[i];
                i++;
            } else {
                temp[k] = arr[j];
                j++;
            }
            k++;
        }
```

// Next page.

```
    //left part
    while (i<=mid) {
        temp[k++] = arr[i++];
    }                          // Time Complexity, ⇒ O(nlogn)
    //right part               // Space Complexity, ⇒ O(n)
    while (j<=ei) {
        temp[k++] = arr[j++];
    }
    //copy temp to original arr
    for (k=0, i=si; k<temp.length; k++, i++) {
        arr[i] = temp[k];
    }
}

public static void mergeSort (int arr[], int si, int ei){
    if (si >= ei) {
        return;
    }
    int mid = si + (ei-si)/2; //(si+ei) ÷ 2
    mergeSort (arr, si, mid); //left part
    mergeSort (arr, mid+1, ei); //right part
    merge (arr, si, mid, ei);
}

public static void printArr (int arr[]) {
    for (int i=0; i<arr.length; i++) {
        System.out.print (arr[i]+" ");
    }
    System.out.println();
}

public static void main (String args[]) {
    int arr[] = {6, 3, 9, 5, 2, 8};
    mergeSort (arr, 0, arr.length-1);
    printArr (arr);
}
}
```

★ QUICK SORT

* Time Complexity → average case ⇒ $O(n \log n)$
  → worst case ⇒ $O(n^2)$
* Space Complexity ⇒ $O(1)$
* Quick Sort uses Pivot & Partition Approach.

S1 → Find the Pivot element.

6	3	9	8	2	5

वो element जिसके आस पास पूरा Array घूमे।

i.e., middle element, first element or last element.

* Our pivot element is 5.

S2 → Let's do Partition (means parts).

3 2 < 5 < 6 9 8 → HOW?

quick sort (Left) | quick sort (Right)

It's the main part to do inside code.

Working:

si=0, ei=5

6	3	9	8	2	5

j=0 j=1 j=2 j=3 j=4 j=5

* pivot = 5.

i = ~~-1~~ 0 ~~1~~ 4

3	2	5		6	9
0	1	2	3	4	5
(6)	(3)	(9)	(8)	(2)	(5)

```
i++;
int temp = arr[j];
arr[i] = arr[i];
arr[i] = temp;
```

* After swapping all the numbers then

2	3	5	6	8	9

⇒
```
public class DivideConquer {
    public static void printArr(int arr[]) {
        for (int i=0; i<arr.length; i++) {
            System.out.print(arr[i]+" ");
        }
        System.out.println();
    }
    public static void quickSort(int arr[], int si, int ei) {
        if (si >= ei) {
            return;
        }
        // last element
        int pIdx = partition(arr, si, ei);
        quickSort(arr, si, pIdx-1); // left
        quickSort(arr, pIdx+1, ei); // right
    }
    public static int partition(int arr[], int si, int ei){
        int pivot = arr[ei];
        int i = si-1; // to make place for elements smaller
                      than pivot.
        for (int j=si; j<ei; j++) {
            if (arr[j] <= pivot) {
                i++;
                //swap
                int temp = arr[j];
                arr[j] = arr[i];
                arr[i] = temp;
            }
        }
        i++;
        int temp = pivot;
        arr[ei] = arr[i];
        arr[i] = temp;
        return i;
```

```
    }
    public static void main (String args[]) {
        int arr [] = {6, 3, 9, 8, 2, 5};
        quickSort (arr, 0, arr.length - 1);
        printArr (arr);
    }
}
```

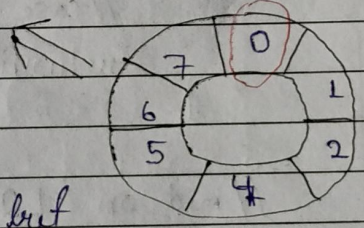
#NOTE: Worst case occurs when pivot is always the smallest or the largest element, i.e, $\Rightarrow O(n^2)$

★ SEARCH IN ROTATED SORTED ARRAY

input: sorted & rotated array with distinct numbers (in ascending order). It is rotated at a pivot point. Find the index of given element.

4	5	6	7	0	1	2
0	1	2	3	4	5	6

*target: 0



output: 4

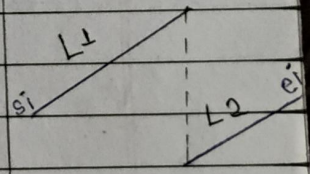
We will use binary search but in a modified way.

4	5	6	7	0	1	2
0	1	2	3	4	5	6

mid

Case 1: mid on L1 → (arr [si] <= mid)
- case a: L1 left (si ≤ tar ≤ mid)
- case b: mid right

Case 2: mid on L2 → (arr [mid] ≤ arr [ei])
- case c: L1 right (mid ≤ tar ≤ ei)
- case d: mid left



⇒
```
public class DividerConquer {
    public static int search (int arr[], int tar, int si, int ei){
        if (si > ei) {
            return -1;
        }
        // kaam
        int mid = si + (ei - si)/2; //or (si+ei)/2

        // Case found
        if (arr[mid] == tar) {
            return mid;
        }
        // mid on L1
        if (arr[si] <= arr[mid]) {
            //case a: Left
            if (arr[si] <= tar && tar <= arr[mid]) {
                return search (arr, tar, si, mid-1);
            } else {
                //case b: right
                return search (arr, tar, mid+1, ei);
            }
        }
        // mid on L2
        else {
            //case c: right
            if (arr[mid] <= tar && tar <= arr[ei]) {
                return search (arr, tar, mid+1, ei);
            } else {
                //case d: Left
                return search (arr, tar, si, mid-1);
            }
        }
    }
```

```
public static void main (String args[]) {
    int arr [] = {4, 5, 6, 7, 0, 1, 2};
    int target = 0;
    int tarIdx = search (arr, target, 0, arr.length-1);
    System.out.println (tarIdx);
  }
}
```

⇒ 4

PROBLEMS

- **Problem - 1**

Apply merge sort to sort an array of Strings. Assume that all the characters in all the strings are in lower case.

input - arr = {"sun", "earth", "mars", "mercury"}

output - arr = {"earth", "mars", "mercury", "sun"}

Solution:

```
public class Solution {
  public static String[] mergeSort (String[] arr, int lo, int hi) {
    if (lo == hi) {
      String[] A = { arr[lo] };
      return A;
    }
    int mid = lo + (hi - lo) / 2;
    String[] arr1 = mergeSort (arr, lo, mid);
    String[] arr2 = mergeSort (arr, mid+1, hi);
    String[] arr3 = merge (arr1, arr2);
    return arr3;
  }
  public static String[] merge (String[] arr1, String[] arr2) {
    int m = arr1.length;
    int n = arr2.length;
    String[] arr3 = new String [m+n];
```

```
int idx = 0;
int i = 0;
int j = 0;
while (i < m && j < n) {
    if (isAlphabeticallySmaller(arr1[i], arr2[j])) {
        arr3[idx] = arr1[i];
        i++;
        idx++;
    }
    else {
        arr3[idx] = arr2[i];
        j++;
        idx++;
    }
}
while (i < m) {
    arr3[idx] = arr1[i];
    i++;
    idx++;
}
while (j < n) {
    arr3[idx] = arr2[j];
    j++;
    idx++;
}
return arr3;
}
public static boolean isAlphabeticallySmaller(String str1, String str2) {
    if (str1.compareTo(str2) < 0) {
        return true;
    }
    return false;
}
```

```
public static void main (String[] args) {
    String [] arr = {"sun", "earth", "mars", "mercury"};
    String [] a = mergeSort (arr, 0, arr.length-1);
    for (int i=0; i< a.length; i++ ) {
        System.out.println (a[i]);
    }
  }
}
```

- Problem - 2

Given an array nums of size n, return the majority element. The majority element is the element that appears more than $[n/2]$ times. We may assume that the majority element always exits in the array.

input - nums=[3,2,3]
output= 3

input - nums = [2,2,1,1,1,2,2]
output= 2

Solutions: Let's count the no. of times each number occurs in the array & find the largest count.

// Time Complexity, $O(n^2)$

Approach 1 - Brute Force Approach

⇒
```
public class Solution {
    public static int majorityElement (int num[]) {
        int majorityCount = nums.length /2;
        for (int i=0; i<nums.length; i++) {
            int count =0;
            for (int j=0; j< nums.length; j++) {
                if (nums[j] == nums [i]) {
                    count += 1;
                }
            }
        }
```

```
        if (count > majorityCount) {
            return nums[i];
        }
    }
    return -1;
}

public static void main (String args[]) {
    int nums[] = {2,2,1,1,1,2,2};
    System.out.println (majorityElement(nums));
}
}
```

Approach 2 - Divide & Conquer

* If we know the majority element in the left & right halves of an array, we can determine which is the majority element in linear time.
// Time Complexity, O(n log n)

⇒
```
public class Solution {
    private static int countInRange (int nums[], int num, int lo, int hi) {
        int count = 0;
        for (int i = lo; i <= hi; i++) {
            if (nums[i] == num) {
                count++;
            }
        }
        return count;
    }
    private static int majorityElementRec (int nums[], int lo, int hi) {
        if (lo == hi) {
            return nums[lo]; //base case
        }
```

```
//recurse on left & right halves of this slice
int mid = (hi-lo)/2 + lo;
int left = majorityElementRec(nums, lo, mid);
int right = majorityElementRec(nums, mid+1, hi);
//if the two halves agree on majority element, return it
if (left == right) {
    return left;
}
//otherwise, count each element & return the "winner".
int leftCount = countInRange(nums, left, lo, hi);
int rightCount = countInRange(nums, right, lo, hi);
return leftCount > rightCount ? left : right;
}
public static int majorityElement(int nums[]) {
    return majorityElementRec(nums, 0, nums.length-1);
}
public static void main(String args[]) {
    int nums[] = {2,2,1,1,1,2,2};
    System.out.println(majorityElement(nums));
}
}
```

- **Problem - 3**

Given an array of integers. Find the Inversion Count in the array.

* **Inversion Count :** For an array, it indicates how far or close the array is from being sorted. If the array is already sorted then the inversion count is 0. If an array is sorted in the reverse order then the inversion count is maximum.

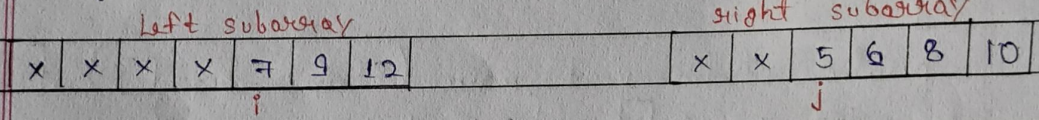
* Formally, two elements a[i] & a[j] form an inversion if a[i] > a[j] & i < j.

input - N=5, arr[] = {2,4,1,3,5}
output - 3 becoz it has 3 inversions - (2,1) (4,1) (4,3)

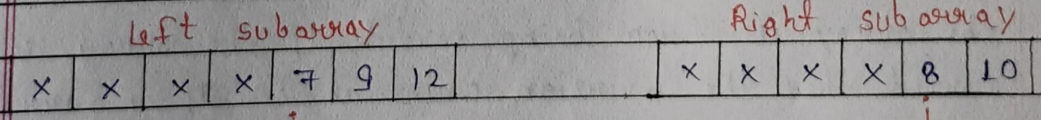
input - N=5, arr[] = {2,3,4,5,6}, input - N=3, arr[] = {5,5,5}
output - 0, becoz already sorted, output - 0, becoz already sorted

Approach - Suppose the no. of inversions in the left half & right half of the array (let be inv1 & inv2).

* To get the total no. of inversions that need to be added are the no. of inversions in the left subarray, right subarray & merge().
* Basically, for each array element count all elements more than it to its left & add the count to the output.
* The whole magic happens inside the merge() of merge sort.



* As 7 > 5, (7,5) pair forms an inversion pair.
* Also, as left subarray is sorted, it is obvious that elements 9 & 12 will also form inversion pairs with element 5 i.e., (9,5) & (12,5).
* So, we can say that for element 5, total inversions are 3, which is exactly equal to the no. of elements left in the left subarray.



7 < 8, no inversion found.

⇒ //Time Complexity - O(nlogn)

```
public class Solution {
    public static int merge (int arr[], int left, int mid, int right) {
        int i = left, j = mid, k = 0;
        int invCount = 0;
        int temp[] = new int [(right - left + 1)];
        while ((i < mid) && (j <= right)) {
            if (arr[i] <= arr[j]) {
                temp[k] = arr[i];
                k++;
                i++;
            } else {
                temp[k] = arr[j];
                invCount += (mid - i);
                k++;
                j++;
            }
        }
        while (i < mid) {
            temp[k] = arr[i];
            k++;
            i++;
        }
        while (j <= right) {
            temp[k] = arr[j];
            k++;
            j++;
        }
        for (i = left; k = 0; i <= right; i++, k++) {
            arr[i] = temp[k];
        }
        return invCount;
    }
```

```
private static int mergeSort (int arr[], int left, int right) {
    int invCount = 0;
    if (right > left) {
        int mid = (right + left) / 2;
        invCount = mergeSort (arr, left, mid);
        invCount += mergeSort (arr, mid+1, right);
        invCount += merge(arr, left, mid+1, right);
    }
    return invCount;
}
public static int getInversions (int arr[]) {
    int n = arr.length;
    return mergeSort (arr, 0, n-1);
}
public static void main (String args[]) {
    int arr[] = {1, 20, 6, 4, 5};
    System.out.println ("Inversion Count = "
                        + getInversions (arr));
}
}
```