Arrays – Medium Part 1

1. **2 Sum**

   Approach 1:- For every element we will traverse the entire array and find if we can get the desired sum. TC:- O(N2) SC:- O(1)

   Approach 2 :- Hashing, save prefix sums check if complement in available if yes then we have desired sum. TC:- O(NlogN) SC:- O(N)

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map <Integer,Integer> mp= new HashMap<>();
        for (int i=0;i<nums.length;i++){
            mp.put(nums[i],i);
        }
        for(int i=0;i<nums.length;i++){
            int rem= target-nums[i];
            if(mp.containsKey(rem) && mp.get(rem) != i){
                return new int []{i,mp.get(rem)};
            }
        }
        return new int[]{};

    }
}
```

   Approach 3 :- 2 pointers Sort array→ one left ptr and one right ptr find sum is sum<desired left++ (go in higher no direction) if sum>desired sum right—(go in lower no direction). TC:- O(N) SC:-O(1)

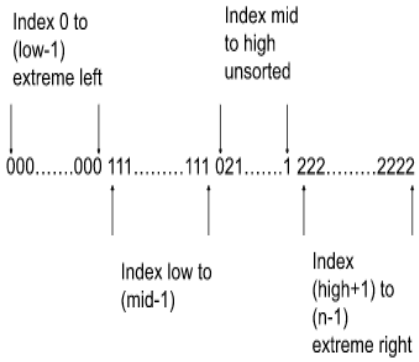
```
public static String twoSum(int n, int []arr, int target) {
    Arrays.sort(arr);
    int left = 0, right = n - 1;
    while (left < right) {
        int sum = arr[left] + arr[right];
        if (sum == target) {
            return "YES";
        } else if (sum < target) left++;
        else right--;
    }
    return "NO";
}
```

2. **Sort 0, 1, 2**

   Approach 1 : Directly sort the array. TC :- O(nlogn) SC:- O(1)

   Approach 2 :- Keep count variables for each 0,1,2 then put that many in new array TC:- O(2n) Sc:- O(1)

   Approach 3 :- Dutch National Flag Algorithm – 3 way partitioning . TC :- O(n) SC :- O(1)




```java
class Solution {
    public void sortColors(int[] nums) {
        int low = 0, mid = 0, high = nums.length - 1;
        while (mid <= high) {
            if (nums[mid] == 0) {
                swap(nums, low, mid);
                low++;
                mid++;
            } else if (nums[mid] == 1) {
                mid++;
            } else {
                swap(nums, mid, high);
                high--;
            }
        }
    }

    public void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

3. **Majority Element in Array :-**

Approach 1 :- Select ele one by one in array, for each ele count its occurrence. If any ele frequency>n/2 return it. TC:- O(N2) SC:- O(1)

Approach 2 :- Use hashmap to store ele and its frequency. Traverse whole array and update frequency. Check if freq>n/2 return it of yes. TC:- O(nlogn) SC:- O(n). My submission:-

```java
class Solution {
    public int majorityElement(int[] nums) {
        Map<Integer, Integer> mp = new HashMap<>();

        for (int i = 0; i < nums.length; i++) {
            mp.put(nums[i], mp.getOrDefault(nums[i], 0) + 1);
        }

        for (Map.Entry<Integer, Integer> entry : mp.entrySet()) {
            if (entry.getValue() > nums.length/2) {
                return entry.getKey();
            }
        }

        return 0;
    }
}
```

Approach 3:- Moore's Voting Algorithm , Someone that appears more than N/2 times will not get cancelled.

Dry run :- 7 7 5 7 5 1 5 7 5 5 7 7 5 5 5

Ct= 1 2 1 2 1 0 1 0 1 2 1 0 1 2 3 4

Ele= 7 5 5 5

- Initialize 2 variables:
  Count – for tracking the count of element
  Element – for which element we are counting
- Traverse through the given array.
  - If **Count** is 0 then store the current element of the array as **Element**.
  - If the current element and **Element** are the same increase the **Count** by 1.
  - If they are different decrease the **Count** by 1.
- The integer present in **Element** should be the result we are expecting

```java
public static int majorityElement(int []v)
{       //size of the given array:
        int n = v.length;
        int cnt = 0; // count
        int el = 0; // Element

        //applying the algorithm:
        for (int i = 0; i < n; i++) {
            if (cnt == 0) {
                cnt = 1;
                el = v[i];
            } else if (el == v[i]) cnt++;
            else cnt--;
        }

        //checking if the stored element
        // is the majority element:
        int cnt1 = 0;
        for (int i = 0; i < n; i++) {
            if (v[i] == el) cnt1++;
        }

        if (cnt1 > (n / 2)) return el;
        return -1;
    }
```

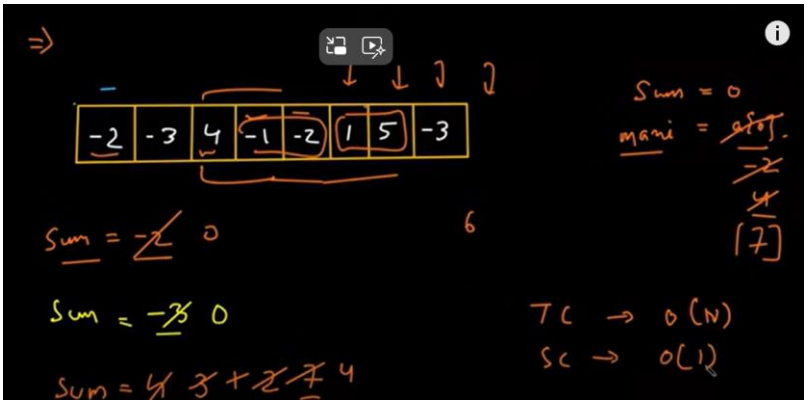
4. **Find maximum subarray sum**

   Approach 1 :- Take sum of all possible sub arrays, see which one is maximum. TC:- O(N3) SC:- O(1)

   Approach 2:- Prefix sum, keep updating maxsum till we find it. TC:- O(N2) SC:- O(1)

   Approach 3: Kadane's Algorithm Intution:- A subarray with sum<0 will always reduce our answer so it shouldn't be a part of the subarray with maximum sum. We carry a sum as long as it gives a positive answer.

   ➔ If sum<0 update the sum to 0




```java
class Solution {
    public int maxSubArray(int[] nums) {
        int sum=0;
        int maxsum=nums[0];
        for(int i=0;i<nums.length;i++){
            sum+=nums[i];
            if(sum > maxsum){
                maxsum=sum;
            }
            if(sum<0){
                sum=0;
            }
        }
        return maxsum;
    }
}
```

5. Extension of Problem 4 : Also print the sub array

```java
class Solution {
    public int maxSubArray(int[] nums) {
        int sum=0;
        int maxsum=nums[0];
        int ansStart=-1, ansEnd=-1;
        for(int i=0;i<nums.length;i++){
            sum+=nums[i];
            if(sum==0){start=i;}
            if(sum > maxsum){
                maxsum=sum;
                ansStart= start;
                ansEnd= ansEnd;
            }
            if(sum<0){
                sum=0;
            }
        }
        return maxsum;
    }
}
```