Experiment 5

Student Name: Abhishek Roushan **UID: 22BCS10187**

Branch: BE-CSE **Section/Group: NTPP_603B**

Semester:6th **Date of Performance:20/02/25**

Subject Name: Ap **Subject Code: 22CSH-359**

1. **Aim : Sort Colors**

2. **Objective :** Given an array nums with n objects colored red, white, or blue, sort them **in-place** so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

   We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.

   You must solve this problem without using the library's sort function.

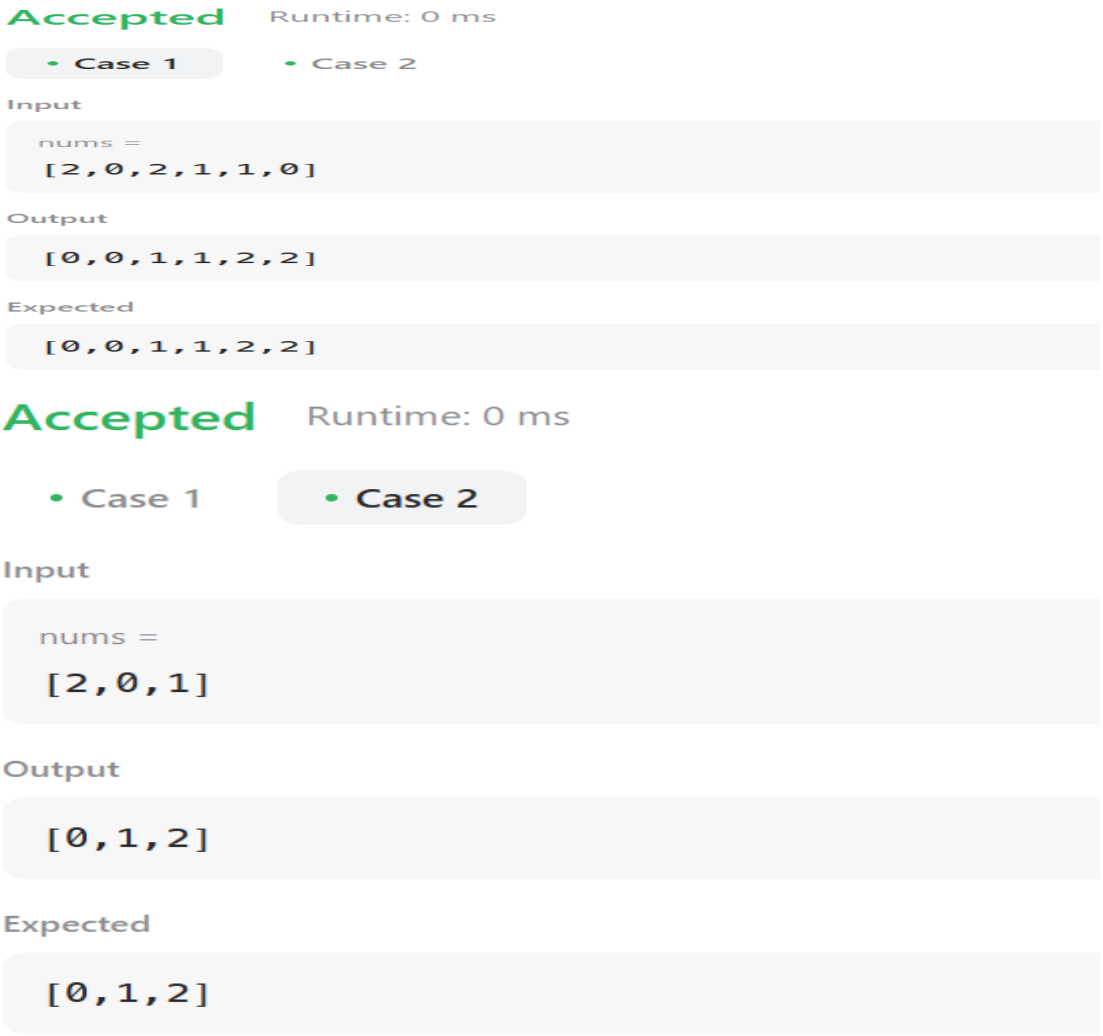
3. **Code:**

```
void sortColors(vector<int>& nums) {
    int low = 0, mid = 0, high = nums.size() - 1;
    while (mid <= high) {
        if (nums[mid] == 0) {
            swap(nums[low++], nums[mid++]);
        } else if (nums[mid] == 1) {
            mid++;
        } else {
            swap(nums[mid], nums[high--]);
        }
```

```
        }

    }
```

4. Output:



5.Learning Outcomes

- Use of Binary Search Approach in it.
- Use of Swap Functon in it.
- Use of Pointer Approach in it.

1. **Aim** :- **Median of Two Sorted Arrays**

2. **Objective** :- Given two sorted arrays nums1 and nums2 of size m and n respectively, return **the median** of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

3. **Code**:-

```
void merge(vector<int>& nums1, vector<int>& nums2,vector<long long >&v){

    int k = 0;
    int idx1 = 0;
    int idx2 = 0;

    while(idx1 < nums1.size() and idx2 < nums2.size()){

       if(nums1[idx1] <= nums2[idx2]){
          v.push_back(nums1[idx1++]);
       }else{
          v.push_back(nums2[idx2++]);
       }

    }

    while(idx1 < nums1.size()){
       v.push_back(nums1[idx1++]);
    }
    while(idx2 < nums2.size()){
       v.push_back(nums2[idx2++]);
    }

  }
public:
   double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {

    vector<long long > v;

    merge(nums1,nums2,v);

    int index =  v.size()/ 2;

    if(v.size() % 2 != 0){
       return v[index];
```

```
        }else{

            return (double)(v[index] + v[index - 1])/2;

        }
    }
```

4. **OutPut**:-

Case 1
Input
nums1 =[1,2]
nums2 =[3,4]
Output
2.50000
Expected
2.50000

Input
nums1 =[1,2]
nums2 =[3,4]
Output
2.50000
Expected
2.50000

5. **Learning Outcomes:-**

1.Using MergeSort Approach.
2.we can make it as an ideal approach in it.
3.using of function call.

1. **Aim :-** Kth Smallest Element in a Sorted Matrix

2. **Objective :-** Given an n x n matrix where each of the rows and columns is sorted in ascending order, return *the* k^{th} *smallest element in the matrix*.
   Note that it is the k^{th} smallest element in the sorted order, not the k^{th} distinct element.
   You must find a solution with a memory complexity better than $O(n^2)$.

3. **Code:-**

```
int kthSmallest(vector<vector<int>>& matrix, int k) {

    priority_queue<int>ans;
    for(int i=0;i<matrix.size();i++){
       for(int j=0;j<matrix.size();j++){

          if(ans.size() < k){
           ans.push(matrix[i][j]);
          }else{

           if(matrix[i][j]<ans.top()){
              ans.pop();
              ans.push(matrix[i][j]);
           }
          }
       }
    }
    return ans.top();}
```

4. **Output :-**

Case 1
Input
matrix =[[1,5,9],[10,11,13],[12,13,15]]
k = 8
Output
13
Expected
13

Input
matrix =[[-5]]
k =1
Output
-5
Expected
-5

5. **Learning Outcomes:-**

1. Use of Priority Queue.
2. Use of neasted loop.
3. Using the approach of queue.