ARRAYS - SET 2 SOLUTIONS

Note: All solutions are written in C++.

Question 1:

- **Problem link**: https://leetcode.com/problems/largest-number-at-least-twice-of-others/
- **Difficulty level**: Easy

Explanation:

Calculate the max and second max element of the array. If the max element is greater than or equal to twice the second max element, then, return the index of the max element. Else return -1.

Solution:

```
class Solution {
public:
    int dominantIndex(vector<int>& nums) {

        int maxIndex = -1;
        int max = -1;
        int secondMax = -1;

        for(int i=0; i<nums.size(); i++)
        {
            if(nums[i] > max)
            {
                secondMax = max;
                max = nums[i];
                maxIndex = i;
            }
            else if(nums[i]>secondMax && nums[i]!=max)
            {
                secondMax = nums[i];
            }
        }

        if(max >= 2*secondMax)
        {
            return maxIndex;
        }
        else
        {
            return -1;
        }

    }
};
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 2:

- **Problem link**: https://leetcode.com/problems/merge-sorted-array/
- **Difficulty level**: Easy

Explanation:

Merge the two sorted arrays into the first array starting from the right end and progressively moving towards the left.

Solution:

```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n)
{

        int i=m-1;
        int j=n-1;
        int k = m+n-1;

        while(i>=0 && j>=0)
        {
            if(nums1[i] > nums2[j])
            {
                nums1[k] = nums1[i];
                k--;
                i--;
            }
            else
            {
                nums1[k] = nums2[j];
                k--;
                j--;
            }
        }

        while(i>=0)
        {
            nums1[k] = nums1[i];
            k--;
            i--;

        }

        while(j>=0)
        {
            nums1[k] = nums2[j];
            k--;
            j--;
        }
    }
};
```

Complexity:

- Time: O(m + n)
- Extra Space: O(1)

Question 3:

- **Problem link**: https://leetcode.com/problems/degree-of-an-array/
- **Difficulty level**: Easy

Explanation:

Maintain two maps: one to store the count of each element and the other to store the index of the first occurrence of each element. Traverse the given array and for each element, increment its counter. If the count of the current element is 1, then record the current position as the first occurrence of that element. If the count is equal to the current degree, then, calculate the size of the subarray [first Index ... current index] and check if this size is smaller. If the count is greater than current degree, update the degree and update the size of the smallest subarray as well.

Solution:

```
class Solution {
public:
    int findShortestSubArray(vector<int>& nums) {
        int degree = 0;
        int minSize = 0;

        map<int,int> count;
        map<int,int> firstIndex;

        for(int i=0; i<nums.size(); i++)
        {
            int num = nums[i];
            count[num]++;

            if(count[num] == 1)
            {
                firstIndex[num] = i;
            }

            if(count[num] == degree)
            {
                int currSize = i-firstIndex[num]+1;
                minSize = min(minSize, currSize);
            }

            if(count[num] > degree)
            {
                degree = count[num];
                minSize = i-firstIndex[num]+1;
            }
        }

        return minSize;
    }
};
```

Complexity:

- Time: O(n)

- Space: O(n)

Question 4:

- **Problem link**: https://leetcode.com/problems/bulb-switcher-iii/
- **Difficulty level**: Medium

Explanation:

If at a moment ‘k’ , the number of bulbs on is equal to max(light[0],..,light[k]) then all the bulbs are blue at that instant.

Solution:

```
class Solution {
public:
    int numTimesAllBlue(vector<int>& light) {
        int count=0,num=0,max=0;
        for(int i=0;i<light.size();i++)
        {
            num++;
            if(light[i]>m) max=light[i];
            if(max==num) count++;
        }
        return count;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 5:

- **Problem link**: https://leetcode.com/problems/number-of-sub-arrays-with-odd-sum/
- **Difficulty level**: Medium

Explanation:
Maintain prefix sum at each step and count the number of times it is even/odd. If the current prefix sum is odd, then all previous even prefix sums will also count for odd subarray sums and vice versa.

Solution:

```
class Solution {
public:
    int numOfSubarrays(vector<int>& arr) {
        int n=arr.size(),ne=0,no=0,sum=0,count=0;
        for(int i=0;i<n;i++)
        {
            sum+=arr[i];
            if(sum%2)
            {
                no++;
                count=(count+ne+1)%1000000007;
            }
            else
            {
                ne++;
                count=(count+no)%1000000007;
            }
        }
        return count;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 6:

- **Problem link**: https://leetcode.com/problems/find-the-winner-of-an-array-game/
- **Difficulty level**: Medium

Explanation:

If ‘k’ is greater than the number of elements in the array, then return the maximum of the array. Otherwise, the number may/may not be the maximum number. Then, we simulate the game until a number wins ‘k’ consecutive games.

Solution:

```
class Solution {
public:
    int getWinner(vector<int>& arr, int k) {
        int max=0,n=arr.size();
        for(int i=0;i<n;i++) if(arr[i]>max) max=arr[i];
        if(k>=n) return max;
        int x=0,j=0,result;
        for(int i=1;i<n;i++)
        {
            if(x==k)
            {
                result=arr[j];
                break;
            }
            else if(arr[i]>arr[j]){
                if(arr[i]==max)
                {
                    result=max;
                    break;
                }
                x=1;
                j=i;
            }
            else x++;
        }
        return result;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 7:

- **Problem link**: https://leetcode.com/problems/reverse-subarray-to-maximize-array-value/
- **Difficulty level**: Hard

Explanation:

Refer to this link to get a detailed description of how the algorithm used to solve this problem actually works.

Solution:

```
class Solution {
public:
    int maxValueAfterReverse(vector<int>& nums) {

        int maxElement = INT_MIN;
        int minElement = INT_MAX;
        int n = nums.size();
        int originalValue = 0;

        // Original Array value
        for(int i=0; i<n-1; i++)
        {
            originalValue = originalValue + abs(nums[i] - nums[i+1]);
        }

        // Total max change that can be done
        for(int i=0; i<n-1; i++)
        {
            int a = nums[i];
            int b = nums[i+1];

            maxElement = max(min(a,b), maxElement);
            minElement = min(max(a,b), minElement);
        }

        int totalChange = max(2*(maxElement - minElement), 0);

        // Boundary case: when the subarray to be reversed includes
the 1st element or the last element
        for(int i=0; i<n-1; i++)
        {
            int a = nums[i];
            int b = nums[i+1];

            // If subarray to be reversed includes the 1st element
            int change1 = (-1)*abs(a-b) + abs(nums[0]-b);

            // If subarray to be reversed includes the 2nd element
            int change2 = (-1)*abs(a-b) + abs(nums[n-1]-a);

            int maxChange = max(change1, change2);
```

```
                totalChange = max(totalChange, maxChange);
            }

            // Return final value of array
            return originalValue + totalChange;
        }
};
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 8:

- **Problem link**: https://leetcode.com/problems/sum-of-subsequence-widths/
- **Difficulty level**: Hard

Explanation:

A very common mistake that happens in this question is that a subsequence of an array is not the same as a subarray. Check this link to learn about the difference between a subsequence and a subarray. The detailed explanation and proof for the problem can be found here.

Solution:

```
class Solution {
public:
    int sumSubseqWidths(vector<int>& A) {
        if(A.size()==1)
        {
            return 0;
        }
        if(A.size()==2)
        {
            return abs(A[0]-A[1]);
        }

        int n = A.size();
        vector<long> power(n,0);
        power[0]=1;

        for(int i=1;i<n;i++)
        {
            power[i]=(power[i-1]*2)%1000000007;
        }

        long result=0;
        sort(A.begin(),A.end());

        for(int i=0;i<n;i++)
        {
            result=long(result+((power[i]-power[n-i-1]))*A[i])%1000000007;
        }

        return result;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(n)