Assignment Questions 11

Question 1

Given a non-negative integer `x`, return *the square root of* `x` *rounded down to the nearest integer*. The returned integer should be **non-negative** as well.

You **must not use** any built-in exponent function or operator.

- For example, do not use `pow(x, 0.5)` in c++ or `x ** 0.5` in python. </aside>

Input: x = 4

Output: 2

Explanation: The square root of 4 is 2, so we return 2.

Solution:-

```
class Solution {
    public int mySqrt(int x)
    {
        long start=1;
        long end=x;
        long ans=0;
        while(start<=end)
    {
        long mid=start +( end-start)/2;
        if(mid*mid==x)
        {
            ans=(int)mid;
            break;

        }
        else if(mid*mid<x)
        {
            start=mid+1;
            ans=mid;
        }
        else
        {
            end=mid-1;
        }
    }
  return (int) ans;

    }
```

```
}
```

Question 2

A peak element is an element that is strictly greater than its neighbors.

Given a **0-indexed** integer array `nums`, find a peak element, and return its index. If the array contains multiple peaks, return the index to **any of the peaks**.

You may imagine that `nums[-1] = nums[n] = -∞`. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

You must write an algorithm that runs in `O(log n)` time.

Example 1:

Input: nums = [1,2,3,1]

Output: 2

Explanation: 3 is a peak element and your function should return the index number 2**.**

Solution:-

```
class Solution {
    public int findPeakElement(int[] nums) {
        int n=nums.length;
        int start=0;
        int end=n-1;

            if(n==1){
            return 0;
            }


            if(nums[0]>nums[1]){
                return 0;
            }
            else if(nums[n-1]>nums[n-2]){
                return n-1;
            }
            start=1;
            end=n-2;
            while(start<=end){

            int mid=start+(end-start)/2;

            if(nums[mid]>nums[mid+1]  && nums[mid]>nums[mid-1]){
```

```
                return mid;
            }

            else if(nums[mid]>nums[mid-1]){
                start=mid+1;
            }

            else if(nums[mid]>nums[mid+1]){
                end=mid-1;
            }

            else if(nums[mid]<nums[mid-1]){
                end=mid-1;
            }

            else if(nums[mid]<nums[mid+1]){
                start=mid+1;
            }

            }
               return -1;


    }
}
```

Question 3

Given an array `nums` containing `n` distinct numbers in the range `[0, n]`, return *the only number in the range that is missing from the array.*

Example 1:

Input: nums = [3,0,1]

Output: 2

Explanation: n = 3 since there are 3 numbers, so all numbers are in the range [0,3]. 2 is the missing number in the range since it does not appear in nums.

Solution:-

```
class Solution {
    public int missingNumber(int[] nums) {
        int n = nums.length;
        int totalSum = n*(n+1)/2;
        for(int i=0;i<n;i++){
```

```
            totalSum -= nums[i];
        }
        return totalSum;
    }
}
```

Question 4

Given an array of integers `nums` containing `n + 1` integers where each integer is in the range `[1, n]` inclusive.

There is only **one repeated number** in `nums`, return *this repeated number*.

You must solve the problem **without** modifying the array `nums` and uses only constant extra space.

Example 1:

Input: nums = [1,3,4,2,2]

Output: 2

Solution:-

```
class Solution {
    public int findDuplicate(int[] nums) {
        HashSet<Integer> set = new HashSet<>();
        for(int num : nums) {
            if(!set.add(num)) {
                return num;
            }
        }
        return -1;
    }
}
```

Question 5

Given two integer arrays `nums1` and `nums2`, return *an array of their intersection*. Each element in the result must be **unique** and you may return the result in **any order**.

Example 1:

Input: nums1 = [1,2,2,1], nums2 = [2,2]

Output: [2]

Solution:-

```
class Solution {
    public int[] intersection(int[] nums1, int[] nums2) {
    Set<Integer> x=new HashSet<>();
      Set<Integer> y=new HashSet<>();
      for(int i=0;i<nums1.length;i++){
          x.add(nums1[i]);
      }
      for(int i=0;i<nums2.length;i++){
          y.add(nums2[i]);
      }
      x.retainAll(y);
      int ans[]=new int[x.size()];
      int i = 0;
        for(Integer n : x) {
            ans[i] = n;
            i++;
        }
     return ans; }
}
```

Question 6

Suppose an array of length `n` sorted in ascending order is **rotated** between `1` and `n` times. For example, the array `nums = [0,1,2,4,5,6,7]` might become:

- `[4,5,6,7,0,1,2]` if it was rotated `4` times.
- `[0,1,2,4,5,6,7]` if it was rotated `7` times.

Notice that **rotating** an array `[a[0], a[1], a[2], ..., a[n-1]]` 1 time results in the array `[a[n-1], a[0], a[1], a[2], ..., a[n-2]]`.

Given the sorted rotated array `nums` of **unique** elements, return *the minimum element of this array*.

You must write an algorithm that runs in `O(log n) time`.

Example 1:

Input: nums = [3,4,5,1,2]

Output: 1

Explanation: The original array was [1,2,3,4,5] rotated 3 times.

Solution;-

```
class Solution {
        public int findMin(int[] A) {
        final int N = A.length;
        if(N == 1) return A[0];
        int start = 0, end = N-1, mid;
        while(start < end){
            mid = (start+end) / 2;
            if(mid > 0 && A[mid] < A[mid-1]) return A[mid];
            if(A[start] <= A[mid] && A[mid] > A[end]) start = mid + 1;
            else end = mid - 1;
        }
        return A[start];
    }
}
```

Question 7

Given an array of integers `nums` sorted in non-decreasing order, find the starting and ending position of a given `target` value.

If `target` is not found in the array, return `[-1, -1]`.

You must write an algorithm with `O(log n)` runtime complexity.

Example 1:

Input: nums = [5,7,7,8,8,10], target = 8

Output: [3,4]

Solution:-

```
class Solution {
    public int[] searchRange(int[] arr, int t) {
        int[] brr=new int[2];
        brr[0]=-1;brr[1]=-1;
        int f=-1;int l=-1;
        for(int i=0;i<arr.length;i++){
            if(t==arr[i]){
                if(f==-1){
              brr[0]=i;
              f++;
                }
```

```
                else{
                    brr[1]=i;
                    l++;
                }
            }
        }
        if(f!=-1&&l==-1){
         brr[1]=brr[0];
                 return brr;
        }

                return brr;
    }
}
```

Question 8

Given two integer arrays `nums1` and `nums2`, return *an array of their intersection*. Each element in the result must appear as many times as it shows in both arrays and you may return the result in **any order**.

Example 1:

Input: nums1 = [1,2,2,1], nums2 = [2,2]

Output: [2,2]

Solution:-

```
class Solution{
  public int[] intersect(int[] nums1, int[] nums2) {
  Arrays.sort(nums1);
  Arrays.sort(nums2);
  int top = 0;
  int bottom = 0;
  List<Integer> h = new ArrayList<>();

  while (true){
    if (top >= nums1.length || bottom >= nums2.length){
      break;
    }
    if (nums1[top] == nums2[bottom]){
      h.add(nums1[top]);
      top ++;
      bottom ++;
    }
    else if (nums1[top] < nums2[bottom]){
```

```
      top ++;
    }
    else if (nums1[top] > nums2[bottom]){
      bottom ++;
    }
  }

  int[] g = new int[h.size()];
  for (int i = 0; i < h.size(); i++) {
    g[i] = h.get(i);
  }
  return g;
}
}
```