Sample Problems on Searching

Problem #1 : Missing and Repeating Number

Description: Given an unsorted array of size **n**. Array elements are in the range from **1 to n**. One number from set {1, 2, ...n} is missing and one number occurs twice in the array. Our task is to find these two numbers.

```
Input
[2, 3, 2, 1, 5]
Output
4 2
```

1. **Solution : Use Sorting** Follow the given steps-

   ```
   1) Sort the input array.
   2) Traverse the array and check for missing and repeating.
   ```

   Time Complexity : O(nLogn)
   Auxiliary Space: O(1)
2. **Solution : Make two equations using Sum and Product**
   1. Let x be the missing and y be the repeating element.
   2. Get the the sum of Array using formula **S = n(n+1)/2 - x + y**
   3. Get product of Array using formula **P = 1*2*3*...*n * y / x**
   4. The above two steps give us two equations, we can solve the equations and get the values of x and y.

   ```
   Example Array : [2, 3, 2, 1, 5]
   S = 13
   (n *(n+1))/2 = 15
   13 = 15 - x + y
   x - y = 2 ....  1
   P = 60
   1*2*3..n = 120
   60 = (120*y)/x
   x = 2y .... 2
   Solving Equation 1 and 2 --
   x = 4 and y = 2
   ```

   Time Complexity: O(n)
   Auxiliary Space : O(1)

   Note: This method can cause arithmetic overflow as we calculate product and sum of all array elements. Can you avoid this?

3. **Solution : Use Hashing** We can create a auxiliary array to count the elements in the Array. We traverse the auxiliary array for finding out missing and repeating number in the array. Can we optimize the space ?

   Pseudo Code

   ```
   //n : size of array
   void repeating_missing(arr, n)
   {
       count[n+1] = {0}
       for (i=0 to n-1 )
           count[a[i]]++

       for (i=1 to n) {
           if (count[i] == 0 )
               missing = i
           if (count[i] == 2 )
               repeating = i
       }
       print(repeating, missing)
   }
   ```

   Time Complexity: O(n)
   Auxiliary Space : O(n)

4. **Solution : Use Negative Indexing** Traverse the array. While traversing, use the absolute value of every element as an index and make the value at this index as negative to mark it visited. If something is already marked negative then this is the repeating element. To find missing, traverse the array again and look for a positive value.

Pseudo Code

```
//n : size of array
void repeating_missing(arr, n)
{
    for ( i=0 to n-1 ) {
        temp = arr[abs(arr[i])- 1]
        if (temp < 0 ) {
            repeating = abs(arr[i])
            break
        }
        arr[abs(arr[i])- 1] = -arr[abs[arr(i)]- 1]
    }

    for (i=0 to n-1) {
        if (arr[i] > 0 )
            missing = i+1
    }
    print(repeating, missing)
}
```

Time Complexity: O(n)
Auxiliary Space : O(1)

Problem #2 : Count number of Occurences in Sorted Array

Description - Given a sorted array **arr[ ]** and a number **x**, We have to count the occurrences of **x** in arr[ ].

```
Input : [1, 1, 2, 2, 2, 2, 3] , x = 2
Output : 4
```

Solution : Linear Search We can traverse the array and count the number of occurences of x in the given input array.
Time Complexity : O(n)
Since Array is sorted, can we optimize the solution using binary search.

Solution: Binary Search We can solve this problem using binary search by reducing the effective search space in each step. We will be using these steps -

1. Use Binary search to get the index of the first occurrence of x in arr[ ]. Let the index of the first occurrence be i.
2. Use Binary search to get the index of the last occurrence of x in arr[ ]. Let the index of the last occurrence be j.
3. Return the count as difference between first and last indices (j – i + 1);

Pseudo Code

```
int first_index(arr, low, high, x, n)
{

    if(high >= low)
    {
        mid = (low + high)/2  /*low + (high - low)/2*/
        if( ( mid == 0 || x > arr[mid-1]) && arr[mid] == x) :
            return mid
        else if(x > arr[mid]) :
            return first_index(arr, (mid + 1), high, x, n)
        else :
            return first_index(arr, low, (mid -1), x, n)
    }
}

int last_index(arr, low, high, x, n):
{
    if (high >= low)
    {
        int mid = (low + high)/2  /*low + (high - low)/2*/
        if( ( mid == n-1 || x < arr[mid+1]) && arr[mid] == x )
            return mid
        else if(x < arr[mid]) :
            return last_index(arr, low, (mid -1), x, n)
        else :
            return last_index(arr, (mid + 1), high, x, n)
    }
}

int count_occurences(arr, n, x)
{
    i = first_index(arr, 0, n-1, x, n)
    j = last_index(arr, 0, n-1, x, n)
    count = j-i + 1
    return count
}
```

Time Complexity : O(Log(n))

Problem #3 : Find the index of first 1 in a sorted array of 0's and 1's

Description - We are given an sorted boolean array, We have to find out the index of first 1 in the Array

```
Input : arr[] = [0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
Output : 6
The index of first 1 in the array is 6.
```

Solution - One simple solution can be traverse the Array and find out the first index of 1. Since the array is sorted, we can optimize the solution using binary search by reducing the effective search space in each step.

Pseudo Code

```
int indexOfFirstOne(arr[], low, high)
{
    while (low <= high)
    {
        int mid = (low + high) / 2

        if (arr[mid] == 1 && (mid == 0 || arr[mid - 1] == 0)) :
            return mid
        else if (arr[mid] == 1) :
            high = mid - 1
        else :
            low = mid + 1
    }
}
```

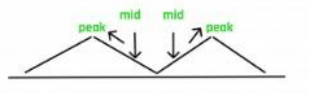
Time Complexity : O(Log(n))

Problem #4 : Find Peak element in Sorted Array

Description - We are given an array of distinct integers. We have to find the peak element (The element which is greater than both the neighbours). There can be many such elements we need to return any of them.

```
Input : [10, 20, 15, 2, 23, 90, 67]
Output : 20 or 90
```

Solution : A simple solution is to traverse the array and as soon as we find a peak element, we return it. The worst case time complexity of this method would be O(n).Can we find a peak element in worst time complexity better than O(n)?
We can use the Divide and Conquer. The idea is Binary Search-based, we compare the middle element with its neighbors. If the middle element is not smaller than any of its neighbors, then we return it. If the middle element is smaller than its left neighbor, then there is always a peak in the left half. If the middle element is smaller than its right neighbor, then there is always a peak in the right half (due to the same reason as left half).



Pseudo Code

```
int findPeak(arr[], low, high, n)
{
    int mid = low + (high - low)/2 /* (low + high)/2 */

    if ((mid == 0 || arr[mid-1] <= arr[mid]) &&
            (mid == n-1 || arr[mid+1] <= arr[mid])) :
        return arr[mid]

    // If middle element is not peak and its left neighbour is greater
    // than it, then left half must have a peak element
    else if (mid > 0 && arr[mid-1] > arr[mid]) :
        return findPeak(arr, low, (mid -1), n)

    // If middle element is not peak and its right neighbour is greater
    // than it, then right half must have a peak element
    else :
        return findPeak(arr, (mid + 1), high, n)
}
```

Time Complexity : O(Log(n))