

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 $\{1, 2, \dots, n\}$ 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

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

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

Time Complexity : $O(n \log n)$

Auxiliary Space: $O(1)$

- **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 $P = 1*2*3*\dots*n * y / x$
- The above two steps give us two equations, we can solve the equations and get the values of x and y .

- **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)
```

Track Progress

12 of 60 Complete. (20%)

for (i=0 to n-1)

Dash



```

    if (count[i] == 0 )
        missing = i
    if (count[i] == 2 )
        repeating = i
}
print(repeating, missing)
}

```



Problems

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

- **Solution : Use Negative Indexing** ^{Quiz} 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. ^{Contest}

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)
}

```

Menu

Time Complexity: $O(n)$

Courses

Tutorials

Jobs 10 Complete. (20%)

Practice



Contests | **cription** - Given a sorted array `arr[]` and a number `x`, We have to count the



Output : 4



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

Articles

Videos

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 -

Problems

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 ($j - i + 1$);



Quiz



Contest

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
    }
}
```

Menu

Track Progress

12 of 60 Complete. (20%)

else :

Dash



```
int count_occurrences(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
}
```



Articles



Videos



Problems

**Time Complexity :** $O(\log(n))$

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



Quiz

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



Contest

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
    }
}
```

Menu



Track Progress

12 of 60 Complete. (20%)

Problem #4 : Find Peak element in Unsorted Array

Dash



Note: For corner elements, we need to consider only one neighbor.



Articles

Example:



Videos

Input: `array[] = {5, 10, 20, 15}`

Output: 20

Explanation: The element 20 has neighbors 10 and 15, both of them are less than 20.



Problems

Input: `array[] = {10, 20, 15, 2, 23, 90, 67}`

Output: 20 or 90



Quiz

Explanation: The element 20 has neighbors 10 and 15, both of them are less than 20, similarly 90 has neighbors 23 and 67.



Contest

The following corner cases give a better idea about the problem.

1. If the input array is sorted in a strictly increasing order, the last element is always a peak element. For example, 50 is peak element in {10, 20, 30, 40, 50}.
2. If the input array is sorted in a strictly decreasing order, the first element is always a peak element. 100 is the peak element in {100, 80, 60, 50, 20}.
3. If all elements of the input array are the same, every element is a peak element.

It is clear from the above examples that there is always a peak element in the input array.

Naive Approach: Below is the idea to solve the problem

The array can be traversed and the element whose neighbors are less than that element can be returned.

Follow the below steps to Implement the idea:

- If the first element is greater than the second or the last element is greater than the second last, print the respective element and terminate the program.
- Else traverse the array from the second index to the second last index i.e. 1 to $N - 1$
 - If for an element `array[i]` is greater than both its neighbors. i.e.,

Menu

Track Progress

12 of 60 Complete. (20%)

array[i] >= array[i+1]

Dash



All

C++

Java



Articles



Videos



Problems



Quiz



Contest

```
// A C++ program to find a peak element
#include <bits/stdc++.h>
using namespace std;

// Find the peak element in the array
int findPeak(int arr[], int n)
{
    // first or last element is peak element
    if (n == 1)
        return 0;
    if (arr[0] >= arr[1])
        return 0;
    if (arr[n - 1] >= arr[n - 2])
        return n - 1;

    // check for every other element
    for (int i = 1; i < n - 1; i++) {

        // check if the neighbors are smaller
        if (arr[i] >= arr[i - 1] && arr[i] >= arr[i + 1])
            return i;
    }
}

// Driver Code
int main()
{
    int arr[] = { 1, 3, 20, 4, 1, 0 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Index of a peak point is " << findPeak(arr, n);
    return 0;
}
```

Menu

Track Progress

12 of 60 Complete. (20%)



Using Binary Search, check if the **middle** element is the peak element or not. If the middle element the peak element terminate the while loop and print **middle element**, then check if the element on the right side is greater than the middle element then there is always a peak element on the **right side**. If the element on the left side is greater than the middle element then there is always a peak element on the **left side**.

Problems

Follow the steps below to implement the idea:

- Create two variables, l and r , initialize $l = 0$ and $r = n-1$
- Run a while loop till $l \leq r$, lowerbound is less than the upperbound
 - Check if the mid value or index $mid = low + (high-low)/2$ is the peak element or not, if yes then print the element and terminate.
 - Else if the element on the left side of the middle element is greater then check for peak element on the left side, i.e. update $r = mid - 1$
 - Else if the element on the right side of the middle element is greater then check for peak element on the right side, i.e. update $l = mid + 1$

The below-given code is the iterative version of the above explained and demonstrated recursive based divide and conquer technique.

C++

Java

```
// A C++ program to find a peak element
// using divide and conquer
#include <bits/stdc++.h>
using namespace std;

// A binary search based function
// that returns index of a peak element
int findPeak(int arr[], int n)
{
    int l = 0;
```

Menu

Track Progress

12 of 60 Complete. (20%)



Articles

```
mid = (l + r) >> 1;

// first case if mid is the answer
if ((mid == 0 || arr[mid - 1] <= arr[mid])
    and (mid == n - 1 || arr[mid + 1] <= arr[mid]))
    break;
```



Videos



Problems

```
// move the right pointer
if (mid > 0 and arr[mid - 1] > arr[mid])
    r = mid - 1;
```



Quiz

```
// move the left pointer
else
    l = mid + 1;
}
```



Contest

```
return mid;
}

// Driver Code
int main()
{
    int arr[] = { 1, 3, 20, 4, 1, 0 };
    int N = sizeof(arr) / sizeof(arr[0]);
    cout << "Index of a peak point is " << findPeak(arr, N);
    return 0;
}
```

Output

Index of a peak point is 2

Time Complexity: $O(\log N)$, Where n is the number of elements in the input array. In each step our search becomes half. So it can be compared to Binary search, So the time complexity is $O(\log N)$

Track Progress

12 of 60 Complete. (20%)

Dash

Marked as Read



All

Report An Issue

If you are facing any issue on this page. Please let us know.

Articles



Videos



Problems



Quiz



Contest



Menu



Track Progress

12 of 60 Complete. (20%)