DATA STRUCTURES (BASICS)

[SEARCHING]

PLACEMENT PREPARATION [EXCLUSIVE NOTES]

SAVE AND SHARE

Curated By- HIMANSHU KUMAR(LINKEDIN) https://www.linkedin.com/in/himanshukumarmahuri

TOPICS COVERED-

- **Linear Search**
- **Binary Search**
- **Ternary Search**
- **Binary Search Functions in C++ STL**
- **BinarySearch using Built-in Mehods in**
- **Java Sample Problems on Searching**

Linear Search-

Linear Search means to sequentially traverse a given list or array and check if an element is present in the respective array or list. The idea is to start traversing the array and compare elements of the array one by one starting from the first element with the given element until a match is found or end of the array is reached.

Examples :

```
Input : arr[] = {10, 20, 80, 30, 60, 50,
                    110, 100, 130, 170}
          key = 110;
Output : 6
Element 110 is present at index 6

Input : arr[] = {10, 20, 80, 30, 60, 50,
                    110, 100, 130, 170}
           key = 175;
Output : -1
Element 175 is not present in arr[].
```

Problem: Given an array **arr[]** of **N** elements, write a function to search a given element **X** in arr[].

Algorithm:

- Start from the leftmost element of arr[] and one by one compare X with each element of arr[].
- If X matches with an element, return the index.
- If X doesn't match with any of elements and end of the array is reached, return -1.

Function:

```
// Function to linearly search the element X in the
// array arr[] of N elements
int search(int arr[], int N, int X)
{
   // Pointer to traverse the array
   int i;

   // Start traversing the array
   for (i = 0; i < N; i++)
   {
```

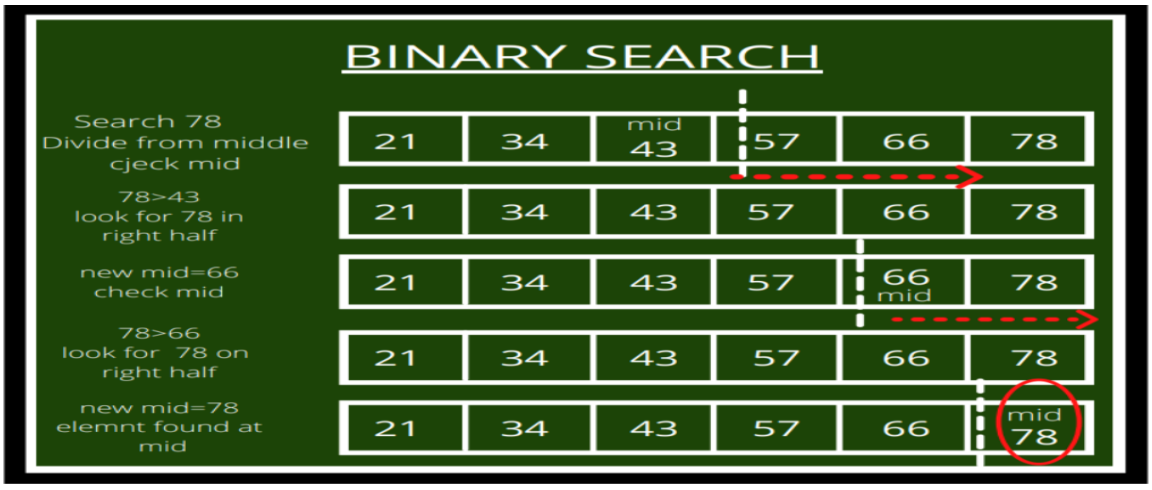
```
        // If a successful match is found,
        // return the index
        if (arr[i] == X)
            return i;
    }

    // If the element is not found,
    // and end of array is reached
    return -1;
}
```

Time Complexity: O(N). Since we are traversing the complete array, so in worst case when the element X does not exists in the array, number of comparisons will be N. Therefore, *worst case time complexity of the linear search algorithm is O(N).*

Binary Search-

Binary Search is a searching algorithm for searching an element in a sorted list or array. Binary Search is efficient than Linear Search algorithm and performs the search operation in logarithmic time complexity for sorted arrays or lists.

Binary Search performs the search operation by repeatedly dividing the search interval in half. The idea is to begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.



Problem: Given a sorted array arr[] of N elements, write a function to search a given element X in arr[] using *Binary Search Algorithm*.

Algorithm: We basically ignore half of the elements just after one comparison.

- Compare X with the middle element of the array.
- If X matches with middle element, we return the mid index.
- Else If X is greater than the mid element, then X can only lie in right half subarray after the mid element. So we will now look for X in only the right half ignoring the complete left half.
- Else if X is smaller, search for X in the left half ignoring the right half.

Implementation: The Binary Search algorithm can be implemented both recursively and iteratively.

Rescursion Function-

```
// A recursive binary search function. It returns
// location of x in given array arr[l..r] if present,
// otherwise -1

// Initially,
// l = 0, first index of arr[].
// r = N-1, last index of arr[].
int binarySearch(int arr[], int l, int r, int x)
{
   if (r >= l) {
      int mid = l + (r - l) / 2;

      // If the element is present at the middle
      // itself
      if (arr[mid] == x)
         return mid;
      // If element is smaller than mid, then
      // it can only be present in left subarray
      if (arr[mid] > x)
         return binarySearch(arr, l, mid - 1, x);

      // Else the element can only be present
      // in right subarray
      return binarySearch(arr, mid + 1, r, x);
   }

   // We reach here when element is not
   // present in array
   return -1;
}
```

Iterative Function:

```
// A iterative binary search function. It returns
// location of x in given array arr[l..r] if present,
// otherwise -1

// Initially,
// l = 0, first index of arr[].
// r = N-1, last index of arr[].
int binarySearch(int arr[], int l, int r, int x)
{
   while (l <= r) {
      int m = l + (r - l) / 2;

      // Check if x is present at mid
      if (arr[m] == x)
         return m;

      // If x greater, ignore left half
      if (arr[m] < x)
         l = m + 1;

      // If x is smaller, ignore right half
      else
         r = m - 1;
   }

   // if we reach here, then element was
   // not present
   return -1;
}
```

Time Complexity: O(Log N), where N is the number of elements in the array.

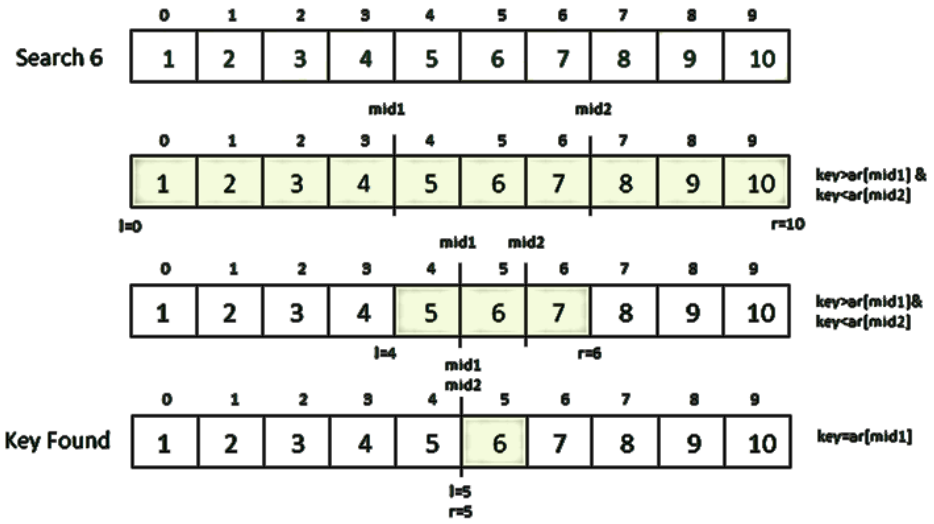
Ternary Search-

Ternary Search is a Divide and Conquer Algorithm used to perform search operation in a **sorted array**. This algorithm is similar to the Binary Search algorithm but rather than dividing the array into two parts, it divides the array into three equal parts.

In this algorithm, the given array is divided into three parts and the key (element to be searched) is compared to find the part in which it lies and that part is further divided into three parts.

We can divide the array into three parts by taking mid1 and mid2 which can be calculated as shown below. Initially, l and r will be equal to 0 and N-1 respectively, where N is the length of the array.

```
mid1 = l + (r-l)/3
mid2 = r – (r-l)/3
```



Note: The array must be sorted in order to perform the Binary Search or Ternary Search operation.

Steps to perform Ternary Search:

- First, we compare the key with the element at mid1. If found equal, we return mid1.

- If not, then we compare the key with the element at mid2. If found equal, we return mid2.
- If not, then we check whether the key is less than the element at mid1. If yes, then recur to the first part.
- If not, then we check whether the key is greater than the element at mid2. If yes, then recur to the third part.
- If not, then we recur to the second (middle) part.

Implementation: The Ternary Search Algorithm can be implemented in both recursive and iterative manner. Below is the implementation of both recursive and iterative function to perform Ternary Search on an array *arr[]* of size *N* to search an element *key*.

Recursive Function-

```
// Recursive Function to perform Ternary Search
// Initially,
// l = 0, starting index of array.
// r = N-1, ending index of array.
int ternarySearch(int l, int r, int key, int ar[])
{
   if (r >= l)
   {
      // Find mid1 and mid2
      int mid1 = l + (r - l) / 3;
      int mid2 = r - (r - l) / 3;

      // Check if key is present at any mid
      if (ar[mid1] == key)
      {
         return mid1;
      }
      if (ar[mid2] == key)
      {
         return mid2;
      }

      // Since key is not present at mid,
      // check in which region it is present
      // then repeat the Search operation
      // in that region
      if (key < ar[mid1])
      {

         // The key lies in between l and mid1
         return ternarySearch(l, mid1 - 1, key,
ar);
      }
      else if (key > ar[mid2])
      {

         // The key lies in between mid2 and r
         return ternarySearch(mid2 + 1, r, key,
ar);
      }
      else
```

```
    {

        // The key lies in between mid1 and
mid2
        return ternarySearch(mid1 + 1, mid2
- 1, key, ar);
```

Iterative function-

```
// Iterative Function to perform Ternary Search
// Initially,
// l = 0, starting index of array.
// r = N-1, ending index of array.
int ternarySearch(int l, int r, int key, int ar[])
{
   while (r >= l) {

      // Find mid1 and mid2
      int mid1 = l + (r - l) / 3;
      int mid2 = r - (r - l) / 3;

      // Check if key is present at any mid
      if (ar[mid1] == key) {
         return mid1;
      }
      if (ar[mid2] == key) {
         return mid2;
      }

      // Since key is not present at mid,
      // check in which region it is present
      // then repeat the Search operation
      // in that region

      if (key < ar[mid1]) {

         // The key lies in between l and mid1
         r = mid1 - 1;
      }
      else if (key > ar[mid2]) {

         // The key lies in between mid2 and r
         l = mid2 + 1;
      }
      else {

         // The key lies in between mid1 and
mid2
         l = mid1 + 1;
         r = mid2 - 1;
      }
   }

   // Key not found
   return -1;
}
```

Time Complexity: $O(Log_3N)$, where N is the number of elements in the array.

Binary Search Functions in C++ STL

So far, we have discussed the Binary Search algorithm and its implementation by writing a function. The C++ standard template library have some built-in functions that can be used to perform Binary Search operation directly on a sequential list or container.

Some of these functions are:

- **binary_search()**
- **upper_bound()**
- **lower_bound()**

binary_search()

This function only checks if a particular element is present in a sorted container or not. It accepts the starting iterator, ending iterator and the element to be checked as parameters and returns True if the element is present otherwise False.

Syntax:

```
binary_search(start_ptr, end_ptr, ele)
```

Below program illustrate the working of binary_search() function with both Arrays and Vectors:

```
// C++ code to demonstrate the working
// of binary_search()

#include<bits/stdc++.h>
using namespace std;

int main()
{
   /*** USING binary_search() ON VECTOR ***/

   // initializing vector of integers
   vector<int> vec = {10, 15, 20, 25, 30, 35};

   // using binary_search to check if 15 exists
   if (binary_search(vec.begin(), vec.end(), 15))
      cout << "15 exists in vector";
   else
      cout << "15 does not exist";

   cout << endl;

   /*** USING binary_search() ON ARRAYS ***/
```

```
int arr[] = {10, 15, 20, 25, 30, 35};
int n = sizeof(arr)/sizeof(arr[0]);

// using binary_search to check if 20 exists
if (binary_search(arr, arr + n, 20))
    cout << "20 exists in Array";
else
    cout << "20 does not exist";

return 0;
}
```

Output:

```
15 exists in vector
20 exists in Array
```

Note: This function only checks if the element is present or not, it does not give any information about the location of the element if it exists.

upper_bound()

The upper_bound() function is used to find the upper bound of an element present in a container. That is it finds the location of an element just greater than a given element in a container. This function accepts the start iterator, end iterator and the element to be checked as parameters and returns the iterator pointing to the element just greater than the element passed as the parameter. If there does not exist any such element than the function returns an iterator pointing to the last element.

Syntax:

```
upper_bound(first_itr, last_itr, ele)
```

Return Value: Returns an iterator pointing to the element just greater than *ele*.

Examples:

```
Input : 10 20 30 30 40 50
Output : upper_bound for element 30 will return
         an iterator pointing to the element 40.

Input : 10 20 30 40 50
Output : upper_bound for element 45 will return
         an iterator pointing to the element 50.

Input : 10 20 30 40 50
Output : upper_bound for element 60 will
         return end iterator.
```

Note: We can calculate the exact index position of the elements by subtracting the beginning iterator from the returned iterator.

Below program illustrate the working of upper_bound() function with both Vectors and Arrays:

```
// CPP program to illustrate using upper_bound()
// with vectors and arrays

#include <bits/stdc++.h>
using namespace std;

// Driver code
int main()
{
    /*** USING upper_bound() WITH VECTOR ***/

    vector<int> v{ 10, 20, 30, 40, 50 };

    // Print vector
    cout << "Vector contains :";
    for (int i = 0; i < v.size(); i++)
        cout << " " << v[i];
    cout << "\n";

    vector<int>::iterator upper1, upper2;

    // std :: upper_bound
    upper1 = upper_bound(v.begin(), v.end(), 35);
    upper2 = upper_bound(v.begin(), v.end(), 45);

    cout << "\nUpper_bound for element 35 is at position : "
         << (upper1 - v.begin());
    cout << "\nUpper_bound for element 45 is at position : "
         << (upper2 - v.begin())<<"\n\n";


    /*** USING upper_bound() WITH ARRAY ***/

    int arr[] = { 10, 20, 30, 40, 50 };

    // Print elements of array
    cout << "Array contains :";
    for (int i = 0; i < 5; i++)
        cout << " " << arr[i];
    cout << "\n";

    // using upper_bound
    int up1 = upper_bound(arr, arr+5, 35) - arr;
    int up2 = upper_bound(arr, arr+5, 45) - arr;

    cout << "\nupper_bound for element 35 is at position : "
         << (up1);
    cout << "\nupper_bound for element 45 is at position : "
         << (up2);

    return 0;
}
```

Output:

```
Vector contains : 10 20 30 40 50

Upper_bound for element 35 is at position : 3
Upper_bound for element 45 is at position : 4

Array contains : 10 20 30 40 50

upper_bound for element 35 is at position : 3
upper_bound for element 45 is at position : 4
```

lower_bound()

The lower_bound() function is used to find the lower bound of an element present in a container. That is it finds the location of an element just smaller than a given element in a container. This function accepts the start iterator, end iterator and the element to be checked as parameters and returns the iterator pointing to the lower bound of the element passed as the parameter. If all elements of the container are smaller are less than the element passed, then it returns the last iterator.

Syntax:

```
lower_bound(first_itr, last_itr, ele)
```

Return Value: Returns an iterator pointing to the lower bound of the element *ele*. That is if *ele* exists in the container, it returns an iterator pointing to *ele* otherwise it returns an iterator pointing to the element just greater than ele.

Below program illustrate the working of lower_bound() function with both Vectors and Arrays:

```
// CPP program to illustrate lower_bound()
// for both vectors and array

#include <bits/stdc++.h>
using namespace std;

// Driver code
int main()
{
    /*** USING lower_bound() ON VECTORS ***/

    vector<int> v{ 10, 20, 30, 40, 50 };

    // Print vector
    cout << "Vector contains :";
    for (int i = 0; i < v.size(); i++)
        cout << " " << v[i];
    cout << "\n";
```

```
    vector<int>::iterator low1, low2;

    // using lower_bound()
    low1 = lower_bound(v.begin(), v.end(), 20);
    low2 = lower_bound(v.begin(), v.end(), 55);

    cout << "\nlower_bound for element 20 at position : "
         << (low1 - v.begin());
    cout << "\nlower_bound for element 55 at position : "
         << (low2 - v.begin());


    /*** USING lower_bound() ON ARRAYS ***/

    int arr[] = { 10, 20, 30, 40, 50 };

    // Print elements of array

    cout << "\n\nArray contains :";
    for (int i = 0; i < 5; i++)
        cout << " " << arr[i];
    cout << "\n";

    // using lower_bound()
    int lb1 = lower_bound(arr, arr + 5, 20) - arr;
    int lb2 = lower_bound(arr, arr + 5, 55) - arr;

    cout << "\nlower_bound for element 20 is at position : "
         << (lb1);
    cout << "\nlower_bound for element 55 is at position : "
         << (lb2);

    return 0;
}
```

Output:

```
Vector contains : 10 20 30 40 50

lower_bound for element 20 at position : 1
lower_bound for element 55 at position : 5

Array contains : 10 20 30 40 50

lower_bound for element 20 is at position : 1
lower_bound for element 55 is at position : 5
```

BinarySearch using Built-in Mehods in Java-

If you are a Java programmer, you must have used built-in methods in Java at some point for basic operations like sorting, reversing etc. Java also provides us methods to perform Binary Search on both Arrays and Collection classes. The most commonly used methods in Java to perform Binary Search are:

- **Arrays.binarySearch()**
- **Collections.binarySearch()**

Let's look at each of the above two methods in details:

Arrays.binarySearch()

Arrays.binarySearch() is the simplest and most efficient method to find an element in a sorted array in Java

Declaration:

```
public static int binarySearch(data_type arr, data_type key )
```

Where **data_type** can be any of the primitive data types: *byte*, *char*, *double*, *int*, *float*, *short*, *long* and *Object* as well.

Description: This method searches the specified array of the given data type for the specified value using the binary search algorithm. The array must be sorted prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found.

Parameters:

- arr - the array to be searched
- key - the value to be searched for

Return Value: It returns the index of the search key, if it is contained in the array; otherwise, (-(insertion point) - 1). The insertion point is defined as the point at which the key would be inserted into the array: the index of the first element greater than the key, or a.length if all elements in the array are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

Examples:

```
Searching for 35 in byteArr[] = {10,20,15,22,35}
will give result as 4 as it is the index of 35

Searching for 35 in charArr[] = {'g','p','q','c','i'}
will give result as 1 as it is the index of 'g'

Searching for 22 in intArr[] = {10,20,15,22,35};
will give result as 3 as it is the index of 22

Searching for 1.5 in doubleArr[] = {10.2,15.1,2.2,3.5}
will give result as -1 as it is the insertion point of 1.5

Searching for 35.0 in floatArr[] = {10.2f,15.1f,2.2f,3.5f}
will give result as -5 as it is the insertion point of 35.0

Searching for 5 in shortArr[] = {10,20,15,22,35}
will give result as -1 as it is the insertion point of 5
```

Implementation:

```
// Java program to demonstrate working of Arrays.
// binarySearch() in a sorted array

import java.util.Arrays;

public class GFG
{
   public static void main(String[] args)
   {
      byte byteArr[] = {10,20,15,22,35};
      char charArr[] = {'g','p','q','c','i'};
      int intArr[] = {10,20,15,22,35};
      double doubleArr[] = {10.2,15.1,2.2,3.5};
      float floatArr[] = {10.2f,15.1f,2.2f,3.5f};
      short shortArr[] = {10,20,15,22,35};

      Arrays.sort(byteArr);
      Arrays.sort(charArr);
      Arrays.sort(intArr);
      Arrays.sort(doubleArr);
      Arrays.sort(floatArr);
      Arrays.sort(shortArr);

      byte byteKey = 35;
      char charKey = 'g';
      int intKey = 22;
      double doubleKey = 1.5;
      float floatKey = 35;
      short shortKey = 5;

      System.out.println(byteKey + " found at index = "
+Arrays.binarySearch(byteArr,byteKey));
      System.out.println(charKey + " found at index = "
+Arrays.binarySearch(charArr,charKey));
      System.out.println(intKey + " found at index = "
+Arrays.binarySearch(intArr,intKey));
      System.out.println(doubleKey + " found at index = "
```

```
+Arrays.binarySearch(doubleArr,doubleKey));

      System.out.println(floatKey + " found at
index = "

+Arrays.binarySearch(floatArr,floatKey));

      System.out.println(shortKey + " found at
index = "

+Arrays.binarySearch(shortArr,shortKey));

   }
}
```

Output:

```
35 found at index = 4
g found at index = 1
22 found at index = 3
1.5 found at index = -1
35.0 found at index = -5
5 found at index = -1
```

Important Points:

- If input list is not sorted, the results are undefined.
- If there are duplicates, there is no guarantee which one will be found.

Collections.binarySearch()

The Collections.binarySearch() method is a Collections class method in Java that returns position of an object in a sorted list.

Declaration:

```
// Returns index of key in sorted list sorted in
// ascending order
public static int binarySearch(List slist, T key)

// Returns index of key in sorted list sorted in
// order defined by Comparator c.
public static int binarySearch(List slist, T key, Comparator c)

If key is not present, the it returns "(-(insertion point) - 1)".
The insertion point is defined as the point at which the key
would be inserted into the list.
```

The method throws **ClassCastException** if elements of list are not comparable using the specified comparator, or the search key is not comparable with the elements.

Searching an int key in a list sorted in ascending order:

```
// Java program to demonstrate working of
Collections.
// binarySearch()
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class GFG
{
   public static void main(String[] args)
   {
      List al = new ArrayList();
      al.add(1);
      al.add(2);
      al.add(3);
      al.add(10);
      al.add(20);

      // 10 is present at index 3.
      int index = Collections.binarySearch(al,
10);
      System.out.println(index);

      // 13 is not present. 13 would have been
inserted
      // at position 4. So the function returns (-4-
1)
      // which is -5.
      index = Collections.binarySearch(al, 15);
      System.out.println(index);
   }
}
```

Output :

```
3
-5
```

Searching an int key in a list sorted in descending order.

```
// Java program to demonstrate working of
Collections.
// binarySearch() in an array sorted in
descending order.
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class GFG
{
   public static void main(String[] args)
   {
      List al = new ArrayList();
      al.add(100);
      al.add(50);
      al.add(30);
      al.add(10);
      al.add(2);

      // The last parameter specifies the
comparator method
      // used for sorting.
      int index = Collections.binarySearch(al, 50,
                  Collections.reverseOrder());

      System.out.println("Found at index " +
index);
   }
}
```

Output :

```
Found at index 1
```

Note: Arrays.binarysearch() works for arrays which can be of primitive data type also. Collections.binarysearch() works for objects Collections like ArrayList and LinkedList.

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
```

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)

1. **Solution : Make two equations using Sum and Product**
    0. Let x be the missing and y be the repeating element.
    1. Get the the sum of Array using formula **S = n(n+1)/2 - x + y**
    2. Get product of Array using formula **P = 1*2*3*...*n * y / x**
    3. 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?

2. **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)

3. **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))

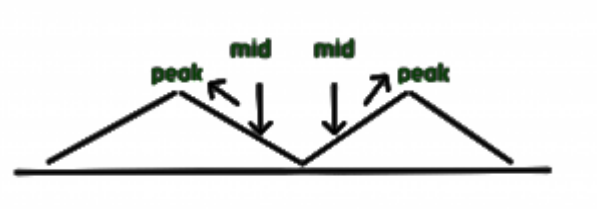
Problem #4 : Find Peak element in Sorted Array

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))

Important problems-

- **Search an Element in an array**
- **Searching an element in a sorted array**
- **Count 1's in binary array**
- **Square root of a number**
- **Majority Element**

HIMANSHU KUMAR(LINKEDIN)

https://www.linkedin.com/in/himanshukumarmahuri

CREDITS- INTERNET.

DISCLOSURE- ALL THE DATA AND IMAGES ARE TAKEN FROM GOOGLE AND INTERNET.