Searching algorithm

1) Majority Element

Approach: Insert elements in BST one by one and if an element is already present then increment the count of the node. At any stage, if the count of a node becomes more than n/2 then return.

```
// C++ program to find Majority element in an array

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

// Function to find Majority element in an array

void findMajority(int arr[], int n)
{
  int maxCount = 0;
  int index = -1; // sentinels
  for (int i = 0; i < n; i++) {
    int count = 0;
    for (int j = 0; j < n; j++) {
      if (arr[i] == arr[j])
        count++;
    }
    // update maxCount if count of current element is greater
    if (count > maxCount) {
      maxCount = count;
      index = i;
    }
  }
 // if maxCount is greater than n/2 return the coresp element
  if (maxCount > n / 2)
    cout << arr[index] << endl;

  else
    cout << "No Majority Element" << endl;
}
// Driver code

```

2) Square root of an integer

The idea is to find the largest integer *i* whose square is less than or equal to the given number. The idea is to use Binary Search to solve the problem. The values of i * i is monotonically increasing, so the problem can be solved using binary search.

```
#include <iostream>

using namespace std;
int floorSqrt(int x)
{
  // Base cases
  if (x == 0 || x == 1)
    return x;

   // Do Binary Search for floor(sqrt(x))
   int start = 1, end = x/2, ans;
   while (start <= end) {
     int mid = (start + end) / 2;

     // If x is a perfect square
     int sqr = mid * mid;
     if (sqr == x)
       return mid;

     // Since we need floor, we update answer when
     // mid*mid is smaller than x, and move closer to
     // sqrt(x)

     /*
     if(mid*mid<=x)
         {
             start = mid+1;
             ans = mid;
```

```
        }
      Here basically if we multiply mid with itsel
 so
    there will be integer overflow which will throw
    tle for larger input so to overcome this
    situation we can use long or we can just divide
      the number by mid which is same as checking
    mid*mid < x
    */
    if (sqr <= x) {
      start = mid + 1;
      ans = mid;
    }
    else // If mid*mid is greater than x
      end = mid - 1;
  }
  return ans;
}

// Driver program
int main()
{
  int x = 20221;
  cout << floorSqrt(x) << endl;
  return 0;
}
```

3) Missing Number in array

Approach: The approach remains the same but there can be an overflow if n is large. In order to avoid integer overflow, pick one number from the known numbers and subtract that one number from the given numbers. This way there won't be any Integer Overflow.

```
class Solution{
 public:
   int MissingNumber(vector<int>& array, int n) {
       long long sum=n*(n+1)/2;
       for(auto n: array)
           sum-=n;
       return sum;
   } };
```

4) Find a peak element

Efficient Approach: Divide and Conquer can be used to find a peak in O(Logn) time. The idea is based on the technique of Binary Search to check if the middle element is the peak element or not. If the middle element is not the peak 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. Form a recursion and the peak element can be found in log n time.

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

// A binary search based function that returns index of a peak element
int findPeakUtil(int arr[], int low,
        int high, int n)
{
  // Find index of middle element
  // low + (high - low) / 2
  int mid = low + (high - low) / 2;

  // Compare middle element with its neighbours (if neighbours exist)
  if ((mid == 0 || arr[mid - 1] <= arr[mid]) &&
    (mid == n - 1 || arr[mid + 1] <= arr[mid]))
    return 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 findPeakUtil(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 findPeakUtil(
      arr, (mid + 1), high, n);
}

```

```cpp
// A wrapper over recursive function findPeakUtil()

int findPeak(int arr[], int n)
{
  return findPeakUtil(arr, 0, n - 1, n);
}

// 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;
}
```