**Square root of an integer**

// Returns floor of square root of x

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, ans;

    while (start <= end)

    {

        int mid = (start + end) / 2;

        // If x is a perfect square

        if (mid\*mid == 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;

        }

        else // If mid\*mid is greater than x

            end = mid - 1;

    }

    return ans;

}

TC : O(logn)

# Find cubic root of a number

**double diff(double n,double mid)**

**{**

**if (n > (mid\*mid\*mid))**

**return (n-(mid\*mid\*mid));**

**else**

**return ((mid\*mid\*mid) - n);**

**}**

**// Returns cube root of a no n**

**double cubicRoot(double n)**

**{**

**// Set start and end for binary search**

**double start = 0, end = n;**

**// Set precision**

**double e = 0.0000001;**

**while (true)**

**{**

**double mid = (start + end)/2;**

**double error = diff(n, mid);**

**// If error is less than e then mid is**

**// our answer so return mid**

**if (error <= e)**

**return mid;**

**// If mid\*mid\*mid is greater than n set**

**// end = mid**

**if ((mid\*mid\*mid) > n)**

**end = mid;**

**// If mid\*mid\*mid is less than n set**

**// start = mid**

**else**

**start = mid;**

**}**

**}**

TC : O(logn)

This is less efficient than O(cuberoot(n)) but this may help in finding the double value (exact).

# Find frequency of each element in a limited range array in less than O(n) time

This problem can be solved in less than O(n) using a modified binary search. The idea is to recursively divide the array into two equal subarrays if its end elements are different. If both its end elements are same, that means that all elements in the subarray is also same as the array is already sorted. We then simply increment the count of the element by size of the subarray.

The time complexity of above approach is O(m log n), where m is number of distinct elements in the array of size n. Since m <= M (a constant), the time complexity of this solution is O(log n).

void findFrequencyUtil(int arr[], int low, int high,

                        vector<int>& freq)

{

    // If element at index low is equal to element

    // at index high in the array

    if (arr[low] == arr[high])

    {

        // increment the frequency of the element

        // by count of elements between high and low

        freq[arr[low]] += high - low + 1;

    }

    else

    {

        // Find mid and recurse for left and right

        // subarray

        int mid = (low + high) / 2;

        findFrequencyUtil(arr, low, mid, freq);

        findFrequencyUtil(arr, mid + 1, high, freq);

    }

}

Worst case space complexity will be O(logn). (stack space - in case every element appears only once).

# Find the element that appears once in a sorted array

Given a sorted array in which all elements appear twice (one after one) and one element appears only once. Find that element in O(log n) complexity.

All elements before the required have first occurrence at even index (0, 2, ..) and next occurrence at odd index (1, 3, …). And all elements after the required element have first occurrence at odd index and next occurrence at even index.

1. Find the middle index, say ‘mid’.
2. If ‘mid’ is even, then compare arr[mid] and arr[mid + 1]. If both are same, then the required element after ‘mid’ else before mid.
3. If ‘mid’ is odd, then compare arr[mid] and arr[mid – 1]. If both are same, then the required element after ‘mid’ else before mid.

# Find the missing number in Arithmetic Progression

Given an array that represents elements of arithmetic progression in order. One element is missing in the progression, find the missing number.

Input: arr[] = {2, 4, 8, 10, 12, 14}

Output: 6

Input: arr[] = {1, 6, 11, 16, 21, 31};

Output: 26

* The idea is to go to the middle element.
* Check if the difference between middle and next to middle is equal to diff or not, if not then the missing element lies between mid and mid+1.
* If the middle element is equal to n/2th term in Arithmetic Series (Let n be the number of elements in input array), then missing element lies in right half. Else element lies in left half.

TC : O(logn)