DAY 31/180

Q1- Find first and last position of element in a sorted array.

```cpp
// This helper function performs binary search to find the start or end position of the target value in the given 'nums' array.
int help(vector<int>& nums, int target, int find) {
    int left = 0;  // Initialize the left boundary of the search range.
    int right = nums.size() - 1;  // Initialize the right boundary of the search range.
    int result = -1;  // Initialize the result to -1 in case the target is not found.

    // Perform binary search as long as the left boundary is less than or equal to the right boundary.
    while (left <= right) {
        int mid = (left + right) / 2;  // Calculate the middle index of the current search range.

        if (nums[mid] == target) {  // If the middle element is equal to the target:
            result = mid;  // Update the result to the current middle index.

            if (find == 1) {
                right = mid - 1;  // If we are finding the start position, move the right boundary to the left of mid.
            } else {
                left = mid + 1;  // If we are finding the end position, move the left boundary to the right of mid.
            }
        } else if (nums[mid] > target) {
            right = mid - 1;  // If the middle element is greater than the target, update the right boundary.
        } else {
            left = mid + 1;  // If the middle element is less than the target, update the left boundary.
        }
    }

    return result;  // Return the result (start or end position of the target value).
}

// This function searches for the range of positions where the target value appears in the 'nums' array.
vector<int> searchRange(vector<int>& nums, int target) {
    int start = help(nums, target, 1);  // Find the start position of the target value.
    int end = help(nums, target, 2);  // Find the end position of the target value.

    return {start, end};  // Return a vector containing the start and end positions of the target value.
}
```

Q2-Search Insert Position

```
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        int s = 0;           // Initialize the start index of the search range.
        int n = nums.size(); // Get the size of the 'nums' array.
        int e = n - 1;       // Initialize the end index of the search range.
        int mid = s + (e - s) / 2;  // Calculate the initial middle index.
        // Perform binary search as long as the start index is less than or equal to the end index.
        while (s <= e) {
            if (nums[mid] > target) {
                e = mid - 1;  // If the middle element is greater than the target, update the end index.
            } else if (nums[mid] < target) {
                s = mid + 1;  // If the middle element is less than the target, update the start index.
            } else if (nums[mid] == target) {
                return mid;  // If the middle element is equal to the target, return the current middle index.
            }
            mid = s + (e - s) / 2;  // Recalculate the middle index for the next iteration.
        }

        return mid;  // Return the 'mid' index, which is the index where 'target' should be inserted.
    }
};
```

Q3- Sqrt(X) by binary search.

```cpp
#define ll long long  // Define 'll' as a shorthand for 'long long'.
class Solution {
public:
    // This function finds the integer square root of the given integer 'x'.
    int mySqrt(int x) {
        ll s = 1;          // Initialize the start of the search range.
        ll e = x;          // Initialize the end of the search range.
        ll ans;            // Initialize a variable to store the answer.
        // Perform binary search as long as the start index is less than or equal to the end index.
        while (s <= e) {
            ll mid = s + (e - s) / 2;  // Calculate the middle point of the search range.

            if (mid * mid == x) {
                return mid;  // If the middle value squared equals 'x', return 'mid' as the square root.
            } else if (mid * mid > x) {
                e = mid - 1;  // If the square of the middle value is greater than 'x', update the end index.
            } else {
                ans = mid;  // Update 'ans' to the current middle value as a potential answer.
                s = mid + 1;  // If the square of the middle value is less than 'x', update the start index.
            }
            mid = s + (e - s) / 2;  // Recalculate the middle point for the next iteration.
        }
        return ans;  // Return 'ans' as the integer square root of 'x'.
    }
};
```

Q4- Kth missing positive number.

```cpp
class Solution {
public:
    // This function finds the k-th missing positive integer in the given sorted 'arr'.
    int findKthPositive(vector<int>& arr, int k) {
        int n = arr.size();  // Get the size of the 'arr' array.
        int s = 0;          // Initialize the start index of the search range.
        int e = n - 1;      // Initialize the end index of the search range.

        // Perform binary search as long as the start index is less than or equal to the end index.
        while (s <= e) {
            int mid = (s + e) / 2;  // Calculate the middle index.

            int missing = arr[mid] - (mid + 1);  // Calculate the number of missing elements up to 'arr[mid]'.

            if (missing < k) {
                s = mid + 1;  // If there are fewer missing elements than 'k', update the start index.
            } else {
                e = mid - 1;  // If there are more or equal missing elements than 'k', update the end index.
            }
        }

        // The 's' index points to the position where the k-th missing positive integer should be.
        return s + k;  // Return the k-th missing positive integer.
    }
};
```

Q5- Count the Zeros

```cpp
class Solution {
public:
    // This function counts the number of zeros in a sorted array of 0s and 1s.
    int countZeroes(int arr[], int n) {
        int ans = 0;  // Initialize the count of zeros.
        int s = 0;    // Initialize the start index of the search range.
        int e = n - 1;  // Initialize the end index of the search range.
        // Perform binary search as long as the start index is less than or equal to the end index.
        while (s <= e) {
            int mid = (s + e) / 2;  // Calculate the middle index.
            if (arr[mid] == 1) {
                s = mid + 1;  // If the middle element is 1, update the start index.
            } else {
                // If the middle element is 0, update 'ans' with the count of zeros in the right
                ans += (e - mid + 1);
                e = mid - 1;  // Update the end index.
            }
        }
        return ans;  // Return the total count of zeros in the array.
    }
};
```

Q6- Number of Occurrences.

```cpp
int count(int arr[], int n, int x) {
    int left = 0;      // Initialize the left boundary of the search range.
    int right = n - 1;  // Initialize the right boundary of the search range.
    int first = -1;    // Initialize 'first' to -1 (no occurrences found).
    int last = -1;     // Initialize 'last' to -1 (no occurrences found).
    // Perform binary search to find the first occurrence of 'x'.
    while (left <= right) {
        int mid = (left + right) / 2;  // Calculate the middle index.
        if (arr[mid] == x) {
            first = mid;  // Update 'first' to the current middle index.
            right = mid - 1;  // Move the right boundary to the left of mid.
        } else if (arr[mid] > x) {
            right = mid - 1;  // If the middle element is greater, update the right boundary.
        } else {
            left = mid + 1;  // If the middle element is smaller, update the left boundary.
        }
    }
    // If 'first' is still -1, 'x' was not found, so return 0 occurrences.
    if (first == -1) {
        return 0;
    }
    left = 0;       // Reset the left boundary.
    right = n - 1;  // Reset the right boundary.
    // Perform binary search to find the last occurrence of 'x'.
    while (left <= right) {
        int mid = left + (right - left) / 2;  // Calculate the middle index.
        if (arr[mid] == x) {
            last = mid;  // Update 'last' to the current middle index.
            left = mid + 1;  // Move the left boundary to the right of mid.
        } else if (arr[mid] < x) {
            left = mid + 1;  // If the middle element is smaller, update the left boundary.
        } else {
            right = mid - 1;  // If the middle element is greater, update the right boundary.
        }
    }
    // Return the count of occurrences by subtracting 'first' from 'last' and adding 1.
    return last - first + 1;
}
```

Q7- Cube Root of a number.

```cpp
class Solution {
public:
    int cubeRoot(int N) {
        // If 'N' is 1, return 1 since the cube root is also 1.
        if (N == 1) return 1;
        ll s = 0;       // Initialize the start value for binary search.
        ll e = N;       // Initialize the end value for binary search.
        ll ans = 0;     // Initialize a variable to store the answer.
        // Perform binary search as long as the start value is less than or equal to the end value.
        while (s <= e) {
            ll mid = (s + e) / 2;  // Calculate the middle value.

            if (mid * mid * mid > N) {
                e = mid - 1;  // If the cube of the middle value is greater than 'N', update the end value.
            } else if (mid * mid * mid == N) {
                return mid;  // If the cube of the middle value equals 'N', return 'mid' as the cube root.
            } else {
                ans = mid;    // Update 'ans' to the current middle value as a potential answer.
                s = mid + 1;  // If the cube of the middle value is less than 'N', update the start value.
            }
        }
        return ans;  // Return 'ans' as the cube root of 'N'.
    }
};
```