Strivers-A2Z-DSA-Sheet-main\02.Binary Search\1D Arrays\01.Find_x_in_sorted_array.cpp

```cpp
/*
QUESTION:-
Given an array of integers nums which is sorted in ascending order, and an integer target, write a function to search target in nums. If target exists, then return its index. Otherwise, return -1.

Example 1:
Input: nums = [-1,0,3,5,9,12], target = 9
Output: 4
Explanation: The target value 9 exists in the nums array, and its index is 4.

Example 2:
Input: nums = [-1,0,3,5,9,12], target = 2
Output: -1
Explanation: The target value 2 does not exist in the nums array, so return -1.
*/

/*
APPROACH:-
1. Initialize low as 0 and high as the last index of the array.
2. Iterate using a while loop until low is less than or equal to high.
3. Calculate the middle index using the formula mid = low + (high - low) / 2.
4. Compare the target value with the element at the middle index:
   - If they are equal, return the middle index.
   - If the target is less than the element, update high to mid - 1 and continue the search in the left half.
   - If the target is greater than the element, update low to mid + 1 and continue the search in the right half.
5. If the target is not found, return -1.
*/

//CODE:-
int search(vector<int>& nums, int target) {
    int low = 0, high = nums.size() - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (nums[mid] == target)
            return mid;
        else if (nums[mid] > target)
            high = mid - 1;
        else
            low = mid + 1;
    }
    return -1;
}


// TIME COMPLEXITY: O(log n)
// - The algorithm divides the search space in half at each step, resulting in a logarithmic time complexity.

// SPACE COMPLEXITY: O(1)
```

Strivers-A2Z-DSA-Sheet-main\02.Binary Search\1D Arrays\02.Implement_lower_bound.cpp

```cpp
/*
QUESTION:
Given a sorted array arr[] of size N without duplicates, and given a value x. Floor of x is defined as the largest element K in arr[] such that K is smaller than or equal to x. Find the index of K (0-based indexing).

Example 1:
Input:
N = 7, x = 0
arr[] = {1,2,8,10,11,12,19}
Output: -1
Explanation: No element less than 0 is found. So the output is "-1".

Example 2:
Input:
N = 7, x = 5
arr[] = {1,2,8,10,11,12,19}
Output: 1
Explanation: Largest number less than 5 is 2 (i.e K = 2), whose index is 1 (0-based indexing).

APPROACH:
- Initialize low as 0 and high as N-1.
- Iterate using a while loop until low is less than or equal to high.
- Calculate the mid index using mid = low + (high - low) / 2.
- Check if the element at mid index is less than or equal to x.
  - If true, update the answer as mid and move the low pointer to mid+1 to search for a larger element.
  - If false, update the high pointer to mid-1 to search in the lower half of the array.
- Finally, return the answer.

CODE:
*/

int findFloor(vector<long long> v, long long n, long long x) {
    long long low = 0, high = n - 1;
    int ans = -1;
    while (low <= high) {
        long long mid = low + (high - low) / 2;
        if (v[mid] <= x) {
            ans = mid;
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return ans;
}

// TIME COMPLEXITY: O(log N)
// SPACE COMPLEXITY: O(1)

```

Strivers-A2Z-DSA-Sheet-main\02.Binary Search\1D Arrays\03.Implement_lower_upper_bound.cpp

```cpp
/*
QUESTION:
Given an unsorted array Arr[] of N integers and an integer X, find floor and ceiling of X in Arr[0..N-1].

Floor of X is the largest element which is smaller than or equal to X. Floor of X doesn’t exist if X is smaller than the smallest element of Arr[].

Ceil of X is the smallest element which is greater than or equal to X. Ceil of X doesn’t exist if X is greater than the greatest element of Arr[].

Example:

Input:
N = 8, X = 7
Arr[] = {5, 6, 8, 9, 6, 5, 5, 6}
Output: 6 8
Explanation:
Floor of 7 is 6 and ceil of 7 is 8.

APPROACH:
1. Sort the array in ascending order.
2. Use binary search to find the floor and ceil values.
3. The floor value is the largest element smaller than or equal to X, and the ceil value is the smallest element greater than or equal to X.
4. If the floor or ceil values are not found, set them to -1.

CODE:
*/

int lowerbound(int arr[], int n, int x){
    int low = 0, high = n-1;
    int ans = -1;
    while(low<=high){
        int mid = low+(high-low)/2;
        if(arr[mid]<=x){
            ans = mid;
            low = mid+1;
        }
        else{
            high = mid-1;
        }
    }
    if(ans!=-1) ans = arr[ans];
    return ans;
}

int upperbound(int arr[], int n, int x){
    int low = 0, high = n-1;
    int ans = -1;
    while(low<=high){
        int mid = low+(high-low)/2;
```

```cpp
        if(arr[mid]>=x){
            ans = mid;
            high = mid-1;
        }
        else{
            low = mid+1;
        }
    }
    if(ans!=-1) ans = arr[ans];
    return ans;
}

pair<int, int> getFloorAndCeil(int arr[], int n, int x) {
    sort(arr,arr+n);
    int Floor = lowerbound(arr,n,x);
    int Ceil = upperbound(arr,n,x);
    return {Floor,Ceil};
}

// TIME COMPLEXITY: O(NlogN)
// SPACE COMPLEXITY: O(1)

```

Strivers-A2Z-DSA-Sheet-main\02.Binary Search\1D Arrays\04.Search_insert_position.cpp

```cpp
/*
QUESTION:
Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with O(log n) runtime complexity.

Example:

Input: nums = [1,3,5,6], target = 5
Output: 2
Example 2:

Input: nums = [1,3,5,6], target = 2
Output: 1
*/

/*
APPROACH:
We can use the lower_bound function from the C++ standard library to find the index where the target should be inserted. The lower_bound function returns an iterator pointing to the first element that is not less than the target. By subtracting the beginning iterator from the lower_bound iterator, we get the index where the target should be inserted.

CODE:
*/

int searchInsert(vector<int>& nums, int target) {
    auto ans = lower_bound(nums.begin(), nums.end(), target) - nums.begin();
    return ans;
}

// TIME COMPLEXITY: O(log n) due to the use of lower_bound function
// SPACE COMPLEXITY: O(1)

```

Strivers-A2Z-DSA-Sheet-main\02.Binary Search\1D Arrays\05.Check_If_array_is_sorted.cpp

```cpp
/*
QUESTION:
Given an array arr[] of size N, check if it is sorted in non-decreasing order or not.

APPROACH:
- We can use a recursive approach to check if the array is sorted in non-decreasing order or not.
- The base case for recursion is when the subarray has only one element or when the subarray is empty, in which case we consider it to be sorted.
- For a non-empty subarray, we compare the middle element with its next element. If they are in non-decreasing order and both the left and right subarrays are also sorted, then we consider the entire array to be sorted.
- We recursively check the left and right subarrays using the same approach.
- If any of the recursive calls returns false, we return false. Otherwise, we return true.

Example:

Input:
N = 5
arr[] = {10, 20, 30, 40, 50}
Output: 1
Explanation: The given array is sorted.

CODE:
*/

bool solve(int arr[], int low, int high) {
    if (low >= high)
        return true;

    int mid = low + (high - low) / 2;
    if (arr[mid] <= arr[mid + 1] && solve(arr, low, mid) && solve(arr, mid + 1, high))
        return true;

    return false;
}

bool arraySortedOrNot(int arr[], int n) {
    return solve(arr, 0, n - 1);
}

// TIME COMPLEXITY: O(log N)
// SPACE COMPLEXITY: O(log N) (for recursion stack)

```

Strivers-A2Z-DSA-Sheet-main\02.Binary Search\1D Arrays\06.First_and_last_position.cpp

```cpp
/*
QUESTION:
Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.

If target is not found in the array, return [-1, -1].

You must write an algorithm with O(log n) runtime complexity.

Example:

Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]

APPROACH:
1. Use lower_bound to find the index of the first occurrence of the target in the array.
2. If the target is not found, return [-1, -1].
3. Use upper_bound to find the index of the last occurrence of the target in the array.
4. Return the range [first, last-1] as the starting and ending positions.

CODE:
*/

vector<int> searchRange(vector<int>& nums, int target) {
    int first = lower_bound(nums.begin(), nums.end(), target) - nums.begin();
    // if the target is not found, return [-1, -1]
    if (first == nums.size() || nums[first] != target)
        return {-1, -1};
    int last = upper_bound(nums.begin(), nums.end(), target) - nums.begin();
    return {first, last-1};
}

// TIME COMPLEXITY: O(log n)
// SPACE COMPLEXITY: O(1)

```

Strivers-A2Z-DSA-Sheet-main\02.Binary Search\1D Arrays\07.Number_of_occurences.cpp

```cpp
/*
QUESTION:
Given a sorted array Arr of size N and a number X, you need to find the number of occurrences of X in Arr.

Example:

Input:
N = 7, X = 2
Arr[] = {1, 1, 2, 2, 2, 2, 3}
Output: 4
Explanation: 2 occurs 4 times in the given array.
Example 2:

Input:
N = 7, X = 4
Arr[] = {1, 1, 2, 2, 2, 2, 3}
Output: 0
Explanation: 4 is not present in the given array.
*/

/*
APPROACH:
1. Use binary search to find the first occurrence of the target element.
2. Use binary search to find the last occurrence of the target element.
3. Return the difference between the indices of the first and last occurrence + 1.

CODE:
*/

int countOccurrences(int arr[], int n, int x) {
    int first = lower_bound(arr, arr + n, x) - arr;
    if (first == n || arr[first] != x)
        return 0;
    int last = upper_bound(arr, arr + n, x) - arr;
    return last - first;
}

// TIME COMPLEXITY: O(log N), where N is the size of the array.
// SPACE COMPLEXITY: O(1).

```

Strivers-A2Z-DSA-Sheet-main\02.Binary Search\1D Arrays\08.Find_peak_element.cpp

```cpp
/*
QUESTION:-
A peak element is an element that is strictly greater than its neighbors.

Given a 0-indexed integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks.

You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

You must write an algorithm that runs in O(log n) time.

Example 1:
Input: nums = [1,2,3,1]
Output: 2
Explanation: 3 is a peak element and your function should return the index number 2.

Example 2:
Input: nums = [1,2,1,3,5,6,4]
Output: 5
Explanation: Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.
*/

/*
APPROACH:-
We can use the binary search approach to find the peak element.
1. Initialize low = 0 and high = n-1, where n is the size of the array.
2. While low < high, calculate mid = low + (high - low) / 2.
3. If nums[mid] < nums[mid+1], it means a peak element exists on the right side of mid, so update low = mid+1.
4. Otherwise, a peak element exists on the left side of mid or mid itself is a peak, so update high = mid.
5. After the loop ends, low will be pointing to the peak element index.
6. Return low as the result.

CODE:-
*/

int findPeakElement(vector<int>& nums) {
    int low = 0, high = nums.size()-1;
    while(low < high){
        int mid = low + (high - low) / 2;
        if(nums[mid] < nums[mid+1])
            low = mid+1;
        else
            high = mid;
    }
    return low;
}

// TIME COMPLEXITY: O(log n)
```

```
48 // SPACE COMPLEXITY: O(1)
49
```

Strivers-A2Z-DSA-Sheet-main\02.Binary Search\1D Arrays\09.Search_in_rotated_sorted_array.cpp

```
/*
QUESTION:
There is an integer array nums sorted in ascending order (with distinct values).

Prior to being passed to your function, nums is possibly rotated at an unknown pivot index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (0-indexed). For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].

Given the array nums after the possible rotation and an integer target, return the index of target if it is in nums, or -1 if it is not in nums.

You must write an algorithm with O(log n) runtime complexity.

Example 1:
Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4

Example 2:
Input: nums = [4,5,6,7,0,1,2], target = 3
Output: -1
*/

/*
APPROACH:
We can use the binary search approach to find the target element in the rotated sorted array.
1. Initialize low = 0 and high = nums.size() - 1, where nums is the input array.
2. Perform binary search using the while loop until low <= high.
3. Calculate mid = low + (high - low) / 2.
4. If nums[mid] is equal to the target, return mid.
5. Check if the left part of the array (nums[low] to nums[mid]) is sorted or the right part (nums[mid] to nums[high]) is sorted.
   - If the left part is sorted:
     - If the target is within the range of nums[low] and nums[mid], update high = mid - 1.
     - Otherwise, update low = mid + 1.
   - If the right part is sorted:
     - If the target is within the range of nums[mid] and nums[high], update low = mid + 1.
     - Otherwise, update high = mid - 1.
6. If the target is not found after the while loop, return -1.

CODE:
*/

int search(vector<int>& nums, int target) {
    int low = 0, high = nums.size() - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (nums[mid] == target)
            return mid;
        if (nums[low] <= nums[mid]) {
            if (nums[low] <= target && target <= nums[mid])
                high = mid - 1;
```

```
48              else
49                  low = mid + 1;
50          } else {
51              if (nums[mid] <= target && target <= nums[high])
52                  low = mid + 1;
53              else
54                  high = mid - 1;
55          }
56      }
57      return -1;
58  }
59
60  // TIME COMPLEXITY: O(log n)
61  // SPACE COMPLEXITY: O(1)
62
```

Strivers-A2Z-DSA-Sheet-main\02.Binary Search\1D Arrays\10.Search_in_rotated_sorted_array_with_duplicates.cpp

```
/*
QUESTION:
There is an integer array nums sorted in non-decreasing order (not necessarily with distinct values).

Before being passed to your function, nums is rotated at an unknown pivot index k (0 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (0-indexed). For example, [0,1,2,4,4,4,5,6,6,7] might be rotated at pivot index 5 and become [4,5,6,6,7,0,1,2,4,4].

Given the array nums after the rotation and an integer target, return true if target is in nums, or false if it is not in nums.

You must decrease the overall operation steps as much as possible.

Example 1:
Input: nums = [2,5,6,0,0,1,2], target = 0
Output: true

Example 2:
Input: nums = [2,5,6,0,0,1,2], target = 3
Output: false
*/

/*
APPROACH:
We can modify the standard binary search algorithm to search for the target element.
1. Initialize low = 0 and high = nums.size() - 1.
2. While low <= high, calculate mid = low + (high - low) / 2.
3. If nums[mid] equals the target, return true.
4. If nums[mid] is equal to nums[low], we are in a situation where we can't determine which part of the array is sorted.
   In this case, we increment low and decrement high to skip the duplicate elements.
5. If the left part of the array from low to mid is sorted, check if the target lies within this range.
   If so, update high = mid - 1. Otherwise, update low = mid + 1.
6. If the right part of the array from mid to high is sorted, check if the target lies within this range.
   If so, update low = mid + 1. Otherwise, update high = mid - 1.
7. If the target is not found, return false.

CODE:
*/

bool search(vector<int>& nums, int target) {
    int low = 0, high = nums.size() - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (nums[mid] == target)
            return true;
        if (nums[mid] == nums[low] && nums[mid] == nums[high]) {
```

```
            low++;
            high--;
            continue;
        }
        if (nums[low] <= nums[mid]) {
            if (nums[low] <= target && target <= nums[mid])
                high = mid - 1;
            else
                low = mid + 1;
        } else {
            if (nums[mid] <= target && target <= nums[high])
                low = mid + 1;
            else
                high = mid - 1;
        }
    }
    return false;
}

// TIME COMPLEXITY: O(log n)
// SPACE COMPLEXITY: O(1)

```

Strivers-A2Z-DSA-Sheet-main\02.Binary Search\1D Arrays\11.Find_the_minimum_element_in_sorted_rotated_array.cpp

```
/*
QUESTION:
Suppose an array of length n sorted in ascending order is rotated between 1 and n times. For example, the array nums = [0,1,2,4,5,6,7] might become:

[4,5,6,7,0,1,2] if it was rotated 4 times.
[0,1,2,4,5,6,7] if it was rotated 7 times.

Given the sorted rotated array nums of unique elements, return the minimum element of this array.

APPROACH:
We can use the binary search approach to find the minimum element.
1. Initialize low = 0 and high = n-1, where n is the size of the array.
2. While low < high, calculate mid = low + (high - low) / 2.
3. If nums[mid] > nums[high], it means the minimum element is on the right side of mid, so update low = mid+1.
4. Otherwise, the minimum element is on the left side of mid or mid itself, so update high = mid.
5. After the loop ends, low will be pointing to the minimum element index.
6. Return nums[low] as the result.

CODE:
*/

int findMin(vector<int>& nums) {
    int low = 0, high = nums.size()-1;
    while(low < high){
        int mid = low + (high - low) / 2;
        if(nums[mid] > nums[high])
            low = mid+1;
        else
            high = mid;
    }
    return nums[low];
}

// TIME COMPLEXITY: O(log n)
// SPACE COMPLEXITY: O(1)

```

Strivers-A2Z-DSA-Sheet-main\02.Binary Search\1D Arrays\12.Find_single_element_in_sorted_array.cpp

```cpp
/*
QUESTION:
You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once.

Return the single element that appears only once.

APPROACH:
Since the array is sorted and every element appears exactly twice except for one element, we can use binary search to find the single element.
1. Initialize low = 0 and high = nums.size()-1, where nums is the input array.
2. While low < high, calculate mid = low + (high - low) / 2.
3. Check if mid is an even index (mid % 2 == 0).
    - If nums[mid] is equal to nums[mid + 1], it means the single element is on the right side, so update low = mid + 1.
    - Otherwise, the single element is on the left side, so update high = mid.
4. If mid is an odd index (mid % 2 == 1).
    - If nums[mid] is not equal to nums[mid + 1], it means the single element is on the right side, so update low = mid + 1.
    - Otherwise, the single element is on the left side, so update high = mid.
5. After the loop ends, low will be pointing to the single element.
6. Return nums[low] as the result.

CODE:
*/

int singleNonDuplicate(vector<int>& nums) {
    int low = 0, high = nums.size() - 1;
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (mid % 2 == 0) {
            if (nums[mid] == nums[mid + 1])
                low = mid + 1;
            else
                high = mid;
        } else {
            if (nums[mid] != nums[mid + 1])
                low = mid + 1;
            else
                high = mid;
        }
    }
    return nums[low];
}

// TIME COMPLEXITY: O(log n)
// SPACE COMPLEXITY: O(1)

```

Strivers-A2Z-DSA-Sheet-main\02.Binary Search\1D Arrays\13.Find_how_many_times_array_is_rotated.cpp

```
/*
QUESTION:
Given an ascending sorted rotated array Arr of distinct integers of size N. The array is right rotated K times. Find the value of K.

Example 1:

Input:
N = 5
Arr[] = {5, 1, 2, 3, 4}
Output: 1
Explanation: The given array is 5 1 2 3 4.
The original sorted array is 1 2 3 4 5.
We can see that the array was rotated
1 times to the right.

APPROACH:
To find the value of K, we can use binary search.
1. Initialize low = 0 and high = N-1, where N is the size of the array.
2. While low < high, calculate mid = low + (high - low) / 2.
3. Check if arr[mid] > arr[n-1].
    - If true, it means the rotation point lies on the right side of mid, so update low = mid + 1.
    - If false, it means the rotation point lies on the left side of mid or mid is the rotation point, so update high = mid.
4. After the loop ends, low will be pointing to the rotation point.
5. Return low as the value of K.

CODE:
*/

int findKRotation(int arr[], int n) {
    int low = 0, high = n - 1;
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] > arr[n - 1])
            low = mid + 1;
        else
            high = mid;
    }
    return low;
}

// TIME COMPLEXITY: O(log n)
// SPACE COMPLEXITY: O(1)

```