Binary Search

Introduction

Binary search is one of the most popular searching algorithms. It is time-efficient and also one of the most commonly used problem-solving techniques. It takes only the order of **$O(\log_2(n))$** time to search an element if there are ***n*** elements present, whereas linear search takes the order of **O(n).** Therefore, it is quite a lot faster and efficient than linear search.

In this technique, we keep dividing our array into halves until we find the element which we are searching for. After every iteration/step, our searching space gets decreased therefore its time-complexity is quite low as compared to linear search.

Before applying binary search the thing which we should keep in our mind is that the set of elements should be in an **ordered fashion (sorted).**

Algorithm

- As mentioned before, we can apply binary search only if the array is sorted. So, if the array is not sorted our first step should be to sort it.
- We take two pointers ***left*** and ***right***, initially ***left=0*** and ***right=n-1***.(where n is the size of the array and indexing is zero based).
- We found the middle index of the left and right pointer **mid=(left + (left - right)/2).**
- Compare the element at index ***mid*** with the ***target number***.

- If the element at index ***mid*** is greater than the target ***number*** it means that the target number should lie between ***left*** and ***mid-1.*** Therefore, we can reduce the searching size by making ***right=mid-1.***
- If the element at index ***mid*** is less than the target ***number*** it means that the target number should lie between ***mid+1*** and ***left.*** Therefore, we can reduce the searching size by making ***left=mid+1.***
- We keep doing this process repeatedly until we find the ***target number*** or our ***left*** is less than ***right.***

Implementation

• Recursive

```
bool binary_search(int nums[], int target_num, int left, int right)
{
      int mid;
      if(right>=left){

      mid=left+(right-left)/2;
      if(nums[mid]==target_num)return true;
      else if(nums[mid]>target_num)return binary_search(nums,target_num,left,mid-1);
      else  return binary_search(nums,target_num,mid+1,right);
      }
      return -1;
}
```

• Iterative

```
bool binary_search(int nums[],int target_num){
     int  left=0,right=nums.length();
     while(left<=right){
           int mid=(left+right)/2;
           if(nums[mid]==target_num)return true;
           if(nums[mid]>target_num)right=mid-1
           else if(nums[mid]<target_num)left=mid+1
     }
     return false;
}
```

Time-complexity: O($\log_2$(n)) where ***n*** is the number of elements.

As we are dividing our array into halves after every iteration, we can divide our array into two halves at max **$\log_2$(n)** time.

Space-complexity: O(1)

As we're just using extra space as variables, therefore, the space-complexity is constant.

Aggressive Cows

Problem statement:
Given an array of length **'N'**, where each element denotes the position of a stall. Now you have **'N'** stalls and an integer **'K'** which denotes the number of cows that are aggressive. To prevent the cows from hurting each other, you need to assign the cows to the stalls, such that the minimum distance between any two of them is as large as possible. Return the largest minimum distance.

Approach:

- First of all, we make a function ***check(x),*** which checks whether the distance ***x*** can be possible between each of the cows or not.
- We can do it greedily by placing the cows in the left-most possible stalls such that the distance between the current cow and the last-placed cow is at least ***x.***
- As the array is not sorted in order, therefore, we have to sort so that we can apply ***binary search.***
- Now, we apply a binary search algorithm using a **check*(x)*** function in order to find the maximum possible distance between each of the cows.
- Initially, we take ***left(l) = 0*** (minimum answer possible) and ***right(r)*** **=100000005** (maximum possible answer without violating constraints).

- Now, we run a while loop until **l** < **r,** and at every iteration, we take the middle of left and right **mid** =((**l+r)/2**) and pass ***mid*** as an argument of the check***(mid)*** function.
- If ***check(mid)*** returns **true,** it means it is possible to place cows at a distance greater than or equal to ***mid.*** It implies that our answer lies in the range***[mid, r].*** Therefore we equate ***l= mid.***
- If ***check(mid)*** returns ***false*** it means it is not possible to place cows at a distance greater than or equal to ***mid.*** It implies that our answer lies in the range***[l, mid].*** Therefore we equate ***r= mid.***

Code:

```
static boolean check(int barn[] , int k , int mid){
    int total=1;  // we already place it at the first available slot i.e barn[0] (GREEDY)
    int last=0;
    int n = barn.length;
    for(int i=1;i<n;i++){
       if(barn[i]-barn[last]>=mid){
          total++;
          last=i;
       }
    }
    return (total>=k);
 }

public static void main(String[] args){
     Scanner sc = new Scanner(System.in);
     int tt = sc.nextInt();
     while(tt>0){
        int n = sc.nextInt();
        int k = sc.nextInt();
        int barn[] = new int[n];
        for(int i=0;i<n;i++) barn[i] = sc.nextInt();
        Arrays.sort(barn);

    //binary search
        int l=1,r=100000005;
        int ans=100000005;
        while(l<=r){
```

```
            int mid = l+(r-l)/2;
            if(check(barn,k,mid)==true){
               ans=mid;
               l=mid+1;
            }
            else{
               r=mid-1;
            }
         }
         System.out.println(ans);
         tt--;
      }
   }
}
```

Time-complexity: O(n*(log(n))) where n is the number of stalls as we are doing sorting which takes O(n*log(n)) time.

Space complexity: O(1) as we're just using extra space as variables, therefore, the space-complexity is constant.

Search in Rotated Sorted Array

Problem statement:

Aahad and Harshit always have fun by solving problems. Harshit took a sorted array and rotated it clockwise by an unknown amount. For example, he took a sorted **array = [1, 2, 3, 4, 5]** and if he rotates it by **2**, then the array becomes: **[4, 5, 1, 2, 3]**. After rotating a sorted array, Aahad needs to answer **Q** queries asked by Harshit; each of them is described by one integer **Q[i]** which Harshit wanted him to search in the array. For each query, if he found it, he had to shout the index of the number, otherwise, he had to shout **-1.**

Approach:

The main idea is to use the binary search technique to determine the position of **X** within **array A**. If we look at the value at position middle right now, we'll encounter three different scenarios:

- **A[middle] = X**: this means that we found our target and returned its index.
- **A[middle] < X**: this means we need to find a larger element, so we'll see if the leftmost element is greater than **A[middle]** and less than or equal to the target **X**. In this case, our target should come after the last element on the left. As a result, we might find it on the array's left side. Otherwise, it should be on the correct side.
- **A[middle] > X**: It implies that we must seek a smaller element. As a result, we look to see if the rightmost element is less than **A[middle]** and greater than or equal to **X**. If this is the case, our target should come before the rightmost element. As a result, we can locate it in the right segment of **A**. If not, it should be on the left side.

Finally, if we finish the binary search without returning any results, then our target **X** does not exist in **A**. As a result, we return a **-1** value.

Code:

```
int binarySearch(int nums[] , int left, int right , int target){
    while(left<=right){
      int mid = left + (right-left)/2;

      if(nums[mid]==target) return mid;
      if(nums[mid]<target) left=mid+1;
      else right=mid-1;
    }
    return -1;
  }

  static int findLargestElement(int nums[] , int left, int right){
    int n = nums.length;
    while(left<=right){
      int mid = left + (right-left)/2;

      if(mid-1>=0 && nums[mid-1]<nums[mid] && mid+1<n && nums[mid+1]<nums[mid])
```

```
        return mid;

        else if(mid==0 && mid+1<n && nums[mid+1]<nums[mid])
        return mid;

        else if(mid==n-1 && mid-1>=0 && nums[mid-1]<nums[mid])
        return mid;

        if(nums[mid]>=nums[left])
         left=mid+1;

        else right=mid-1;
      }
      return -1;
    }

public static void main(String[] args){
      Scanner sc= new Scanner(System.in);
      int tt = sc.nextInt();
      while(tt>0){
        int n = sc.nextInt();
        int nums[]=new int[n];
        for(int i=0;i<n;i++) nums[i]=sc.nextInt();

        int pivot = findLargestElement(nums,0,n-1);

        int q;
        q = sc.nextInt();
        while(q>0){
          int target = sc.nextInt();
          int res1 = binarySearch(nums,pivot+1,n-1,target);
          int res2 = binarySearch(nums,0,pivot,target);

          if(res1!=-1) System.out.println(res1);
          else if(res2!=-1) System.out.println(res2);
          else System.out.println(-1);
          q--;
        }
        tt--;
      }
 }
```

Time-complexity: **O($\log_2$(n))**, Where **n** is the total number of elements in the array.
Space-complexity: **O(1).**