

Searching 1

Rajat Garg

17th February, 2023



Searching 1

Agenda

1. Concept of Binary Search
2. Problems based on Binary Search

Binary Search

Binary search is an efficient algorithm for finding an item from an ordered list of items. It works by repeatedly dividing in half the portion of the list that could contain the item until you've narrowed down the possible locations to just one. Binary Search works mostly in a sorted array.

Searching for the middle element takes constant time and in every recursive call problem size is reduced to half.

Hence, $T(n)=T(n/2)+k$ is the recursive relation for the time complexity of the binary search.

About Bosscoder !

BossCoder Academy is a platform for ambitious engineers who want to upskill and achieve great heights in their careers.

Our world-class program provides the learners with:

- ✓ Structured curriculum designed by experts.
- ✓ Covers Data Structures & Algorithms, System Design & Project Development.
- ✓ Live Interactive Classes from Top PBC engineers.
- ✓ 1:1 Mentorship
- ✓ Quick Doubt Support
- ✓ Advanced Placement Support

Want to upskill to achieve your dream of working at

[Learn now we can help.](#)

searchn

In the worst case, Binary search does $2\log_2(n) + 1$ comparisons where Ternary search does $4\log_3(n) + 1$ comparisons which comes down to $\log_2(n)$ and $2\log_3(n)$. Hence Binary Search is better than ternary search.

Implementation of Binary Search

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

int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r)
    {
        int m = l + (r - l) / 2;

        if (arr[m] == x)
            return m;

        if (arr[m] < x)
            l = m + 1;

        else
            r = m - 1;
    }

    return -1;
}

int main()
{
    int n;
    cin >> n;
```

Complete Guide

to **Upskill yourself** for**Download PDF**

1543+ downloads.

```

        cin >> a[i];
    int x;
    cin >> x;
    int result = binarySearch(a, 0, n - 1,
    (result == -1)
        ? cout << "Element is not present i
        : cout << "Element is present at ir
    return 0;
}

```

If the key is not present in the array then after the $low \leq high$ is violated the low point to the **ceil of the key** in the array and **high points to the floor of the key**.

Problems

1. Find the First Position of an element in an Array

Given an array of n integers find the first occurrence of a number and its index. If it's not present return -1.

Example

Input: n=7, arr[]={1,2,3,3,4,5,6}, key=3

Output:2

Brute Force Method

Iterate over the array elements. If the element is found return the index and come out of the loop else return -1.

Implementation

```

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

```

```
    for (int i = 0; i <= r; i++)
    {
        if (arr[i] == x)
        {
            cout << i;
            return 0;
        }
    }
    cout << -1;
}

int main()
{
    int n;
    cin >> n;
    int a[n];
    for (int i = 0; i < n; i++)
        cin >> a[i];
    int x;
    cin >> x;
    first(a, 0, n - 1, x);
    return 0;
}
```

Optimised Approach

We can use Binary Search for finding the first occurrence. We can do a normal binary search just when the key is found we initialise the end element to mid-1 so that we can find the leftmost occurrence of the key number.

Implementation

```
int first(int arr[], int l, int r, int x,int n)
{
    int low = 0, high = n - 1, res = -1;
    while (low <= high) {

        int mid = (low + high) / 2;

        if (arr[mid] > x)
            high = mid - 1;
        else if (arr[mid] < x)
            low = mid + 1;

        else {
            res = mid;
            low = mid + 1;
        }
    }
    return res;
}
```

```
int main()
{
#ifdef ONLIINE_JUDGE
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
#endif
    int n;
    cin >> n;
    int a[n];
    for (int i = 0; i < n; i++)
        cin >> a[i];
    int x;
    cin >> x;
    int ans= first(a, 0, n - 1, x,n);
}
```

}

Time Complexity: $O(N)$

Space Complexity: $O(N)$

2. Find pivot element in a rotated sorted array.

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

Example

Input: $n=6$, $arr[]=\{4,5,6,1,2,3\}$

Output: 1

Brute Force Method

We can just sort the array and print the first element.

Implementation

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

int pivot(int arr[],int n)
{
    sort(arr,arr+n);
    return arr[0];
}

int main()
{
    #ifndef ONLINE_JUDGE
        freopen("input.txt", "r", stdin);
        freopen("output.txt", "w", stdout);
    #endif
```

```
int a[n];  
for (int i = 0; i < n; i++)  
    cin >> a[i];  
int ans=pivot(a,n);  
cout<<ans;  
return 0;  
}
```

Optimised Approach

We can use binary search. Keep two pointers low and high. Whenever the mid element is higher than the low element it means that we are still in the rotated part of the array. So we shift your low 1 place ahead of the mid-element. Else if that is not the case, it means we are already in the non-rotated part of the array but in the farther right side, so we shift the high 1 place behind the mid. We do this to reach that part of the non-rotated array which has lesser value elements. Each time during the search we take a mini variable which stores the minimum between mid element, low element and mini in mini.

Implementation

```
#include <bits/stdc++.h>  
using namespace std;  
  
int pivot(int arr[], int n)  
{  
    int l = 0;  
    int h = n - 1;  
    int mini = INT_MAX;  
    while (l <= h)
```

```
        mini = min(mini, arr[mid]);
        mini = min(mini, arr[l]);
        if (arr[mid] >= arr[l])
        {
            l = mid + 1;
        }
        else
        {
            h = mid - 1;
        }
    }
    return mini;
}
```

```
int main()
{
    int n;
    cin >> n;
    int a[n];
    for (int i = 0; i < n; i++)
        cin >> a[i];
    int ans = pivot(a, n);
    cout << ans;
    return 0;
}
```

Time Complexity: $O(\log n)$

Space Complexity: $O(1)$

3. [Find Peak Element](#)

A peak element is an element that is strictly greater than its neighbours. Given a 0-indexed integer array `nums[]`, find a peak element, and return its index. If the array contains multiple

Input: nums = [1,2,5,1]

Output: 2

Brute Force Method

Return the greatest element of the array, it will always be the peak element as it is the largest.

Implementation

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

int findPeakElement(int nums[],int n) {
    int mx=max_element(nums,nums+n)-nums[0];
    return mx;
}

int main()
{
    int n;
    cin >> n;
    int a[n];
    for (int i = 0; i < n; i++)
        cin >> a[i];
    int ans = findPeakElement(a, n);
    cout << ans;
    return 0;
}
```

Optimised Approach

Use Binary Search on mid and mid+1 if
mid>mid+1 then mid can be a peak else mid+1
can be a peak

Implementation

```
int findPeakElement(int nums[], int s, int
{
    if (s == end)
        return s;
    else
    {
        int mid1 = (s + end) / 2;
        int mid2 = mid1 + 1;
        if (nums[mid1] > nums[mid2])
            return findPeakElement(nums, s,
        else
            return findPeakElement(nums, mi
    }
}

int main()
{
    int n;
    cin >> n;
    int a[n];
    for (int i = 0; i < n; i++)
        cin >> a[i];
    int ans = findPeakElement(a, 0, n - 1);
    cout << ans;
    return 0;
}
```

Time Complexity: $O(\log N)$

Space Complexity: $O(1)$

4. [Find the element that appears once the rest of the element appears twice](#)

Given an array of integers. All numbers occur

Example :

Input: n=7, arr[] = {2, 3, 5, 4, 5, 3, 4}

Output: 2

Brute Force Method

Check every element if it appears once or not.
Once an element with a single occurrence is found, return it.

Implementation

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

int SingleOccuringElement(int a[], int n)
{
    for (int i = 0; i < n; i++)
    {
        int count = 0;
        for (int j = 0; j < n; j++)
        {
            if (a[i] == a[j])
            {
                count++;
            }
        }
        if (count == 1)
        {
            return a[i];
        }
    }

    return -1;
}
```

```
int n;  
cin >> n;  
int a[n];  
for (int i = 0; i < n; i++)  
    cin >> a[i];  
cout << SingleOccuringElement(a, n);  
return 0;  
}
```

Another Approach

We can simply use a hashmap to store the frequency of the elements and after that, we can iterate the hashmap to find the element with frequency 1.

Implementation

```
#include <bits/stdc++.h>  
using namespace std;  
  
int SingleOccuringElement(int a[],int n)  
{  
    unordered_map<int,int> mm;  
    for(int i=0;i<n;i++)  
    {  
        mm[a[i]]++;  
    }  
    for(auto x:mm)  
    {  
        if(x.second==1) return x.first;  
    }  
}  
  
int main()  
{
```

```
int a[n];  
for(int i=0;i<n;i++) cin>>a[i];  
cout << SingleOccuringElement(a,n);  
return 0;  
}
```

Bitwise Based Approach

The best solution is to use XOR. XOR of all array elements gives us the number with a single occurrence. The idea is based on the following two facts.

1. XOR of a number with itself is 0.
2. XOR of a number with 0 is the number itself.

Implementation

```
#include <bits/stdc++.h>  
using namespace std;  
  
int SingleOccuringElement(int a[],int n)  
{  
    int res = a[0];  
    for (int i = 1; i < n; i++)  
        res = res ^ a[i];  
  
    return res;  
}  
  
int main()  
{  
    int n;  
    cin>>n;  
    int a[n];  
    for(int i=0;i<n;i++) cin>>a[i];  
    cout << SingleOccuringElement(a,n);  
}
```

Binary Search-Based Approach

This is an efficient approach for finding a single element in a list of duplicate elements. In this approach, we are using a binary search algorithm to find the single element in the list of duplicate elements. Before that, we need to make sure the array is sorted. The first step is to sort the array because the binary search algorithm won't work if the array is not sorted. Now check if the mid-index value falls in the left half or the right half. If it falls in the left half then we change the low value to mid+1 and if it falls in the right half, then we change the high index to mid-1. To check it, we used a logic (**if(arr[mid]==arr[mid^1])**).

Implementation

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

int SingleOccuringElement(int a[],int n)
{
    int low = 0, high = n - 2;
    int mid;
    while (low <= high) {
        mid = (low + high) / 2;
        if (a[mid] == a[mid ^ 1])
            low = mid + 1;
        else
            high = mid - 1;
    }
    return a[low];
}
```

```
{  
    int n;  
    cin>>n;  
    int a[n];  
    for(int i=0;i<n;i++) cin>>a[i];  
    cout << SingleOccuringElement(a,n);  
    return 0;  
}
```

Time Complexity: $O(\log n)$ [If the array is sorted]

Auxiliary Space: $O(1)$



"BossCoder had everything I was looking for"

Udit Sharma
Software Developer

BossCoder had everything I was looking for



"BossCoder helped me clear my basics of DSA and System Design"

Rachit Arora
Head of Engineering


BossCoder helped me clear my basics of DSA and System Design

[View All blogs](#)

Helping
ambitious
learners
upskill
themselves
& shift to
top
product
based
companies.

Lets hear
all about
it.






Who are we

About us
Blog
Attend a FREE Event
Privacy Policy
Terms & Condition
Pricing and Refund Policy

Contact Us

Email: ask@bosscoderacademy.com

Follow us on

 LinkedIn
 Youtube
 Instagram
 Telegram
 Reviews on Quora