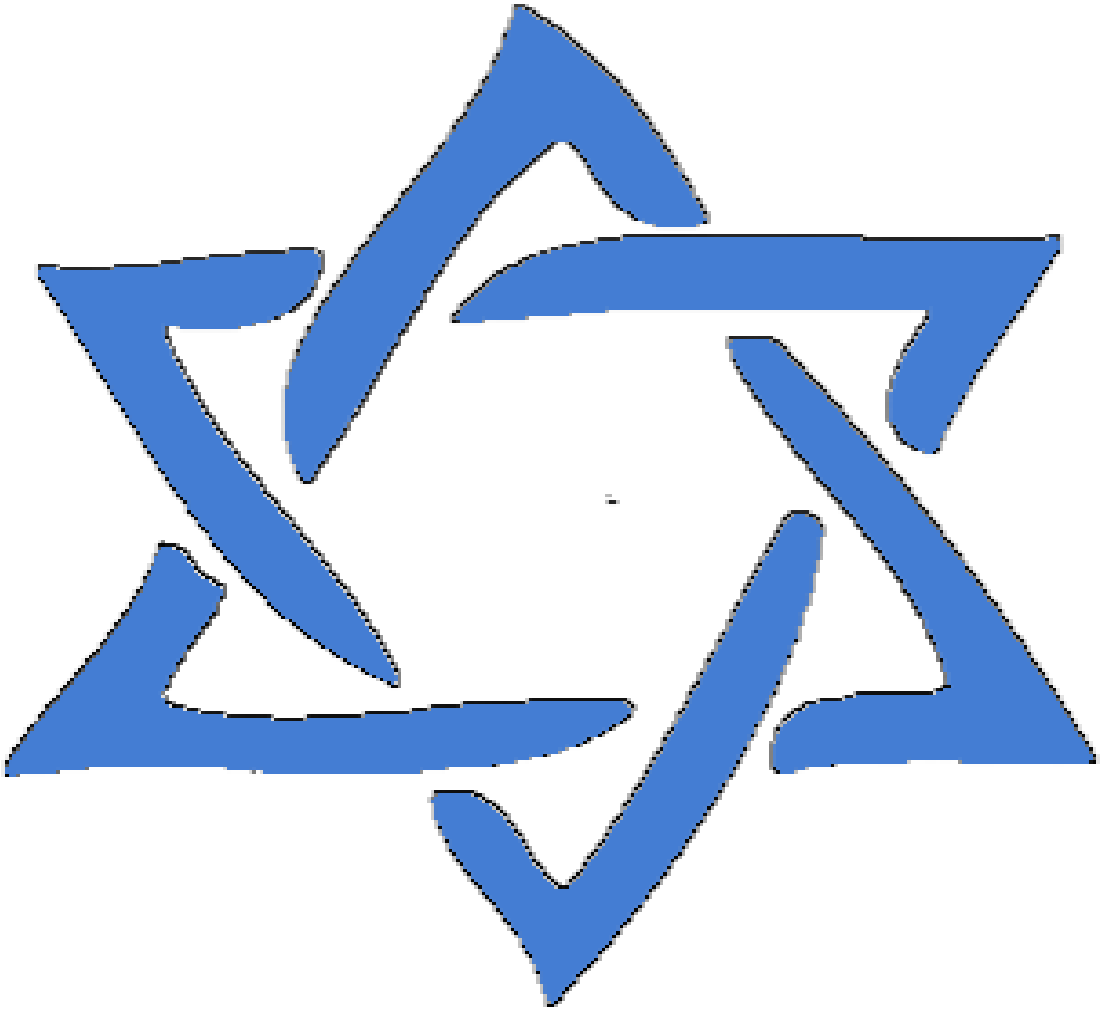
Binary search
Lesson 1



Linear Search

Linear search means searching an element linearly.
E.g., searching an element in an array by simply traversing the array.

Problem. Given an array arr[] of n elements, write a function to search a given element x in arr[].

Examples :

Input : arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}
x = 110;

Output : 6
Element x is present at index 6

Algorithm:

A simple approach is to do a linear search, i.e

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.

CODE:

```
1. #include <iostream>
2. using namespace std;
3. int search(int arr[], int n, int x)
```

```
4. {
5.     int i;
6.     for (i = 0; i < n; i++)
7.         if (arr[i] == x)
8.             return i;
9.     return -1;
10.    }
11.
12.    // Driver code
13.    int main()
14.    {
15.        int arr[] = { 2, 3, 4, 10, 40 };
16.        int x = 10;
17.        int n = sizeof(arr) / sizeof(arr[0]);
18.
19.        // Function call
20.        int result = search(arr, n, x);
21.          if(result == -1)
22.            cout << "Element is not present in array";
23.          else
24.            cout << "Element is present at index " << result;
25.        return 0;
26.    }
```

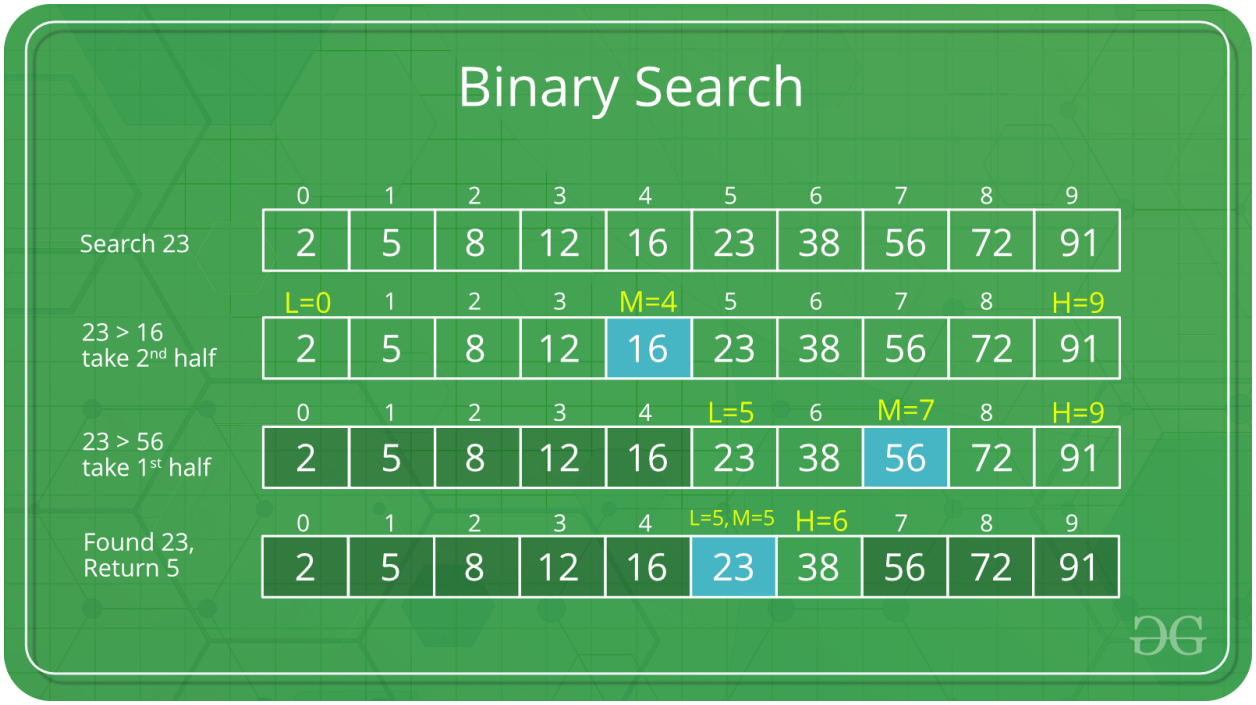
Time complexity: O(N),where N is the number of elements in the array.

But if we further want to optimize this ,we can use binary search.

But binary search is applicable only if the given array is sorted.

Binary search:

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.



The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

ALGORITHM:

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recursive call for right half.
4. Else (x is smaller) recursive call for the left half.

RECURSIVE IMPLEMENTATION OF BINARY SEARCH:

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. // A recursive binary search function. It returns
5. // location of x in given array arr[low....high] is present,
6. // otherwise -1
7. int binarySearch(int arr[], int low, int high, int x)
8. {
9.    if (low<=high) {
10.          int mid = (low + high) / 2;
11.
```

```
12.         // If the element is present at the middle
13.         // itself
14.         if (arr[mid] == x)
15.             return mid;
16.
17.         // If element is smaller than mid, then
18.         // it can only be present in left subarray
19.         if (arr[mid] > x)
20.             return binarySearch(arr, low, mid - 1, x);
21.
22.         // Else the element can only be present
23.         // in right subarray
24.         return binarySearch(arr, mid + 1, high, x);
25.     }
26.
27.     // We reach here when element is not
28.     // present in array
29.     return -1;
30. }
31. int main()
32. {
33.     int arr[] = { 2, 3, 4, 10, 40 };
34.     int x = 10;
35.     int n = sizeof(arr) / sizeof(arr[0]);
```

```
36.     int result = binarySearch(arr, 0, n - 1, x);
37.     if(result == -1)
38.         cout << "Element is not present in array";
39.     else
40.         cout << "Element is present at index " << result;
41.     return 0;
42.  }
```

ITERATIVE APPROACH FOR BINARY SEARCH:

```
1.  #include <bits/stdc++.h>
2.  using namespace std;
3.
4.  // A iterative binary search function. It returns
5.  // location of x in given array arr[low...high] if present,
6.  // otherwise -1
7.  int binarySearch(int arr[], int low, int high, int x)
8.  {
9.     while (low<=high) {
10.         int mid=(low+high)/2;
11.
12.         // Check if x is present at mid
```

```
13.         if (arr[mid] == x)
14.             return mid;
15.
16.         // If x greater, ignore left half
17.         if (arr[mid] < x)
18.             low = mid + 1;
19.
20.         // If x is smaller, ignore right half
21.         else
22.             high = mid - 1;
23.     }
24.
25.     // if we reach here, then element was
26.     // not present
27.     return -1;
28. }
29.
30. int main()
31. {
32.     int arr[] = { 2, 3, 4, 10, 40 };
33.     int x = 10;
34.     int n = sizeof(arr) / sizeof(arr[0]);
35.     int result = binarySearch(arr, 0, n - 1, x);
36.     if(result == -1)
```

```
37.         cout << "Element is not present in array";
38.       else
39.         cout << "Element is present at index " << result;
40.       return 0;
41.    }
```

Auxiliary Space: O(1) in case of iterative implementation. In case of recursive implementation, O(Logn) recursion call stack space.

Que link:[searching an element in sorted array]

MAGICAL NUMBER

Your friend loves magic and he has coined a new term - "Magical number". To perform his magic, he needs that Magic number. There are N number of people in the magic show, seated according to their ages in an ascending order. Magical number is that seat no. where the person has the same age as that of the given seat number.

Help your friend in finding out that "Magical number"

Input:

The first line of input contains an integer T denoting the number of test cases.

The first line of each test case is N, size of an array.

The second line of each test case contains N input A[].

Output:

Print "Magical Number"

Print "-1" when index value does not match with value.

Example:

Input:

1

10

-10 -1 0 3 10 11 30 50 100 150

Output:

3

ALGORITHM:

NAIVE APPROACH:

O(N), if we simply traverse and if there is number whose index number is equal to the value at that particular index in the array,return the index . Else return -1.

USING BINARY SEARCH:

If the value at mid is greater than index value,we check in left side,else in right side of the mid. Time complexity:O(logN)

CODE:

```
1.  int binarySearch(int arr[], int low, int high)
2.  {
3.      while(low<=high)
4.      {
5.          int mid=(low+high)/2;
6.          if(arr[mid]==mid)
7.            return arr[mid];
8.          else if(arr[mid]>mid)
9.            high=mid-1;
10.         else
11.           low=mid+1;
12.         }
13.         return -1;
14.   }
```

Que link:[magical number]

FIND FIRST AND LAST POSITION OF AN ELEMENT IN A SORTED ARRAY

Given a sorted array arr containing n elements with possibly duplicate elements, the task is to find indexes of first and last occurrences of an element x in the given array.

Input:

n=9, x=5

arr[] = { 1, 3, 5, 5, 5, 5, 67, 123, 125 }

Output: 2 5

Explanation: First occurrence of 5 is at index 2 and last occurrence of 5 is at index 5.

ALGORITHM :

NAIVE APPROACH

1. Run a for loop and for i = 0 to n-1
2. When we find element first time then we update first = i
3. We always update last=i whenever we find the element.

Time complexity : O(n)

CODE:

```
1. void findFirstAndLast(int arr[], int n, int x)
```

```
2. {
3.     int first = -1, last = -1;
4.     for (int i = 0; i < n; i++) {
5.         if (x != arr[i])
6.             continue;
7.         if (first == -1)
8.             first = i;
9.         last = i;
10.        }
11.    if (first != -1)
12.            cout << "First Occurrence = " << first
13.                << "\nLast Occurrence = " << last;
14.        else
15.            cout << "Not Found";
16.    }
```

USING BINARY SEARCH (iterative)

1. Finding first occurrence:

- Find the mid element :
- If it is greater than x search in left half
- If it is lesser than x, search in right half
- If it is equal to x, assign value of index to first variable and search in left half

2. Finding last occurrence:

- Find the mid element :
- If it is greater than x search in left half
- If it is lesser than x, search in right half
- If it is equal to x, assign value of index to last variable and search in right half

CODE:

```
int first(int arr[], 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;

            //update res and search in left half
            else {
                res = mid;
                high = mid - 1;
            }
```

```
17.     }
18.     return res;
19.   }
20.
21.   int last(int arr[], int x, int n)
22.   {
23.     int low = 0, high = n - 1, res = -1;
24.     while (low <= high)
25.     {
26.         int mid = (low + high) / 2;
27.         if (arr[mid] > x)
28.             high = mid - 1;
29.         else if (arr[mid] < x)
30.             low = mid + 1;
31.
32.         //update res and search in right half
33.         else {
34.             res = mid;
35.             low = mid + 1;
36.         }
37.     }
38.     return res;
39.   }
```

USING BINARY SEARCH (recursive)

1. Finding first occurrence:

- If low <= high
- Find the mid element :
- If (arr[mid] == x && (mid == 0 || arr[mid-1] < x)) , return mid
- If it is greater than x, call function for left half i.e. high = mid -1
- If it is lesser than x, call function for right half i.e. low = mid + 1

2. Finding last occurrence:

- If low <= high
- Find the mid element :
- If (arr[mid] == x && (mid == n - 1 || arr[mid+1] > x)) , return mid
- If it is greater than x, call function for left half i.e. high = mid -1
- If it is lesser than x, call function for right half i.e. low = mid + 1

Ques link: **First and last occurrences of x**

leftmost and rightmost index

Question for practice:
the problem of identical arrays