17/02/2023

Searching

We have discussed an algorithm named linear search. Search means to find whether key is present or not. In linear search, in the worst case we will be comparing all the elements. Say the size of array is n, then in worst case n comparasion will happen & the time complexity = O(n)

```
for(int i=0 ; i<n ; i++) {
        if(arr[i] == key) {
            cout << "Found";
            break;
        }
}
```

The above is the code of linear search & has time complexity = O(n)

Let's say in our array we have 1000 elements then in worst case 1000 comparasions is done which is a lot.

Binary Search

If we 1000 elements, then binary search just takes 10 comparasions which is far less than 1000 comparasions.

Note→ To apply binary search, there is one condition which is the elements should be in monotonic order i.e elements should be sorted either

in increasing or decreasing order.

1 2 4 6 9 11 15

The above elements in sorted fashion & currently we are at 9 value & target = 15 which is greater than 9, hence we need to search in the right.

Working of binary search

index →	0	1	2	3	4	5	6	7
	1	3	7	9	11	13	15	19

target = 15

1) Initialize start = 0 and end = n-1 where n is the size of array.

2) Find mid index, $mid = \frac{start + end}{2}$

$$mid = \frac{0+7}{2} = 3$$

arr[3] = 9

3) Compare target value with value at mid index.

15 > 9 } This means search in right array

4) 11 13 15 19

start = 4, end = 7

$mid = \frac{4+7}{2} = 5$

Now compare target with value at mid index.

15 > 13 } Search in right array.

5)

15 19

start = 6, end = 7

mid = $\frac{6+7}{2}$ = 6

Compare target with value present at the mid index.

15 == 15 } return index

If in case, the element is not present, then return - 1 . This is basically achieved after start > end i.e. start goes ahead of end.

Code:

```
int binarySearch (int arr[], int size, int target)
{
    int start = 0;
    int end = size - 1;
    int mid = (start + end)/2;

    while (start <= end) {
        // Return mid index if found
        if (arr[mid] == target) {
            return mid;
        }
        // Search in right part
        else if (target > element arr[mid])
        {
            start = mid + 1;
        }
```

```
//Search in left part
else {
        end = mid-1;
}   mid = (start + end)/2; → Update mid with
}                              updated start & end values.
return -1; // If element not found
}
```

↑ mid is an index.

Issue in `mid = (start + end) /2;`

There is an issue in this statement & the issue is as follows :

There can be integer overflow here. Let's say

$start = 2^{31} - 1$

$end = 2^{31} - 1$

$mid = \frac{(2^{31} - 1 + 2^{31} - 1)}{2}$ → Numerator will go out of range

Solution

$mid = s + \frac{(e-s)}{2}$;

s → start

e → end

$start = 2^{31} - 1$

$end = 2^{31} - 1$

$2^{31} - 1 + \frac{(2^{31} - 1 - 2^{31} + 1)}{2}$

$2^{31} - 1 + 0 \Rightarrow 2^{31} - 1$

Hence there is no integer overflow.

Note → Other solution is $mid = \frac{start}{2} + \frac{end}{2}$;

This will also won't create any integer overflow but we will follow $s + \frac{(e-s)}{2}$ only.

Time complexity of binary search

n size array

1st iteration ⇒ $\frac{n}{2^1}$

2nd iteration ⇒ $\frac{n}{2^2}$

:

kth iteration ⇒ $\frac{n}{2^k}$

$\frac{n}{2^k} = 1$ as if only one element is present, then we just have compare it & if found equal, then return index, else return -1.

$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

$$k = \log_2 n$$

Time complexity = $O(k) = O(\log_2 n)$

Note → Binary search stops after we get only 1 element. Everytime the size of array reduces by half.

STL function for binary Search
STL stands for Standard Template Library and this have already predefined function of binary search.

```
vector <int> v {1, 2, 3, 4, 5, 6};
                    start    end
if (binary_search (v.begin(), v.end(), 3)) {
        cout << "Found";
}
else {
        cout << "Not found";
}
```

To use binary_search, we need to include <algorithm> in our program.

Note →

```
int arr[] = {1, 2, 3, 4};
int size = 4;
if (binary_search (arr, arr+size, 3)) {
        cout << "Found" << endl;
}
else {
        cout << "Not found" << endl;
}
```

Also note that arr + size will be the last index.

Problem Solving

Q1 Find the first occurence of an element.

i/p →

1	3	4	4	4	4	4	6	7	9

o/p → 2

Here we can use binary search as elements

are in monotonic fashion.

Algorithm

1) start = 0

end = size - 1 = 9

mid = $\frac{start + end}{2} = \frac{0+9}{2} = 4$

arr[mid] = 4

First of all store the answer as index = 4 and search in the left part.

2)

1	3	4	4

start = 0

end = 3

mid = $\frac{0+3}{2} = 1$

arr[mid] = 3 < 4 . Simply search in right part i.e start = mid + 1.

3)

4	4

start = 2

end = 3

mid = $\frac{2+3}{2} = 2$

arr[mid] = 4, answer store as index = 2, & search in left part i.e end = mid - 1.

Now start > end, end loop & answer = 2.

Also if target > arr[mid], then search in right part by

start = mid + 1

Code

```
int firstOccurence(vector<int> v, int t){

    int s=0;
    int e = v.size()-1;
    int mid = s + (e-s)/2;
    int ans = -1;

    while(s<=e){

        if(v[mid]==t){
            ans = mid; //Store answer
            e = mid-1; //Search in left
        }
        else if(target < v[mid]){
            e = mid-1; //Search in left
        }
        else{
            s = mid+1; //Search in right
        }
        mid = s + (e-s)/2; //Update mid again.
    }
    return ans;
}
```

Time complexity same as binary search i.e O(logn)

Note → We can code the question of last occurence of an element by changing e=mid-1 inside the 1st if condition of the above code to s=mid+1 i.e we

need to search in the right part to find the last occurence of the element.

Code of Last Occurence

```
int lastOccurence (vector<int> v, int t){
                                    t = target
                                    in code
    int s = 0;
    int e = v.size() - 1;
    int mid = s + (e-s)/2;
    int ans = -1;

    while (s <= e){

        if (v[mid] == target){
                ans = mid; //Store answer
                s = mid+1; // Search in right
        }
        else if (target < v[mid]){
                e = mid-1; // Search in left
        }
        else{
                s = mid + 1; // Search in right
        }
        mid = s+(e-s)/2; //Again update mid
    }
    return ans;
}
```

Time complexity O(logn)

STL functions for first & last occurence

```
lower_bound (v.begin(), v.end(), 20);
                                 ↳ target element
```

Lower bound is used to find the first occurence & it returns an iterator. Similarly upper-bound is used to find the last occurence.

```
upper_bound (v.begin(), v.end(), 20);
                                 ↳ Target element
```

Q2 Total number of occurences of the element.

```
i/p → 2 4 4 4 4 5 6
o/p → 4
```

Total occurence = lastOccurence - firstOccurence + 1.

Code

```
int first = firstOccurence (V, 4);
int last = lastOccurence (V, 4);
cout << (last - first + 1) << endl;
```

Q3 Find the missing element.

```
i/p → 1 2 3 4 6 7 8
o/p → 5 (missing here)
```

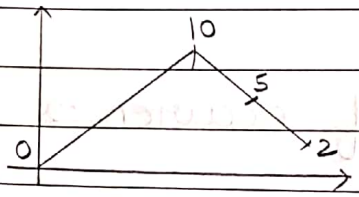
1	2	3	4	6	7	8
0	1	2	3	4	5	6

On left part, there is a pattern which i°+1 == arr[i] but on right side this pattern breaks.

Explore this question & we have to solve it with the help of binary search.

Q4 Peak element in mountain array.

i/p → 0 10 5 2



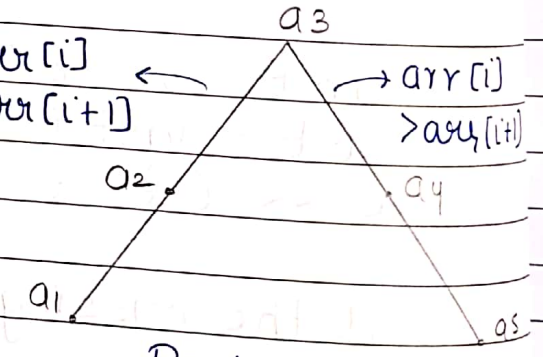
o/p → 10

A brute force solution can be of linear search & find the maximum array value. This has time complexity = O(n). We can do it with the help of binary search. This approach will have time complexity = O(logn)

Algorithm

1) start = 0
end = 2
mid = $\frac{0+2}{2} = 1$

arr[mid] = 10



Peak element
arr[i] > arr[i+1] &&
arr[i] > arr[i-1]

2) if (arr[mid] > arr[mid+1]) { }

✓ mid element may be a peak element or in the descending line

✓ mid in line a3, a4, a5.

3) if (arr[mid] < arr[mid+1]) { }

✓ mid element can not be peak element & check in the right part i.e s = mid+1;

```
else {
    e = mid;
}
```

Code

```
int peakElement (int arr[], int size){

        int s = 0;
        int e = size - 1;
        int mid = s + (e-s)/2;
        while(s<e){
            if(arr[mid] < arr[mid+1]){
                s = mid+1;
            }
            else{
                e = mid;
            }
            mid = s + (e-s)/2;
        }
        return s; // We can return e; also
}
```

Note → if (arr[mid] < arr[mid+1]) { __ }
else { __ }

↳ This means that we are on either peak element or on the descending line.

If we write e = mid - 1, then if we are on peak element, then we might loose it & hence e = mid is written.

s = mid; } while (s <= e) { __ }
e = mid; } Infinite Loop