Binary Search

22 February 2022 19:35

Binary Search is a searching algorithm that works in $O(\log n)$ time complexity.

IMPORTANT :

Binary Seach is only applicable for 'monotonic search spaces'. For example, B.S. can be applied for searching an element in a SORTED ARRAY only.

A sorted array where elements are arranged in a non-decreasing order is a monotonically increasing search space. Similarly, if elements are arranged in a non-increasing order, then it's a monotonically decreasing search space.

Eg : arr[] : {1, 3, 3, 6, 7, 10} ⟶ monotonically increasing search space

APPLYING BINARY SEARCH :

arr[] : {1, 3, 3, 6, 7, 10} search = 7

① Fix a low and high index for the array where you want to search.

arr[] : {1, 3, 3, 6, 7, 10}

low high

② Find the mid index for the given low & high index.

mid = (low + high)/2; ⟶ (0+5)/2 = 2

0 1 2 3 4 5

arr[] : {1, 3, 3, 6, 7, 10}

low mid high

③ Check the following :

(i) If arr[mid] == search, return mid, you found it!

(ii) If arr[mid] < search, then all elements arr[0....mid] will definitely be smaller than search. Thus, update your search space : low = mid + 1.

(You have discarded 0.... mid indices)

(iii) If arr[mid] > search, then all elements arr[mid..... n-1] will definitely be greater than search. Thus, update your search space : high = mid - 1.

(You have discarded mid.... n-1 indices)

	0	1	2	3	4	5
arr[ ] :	{1,	3,	3,	6,	7,	10}
	low		mid			high

arr[mid] = 3 and 3 < 7

low = mid+1 = 3.

⇓

	0	1	2	3	4	5
arr[ ] :	{1,	3,	3,	6,	7,	10}
			mid	low		high

4 Repeat steps 2 and 3 untill low <= high.

	0	1	2	3	4	5
arr[ ] :	{1,	3,	3,	6,	7,	10}
				low	mid	high

arr[mid] == 7, return mid;

$\boxed{\text{return 4;}}$

EXAMPLE : arr[ ] : {4, 8, 16, 32, 64} K = 4

1 {4, 8, 16, 32, 64}

4,	8,	16,	32,	64
low		mid		high

② $(arr[mid] = 16) > 4 \Rightarrow high = mid - 1.$

$\{4, 8, 16, 32, 64\}$

low high mid

③ $(arr[mid] = 8) > 4 \Rightarrow high = mid - 1$

mid

$\{4, 8, 16, 32, 64\}$

low high

④ $(arr[mid] = 4)$ We found it !

Note : If low > high and we haven't found the element, then the element doesn't exist at all.

Code :

```
int binarySearch(int arr[], int size, int key) {

    int start = 0;
    int end = size-1;

    int mid = (start + end)/2;

    while(start <= end) {

        if(arr[mid] == key) {
            return mid;
        }

        //go to right wala part
        if(key > arr[mid]) {
            start = mid + 1;
        }
        else{
            end = mid - 1;
        }

        mid = (start+end)/2;
    }
    return -1;
}
```

```cpp
int even[6] = {2,4,6,8,12,18};
int odd[5] = {3, 8, 11, 14, 16};

int evenIndex = binarySearch(even, 6, 20);

cout << " Index of 18 is " << evenIndex << endl;

int oddIndex = binarySearch(odd, 5, 8);

cout << " Index of 8 is " << oddIndex << endl;
```

```
lovebabbar@192 ~ % cd "/Users/lovebabbar/" && g++ binarySearch.cpp -
o binarySearch && "/Users/lovebabbar/"binarySearch
 Index of 18 is -1
 Index of 8 is 1
```

CAUTION :

What if you had an array of size $2^{31} - 1$ and during some iteration :

$low = 2^{31} - 2$ and $high = 2^{31} - 1$,

$$mid = \frac{(low + high)}{2}$$

will throw integer overflow error because $2^{31} + 2^{31} - 1 = 2^{32} - 1$

which is greater than INT_MAX. Thus, we can either use long long or even better,

$$mid = low + \frac{(high - low)}{2}$$

New Code :

```cpp
int binarySearch(int arr[], int size, int key) {

    int start = 0;
    int end = size-1;

    int mid = start + (end-start)/2;

    while(start <= end) {

        if(arr[mid] == key) {
            return mid;
        }

        //go to right wala part
        if(key > arr[mid]) {
            start = mid + 1;
        }
        else{ //key < arr[mid]
            end = mid - 1;
        }

        mid = start + (end-start)/2;
    }
    return -1;
}
```

Linear Search v/s Binary Search :

Linear Search's Time Complexity = $O(n)$.

In Binary Search, we are essentially halving our search space in every iteration.

$$\Rightarrow \quad n \rightarrow \frac{n}{2} \rightarrow \frac{n}{4} \rightarrow \frac{n}{8} \rightarrow \ldots\ldots \rightarrow 1.$$

Applying Geometric Progression :

$a = n$

$r = 1/2$

we want to know the no. of elements passed by the time we reach 1.

Say 1 is the z^{th} element.

$$ar^{z-1} = 1$$

$$\rightarrow n\left(\frac{1}{2}\right)^{z-1} = 1$$

$$\rightarrow 2^{z-1} = n$$

$$\rightarrow 2^{z} = 2n$$

$\rightarrow$ Taking $\log_2$ on both sides :

$$\log 2^{z} = \log 2n$$

$$\rightarrow z = \log(2n) \approx \boxed{O(\log n)}$$