Binary Search Algorithm

Binary Search is applied on the sorted array or list of large size. It's time complexity of **O(log n)** makes it very fast as compared to other sorting algorithms. The only limitation is that the array or list of elements must be sorted for the binary search algorithm to work on it.

Implementing Binary Search Algorithm

Following are the steps of implementation that we will be following:

1. Start with the middle element:
    - If the **target** value is equal to the middle element of the array, then return the index of the middle element.
    - If not, then compare the middle element with the target value,
        - If the target value is greater than the number in the middle index, then pick the elements to the right of the middle index, and start with Step 1.
        - If the target value is less than the number in the middle index, then pick the elements to the left of the middle index, and start with Step 1.
2. When a match is found, return the index of the element matched.
3. If no match is found, then return `-1`

```
/*
    function for carrying out binary search on given array
    - values[] => given sorted array
    - len => length of the array
    - target => value to be searched
*/
int binarySearch(int values[], int len, int target)
{
    int max = (len - 1);
    int min = 0;

    int guess;  // this will hold the index of middle elements
    int step = 0;  // to find out in how many steps we completed the search

    while(max >= min)
```

```
    {
        guess = (max + min) / 2;
        // we made the first guess, incrementing step by 1
        step++;

        if(values[guess] ==  target)
        {
            printf("Number of steps required for search: %d \n", step);
            return guess;
        }
        else if(values[guess] >  target)
        {
            // target would be in the left half
            max = (guess - 1);
        }
        else
        {
            // target would be in the right half
            min = (guess + 1);
        }
    }

    // We reach here when element is not
    // present in array
    return -1;
}

int main(void)
{
    int values[] = {13, 21, 54, 81, 90};
```

```
    int n = sizeof(values) / sizeof(values[0]);

    int target = 81;

    int result = binarySearch(values, n, target);

    if(result == -1)

    {

        printf("Element is not present in the given array.");

    }

    else

    {

        printf("Element is present at index: %d", result);

    }

    return 0;

}
```

Copy

We hope the above code is clear, if you have any confusion, post your question in our Q & A Forum.

Now let's try to understand, why is the time complexity of binary search **O(log n)** and how can we calculate the number of steps required to search an element from a given array using binary search without doing any calculations. It's super easy! Are you ready?

Time Complexity of Binary Search $O(\log n)$

When we say the time complexity is $\log n$, we actually mean $\log_2 n$, although the **base** of the log doesn't matter in asymptotic notations, but still to understand this better, we generally consider a base of 2.

Let's first understand what $\log_2(n)$ means.

```
Expression: log₂(n)

- - - - - - - - - - - - - - -

For n = 2:

log₂(2¹) = 1

Output = 1

- - - - - - - - - - - - - - -

For n = 4

log₂(2²) = 2

Output = 2

- - - - - - - - - - - - - - -

For n = 8
```

```
log₂(2³) = 3

Output = 3

- - - - - - - - - - - - - - -

For n = 256

log₂(2⁸) = 8

Output = 8

- - - - - - - - - - - - - - -

For n = 2048

log₂(2¹¹) = 11

Output = 11
```

Now that we know how $\log_2(n)$ works with different values of n, it will be easier for us to relate it with the time complexity of the binary search algorithm and also to understand how we can find out the number of steps required to search any number using binary search for any value of n.

Counting the Number of Steps

As we have already seen, that with every incorrect guess, binary search cuts down the list of elements into half. So if we start with 32 elements, after first unsuccessful guess, we will be left with 16 elements.

So consider an array with 8 elements, after the first unsuccessful, binary search will cut down the list to half, leaving behind 4 elements, then 2 elements after the second unsuccessful guess, and finally only 1 element will be left, which will either be the target or not, checking that will involve one more step. So all in all binary search needed at most 4 guesses to search the target in an array with 8 elements.

If the size of the list would have been 16, then after the first unsuccessful guess, we would have been left with 8 elements. And after that, as we know, we need atmost 4 guesses, add 1 guess to cut down the list from 16 to 8, that brings us to 5 guesses.

So we can say, as the number of elements are getting doubled, the number of guesses required to find the target increments by 1.

Seeing the pattern, right?

Generalizing this, we can say, for an array with n elements,

the number of times we can repeatedly halve, starting at n, until we get the value 1, plus one.

And guess what, in mathematics, the function $\log_2 n$ means exactly same. We have already seen how the **log** function works above, did you notice something there?

For $n = 8$, the output of $\log_2 n$ comes out to be 3, which means the array can be halved 3 times maximum, hence the number of steps(at most) to find the target value will be (3 + 1) = 4.

Question for you: What will be the maximum number of guesses required by Binary Search, to search a number in a list of **2,097,152** elements?

Now that we have learned the Binary Search Algorithms, you can also learn other types of Searching Algorithms and their applications:

- Linear Search
- Jump Search