Binary Search - Time and Space Complexity Analysis

Binary Search works on sorted arrays or lists by repeatedly dividing the search interval in half.

1. Time Complexity (TC)

How it works:

- At each step, it eliminates half of the remaining elements.
- So the number of steps is proportional to the number of times you can divide n by 2 -> which is log2(n).

Time Complexity:

Case	Complexity
Best	O(1) - element found at the first mid check
Average	O(log n) - repeatedly halve the array
Worst	O(log n) - element not found after log n divisions

Why O(log n)?

At each iteration:

n -> n/2 -> n/4 -> ... -> 1

Total steps = log2(n)

2. Space Complexity (SC)

Implementation	Space Complexity	Why?
Iterative	O(1)	Uses constant space (variables only)
Recursive	O(log n)	Due to recursion stack (depth = log n)

Example (Iterative):

```
int binarySearch(int[] arr, int target) {
    int start = 0, end = arr.length - 1;
```

```
    while (start <= end) {
        int mid = start + (end - start) / 2;
        if (arr[mid] == target)
            return mid;
        else if (arr[mid] < target)
            start = mid + 1;
        else
            end = mid - 1;
    }
    return -1;
}
```

- TC: O(log n)
- SC: O(1)

Example (Recursive):

```
int binarySearch(int[] arr, int start, int end, int target) {
    if (start > end) return -1;
    int mid = start + (end - start) / 2;
    if (arr[mid] == target) return mid;
    if (arr[mid] > target)
        return binarySearch(arr, start, mid - 1, target);
    else
        return binarySearch(arr, mid + 1, end, target);
}
```

- TC: O(log n)
- SC: O(log n) (due to recursive stack)

Summary Table

| Approach | Time Complexity | Space Complexity |

|--------------|------------------|-------------------|

| Iterative | O(log n) | O(1) |

| Recursive | O(log n) | O(log n) |