

Binary Search Intro

Cut the problem in half every time. >=■

■ What You Will Learn

- Why binary search is faster than linear search
- The key requirement: array MUST be sorted
- How to halve the search space each step
- Implement iterative binary search in Java

■ Intuition First

Think of the number guessing game. I'm thinking of a number from 1 to 100. You guess 50. I say 'higher'. Now you only need to search 51-100. You guess 75. I say 'lower'. 51-74. Each guess **eliminates half the possibilities**. That's binary search.

■ Why SORTED matters

Binary search only works because in a sorted array, you can confidently say: 'Everything to my left is smaller. Everything to my right is larger.' Without sorting, this logic breaks.

■ Visual: Search for 23

```
Sorted Array: [2, 5, 8, 12, 16, 23, 38, 45, 72]
Indices: 0 1 2 3 4 5 6 7 8
Step 1: low=0, high=8, mid=4 → arr[4]=16 < 23 → go RIGHT
[2, 5, 8, 12, 16, | 23, 38, 45, 72]
↑ search here only
Step 2: low=5, high=8, mid=6 → arr[6]=38 > 23 → go LEFT
[23, | 38, 45, 72]
↑ search here
Step 3: low=5, high=5, mid=5 → arr[5]=23 = 23 ✓ FOUND!
Only 3 steps for 9 elements! (Linear would take 6)
```

■■ How It Works

1

Set low=0, high=n-1

Start with the full array range.

2	Find mid = (low+high)/2 The middle index of current range.
3	Compare arr[mid] to target Is it equal? Greater? Less?
4	Equal → return mid Found it!
5	Target > arr[mid] → low = mid+1 Target is in right half. Ignore left.
6	Target < arr[mid] → high = mid-1 Target is in left half. Ignore right.
7	low > high → return -1 Entire range searched, not found.

■ Code Implementation (Java)

```
public class BinarySearch {  
    static int binarySearch(int[] arr, int target) {  
        int low = 0, high = arr.length - 1;  
        while (low <= high) {  
            int mid = low + (high - low) / 2; // avoids overflow  
            if (arr[mid] == target) return mid; // found!  
            else if (arr[mid] < target) low = mid+1; // go right  
            else high = mid - 1; // go left  
        }  
        return -1; // not found  
    }  
    public static void main(String[] args) {  
        int[] arr = {2, 5, 8, 12, 16, 23, 38, 45, 72};  
        System.out.println(binarySearch(arr, 23)); // 5  
        System.out.println(binarySearch(arr, 10)); // -1  
    }  
}
```

■■ Time & Space Complexity

■ TIME COMPLEXITY

$O(\log n)$

■ SPACE COMPLEXITY

$O(1)$

Each step halves the search space. For $n=1,000,000$ elements, you only need about 20 steps! That's the power of logarithmic complexity.

■ Common Mistakes

■ Using $\text{mid} = (\text{low} + \text{high}) / 2$

For very large arrays, $(\text{low} + \text{high})$ can overflow! Use $\text{mid} = \text{low} + (\text{high} - \text{low}) / 2$ instead. Safer and identical result.

■ Running on unsorted array

Binary search on an unsorted array gives WRONG results silently. Always sort first, or verify the array is sorted.

■ Practice Problems

#	Problem	Difficulty
1	Implement binary search and test on a sorted array	Easy
2	Find first occurrence of a repeated element	Medium
3	Find last occurrence of a repeated element	Medium
4	Binary search on a 2D sorted matrix	Medium
5	Find the square root of a number using binary search	Medium

■ Quick Revision Box

- ✓ Binary search REQUIRES a sorted array
- ✓ Each step eliminates HALF the remaining elements
- ✓ Time: $O(\log n)$ — extremely fast
- ✓ Space: $O(1)$ for iterative, $O(\log n)$ for recursive
- ✓ Use $\text{mid} = \text{low} + (\text{high} - \text{low}) / 2$ to avoid overflow
- ✓ Return -1 when $\text{low} > \text{high}$ (not found)

■ Visual Thinking

■ Imagine It Like This

Think of binary search as opening a dictionary to the middle. The word starts with 'M'. Is your word before or after M? You immediately ignore half the book and open to the quarter point. Repeat. Animation: Show an array shrinking from both ends as the window tightens around the target.