Explaining Binary Search: Finding Elements Efficiently 🔍

(Note: read and understand this document first before doing your Lab14)

CRITICAL: Binary search **only works on sorted arrays**. If your array isn't sorted, binary search will give wrong results!

Before you use binary search:

1. ✅ Sort your array using `Arrays.sort()`
2. ✅ Then use binary search

Quick reminder:

```
import java.util.Arrays;

int[] data = {5, 2, 8, 1, 9};  // Unsorted
Arrays.sort(data);              // MUST DO THIS FIRST!
// Now data = [1, 2, 5, 8, 9]

int index = binarySearch(data, 5);  // Now works correctly!
```

What is Binary Search?

Binary Search is a fast way to find an element in a **sorted array**. Instead of checking every element one by one (which is slow), binary search cuts the search space in half with each step.

Key idea: Like looking up a word in a dictionary by opening to the middle, then choosing left or right based on whether your word comes before or after!

Why Binary Search?

Linear Search (the slow way)

```
Array: [2, 5, 8, 12, 16, 23, 38, 45, 56, 67, 78]
Looking for: 23

Step 1: Check 2? No
Step 2: Check 5? No
Step 3: Check 8? No
Step 4: Check 12? No
Step 5: Check 16? No
Step 6: Check 23? YES! Found it!
```

```
Time: 6 checks for 11 elements (O(n) - slow!)
```

Binary Search (the fast way)

```
Array: [2, 5, 8, 12, 16, 23, 38, 45, 56, 67, 78]
Looking for: 23

Step 1: Check middle (23)? YES! Found it in 1 step!

Time: 1 check! (O(log n) - much faster!)
```

Speed comparison:

- 1,000 elements: Linear = ~500 checks, Binary = ~10 checks
- 1,000,000 elements: Linear = ~500,000 checks, Binary = ~20 checks

---

How Binary Search Works: Step by Step

The Algorithm

1. **Start** with left pointer at the beginning, right pointer at the end
2. **Find** the middle element
3. **Compare** the middle element with the target:
    - If **equal** → Found it! ✓
    - If **target < middle** → Search left half
    - If **target > middle** → Search right half
4. **Repeat** until found or search space is empty

Example: Finding 23 in [2, 5, 8, 12, 16, 23, 38, 45, 56, 67, 78]

```
Initial Array: [2, 5, 8, 12, 16, 23, 38, 45, 56, 67, 78]
                0  1  2   3   4   5   6   7   8   9  10

Target: 23

STEP 1:
left = 0, right = 10
middle index = (0 + 10) / 2 = 5
middle value = array[5] = 23
Compare: 23 == 23? YES! FOUND!
Result: Found 23 at index 5 ✓
```

Another Example: Finding 56

```
Initial Array: [2, 5, 8, 12, 16, 23, 38, 45, 56, 67, 78]
                0  1  2   3   4   5   6   7   8   9  10

Target: 56

STEP 1:
left = 0, right = 10
middle index = (0 + 10) / 2 = 5
middle value = array[5] = 23
Compare: 56 > 23? YES! Search right half
Now: left = 6, right = 10

STEP 2:
left = 6, right = 10
middle index = (6 + 10) / 2 = 8
middle value = array[8] = 56
Compare: 56 == 56? YES! FOUND!
Result: Found 56 at index 8 ✓
```

Java Implementation: Iterative Approach

Simple & Easy to Follow

```java
import java.util.Arrays;

public class BinarySearch {

    // Returns the index of target in array, or -1 if not found
    public static int binarySearch(int[] array, int target) {
        int left = 0;                // Start at beginning
        int right = array.length - 1;  // Start at end

        // Keep searching while search space isn't empty
        while (left <= right) {
            // Find middle index (avoid overflow: left + (right - left) /
2)
            int middle = left + (right - left) / 2;
            int middleValue = array[middle];

            // Check middle value
            if (middleValue == target) {
                return middle;  // Found it!
            }
            else if (target < middleValue) {
                // Target is smaller, search left half
                right = middle - 1;
            }
            else {
                // Target is larger, search right half
                left = middle + 1;
            }
        }

        // Not found
        return -1;
    }

    public static void main(String[] args) {
        int[] numbers = {2, 5, 8, 12, 16, 23, 38, 45, 56, 67, 78};

        // ✓ CORRECT: Array is already sorted

        // Search for 23
        int index = binarySearch(numbers, 23);
        if (index != -1) {
            System.out.println("Found 23 at index: " + index);
        } else {
            System.out.println("23 not found");
        }

        // Search for 56
        index = binarySearch(numbers, 56);
        if (index != -1) {
            System.out.println("Found 56 at index: " + index);
        } else {
            System.out.println("56 not found");
        }

        // Search for 100 (doesn't exist)
        index = binarySearch(numbers, 100);
        if (index != -1) {
            System.out.println("Found 100 at index: " + index);
        } else {
            System.out.println("100 not found");
        }
    }
}
```

Output:

```
Found 23 at index: 5
Found 56 at index: 8
100 not found
```

❌ Important: MUST Use Sorted Array!

Example of WRONG usage:

```
// ✗ WRONG - Array is NOT sorted
int[] unsorted = {23, 8, 56, 2, 45, 12, 78, 5, 67, 38, 16};
int index = binarySearch(unsorted, 23);
// Result: Wrong answer! Don't do this!

// ✓ CORRECT - Sort first, then search
Arrays.sort(unsorted);  // Sort first!
// Now unsorted = [2, 5, 8, 12, 16, 23, 38, 45, 56, 67, 78]
index = binarySearch(unsorted, 23);
// Result: Correct! Index 5 ✓
```

Tracing Through the Code

Let's trace finding 45 in [2, 5, 8, 12, 16, 23, 38, 45, 56, 67, 78]:

```
Initial state:
array = [2, 5, 8, 12, 16, 23, 38, 45, 56, 67, 78]
target = 45
left = 0
right = 10

ITERATION 1:
  middle = 0 + (10 - 0) / 2 = 5
  middleValue = array[5] = 23
  45 > 23? YES, search right
  left = 6

ITERATION 2:
  middle = 6 + (10 - 6) / 2 = 8
  middleValue = array[8] = 56
  45 < 56? YES, search left
  right = 7

ITERATION 3:
  middle = 6 + (7 - 6) / 2 = 6
  middleValue = array[6] = 38
  45 > 38? YES, search right
  left = 7

ITERATION 4:
  middle = 7 + (7 - 7) / 2 = 7
  middleValue = array[7] = 45
  45 == 45? YES! FOUND!
  return 7

Result: Found 45 at index 7 ✓
```

Complexity Analysis

Metric	Value
Time Complexity	O(log n) - splits array in half each time
Space Complexity	O(1) - only uses variables (iterative)
Best Case	O(1) - target is at middle
Worst Case	O(log n) - element at end or not found

Comparison

```
Array Size: 1,000,000 elements

Linear Search (check each):
- Average: 500,000 checks
- Worst case: 1,000,000 checks

Binary Search (on sorted array):
- Average: ~20 checks
- Worst case: ~20 checks

Binary search is 25,000x FASTER! 🚀
```

The Complete Process

```
1. GET RAW DATA
   [52, 78, 52, 95, 82, 52, 88, 78, 91, 65, 88, 98, 75, 92, 85]

2. SORT THE DATA (using Arrays.sort)
   [52, 52, 52, 65, 75, 78, 78, 82, 85, 88, 88, 91, 92, 95, 98]

3. NOW USE BINARY SEARCH (on sorted array)
   Search for 85? Found at index 8 in 4 comparisons!

4. REASON FOR BOTH:
   - Without sorting: Would need ~7.5 comparisons on average (linear)
   - With sorting + binary search: 1 sort + 4 searches = efficient!
```

Common Mistakes

❌ Wrong	✅ Correct
Using binary search on unsorted array	Always sort first using `Arrays.sort()`

❌ Wrong	✅ Correct
`int middle = (left + right) / 2`	`int middle = left + (right - left) / 2` (avoid overflow)
`while (left < right)`	`while (left <= right)` (check last element)
Not returning -1 when not found	Return -1 if search fails
Forgetting array is 0-indexed	Array indices start at 0

Key Takeaways

1. ✅ **Array MUST be sorted** - Use `Arrays.sort()` first!
2. ✅ **Binary search is FAST** - O(log n) vs O(n)
3. ✅ **Returns index** - Position in array (0-indexed)
4. ✅ **Returns -1 if not found** - Indicates element doesn't exist
5. ✅ **Divide and conquer** - Cuts search space in half each time

---

Real-World Examples

Where is binary search used?

- 💻 **Database queries** - Index searches on sorted data
- 📊 **Data analysis** - Finding values in sorted datasets
- 🔎 **Search engines** - Finding cached pages
- 📱 **App suggestions** - Autocomplete with sorted data

In ALL cases, data must be sorted first!
