# Lab 14

## Recap: Lab (Github) Workflow 📄 - How to Work on Labs

Follow these steps for every lab carefully to access, complete, and submit your assignment.

1. **Accept the Assignment**

   - Open the Lab Assignment Link the professor provided.
   - Click **"Accept the assignment"**. This will create your personal assignment repository on GitHub under the `OOP-Fall-2025` organization.
   - You'll be taken to your repository page. Verify that the URL looks like `github.com/OOP-Fall-2025/lab-number-yourusername`.

2. **Clone the Repository to Your Computer**

   - On your repository page, click the blue **<> Code** button.
   - In the dropdown menu, choose **"Open with GitHub Desktop"**.
   - GitHub Desktop will launch. Choose a preferred local folder on your computer to save the project and click **"Clone"**.
   - If asked "How are you planning on using this fork?", select **"For my own purpose"** and continue.

3. **Open in VS Code and Start Coding**

   - In GitHub Desktop, ensure the "Current repository" is the one for this lab.
   - Click the **"Open in Visual Studio Code"** button.
   - VS Code will open the project folder. You can now begin writing your solutions in the `Lab14.java` file.

4. **Save and Submit Your Work**

   - **Commit (Save) Changes**: As you work, save your file in VS Code (`Ctrl+S` or `Cmd+S`). To record your progress, go to the **Source Control** tab (the fork icon) on the left sidebar in VS Code. Type a descriptive message in the message box (e.g., "Finished Lab 14") and click **"Commit"**. You must enter a message.
   - **Push (Submit) to GitHub**: When you are finished with the lab or want to back up your work, go back to GitHub Desktop. Click the **"Push origin"** button at the top of the window. This sends your committed changes from your computer to your GitHub repository online.

5. **Verify Your Submission**

   - After you push, you can click **"View on GitHub"** in GitHub Desktop to open your repository in the browser.
   - On the GitHub website, make sure you are viewing the `main` branch and confirm that all of your latest code is visible.

---

## Lab 14 Task

### Binary Search: Monster Power Level Scanner 🔍

*Master binary search by implementing an intelligent monster encounter system! Use sorted arrays and binary search to find appropriate combat encounters in a fantasy dungeon.*

---

## The Challenge

You are building the **Monster Encounter System** for an RPG dungeon! The kingdom's Monster Database contains all known creatures sorted by **power level** (difficulty rating from 0-100). As adventurers enter the dungeon with their own power level, your system uses **binary search** to intelligently match them with appropriate monsters.

**Why Binary Search?** Imagine searching through 10,000 monsters one by one (O(n) - slow!). Binary search cuts this to just ~14 comparisons (O(log n) - blazingly fast!). This is the difference between a laggy game and smooth gameplay!

**Important Note:** This lab requires you to understand and apply the **binary search algorithm from the Binary Search README**. You will NOT just copy the template - you will need to think about how to modify binary search for different purposes (exact match vs. finding boundaries).

---

## Part 1: Create Monster Database & Implement Binary Search 🐉

### What You Need to Do

In the `part1()` method, you will:

1. Display the monster database array nicely formatted
2. Show the total count of monsters
3. Implement the standard binary search algorithm
4. Test your binary search with 3 different searches

### Understanding the Data

```
Monster Database (sorted):
Index:  0   1   2   3   4   5   6   7   8   9  10
Power:  5  12  18  25  32  40  55  68  75  88  95
```

The array is **already sorted** - this is critical for binary search to work!

### Part 1A: Display the Database

You need to print the array nicely. Think about:

- How do you access each element in the array?
- How do you get the total length of an array?
- When printing elements separated by commas, should you print a comma after the last element?

**Hint:** Use a `for` loop to iterate through the array. Remember that `array.length` gives you the number of elements.

**Expected Output:**

```
Power Levels: [5, 12, 18, 25, 32, 40, 55, 68, 75, 88, 95]
Total monsters: 11
Database status: ✓ Sorted and ready!
```

### Part 1B: Implement Binary Search

This is the core algorithm you studied in the Binary Search README. Refer to that document for the exact steps:

**Algorithm Reminder:**

1. Start with `left = 0` and `right = array.length - 1`
2. While `left <= right`:
   - Calculate middle index (use the safe formula from README: `left + (right - left) / 2`)
   - Get the middle value
   - If middle value equals target: **return the index**
   - If target < middle value: search the left half (move right boundary)
   - If target > middle value: search the right half (move left boundary)
3. If loop exits without finding: **return -1**

**Key Points:**

- The method signature is already written: `public static int binarySearch(int[] array, int target)`
- You are returning an **index** (0-10), not the value itself
- Return **-1** if not found (this is a standard convention)
- Use the safe middle calculation to avoid overflow: `left + (right - left) / 2`

### Part 1C: Test Your Binary Search

Test with three scenarios:

1. **Power 40** - This EXISTS in the database (should find it)
2. **Power 50** - This does NOT exist (should return -1)
3. **Power 5** - Boundary test - smallest value (should find it)

For each test, call your `binarySearch()` method and check if the result is -1 or a valid index.

**Expected Output:**

```
--- Testing Binary Search ---
Searching for power 40... ✓ Found at index 5!
Searching for power 50... ✗ Not found (returned -1)
Searching for power 5... ✓ Found at index 0!
```

---

## Part 2: Get Player Power & Find Matching Monsters 👤

### What You Need to Do

In the `part2()` method, you will:

1. Get the player's power level using Scanner
2. Use binary search to find three different types of monsters:
    - **Exact Match**: A monster at exactly the player's power
    - **Beatable Monster**: The strongest monster the player can beat
    - **Challenge Monster**: The weakest monster that would challenge the player

### Part 2A: Get Player Input

**Steps:**

1. Create a `Scanner` object to read user input
2. Ask the user: "What is your power level? (1-100): "
3. Read their integer response

**Hint:** Refer to Lab 13 or other labs where you used Scanner if you need a reminder on how to use it.

### Part 2B: Three Types of Searches

#### Search Type 1: Exact Match

Use your standard `binarySearch()` method from Part 1. This searches for a monster at EXACTLY the player's power level.

**Example:** If player power is 35:

- Look through the database: [5, 12, 18, 25, 32, 40, 55, 68, 75, 88, 95]
- Is there a 35? No! So return -1
- Display: "❌ No monster at exactly power 35"

#### Search Type 2: Beatable Monster (Strongest You Can Beat)

**The Goal:** Find the LARGEST monster power that is ≤ player power

**Example with player power 35:**

```
Array: [5, 12, 18, 25, 32, 40, 55, 68, 75, 88, 95]
         ✓   ✓   ✓   ✓   ✓ ✗ ✗ ✗ ✗ ✗ ✗
All these are ≤ 35, but 32 is the STRONGEST
```

**Algorithm Hint - Modify Binary Search:**

- Use binary search boundaries, but track a **result** variable
- When you find a value ≤ player power, **save it** and keep searching **right** for a stronger one
- When you find a value > player power, search **left**
- After the loop, return your best result (or -1 if none found)

This is a **boundary search** - you're not looking for an exact match, but for the edge/boundary!

**Search Type 3: Challenge Monster (Weakest Challenge)**

**The Goal:** Find the SMALLEST monster power that is > player power

**Example with player power 35:**

```
Array: [5, 12, 18, 25, 32, 40, 55, 68, 75, 88, 95]
        ✗  ✗  ✗  ✗  ✗  ✓  ✓  ✓  ✓  ✓  ✓
All these are > 35, but 40 is the WEAKEST
```

**Algorithm Hint - Modify Binary Search:**

- Similar to the beatable search, but the conditions are opposite
- When you find a value > player power, **save it** and keep searching **left** for a weaker challenge
- When you find a value ≤ player power, search **right**
- After the loop, return your best result (or -1 if no challenge exists)

**Key Insight:** Both `findBeatableMonster()` and `findChallengeMonster()` use binary search logic, but with modified conditions. You're not looking for an exact value - you're finding boundaries!

---

# Part 3: Interactive Monster Hunter Game 🎮

## What You Need to Do

Implement a game loop where:

1. Player sees their current power and available monsters
2. Player chooses from 4 menu options
3. Different outcomes based on their choice
4. Game continues until they exit

## Part 3A: Game Variables

Initialize these at the start:

- `playerPower` - Start at 35
- `startingPower` - Remember the starting value
- `wins` - Count successful hunts
- `losses` - Count failed attempts
- `playing` - Boolean to control the loop

Also create:

- `Scanner` for reading menu choices
- `Random` for 50/50 chance on challenges

### Part 3B: The Main Game Loop

**Structure:**

```
while (playing) {
    Find available monsters
    Display status
    Show menu
    Get player choice
    Handle choice (switch statement)
}
```

### Part 3C: Menu Options

**Option 1: Hunt a Beatable Monster (Guaranteed Win)**

**Mechanics:**

- Find the beatable monster
- If one exists:
  - Announce the fight
  - Player automatically wins
  - Gain **+2 power**
  - Increment **wins**
- If none exists:
  - Tell player they have no beatable monsters

**Hint:** Save the old power before changing it so you can show "35 → 37"

**Option 2: Attempt a Challenge (50/50 Chance)**

**Mechanics:**

- Find the challenge monster
- If one exists:
  - Announce the fight
  - Generate a random outcome (use `random.nextBoolean()` for 50/50)
  - If victory:
    - Player wins
    - Gain **+5 power** (more reward for harder fight)
    - Increment **wins**
  - If defeat:
    - Player loses and escapes

- Power stays the same (no penalty, no reward)
- Increment **losses**
- If none exists:
  - Tell player they can beat all monsters

**Hint:** `random.nextBoolean()` returns `true` or `false` with 50/50 probability

**Option 3: Search for Specific Monster**

**Mechanics:**

- Ask: "What power level to search for?"
- Use `binarySearch()` to find it
- Display whether it was found and at what index

**Example Output:**

```
What power level to search for? 55
🔍 Using binary search...
✓ Found monster at power 55!
Located at index 6! ⚡
```

**Option 4: Exit**

**Mechanics:**

- Set `playing = false`
- This exits the loop
- Game ends and shows final report

### Part 3D: Final Report

After the game loop ends, display:

```
=== SESSION COMPLETE ===
Starting power: 35
Final power: [whatever it is now]
Hunts completed: [wins] wins, [losses] losses
Power gained: [final - starting]
```

---

## Implementing findBeatableMonster() 💪

### Understanding the Problem

```
```

```
Database: [5, 12, 18, 25, 32, 40, 55, 68, 75, 88, 95]
Player Power: 35

Question: What's the strongest monster I can beat?
Answer: 32 (it's the largest value ≤ 35)
```

### Algorithm (Using Binary Search Logic)

**Key Idea:** Use binary search but keep track of the **best result found so far**.

1. Initialize: `left = 0`, `right = array.length - 1`, `result = -1`
2. While `left <= right`:
   - Calculate `middle = left + (right - left) / 2`
   - Get `middleValue = array[middle]`
   - If `middleValue <= playerPower`:
     - **Save this as a potential answer:** `result = middleValue`
     - **Search right for something stronger:** `left = middle + 1`
   - Else (value is too strong):
     - **Search left for something weaker:** `right = middle - 1`
3. Return `result` (which will be -1 if nothing found, or the strongest beatable monster)

### Why This Works

By moving `left` to `middle + 1` when we find a valid candidate, we keep trying to find something **stronger** but still beatable. This efficiently finds the boundary between beatable and unbeatable.

### Example Walkthrough

```
Array: [5, 12, 18, 25, 32, 40, 55, 68, 75, 88, 95]
Player: 35

Iteration 1:
  left=0, right=10
  middle = 5
  array[5] = 40
  40 > 35? YES
  So search left: right = 4

Iteration 2:
  left=0, right=4
  middle = 2
  array[2] = 18
  18 <= 35? YES
  Save 18, search right: left = 3, result = 18

Iteration 3:
  left=3, right=4
  middle = 3
  array[3] = 25
```

```
  25 <= 35? YES
  Save 25, search right: left = 4, result = 25

Iteration 4:
  left=4, right=4
  middle = 4
  array[4] = 32
  32 <= 35? YES
  Save 32, search right: left = 5, result = 32

Iteration 5:
  left=5, right=4
  left > right, exit loop

Return: 32 ✓
```

---

## Implementing findChallengeMonster() 🔥

### Understanding the Problem

```
Database: [5, 12, 18, 25, 32, 40, 55, 68, 75, 88, 95]
Player Power: 35

Question: What's the weakest monster that's stronger than me?
Answer: 40 (it's the smallest value > 35)
```

### Algorithm (Using Binary Search Logic)

**Key Idea:** Similar to findBeatable, but the logic is **reversed**.

1. Initialize: `left = 0`, `right = array.length - 1`, `result = -1`
2. While `left <= right`:
   - Calculate `middle = left + (right - left) / 2`
   - Get `middleValue = array[middle]`
   - If `middleValue > playerPower`:
     - **Save this as a potential answer:** `result = middleValue`
     - **Search left for something weaker:** `right = middle - 1`
   - Else (value is not strong enough):
     - **Search right for something stronger:** `left = middle + 1`
3. Return `result` (which will be -1 if nothing found, or the weakest challenge)

### Why This Works

By moving `right` to `middle - 1` when we find a valid challenge, we keep trying to find something **weaker** but still challenging. This efficiently finds the boundary between what we can beat and what challenges us.

---

## Common Pitfalls & Debugging Tips ⚠

### Pitfall 1: Forgetting Array is 0-Indexed

```
❌ WRONG: Array has elements at positions 0-11? No!
✅ RIGHT: Array has 11 elements at positions 0-10
```

### Pitfall 2: Not Checking if Array is Sorted

Binary search only works on sorted arrays! The array `{5, 12, 18, ...}` IS sorted, so you don't need to sort it. But always verify this assumption!

### Pitfall 3: Infinite Loop in Binary Search

If you forget to update `left` or `right`, the loop will never exit!

```
✅ Make sure: left = middle + 1 OR right = middle - 1 happens
```

### Pitfall 4: Off-by-One Errors

```
❌ WRONG: while (left < right)     // Misses when left == right
✅ RIGHT: while (left <= right)    // Checks all elements
```

### Pitfall 5: Confusing Index vs. Value

```
❌ WRONG: return array[target];    // This is the wrong thing!
✅ RIGHT: return middle;            // Return the INDEX
```

### Pitfall 6: Not Handling Edge Cases

Test your `findBeatableMonster()` with:

- Player power = 2 (no beatable monsters)
- Player power = 100 (all monsters beatable)
- Player power = 32 (exact match)
- Player power = 35 (no exact match)

### Debugging Strategy

Add print statements inside your methods to trace execution:

```
```

```
System.out.println("left=" + left + ", right=" + right + ", middle=" +
middle + ", value=" + middleValue);
```

This helps you see what the algorithm is doing at each step!

---

## Code Structure Hints

### Scanner Usage

```
Scanner scanner = new Scanner(System.in);
System.out.print("Prompt: ");
int value = scanner.nextInt();
```

### If/Else Chains for Display

```
if (result != -1) {
    System.out.println("✓ Found: " + result);
} else {
    System.out.println("✗ Not found");
}
```

### Switch Statement

```
switch (choice) {
    case 1:
        // Handle option 1
        break;
    case 2:
        // Handle option 2
        break;
    // ... more cases
    default:
        System.out.println("Invalid choice");
}
```

### Random Boolean

```
Random random = new Random();
boolean victory = random.nextBoolean();  // true or false, 50/50
if (victory) {
    // Win
} else {
```

```
        // Lose
    }
```

---

## Testing Your Code

### Test Cases to Verify

**Part 1 - Binary Search:**

- [ ] Searching for 40 returns index 5
- [ ] Searching for 50 returns -1
- [ ] Searching for 5 returns index 0
- [ ] Searching for 95 returns index 10

**Part 2 - Player Matching:**

- [ ] With player power 35:
  - Exact match returns -1 (no 35 in array)
  - Beatable returns 32
  - Challenge returns 40
- [ ] With player power 40:
  - Exact match returns 5 (40 exists!)
  - Beatable returns 40
  - Challenge returns 55

**Part 3 - Game Loop:**

- [ ] Menu displays correctly
- [ ] Option 1 increases power by 2
- [ ] Option 2 randomly wins (+5) or loses (no change)
- [ ] Option 3 searches correctly
- [ ] Option 4 exits the loop
- [ ] Final report shows correct stats

---

## Why This Matters

This lab teaches you:

1. **Binary search** - One of the most important algorithms in computer science
2. **Algorithm adaptation** - Taking a standard algorithm and modifying it for new problems
3. **Game design** - How difficulty matching works in real games
4. **Problem-solving** - Breaking complex systems into manageable parts

Professional game developers use binary search for matchmaking, difficulty scaling, and asset loading!

---

## Need Help?

**Stuck on binary search?** Re-read the BinarySearchExplained.md document and trace through the algorithm by hand with a small example.

**Stuck on findBeatableMonster?** Draw the array on paper and trace which direction you should search at each step.

**Stuck on findChallengeMonster?** This is the REVERSE of findBeatableMonster - think about what changes!

**Game loop not working?** Test each option individually before connecting them together.

---

## Finished?

When done with the lab (committed and pushed on GitHub), show instructor and state your name to be marked as done!

---