Topic Introduction

Today we’re diving deep into **binary search** — the classic algorithm that slices problems in half. If you’ve ever looked for a word in a dictionary or found a song in a sorted playlist, you’ve used binary search (maybe without realizing it!).

What is Binary Search?
Binary search is a fast way to find an element in a sorted list. Instead of scanning each element one by one, you repeatedly cut the search space in half. You start by checking the middle element. Is it what you want? If not, you decide whether to search the left or right half next, based on sorting.

How does it work?
Here’s a high-level process:

- Start with the full range: `left = 0`, `right = n - 1`.
- Calculate the middle: `mid = left + (right - left) // 2`.
- Compare your target to `nums[mid]`.
  - If equal, you found it!
  - If the target is smaller, search the left half.
  - If the target is larger, search the right half.
- Repeat until you find the target or run out of elements.

When is it useful?
Binary search shines whenever you need to find something in a sorted array, list, or even in more abstract "search spaces" (like version numbers or possible answer ranges). It’s often used in interviews to solve problems efficiently that would otherwise be slow with brute force.

Quick Example (not one of today’s problems):
Suppose you have the array `[2, 4, 6, 8, 10]` and want to check if `6` exists.

- Check middle: `6` at index 2. Found it! Only one comparison.
- With linear search, you’d check 2, then 4, then 6 (three checks).

Why These Problems?

Today’s three problems are all about finding *positions* in a sorted space using binary search:

- **First Bad Version**: Find the first occurrence of a "bad" version.
- **Search Insert Position**: Find where a target value fits in a sorted list.
- **Find Peak Element**: Find a "peak" (local maximum) in a list.

They look different on the surface but all use binary search to efficiently find a specific index rather than a specific value. Let’s see how.

Problem 1: First Bad Version

First Bad Version – LeetCode #278

Problem Statement (Reworded):

Imagine you have a series of product versions, numbered from `1` to `n`. At some point, a bad version was introduced. Every version after that is also bad. You're given an API `isBadVersion(version)` that tells you if a version is bad. Your job: find the *first* bad version.

Example:
Suppose `n = 5`, and version 4 is the first bad one.
Calling `isBadVersion(3)` returns `False`
Calling `isBadVersion(4)` returns `True`
So the answer is 4.

Thought Process:
You need to minimize calls to `isBadVersion`. If you check every version, that's slow (O(n)). But since once a version is bad, all after it are bad, you can treat this like a sorted array: False (good), False (good), ..., True (bad), True (bad), etc. You want the first `True`.

Walkthrough Example:
Say `n = 5`, first bad version = 4:

- left = 1, right = 5, mid = 3: isBadVersion(3) -> False
- So, bad version is after 3. Move left to 4.
- left = 4, right = 5, mid = 4: isBadVersion(4) -> True
- Now, maybe the first bad is before 4. Move right to 4.
- When left == right, done. Return 4.

Try This Example:
`n = 7`, first bad version = 5

Brute-Force Approach:
Check `isBadVersion(version)` for every version from 1 to n.

- Time: O(n)
- Space: O(1)

Optimal Approach:
Use binary search.

- Always check the middle version.
- If it's bad, first bad could be earlier, so move right pointer to mid.
- If it's good, move left pointer to mid + 1.

Python Solution

```python
# Suppose isBadVersion is provided by the system
def firstBadVersion(n):
    left = 1
    right = n
    while left < right:
        mid = left + (right - left) // 2  # Avoid overflow
        if isBadVersion(mid):
            right = mid  # Could be the answer, so include mid
        else:
            left = mid + 1  # First bad must be after mid
    return left  # Or right, since left == right
```

- **Time Complexity:** O(log n) — We halve the range each time.
- **Space Complexity:** O(1) — Just a few variables.

What does each part do?

- `left` and `right`: bounds of our search.
- `mid`: current version to check.
- If `isBadVersion(mid)` is True, we might have found the first bad version, so search the left half (including mid).
- Otherwise, search the right half (after mid).
- When left meets right, found the answer.

Trace Example:

Suppose `n = 7`, first bad version is 5:

- left = 1, right = 7, mid = 4: isBadVersion(4) -> False
- left = 5, right = 7, mid = 6: isBadVersion(6) -> True
- left = 5, right = 6, mid = 5: isBadVersion(5) -> True
- left = 5, right = 5: Done. Return 5.

Try this test case yourself:

`n = 8`, first bad version = 2

Take a moment to solve this on your own before jumping into the solution!

Did you know: This problem is a classic *search for the boundary* using binary search. Once you're comfortable with this, many "find the first/last X" problems become approachable.

Problem 2: Search Insert Position

Search Insert Position – LeetCode #35

Problem Statement (Reworded):

Given a sorted array, find the index where a given target value is located. If the target isn't present, return the index where it should be inserted to keep the array sorted.

Example:

Input: `nums = [1, 3, 5, 6], target = 5`

Output: `2`

Another Example:

Input: `nums = [1, 3, 5, 6], target = 2`

Output: `1` (2 fits between 1 and 3)

How is this similar to Problem 1?

Like "First Bad Version," we're finding a *position* in a sorted array, not just checking for existence. The "boundary" here is where the target belongs.

Brute-Force Approach:

Scan from left to right, return index if found, otherwise return index of the first element greater than target.

- Time: O(n)
- Space: O(1)

Optimal Approach:

We can use binary search:

- Search for the target.
- If found, return index.
- If not found, `left` will be at the correct insert position.

Step-by-Step Solution:

- Set `left = 0`, `right = len(nums) - 1`.
- While left <= right:
  - Compute mid.
  - If nums[mid] == target: return mid.
  - If nums[mid] < target: left = mid + 1.
  - Else: right = mid - 1.
- If not found, return `left`.

Pseudocode:

```
function searchInsert(nums, target):
    left = 0
    right = length(nums) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if nums[mid] == target:
            return mid
        else if nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return left
```

Example Trace:

Input: `nums = [1, 3, 5, 6], target = 2`

- left = 0, right = 3, mid = 1, nums[mid] = 3
- 2 < 3: right = 0
- left = 0, right = 0, mid = 0, nums[mid] = 1
- 2 > 1: left = 1
- Now left > right, return left = 1

Try This Test Case:

Input: `nums = [1, 3, 5, 6], target = 7` (should return 4)

Complexity:

- Time: O(log n)
- Space: O(1)

Problem 3: Find Peak Element

[Find Peak Element – LeetCode #162]

Problem Statement (Reworded):
Given an array of numbers, find an index where the value is greater than its neighbors (a "peak"). The array may have multiple peaks; return any peak's index.

How is this different?
We're not necessarily searching for a specific value, but for a *local maximum*. The array isn't sorted, but we can treat it like a search for a boundary — if we're not at a peak, we can use binary search logic to move towards one.

Brute-Force Approach:
Scan the array, compare each element to its neighbors.

- Time: O(n)
- Space: O(1)

Optimal Approach:
Use binary search:

- Check mid.
- If nums[mid] < nums[mid + 1], peak must be to the right.
- Else, peak is to the left or at mid.

Pseudocode:

```
function findPeakElement(nums):
    left = 0
    right = length(nums) - 1
    while left < right:
        mid = left + (right - left) // 2
        if nums[mid] < nums[mid + 1]:
            left = mid + 1
        else:
            right = mid
    return left  // or right
```

Example Trace:
Input: `nums = [1, 2, 1, 3, 5, 6, 4]`

- left = 0, right = 6, mid = 3, nums[3]=3, nums[4]=5
- 3 < 5: left = 4

- left = 4, right = 6, mid = 5, nums[5]=6, nums[6]=4
- 6 > 4: right = 5
- left = 4, right = 5, mid = 4, nums[4]=5, nums[5]=6
- 5 < 6: left = 5
- left = right = 5: return 5

So, index 5 is a peak (value 6).

Try This Test Case:
Input: `nums = [1, 3, 2, 1]` (should return 1, since 3 is a peak)

Complexity:

- Time: O(log n)
- Space: O(1)

Nudge:
Notice how, in each of these problems, binary search helps you "zero in" on the answer by repeatedly halving the problem space. Can you think of other variations where you’re not looking for a value, but a position or boundary?

Summary and Next Steps

What did we cover?

- Three problems, all using binary search to find a *position* or *boundary* — not just a value.
- Key insights:
  - Binary search isn’t just for finding numbers: it also finds first/last occurrences, insertion points, and even special positions like peaks.
  - The pattern: Use sorted/monotonic properties (or local clues) to halve the search space.
  - Move bounds (`left` and `right`) carefully to avoid missing the correct answer or running into infinite loops.

Common Traps:

- Off-by-one errors (is it `left < right`, or `left <= right`?).
- Forgetting to check boundaries (e.g., is the first or last element a peak?).
- Not updating pointers properly after checking `mid`.

Action List

- **Solve all three problems on your own**, even the one we gave you code for. Dry-run test cases on paper to build intuition.
- **Try solving Problem 2 and 3 using a different technique** (e.g., for Problem 2, try linear search; for Problem 3, brute-force).
- **Explore other problems** that use binary search to find boundaries, like "Find Minimum in Rotated Sorted Array" or "Find Kth Smallest Element."
- **Compare your solution with others** — look for elegant pointer updates and edge-case handling.
- **Practice, practice, practice!** If you get stuck, trace through examples by hand. The more you practice, the more natural these patterns become.

Keep going — you’re building real, interview-ready binary search muscle!