Hi everyone!

Today our topic is **Binary Search** — one of the most classic and elegant algorithms you'll ever use in programming.

Let's start with the basics.

Basic Concepts

What is Binary Search?

So, what exactly *is* Binary Search?

If I had to describe it in one sentence, I'd say: Binary Search is a strategy that **keeps shrinking the search space** until we find the answer.

Here's the key idea:

When you have a **sorted** or **monotonic** range — meaning something that goes steadily up or down — you can pick a middle point, `mid`, check whether it meets your condition, and **eliminate half** of the possibilities right away.

In other words, every step cuts the problem size in half.

Let's take a classic example: finding a target value in a sorted array.

```
nums = [-1, 0, 3, 5, 9, 12]
target = 9

[-1, 0, 3, 5, 9, 12]  mid = (start + end) // 2 = 5//2 = 2
       mid < target
          [5, 9, 12]  mid = (start + end) // 2 = (3+5) // 2 = 4
              mid = target
```

So what happens here?

- If `nums[mid] == target`, we're done.
- If it's smaller than the target, we move to the **right half**.
- If it's larger, we move to the **left half**.

Each time, the search range becomes half as big. That's why the time complexity goes from **O(n)** down to **O(log n)** — a huge improvement.

You can think of it this way: The goal of Binary Search isn't really to *find* a number, it's to **eliminate what's impossible**, step by step, until only the answer remains.

Why learn Binary Search?

Now let’s talk about *why* we should learn binary search.

Binary Search isn’t just about finding a number in a sorted array. It’s a structured and efficient problem-solving strategy.

The magic is this:

If you can turn a problem into one where the range of valid answers is monotonic — then you can almost always solve it with binary search. We first define the **search space**, then use the **monotonic property** to zoom in on the answer step by step.

That’s why binary search is one of the most widely used and powerful tools in algorithm interviews. It’s not only for “finding a number” — it also applies to any problem that has some kind of monotonic structure.

There are some examples:

Type	Example	Leetcode Problem
Searching	Find if an element exists	Leetcode 704
Boundary	Find the first element ≥ target	Leetcode 34
Optimization	Minimize the maximum / Maximize the minimum	Leetcode 875 *(Koko Eating Bananas)*
Feasibility	Check if a value satisfies a condition	Leetcode 1011 *(Capacity to Ship Packages Within D Days)*

Even though these problems *look* different, they’re actually the same underneath. In all of them, we’re working within a monotonic range and **eliminating impossible regions** step by step. Once you learn how to reformulate a problem into something that can be “binary-searched,” many seemingly complex problems suddenly become simple and elegant.

When to Use Binary Search?

Now that we know what binary search *is*, let’s talk about something even more important: **when** should we actually use it?

There are three key words to remember.

Monotonicity

The first keyword is **Monotonicity**. If a function or a condition is monotonic, meaning that once it becomes *true*, it stays *true* afterwards(or once it becomes *false*, it stays *false*), then binary search can usually be applied.

For example, imagine checking whether you can finish eating bananas at a certain speed. If speed `x` works, then any speed faster than `x` will also work. That's a monotonic relationship.

Decision Function

The second keyword is **Decision Function**.

If you can write a function like `check(mid)` that simply answers "Is this `mid` valid?" or "Does this value satisfy the condition?", and that function returns a clear **True or False**, then you can use that to decide which half of the range to keep.

Bounded Range

The third keyword is **Bounded Range**.

You need to have a clearly defined search space — a finite range `[l, r]`. That range could be:

- an **array index**,
- a **numeric range** like speed, time, or capacity,
- or even a **logical space** of possible answers.

As long as that space is bounded and can be continuously narrowed down, binary search will work beautifully.

In summary, whenever a problem can be defined by a **monotonic condition** over a **bounded search space**, and you can evaluate it with a **decision function**, you can and should apply binary search.

Template

Now we're getting into the most important part of today — **the Binary Search Template**.

When I first started learning binary search, my first instinct was just to write `while left <= right` and try to fill in the blanks by memory.

But actually, a correct binary search isn't about memorizing a template, it's about **logical reasoning**.

If you can clearly answer these three questions:

1. What are you searching over?

2. How do you check the condition?
3. When do you stop?

Then you've truly understood the essence of binary search.

The Core Template

There are three logical steps to every binary search:

1. Define the **search space**.
2. Design a **check function**.
3. Decide the **loop condition and exit rule** (also known as the loop invariant).

Step 1: Define the Search Space

First, you have to know *what* you're actually binary searching on.

Binary search is about narrowing down a **bounded range** where something is true or false.

- Sometimes it's **array indices** → for example, Leetcode 704 (searching for a target in a sorted array).
- Sometimes it's a **numeric range** → like Leetcode 875, where Koko's eating speed is between `[1, max(piles)]`.
- And sometimes it's an **answer range** → like finding the *minimum ship capacity* that can carry all packages.

So remember: binary search isn't just "finding a number." It's about **eliminating the impossible** within a range `[left, right]`, step by step, until you converge on the answer.

Step 2: Define the Check Function

ext comes the most critical part — the **check function**.

You want to ask: "Given a `mid`, does it satisfy the condition?"

Take the Koko Eating Bananas problem as an example. `check(mid)` means: *If Koko eats at speed `mid`, can she finish all the bananas within H hours?*

比如在 Koko 吃香蕉那题：`check(mid)` 表示 '以速度 mid 能不能在规定时间内吃完香蕉？'

- If the answer is simply **yes or no**,
- and that answer changes in a **monotonic way**, meaning once it becomes `True`, it always stays `True`

Then binary search will work perfectly. Once you have this monotonic behavior, you can safely narrow down your range toward the boundary where the condition flips.

Step 3: Decide the Exit Condition (Loop Invariant)

Finally, the part where most people make mistakes — **when to stop**. There are two common loop patterns, each with its own use case.

(1) `while left <= right` — Searching for an Exact Value

This is the most direct version, used when you're looking for a specific target. When the loop ends without returning, it means the target doesn't exist.

```
# The logic is straightforward: if the midpoint is equal to the target, return it; otherwise, move the bounds based on the comparison result.
# The key here is that the range must strictly shrink each time.
while left <= right:
    mid = (left + right) // 2
    if nums[mid] == target:
        return mid
    elif nums[mid] < target:
        left = mid + 1
    else:
        right = mid - 1
```

(2) `while left < right` — Finding a Boundary or Extreme Value

This pattern is used when you want to find a boundary or an optimal value. The loop ends when `left == right`, which represents the smallest feasible answer.

```
# The idea is to keep shrinking the interval until left == right.
## If check(mid) holds, we keep mid (indicating it might still be a feasible solution).
## Otherwise, we shift the left boundary right to eliminate the infeasible portion.
## When the loop ends, left and right will coincide, and this point is the 'minimum feasible solution' or 'boundary point' we're looking for.
while left < right:
    mid = (left + right) // 2
    if check(mid):
        right = mid      # keep the feasible candidate
    else:
        left = mid + 1   # discard the infeasible part
return left
```

Here, we're not looking for an exact match. Instead, we're *shrinking the range* until only one valid candidate remains — the smallest or largest value that still satisfies the condition.

So by now, you can see that the binary search “template” isn’t something to memorize. It’s something you can derive logically.

As long as you think through these three questions:

1. What’s my **search space**?
2. What’s my **check function**?
3. What’s my **exit condition**?

You’ll be able to write any binary search from scratch — cleanly, confidently, and correctly.

Common Mistakes

Now that we’ve covered the template, let’s talk about where things *usually* go wrong.

Binary search looks simple — but it’s full of small details. One tiny wrong boundary can create really weird bugs: infinite loops, missing answers, or wrong indices.

So I’ve summarized a few of the most common mistakes, see if you’ve made any of these before,

Infinite Loop — Failing to Shrink the Range

This is the **#1 most common mistake**.

```
if nums[mid] < target:
    left = mid    # should be mid + 1
```

Here’s what happens: you calculate `mid`, but if you don’t add `+1`, then the next iteration will use the same `mid` again. The range `[left, right]` never gets smaller, and you’ve got yourself an infinite loop

Always remember: In a `while left <= right` loop, you must **shrink** the range every single time. That means updating `left = mid + 1` or `right = mid - 1` accordingly.

Non-Monotonic Check Function

If your `check(mid)` sometimes returns `True`, then later goes back to `False`, the entire binary search will break. Because it doesn’t know which side to search anymore.

To make binary search work, your `check()` must behave **monotonically**, as `mid` increases, the result should change only once: either from `False → True` or from `True → False`. That's what gives binary search its direction.

Confusing Return Value vs. Index

And finally — a small but very common mistake: not knowing what to return at the end.

- If you're doing a **search problem**, like finding a target in an array → return the **index**.
- If you're doing an **optimization problem**, like finding the *minimum speed* or *minimum capacity*, then you're returning the **value itself**, usually just `left` (or sometimes `right`, depending on your loop).

So always ask yourself at the end: Am I returning a **position** or an **answer value**?

Types of Problems

At its core, binary search is really just **monotonicity + a check function**. That's why it shows up in so many different types of problems.

In fact, almost every binary search problem can be grouped into four major categories. Let's go through them one by one.

Searching for a Target

This is the **classic** type of binary search — the one we all start with.

Keyword: "Find a specific value in a sorted array."

Typical features:

- The search space is the array indices.
- The check condition is simply `nums[mid] == target`.
- The loop condition is usually `while left <= right`.

This is the foundation — the simplest form of binary search.

Finding Boundaries

The second category is **boundary search**, which is slightly trickier. Because the goal is not the 'specific value', but the 'critical threshold'. This type appears in problems like finding the first occurrence of a number, or the first element greater than or equal to a target.

Keyword: "Find the first or last element that satisfies a condition."

Keyword: "Find the first or last element that satisfies a condition."

Typical features:

- The answer you're returning is a **boundary index**, not a value.
- The loop usually uses `while left < right`.
- The update logic must be handled **very carefully**, because if you move one step wrong, you'll skip over the boundary.

The core question here is: How do I design my `check(mid)` so I can move toward the boundary **without losing it**?

Binary Search on Answer Space

Now, this one is especially fun and powerful. Here, we're not searching inside an array, we're searching within a **range of possible answers** `[low, high]`. So what we're "binary searching" is not the data itself, but the **answer**.

This category is perfect for problems that ask you to "minimize the maximum" or "maximize the minimum."

The key steps are:

1. Define a monotonic `check(mid)` — a function that says whether a certain `mid` is feasible.
2. Binary search over `[low, high]` to find the smallest or largest valid `mid`.

You'll see this pattern in problems like *Koko Eating Bananas* or *Ship Packages Within D Days.*

Structural Binary Search

Finally, we have the **structural** or **special structure** type.

Keyword: "The data isn't globally sorted, but it has a structure that allows binary search."

Typical features:

- The array itself isn't monotonic overall,but it follows some **piecewise or local property**.
- You can still decide which side to move toward by checking `mid`.

These problems look like logic puzzles at first, but underneath, you're still doing the same thing, using binary search to keep narrowing down the possible region.

No matter the surface difference — whether you're finding a value, a boundary, an answer, or a structure — binary search always follows the same pattern: Define a range, use a monotonic check, and shrink your space until the answer emerges.

Practice

Let's walk through one of the most classic binary search problems — **Leetcode 704: Binary Search.**

Problem: You're given a sorted integer array `nums` and a target value `target`. Return the index of `target` if it exists, otherwise return `-1`.

```
Input: nums = [-1,0,3,5,9,12], target = 9
Output: 4
Explanation: 9 appears at index 4
```

Now let's solve it step by step, using our three-step logic from earlier.

> ## Define the Search Space
>
> What are we actually binary searching over?
>
> Here, the data is a sorted array, so our search space is the index range — from `0` to `len(nums) - 1`. That's our initial interval `[left, right]`.

> ## Define the Check Function
>
> Next, we define our check condition.
>
> At each step, we compute `mid = (left + right) // 2`, and compare `nums[mid]` with `target`:
>
> - If `nums[mid] == target` → we found it!
> - If `nums[mid] < target` → the target must be in the **right half**.
> - If `nums[mid] > target` → the target must be in the **left half**.
>
> Each check lets us **eliminate half of the search space**.

> ## Define the Exit Condition
>
> Finally, when do we stop?
>
> The loop continues while the range is valid — `left <= right`. Once `left` goes beyond `right`, it means the search space is empty, and if we haven't found the target yet, it doesn't exist.

Here's the full implementation:

```python
def search(nums, target):
    left, right = 0, len(nums) - 1
    while left <= right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Each loop halves the remaining range, so the time complexity is **O(log n)**, and the space complexity is **O(1)**.

Even though this is the simplest binary search problem, it captures the essence of the whole idea: Define your space → check the middle → shrink correctly → stop when it's empty.

Classic Question Classification

Searching for a Target

- Leetcode 704 Binary Search
- Leetcode 374 Guess Number Higher or Lower
- Leetcode 702 Search in a Sorted Array of Unknown Size

Finding Boundaries

- Leetcode 34 Find First and Last Position of Element in Sorted Array
- Leetcode 35 Search Insert Position
- Leetcode 744 Find Smallest Letter Greater Than Target
- Leetcode 658 Find K Closest Elements

Binary Search on Answer

- Leetcode 875 Koko Eating Bananas
- Leetcode 1011 Capacity To Ship Packages Within D Days
- Leetcode 1760 Minimum Limit of Balls in a Bag

Special Structure

- Leetcode 162 Find Peak Element
- Leetcode 153 Find Minimum in Rotated Sorted Array
- Leetcode 33 Search in Rotated Sorted Array

- Leetcode 33 Search in Rotated Sorted Array
- Leetcode 852 Peak Index in a Mountain Array