Basic Concepts

What is Binary Search?

- Binary Search is a strategy that **keeps shrinking the search space** until we find the answer.

Here's the key idea:

- The range is **sorted or monotonic** (values steadily go up or down).
- Pick the middle index `mid`.
- Check the condition at `mid`.
- Eliminate half of the range based on the result.

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
```

- 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 step halves the range → **O(log n)** time, **O(1)** space.

> 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.

It's a structured, efficient problem-solving strategy.

> 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.
>
> 1. Define the search space (indices / numeric range / answer range).
> 2. Write a decision function `check(mid) -> True/False`.
> 3. Ensure monotonicity of `check`.
> 4. Shrink the range toward the boundary where `check` flips.

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)*

When to Use Binary Search?

Monotonicity

The condition must be **monotonic**: Once it becomes True, it stays True; or once False, stays False. The result changes **only once**, allowing us to narrow toward the boundary.

Example: If speed `x` works for Koko to finish bananas, then any faster speed also works.

Decision Function

We can define a **check function** → `check(mid)`. This function divides the space into two regions: one valid (True) and one invalid (False).

Bounded Range

We need a finite, defined search space `[l, r]`. It could be:

- Array indices
- Numeric values (speed, time, capacity)
- Logical possibilities

The range must be continuous and shrinks each step.

Template

Don't **memorize a template — reason it out**.

Answer three questions:

1. What are you searching over?
2. How do you check the condition?
3. When do you stop?

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

Binary search works on a **bounded range** where something is True/False.

Common types:

- Array indices → *Leetcode 704*
- Numeric range (speed, time, capacity) → *Leetcode 875*
- Answer range → minimum feasible solution

It's not about *finding a number*, but **eliminating the impossible** in `[left, right]` until convergence.

Step 2: Define the Check Function

We should ask: “Given a `mid`, does it satisfy the condition?”

Example (*Koko Eating Bananas*): `check(mid)` means: *If Koko eats at speed `mid`, can she finish all the bananas within H hours?*

Requirements:

- Returns Boolean (yes/no)
- Must be monotonic → once True, always True (or vice versa)

Once monotonicity holds, binary search can locate the boundary efficiently.

Step 3: Decide the Exit Condition (Loop Invariant)

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

- Use when looking for specific element.
- The range must strictly shrink (`+1` / `-1`).

```
# 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

- Use when looking for boundary / minimal feasible value.
- Stop when `left == right`.

```
# 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
```

Common Mistakes

Infinite Loop — Failing to Shrink the Range

Most common bug!

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

Without `+1` or `-1`, `mid` repeats → range `[left, right]` never shrinks → infinite loop.

Rule: In `while left <= right`, the range must shrink every loop.

Non-Monotonic Check Function

If `check(mid)` flips back and forth (True → False → True), binary search loses direction.

Rule: Ensure monotonicity — result changes only once:

- `False → True` (finding minimal feasible)
- `True → False` (finding maximal feasible)

Confusing Return Value vs. Index

Don’t confuse index vs value at the end.

- Search problem → return the index (e.g. Leetcode 704).
- Optimization problem → return the value itself (`left` or `right`, depending on loop).

Always double-check: "Am I returning a **position** (index) or a **solution value**?"

Types of Problems

Searching for a Target

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

Typical features:

- Search space → array indices
- Condition → `nums[mid] == target`
- Loop → `while left <= right`

Finding Boundaries

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

Typical features:

- Return → boundary index, not value
- Loop → `while left < right`
- Must update carefully to avoid skipping 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

Keyword: "Search for the optimal answer, not an element."

Features:

- Search space → range of possible answers `[low, high]`
- Define monotonic `check(mid)` → determines if `mid` is feasible
- Goal → find smallest/largest feasible value

Structural Binary Search

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**.
- Use `mid` to decide which half to keep

Practice

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?
>
> The index range of the sorted array.
>
> Initial range: `[left, right] = [0, len(nums) - 1]`.

> ### Define the Check Function
>
> Next, we define our check condition.
>
> Compute midpoint: `mid = (left + right) // 2`.
>
> 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 comparison removes half the search space.

> ### Define the Exit Condition
>
> Finally, when do we stop?

> - Continue while the range is valid: `while left <= right`.
> - Stop when `left > right` → means the range is empty.
> - If we haven't found the target → return `-1`.

Here's the full implementation:

```
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
```

Complexity

- Time: O(log n) — halves the range every loop
- Space: O(1) — constant memory

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

- 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 852 Peak Index in a Mountain Array