Explanation

The intuition behind this code is based on a clever observation about maintaining the **smallest possible tail elements** for increasing subsequences of each length.

Key Insight

Instead of finding the actual longest increasing subsequence, we maintain an array `sub` where:

- `sub[i]` = the smallest ending element of all increasing subsequences of length `i+1`

Why This Works

Example walkthrough with `nums = [10, 9, 2, 5, 3, 7, 101, 18]`:

1. **Start with `sub = []`**
2. **Process 10**: `sub = [10]`
   - Length 1 subsequences: `[10]` (smallest tail: 10)
3. **Process 9**: `sub = [9]`
   - 9 < 10, so replace 10 with 9
   - Length 1 subsequences: `[9]` (smaller tail is better!)
4. **Process 2**: `sub = [2]`
   - 2 < 9, so replace 9 with 2
   - Length 1 subsequences: `[2]` (even smaller tail)
5. **Process 5**: `sub = [2, 5]`
   - 5 > 2, so append
   - Length 1: `[2]`, Length 2: `[2,5]`

6. **Process 3**: `sub = [2, 3]`
    - 3 fits at position 1 (replacing 5)
    - Length 1: `[2]`, Length 2: `[2,3]` (better than `[2,5]`)
7. **Process 7**: `sub = [2, 3, 7]`
    - 7 > 3, so append
    - Length 1: `[2]`, Length 2: `[2,3]`, Length 3: `[2,3,7]`

The Magic

Why replace instead of keeping multiple subsequences?

When we see a number that could extend a subsequence OR replace an element in a longer subsequence, we choose to replace because:

- A smaller tail gives us **more flexibility** for future elements
- We only care about the **length**, not the actual subsequence
- If `[2,3,7]` and `[2,5,7]` both have length 3, keeping the one with smaller middle element (3) is always better

Binary Search Purpose

We use binary search to quickly find where each new number should go:

- If it's larger than all elements in `sub`: append (extend longest subsequence)
- Otherwise: find the leftmost element ≥ current number and replace it

The final length of `sub` equals the length of the longest increasing subsequence, even though `sub` itself might not be a valid subsequence from the original array.

This approach is brilliant because it maintains **optimal substructure** - at each step, we keep the best possible foundation for building longer subsequences.

---

mid or mid-1

The Key Insight: What Are We Actually Searching For?

In this specific case, we want to find the **leftmost position** where we can insert `target` to keep the array sorted. This is called "lower bound" or "left insertion point."

Why `right = mid` (not `mid - 1`)?

Think of it this way:

```
arr = [1, 3, 3, 3, 5, 7]
target = 3
```

When we find `arr[mid] == target` (or `arr[mid] >= target`), we **can't eliminate mid** because:

- Mid might be the **first occurrence** of target
- We're looking for the **leftmost** position

So we set `right = mid` to keep mid in our search space.

Visual Example

```
arr = [1, 3, 3, 3, 5, 7], target = 3
    0  1  2  3  4  5

Step 1: left=0, right=6, mid=3
arr[3] = 3, which is >= target
Since 3 could be our answer, keep it: right = mid = 3

Step 2: left=0, right=3, mid=1
arr[1] = 3, which is >= target
Since 3 could be our answer, keep it: right = mid = 1

Step 3: left=0, right=1, mid=0
arr[0] = 1, which is < target
We can eliminate 1: left = mid + 1 = 1

Step 4: left=1, right=1 → loop ends
Answer: position 1 (leftmost 3)
```

The Pattern Rules

For finding leftmost position (lower bound):

```
if arr[mid] < target:
    left = mid + 1    # mid is too small, eliminate it
else:
    right = mid       # mid might be our answer, keep it
```

For finding rightmost position (upper bound):

```
if arr[mid] <= target:
    left = mid + 1    # we want something after mid
else:
    right = mid       # mid might be our answer
```

Why Sometimes `mid - 1`?

You see `mid - 1` in **different binary search variants**:

```
# Classic "find exact target" search
if arr[mid] == target:
    return mid
elif arr[mid] < target:
    left = mid + 1
else:
    right = mid - 1   # We know mid is too big, so exclude it
```

Memory Trick

Think about what you're keeping vs. eliminating:

- `left = mid + 1` → "mid is definitely not the answer, eliminate it"
- `right = mid` → "mid might be the answer, keep it in search space"
- `right = mid - 1` → "mid is definitely not the answer, eliminate it"

The key is understanding **what question you're asking** and **whether mid could be your final answer**.