Binary Search

1. Binary Search

Pattern: Binary Search

Problem Statement

> Given an array of integers `nums` which is sorted in ascending order, and an integer `target`, write a function to search for `target` in `nums`. If `target` exists, return its index. Otherwise, return `-1`.

You must write an algorithm with `O(log n)` runtime complexity.

Sample Input & Output

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

```
Input: nums = [-1,0,3,5,9,12], target = 2
Output: -1
Explanation: 2 is not in nums.
```

```
Input: nums = [5], target = 5
Output: 0
Explanation: Single-element array contains the target.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def search(self, nums: List[int], target: int) -> int:
        # STEP 1: Initialize structures
        #   - Use two pointers to represent the current search range.
        left, right = 0, len(nums) - 1

        # STEP 2: Main loop / recursion
        #   - Maintain invariant: target is in [left, right] if present.
        #   - Loop while search space is valid (left <= right).
        while left <= right:
            # Avoid overflow (not needed in Python but good practice)
            mid = left + (right - left) // 2

            # STEP 3: Update state / bookkeeping
            #   - Compare mid value to target to narrow search space.
            if nums[mid] == target:
                return mid  # Found target
            elif nums[mid] < target:
                left = mid + 1  # Search right half
            else:
                right = mid - 1  # Search left half

        # STEP 4: Return result
        #   - If loop ends without return, target not found.
        return -1

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.search([-1, 0, 3, 5, 9, 12], 9) == 4

    #  Test 2: Edge case
    assert sol.search([5], 5) == 0
```

```
    #   Test 3: Tricky/negative
    assert sol.search([-1, 0, 3, 5, 9, 12], 2) == -1

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `search([-1, 0, 3, 5, 9, 12], 9)` step by step.

Initial state:
- `nums = [-1, 0, 3, 5, 9, 12]`
- `target = 9`
- `left = 0`, `right = 5` (indices)

Step 1: Enter `while left <= right` → `0 <= 5` → **true**
- Compute `mid = 0 + (5 - 0) // 2 = 2`
- `nums[2] = 3`
- Compare: `3 < 9` → so set `left = mid + 1 = 3`
- **State**: `left=3`, `right=5`

Step 2: Loop condition → `3 <= 5` → **true**
- `mid = 3 + (5 - 3) // 2 = 3 + 1 = 4`
- `nums[4] = 9`
- Compare: `9 == 9` → **return 4**

Final output: `4`

Now trace `search([-1, 0, 3, 5, 9, 12], 2)`:

Initial: `left=0`, `right=5`

- **Iter 1**: `mid=2`, `nums[2]=3` → `3 > 2` → `right = 1`

- **Iter 2**: `left=0`, `right=1` → `mid=0`, `nums[0]=-1` → `-1 < 2` → `left=1`
- **Iter 3**: `left=1`, `right=1` → `mid=1`, `nums[1]=0` → `0 < 2` → `left=2`
- Now `left=2`, `right=1` → loop condition `2 <= 1` → **false**
- Exit loop → **return -1**

Complexity Analysis

- **Time Complexity**: `O(log n)`

  Each iteration cuts the search space in half. After `k` steps, size = `n / 2^k`. When `n / 2^k = 1` → `k = log n`. So at most `log n` iterations.

- **Space Complexity**: `O(1)`

  Only uses a constant amount of extra space (`left`, `right`, `mid`).
  No recursion or dynamic structures that grow with input.

2. First Bad Version

Pattern: Binary Search

Problem Statement

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have `n` versions `[1, 2, ..., n]` and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API `bool isBadVersion(version)` which returns whether `version` is bad. Implement a function to find the **first bad version**. You should minimize the number of calls to the API.

Sample Input & Output

```
Input: n = 5, bad = 4
Output: 4
Explanation: call isBadVersion(3) → false
             call isBadVersion(5) → true
             call isBadVersion(4) → true
             → first bad version is 4.
```

```
Input: n = 1, bad = 1
Output: 1
Explanation: Only one version; it's bad.
```

```
Input: n = 10, bad = 2
Output: 2
Explanation: First bad version is early; binary search should find it quickly.
```

LeetCode Editorial Solution + Inline Tests

```python
# The isBadVersion API is already defined for you.
# def isBadVersion(version: int) -> bool:

class Solution:
    def firstBadVersion(self, n: int) -> int:
        # STEP 1: Initialize pointers for binary search
        #   - left = 1 (first version)
        #   - right = n (last version)
        left, right = 1, n

        # STEP 2: Main loop - narrow search space
        #   - Invariant: first bad version is in [left, right]
        #   - Stop when left == right (converged to answer)
        while left < right:
            mid = left + (right - left) // 2

            # STEP 3: Check if mid is bad
            #   - If yes, first bad is at mid or earlier → move right
```

```
            #   - If no, first bad is after mid → move left
            if isBadVersion(mid):
                right = mid
            else:
                left = mid + 1

        # STEP 4: Return result
        #   - left == right == first bad version
        return left

# ------------------- INLINE TESTS -------------------
# Mock isBadVersion for testing
def isBadVersion(version: int) -> bool:
    # This will be overridden per test case
    pass

if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    bad = 4
    isBadVersion = lambda v: v >= bad
    assert sol.firstBadVersion(5) == 4
    print("  Test 1 passed: n=5, bad=4 → output=4")

    #  Test 2: Edge case (only one version)
    bad = 1
    isBadVersion = lambda v: v >= bad
    assert sol.firstBadVersion(1) == 1
    print("  Test 2 passed: n=1, bad=1 → output=1")

    #  Test 3: Tricky case (early bad version)
    bad = 2
    isBadVersion = lambda v: v >= bad
    assert sol.firstBadVersion(10) == 2
    print("  Test 3 passed: n=10, bad=2 → output=2")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 1**: `n = 5`, `bad = 4` → `isBadVersion(v) = (v >= 4)`

Initial state:
- `left = 1`, `right = 5`

Iteration 1:
- `left (1) < right (5)` → enter loop
- `mid = 1 + (5 - 1) // 2 = 1 + 2 = 3`
- `isBadVersion(3)` → `False`
- So, bad version is **after** 3 → `left = mid + 1 = 4`
- **State**: `left = 4`, `right = 5`

Iteration 2:
- `4 < 5` → continue
- `mid = 4 + (5 - 4) // 2 = 4 + 0 = 4`
- `isBadVersion(4)` → `True`
- So, first bad is **at or before** 4 → `right = 4`
- **State**: `left = 4`, `right = 4`

Loop ends (`left == right`)
- Return `left = 4`

Output: `4` — correct!

Key insight:
We never check versions unnecessarily. Only **3 API calls** (mid=3,4) — optimal!

Complexity Analysis

- **Time Complexity**: `O(log n)`

  Binary search halves the search space each iteration → at most `log (n)` calls to `isBadVersion`.

- **Space Complexity**: `O(1)`

  Only two integer pointers (`left`, `right`) used — constant extra space.

3. Sqrt(x)

Pattern: Binary Search

Problem Statement

Given a non-negative integer `x`, return the square root of `x` rounded down to the nearest integer. The returned integer should be **non-negative** as well.

You must not use any built-in exponent function or operator.

- For example, do not use `pow(x, 0.5)` in C++ or `x ** 0.5` in Python.

Sample Input & Output

```
Input: x = 4
Output: 2
Explanation: The square root of 4 is exactly 2.
```

```
Input: x = 8
Output: 2
Explanation: The square root of 8 is ~2.828, which floors to 2.
```

```
Input: x = 0
Output: 0
Explanation: Edge case - square root of 0 is 0.
```

LeetCode Editorial Solution + Inline Tests

```python
class Solution:
    def mySqrt(self, x: int) -> int:
        # STEP 1: Initialize structures
        #   - Use binary search bounds: left = 0, right = x
        #   - For x=0 or 1, answer is x itself
        if x < 2:
            return x

        left, right = 1, x // 2  # sqrt(x)   x/2 for x   2

        # STEP 2: Main loop / recursion
        #   - Maintain invariant: answer is in [left, right]
        #   - Stop when left > right → right holds floor(sqrt(x))
        while left <= right:
            mid = left + (right - left) // 2
            sq = mid * mid

            # STEP 3: Update state / bookkeeping
            #   - If sq == x, we found exact root
            #   - If sq < x, root is   mid → move left
            #   - If sq > x, root is < mid → move right
            if sq == x:
                return mid
            elif sq < x:
                left = mid + 1
            else:
                right = mid - 1

        # STEP 4: Return result
        #   - Loop ends with right = floor(sqrt(x))
        return right

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.mySqrt(4) == 2, f"Expected 2, got {sol.mySqrt(4)}"

    #  Test 2: Edge case
    assert sol.mySqrt(0) == 0, f"Expected 0, got {sol.mySqrt(0)}"
    assert sol.mySqrt(1) == 1, f"Expected 1, got {sol.mySqrt(1)}"
```

```
    #   Test 3: Tricky/negative
    assert sol.mySqrt(8) == 2, f"Expected 2, got {sol.mySqrt(8)}"
    assert sol.mySqrt(2147395600) == 46340, \
        f"Expected 46340, got {sol.mySqrt(2147395600)}"

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `mySqrt(8)` step by step.

Initial state:
- `x = 8`
- Since `x >= 2`, skip early return.
- Set `left = 1`, `right = 8 // 2 = 4`.

Iteration 1:
- `left = 1`, `right = 4` → `left <= right` → enter loop.
- `mid = 1 + (4 - 1) // 2 = 1 + 1 = 2`
- `sq = 2 * 2 = 4`
- `4 < 8` → move `left = mid + 1 = 3`
- **State**: `left=3`, `right=4`

Iteration 2:
- `left = 3`, `right = 4` → enter loop.
- `mid = 3 + (4 - 3) // 2 = 3 + 0 = 3`
- `sq = 3 * 3 = 9`
- `9 > 8` → move `right = mid - 1 = 2`
- **State**: `left=3`, `right=2`

Loop condition check:
- `left (3) > right (2)` → exit loop.

Return: `right = 2` → correct floor of $\sqrt{8}$.

Complexity Analysis

- **Time Complexity**: `O(log x)`

  Binary search halves the search space each iteration. Max iterations log (x).

- **Space Complexity**: `O(1)`

  Only a few integer variables used — no recursion or dynamic structures.

4. Search Insert Position

Pattern: Binary Search

Problem Statement

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.
You must write an algorithm with `O(log n)` runtime complexity.

Sample Input & Output

```
Input: nums = [1,3,5,6], target = 5
Output: 2
Explanation: Target 5 is found at index 2.
```

```
Input: nums = [1,3,5,6], target = 2
Output: 1
Explanation: 2 would be inserted between 1 and 3 → index 1.
```

```
Input: nums = [1,3,5,6], target = 0
Output: 0
Explanation: 0 is smaller than all → insert at beginning.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def searchInsert(self, nums: List[int], target: int) -> int:
        # STEP 1: Initialize pointers for binary search
        #   - left = start of search space
        #   - right = end of search space (exclusive in this variant)
        left, right = 0, len(nums)

        # STEP 2: Main loop - narrow search space until empty
        #   - Invariant: target must be in [left, right)
        #   - Loop ends when left == right → insertion point
        while left < right:
            mid = (left + right) // 2

            # STEP 3: Compare mid value with target
            #   - If mid value >= target, target belongs in left half
            #   - Else, target belongs in right half
            if nums[mid] >= target:
                right = mid      # Keep mid as candidate
            else:
                left = mid + 1   # Exclude mid

        # STEP 4: Return result
        #   - left == right is the smallest index where
        #     nums[index] >= target → correct insert position
        return left

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case - target exists
    assert sol.searchInsert([1,3,5,6], 5) == 2

    #  Test 2: Edge case - insert at beginning
    assert sol.searchInsert([1,3,5,6], 0) == 0

    #  Test 3: Tricky/negative - insert at end
    assert sol.searchInsert([1,3,5,6], 7) == 4
```

```
    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `searchInsert([1,3,5,6], 2)` step by step:

1. **Initialize**:
   `left = 0`, `right = 4` (length of `nums`)
   → Search space: indices `[0, 4)` → entire array.
2. **First loop iteration** (`left=0`, `right=4`):
   - `mid = (0 + 4) // 2 = 2`
   - `nums[2] = 5`
   - Since `5 >= 2` → move `right = mid = 2`
     → Now search space: `[0, 2)`
3. **Second loop iteration** (`left=0`, `right=2`):
   - `mid = (0 + 2) // 2 = 1`
   - `nums[1] = 3`
   - Since `3 >= 2` → move `right = 1`
     → Now search space: `[0, 1)`
4. **Third loop iteration** (`left=0`, `right=1`):
   - `mid = (0 + 1) // 2 = 0`
   - `nums[0] = 1`
   - Since `1 < 2` → move `left = mid + 1 = 1`
5. **Loop ends**: `left = 1`, `right = 1` → exit
   → Return `left = 1`

Final output: `1` — correct insertion index for `2`.

Key insight: The loop maintains the invariant that the answer is always in `[left, right)`. When the interval collapses, `left` is the first position where `nums[i] >= target`.

Complexity Analysis

- **Time Complexity**: `O(log n)`

  Binary search halves the search space each iteration → at most `log (n)` steps.

- **Space Complexity**: `O(1)`

  Only uses two integer pointers (`left`, `right`) — constant extra space.

5. Find First and Last Position of Element in Sorted Array

Pattern: Binary Search (Two-Pass Variant)

Problem Statement

Given an array of integers `nums` sorted in non-decreasing order, find the starting and ending position of a given `target` value.
If `target` is not found in the array, return `[-1, -1]`.
You must write an algorithm with `O(log n)` runtime complexity.

Sample Input & Output

```
Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]
Explanation: The target 8 appears from index 3 to 4.
```

```
Input: nums = [5,7,7,8,8,10], target = 6
Output: [-1,-1]
Explanation: Target 6 is not present.
```

```
Input: nums = [], target = 0
Output: [-1,-1]
Explanation: Empty array edge case.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        # STEP 1: Initialize structures
        #   - Use helper to find leftmost and rightmost indices
        #   - Avoid linear scan to maintain O(log n)

        def find_first():
            left, right = 0, len(nums) - 1
            first_pos = -1
            while left <= right:
                mid = (left + right) // 2
                if nums[mid] == target:
                    first_pos = mid      # Record candidate
                    right = mid - 1      # Search left for earlier
                elif nums[mid] < target:
                    left = mid + 1
                else:
                    right = mid - 1
            return first_pos

        def find_last():
            left, right = 0, len(nums) - 1
            last_pos = -1
            while left <= right:
                mid = (left + right) // 2
```

```
                if nums[mid] == target:
                    last_pos = mid       # Record candidate
                    left = mid + 1       # Search right for later
                elif nums[mid] < target:
                    left = mid + 1
                else:
                    right = mid - 1
            return last_pos

        # STEP 2: Main logic
        #   - Run two binary searches: one for first, one for last
        #   - Both are O(log n), total remains O(log n)

        # STEP 3: Return result
        #   - If either is -1, target absent → return [-1, -1]
        first = find_first()
        if first == -1:
            return [-1, -1]
        last = find_last()
        return [first, last]

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.searchRange([5,7,7,8,8,10], 8) == [3, 4]

    #  Test 2: Edge case - target not present
    assert sol.searchRange([5,7,7,8,8,10], 6) == [-1, -1]

    #  Test 3: Tricky/negative - empty array
    assert sol.searchRange([], 0) == [-1, -1]

    print(" All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `searchRange([5,7,7,8,8,10], 8)` step by step.

Initial state:
- `nums = [5,7,7,8,8,10]`, `target = 8`
- Call `find_first()`:

Inside `find_first()`: - `left=0`, `right=5`, `first_pos=-1` - **Iteration 1**:
- `mid = (0+5)//2 = 2` → `nums[2] = 7`
- $7 < 8$ → `left = 3` - **Iteration 2**:
- `mid = (3+5)//2 = 4` → `nums[4] = 8` → match!
- Set `first_pos = 4`, then `right = 3` (search left half) - **Iteration 3**:
- `left=3`, `right=3` → `mid=3` → `nums[3]=8` → match!
- Set `first_pos = 3`, `right = 2` - Now `left=3 > right=2` → loop ends → return `3`

Now call `find_last()`: - `left=0`, `right=5`, `last_pos=-1` - **Iteration 1**:
- `mid=2` → `7 < 8` → `left=3` - **Iteration 2**:
- `mid=4` → `8 == 8` → `last_pos=4`, `left=5` - **Iteration 3**:
- `left=5`, `right=5` → `mid=5` → `nums[5]=10 > 8` → `right=4` - Now `left=5 > right=4` → loop ends → return `4`

Final return: `[3, 4]`

Key insight: Two binary searches—first shrinks right on match, last shrinks left on match.

Complexity Analysis

- **Time Complexity**: `O(log n)`

  Two binary searches, each O(log n). Constants ignored → still O(log n).

- **Space Complexity**: `O(1)`

  Only a few integer variables used. No recursion stack (iterative). Input size doesn't affect extra space.

6. Single Element in a Sorted Array

Pattern: Binary Search on Answer / Search in Sorted Array

Problem Statement

You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once.

Return the single element that appears only once.

Your solution must run in `O(log n)` time and `O(1)` space.

Sample Input & Output

```
Input: [1,1,2,3,3,4,4,8,8]
Output: 2
Explanation: 2 appears only once; all others appear twice.
```

```
Input: [3,3,7,7,10,11,11]
Output: 10
Explanation: 10 is the lone element in the second half.
```

```
Input: [1]
Output: 1
Explanation: Edge case - single-element array.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def singleNonDuplicate(self, nums: List[int]) -> int:
        # STEP 1: Initialize binary search bounds
        #   - We search only on even indices to maintain pair alignment
        left, right = 0, len(nums) - 1

        # STEP 2: Main loop - binary search on even indices
```

```
        #   - Invariant: the single element is always at an even index
        #   - Before the single element: pairs are (even, odd)
        #   - After the single element: pairs shift to (odd, even)
        while left < right:
            # Ensure mid is even to keep pair structure intact
            mid = (left + right) // 2
            if mid % 2 == 1:
                mid -= 1  # force mid to be even

            # STEP 3: Compare nums[mid] with its pair (mid+1)
            #   - If equal, single element is to the right
            #   - If not equal, single element is at or left of mid
            if nums[mid] == nums[mid + 1]:
                left = mid + 2
            else:
                right = mid

        # STEP 4: Return result
        #   - left == right points to the single element
        return nums[left]

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.singleNonDuplicate([1,1,2,3,3,4,4,8,8]) == 2

    #  Test 2: Edge case - single element
    assert sol.singleNonDuplicate([1]) == 1

    #  Test 3: Tricky/negative - single at end
    assert sol.singleNonDuplicate([3,3,7,7,10,11,11]) == 10

    print(" All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `nums = [1,1,2,3,3,4,4,8,8]` step by step:

1. **Initialization**:
   - `left = 0`, `right = 8` (length 9 → last index 8)
   - Array indices: `[0:1, 1:1, 2:2, 3:3, 4:3, 5:4, 6:4, 7:8, 8:8]`
2. **First loop iteration**:
   - `mid = (0 + 8) // 2 = 4` → odd → adjust to `mid = 3`? Wait—no! Actually: `mid = 4` → `4 % 2 == 0` → **no adjustment** (4 is even).
   - Compare `nums[4]` and `nums[5]`: `3 vs 4` → **not equal**
   - So single element is **at or left of mid=4** → `right = 4`
3. **State**: `left=0`, `right=4`
4. **Second iteration**:
   - `mid = (0 + 4) // 2 = 2` → even → keep `mid=2`
   - Compare `nums[2]` and `nums[3]`: `2 vs 3` → **not equal**
   - So `right = 2`
5. **State**: `left=0`, `right=2`
6. **Third iteration**:
   - `mid = (0 + 2) // 2 = 1` → odd → adjust to `mid = 0`
   - Compare `nums[0]` and `nums[1]`: `1 vs 1` → **equal**
   - So single element is **right of this pair** → `left = 0 + 2 = 2`
7. **Now**: `left = 2`, `right = 2` → loop ends
8. **Return** `nums[2] = 2`

Key Insight:
By forcing `mid` to be even, we always land at the **first element of a pair**.
- If the pair is intact (`nums[mid] == nums[mid+1]`), the disruption (single element) is **after**.
- If broken, the disruption is **at or before**.

Complexity Analysis

- **Time Complexity**: `O(log n)`

  Binary search halves the search space each time. Only `O(log n)` iterations.

- **Space Complexity**: `O(1)`

  Only a few integer variables (`left`, `right`, `mid`) — no recursion or extra arrays.

7. Find Peak Element

Pattern: Binary Search

Problem Statement

A peak element is an element that is strictly greater than its neighbors.

Given a **0-indexed** integer array `nums`, find a peak element, and return its index. If the array contains multiple peaks, return the index to **any** of the peaks.

You may imagine that `nums[-1] = nums[n] = -∞`. In other words, an element is always considered greater than a non-existent neighbor.

Constraints:
- `1 <= nums.length <= 1000`
- -2^{31} `<= nums[i] <=` 2^{31} `- 1`
- `nums[i] != nums[i + 1]` for all valid `i`.

Sample Input & Output

```
Input: nums = [1, 2, 3, 1]
Output: 2
Explanation: 3 is a peak element (3 > 2 and 3 > 1), at index 2.
```

```
Input: nums = [1, 2, 1, 3, 5, 6, 4]
Output: 5  (or 1 - both 2 and 6 are peaks)
Explanation: 6 is a peak (6 > 5 and 6 > 4) at index 5.
```

```
Input: nums = [1]
Output: 0
Explanation: Single element is always a peak due to -ϖ neighbors.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def findPeakElement(self, nums: List[int]) -> int:
        # STEP 1: Initialize binary search bounds
        #   - We search in [left, right] inclusive
        left, right = 0, len(nums) - 1

        # STEP 2: Main loop - maintain invariant:
        #   - At least one peak exists in [left, right]
        #   - Because boundaries are -ϖ, a local max must exist
        while left < right:
            mid = (left + right) // 2

            # STEP 3: Compare mid with its right neighbor
            #   - If nums[mid] < nums[mid + 1], peak must be to the right
            #     (since values rise toward mid+1 and eventually fall to -ϖ)
            if nums[mid] < nums[mid + 1]:
                left = mid + 1
            else:
```

```
                # nums[mid] > nums[mid + 1] (no equal per constraints)
                # Peak is at mid or to the left
                right = mid

        # STEP 4: Return result
        #   - When left == right, we've converged to a peak
        return left

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.findPeakElement([1, 2, 3, 1]) == 2

    #  Test 2: Edge case - single element
    assert sol.findPeakElement([1]) == 0

    #  Test 3: Tricky/negative - descending then ascending
    # Multiple valid answers; we accept any peak index
    result = sol.findPeakElement([1, 2, 1, 3, 5, 6, 4])
    # Valid peaks at index 1 (value 2) or 5 (value 6)
    assert result in {1, 5}

    print(" All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `findPeakElement([1, 2, 3, 1])` step by step:

1. **Initialize**:
   - `left = 0`, `right = 3` (since `len(nums) = 4`)
   - Array: `[1, 2, 3, 1]`
2. **First loop iteration** (`left=0`, `right=3`):

- `mid = (0 + 3) // 2 = 1`
- Compare `nums[1] = 2` vs `nums[2] = 3` → `2 < 3` → go right
- Update: `left = mid + 1 = 2`

3. **Second loop iteration** (`left=2`, `right=3`):
   - `mid = (2 + 3) // 2 = 2`
   - Compare `nums[2] = 3` vs `nums[3] = 1` → `3 > 1` → go left
   - Update: `right = mid = 2`
4. **Loop ends**: `left == right == 2` → return `2`

Final output: `2` — correct peak index.

Why it works:
- We never need to check both neighbors.
- By comparing `mid` and `mid+1`, we decide which half **must** contain a peak due to the "cliff" at the end (`-ϖ`).
- This is a classic **binary search on condition**, not on value.

Complexity Analysis

- **Time Complexity**: `O(log n)`

  We halve the search space each iteration — standard binary search behavior.

- **Space Complexity**: `O(1)`

  Only a few integer variables (`left`, `right`, `mid`) — no recursion or extra arrays.

8. Peak Index in a Mountain Array

Pattern: Binary Search

Problem Statement

Let's call an array `arr` a **mountain** if the following properties hold:
- `arr.length >= 3`
- There exists some `i` with `0 < i < arr.length - 1` such that:
- `arr[0] < arr[1] < ... < arr[i - 1] < arr[i]`
- `arr[i] > arr[i + 1] > ... > arr[arr.length - 1]`

Given a mountain array `arr`, return the index `i` such that `arr[i]` is the peak of the mountain.
You must solve it in `O(log(arr.length))` time.

Sample Input & Output

```
Input: [0,1,0]
Output: 1
Explanation: Peak is at index 1 (value = 1).
```

```
Input: [0,2,1,0]
Output: 1
Explanation: Values increase to index 1 (2), then decrease.
```

```
Input: [0,10,5,2]
Output: 1
Explanation: Strictly increasing until index 1, then decreasing.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def peakIndexInMountainArray(self, arr: List[int]) -> int:
        # STEP 1: Initialize binary search bounds
        #   - left starts at 1 (peak can't be first)
```

```
        #   - right ends at len(arr)-2 (peak can't be last)
        left, right = 1, len(arr) - 2

        # STEP 2: Main loop - binary search for peak
        #   - Invariant: peak always lies in [left, right]
        #   - Compare mid with its right neighbor to decide slope
        while left <= right:
            mid = (left + right) // 2

            # If arr[mid] < arr[mid+1], we're on ascending slope
            if arr[mid] < arr[mid + 1]:
                left = mid + 1  # Peak must be to the right
            else:
                # arr[mid] > arr[mid+1] → descending slope
                # Peak is at mid or to the left
                right = mid - 1

        # STEP 3: Return result
        #   - Loop ends when left > right; 'left' points to peak
        return left

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.peakIndexInMountainArray([0, 1, 0]) == 1

    #  Test 2: Edge case - minimal mountain
    assert sol.peakIndexInMountainArray([0, 2, 1, 0]) == 1

    #  Test 3: Tricky/negative - steep drop after peak
    assert sol.peakIndexInMountainArray([0, 10, 5, 2]) == 1

    print(" All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `arr = [0, 10, 5, 2]` step by step:

1. **Initialization**:
   - `left = 1`, `right = 2` (since `len(arr) = 4`, so `4 - 2 = 2`)
   - Valid peak indices: only 1 or 2 (but 2 can't be peak because it's not followed by decrease? Actually, index 2 has value 5, next is 2 → it *could* be peak, but in this array, $10 > 5$, so peak is at 1).
2. **First loop iteration**:
   - `mid = (1 + 2) // 2 = 1`
   - Check `arr[1] < arr[2]` → `10 < 5`? **False**
   - So enter `else`: `right = mid - 1 = 0`
3. **Loop condition check**:
   - Now `left = 1`, `right = 0` → `left > right` → exit loop
4. **Return `left = 1`** → correct peak index.

State progression:
- Start: `left=1, right=2`
- After mid=1: since descending, move `right=0`
- Loop ends → return `left=1`

This works because whenever we see a descending slope (`arr[mid] > arr[mid+1]`), the peak is *at or before* `mid`. When ascending, peak is *after* `mid`. Binary search narrows down to the exact turning point.

Complexity Analysis

- **Time Complexity**: `O(log n)`

  We halve the search space each iteration using binary search. Only one pass over `log(n)` elements.

- **Space Complexity**: `O(1)`

  Only a few integer variables (`left`, `right`, `mid`) are used — no extra arrays or recursion stack.

9. Capacity to Ship Packages Within D Days

Pattern: Binary Search on Answer

Problem Statement

A conveyor belt has packages that must be shipped from one port to another within `days` days.
The `i`th package on the conveyor belt has a weight of `weights[i]`. Each day, we load the ship with packages in the order given by `weights`. We may not load more weight than the maximum weight capacity of the ship.
Return the **least weight capacity** of the ship that will result in all the packages being shipped within `days` days.

Sample Input & Output

```
Input: weights = [1,2,3,4,5,6,7,8,9,10], days = 5
Output: 15
Explanation: A ship capacity of 15 is the minimum to ship all packages in 5 days:
[1,2,3,4,5], [6,7], [8], [9], [10]
```

```
Input: weights = [3,2,2,4,1,4], days = 3
Output: 6
Explanation: [3,2], [2,4], [1,4]
```

```
Input: weights = [1,2,3,1,1], days = 4
Output: 3
Explanation: [1,2], [3], [1], [1] - capacity must be at least max(weights) = 3
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def shipWithinDays(self, weights: List[int], days: int) -> int:
        # STEP 1: Initialize search bounds
        #   - Lower bound: max(weights) → ship must carry heaviest pkg
        #   - Upper bound: sum(weights) → ship carries all in 1 day
        left = max(weights)
        right = sum(weights)

        # STEP 2: Binary search on capacity
        #   - For mid capacity, simulate shipping
        #   - If feasible in <= days, try smaller capacity (right = mid)
        #   - Else, need larger capacity (left = mid + 1)
        while left < right:
            mid = (left + right) // 2
            if self._can_ship(weights, days, mid):
                right = mid
            else:
                left = mid + 1

        # STEP 3: Return minimal feasible capacity
        #   - Loop ends when left == right → minimal valid answer
        return left

    def _can_ship(self, weights: List[int], days: int, capacity: int) -> bool:
        # Simulate shipping with given capacity
        current_load = 0
        days_needed = 1  # at least one day

        for weight in weights:
            # If adding this pkg exceeds capacity, start new day
            if current_load + weight > capacity:
                days_needed += 1
                current_load = weight
                # Early exit if already over days
                if days_needed > days:
                    return False
            else:
                current_load += weight

        return True
```

```python
# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.shipWithinDays([1,2,3,4,5,6,7,8,9,10], 5) == 15

    #  Test 2: Edge case - days = len(weights)
    #   → capacity must be max(weights)
    assert sol.shipWithinDays([1,2,3,1,1], 4) == 3

    #  Test 3: Tricky/negative - high variance weights
    assert sol.shipWithinDays([3,2,2,4,1,4], 3) == 6

    print(" All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `weights = [1,2,3,4,5,6,7,8,9,10]`, `days = 5`.

Step 1: Set binary search bounds
- `left = max(weights) = 10`
- `right = sum(weights) = 55`

Step 2: First iteration — mid = (10 + 55) // 2 = 32
- Simulate shipping with capacity 32:
- Day 1: 1+2+3+4+5+6+7 = 28 → add 8? 28+8=36 > 32 → new day
- Day 2: 8+9 = 17 → add 10? 17+10=27 32 → done
- Total days = 3 5 → feasible → set `right = 32`

Step 3: Next mid = (10 + 32) // 2 = 21
- Simulate:
- Day1: 1+2+3+4+5+6=21 → next day
- Day2: 7+8=15 → +9=24 >21 → Day3: 9+10=19
- Total days = 3 → feasible → `right = 21`

Step 4: mid = (10 + 21) // 2 = 15
- Simulate:

- Day1: 1+2+3+4+5=15
- Day2: 6+7=13 → +8=21 >15 → Day3: 8
- Day4: 9
- Day5: 10
- Days = 5 → feasible → `right = 15`

Step 5: mid = (10 + 15) // 2 = 12
- Simulate:
- Day1: 1+2+3+4=10 → +5=15 >12 → Day2: 5+6=11 → +7=18 >12 → Day3:7+8=15>12 → Day4:8 → Day5:9 → Day6:10
- Days = 6 > 5 → not feasible → `left = 13`

Continue… until `left = right = 15`.

Final answer: **15**

Key insight: We **binary search on the answer space** (capacity), not the input array. The feasibility check is greedy and linear.

Complexity Analysis

- **Time Complexity**: `O(n log(sum(weights)))`

  Binary search runs in `O(log(sum))` iterations. Each feasibility check scans all `n` weights → `O(n)` per iteration.

- **Space Complexity**: `O(1)`

  Only a few integer variables used; no extra space proportional to input.

10. Split Array Largest Sum

Pattern: Binary Search on Answer + Greedy Validation

Problem Statement

> Given an integer array `nums` and an integer `k`, split `nums` into `k` non-empty contiguous subarrays such that the **largest sum among these subarrays is minimized**. Return the minimized largest sum.

You must split the array into **exactly** `k` contiguous subarrays.

Sample Input & Output

```
Input: nums = [7,2,5,10,8], k = 2
Output: 18
Explanation: Split as [7,2,5] and [10,8]. Sums = 14 and 18 → max = 18 (minimal possible).
```

```
Input: nums = [1,2,3,4,5], k = 2
Output: 9
Explanation: [1,2,3,4] + [5] → sums = 10 and 5 → max = 10
Better: [1,2,3] + [4,5] → 6 and 9 → max = 9
```

```
Input: nums = [1], k = 1
Output: 1
Explanation: Only one subarray possible.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def splitArray(self, nums: List[int], k: int) -> int:
        # STEP 1: Initialize binary search bounds
        #   - Lower bound: max(nums) → at least one subarray must hold the max
        #   - Upper bound: sum(nums) → one subarray holds everything
        left = max(nums)
        right = sum(nums)
```

```
        # STEP 2: Binary search on the answer (minimized max sum)
        #   - For a candidate "mid", check if we can split into <= k subarrays
        #     where each subarray sum <= mid
        while left < right:
            mid = (left + right) // 2

            # Validate if "mid" is feasible
            subarrays_needed = 1
            current_sum = 0
            for num in nums:
                # If adding this num exceeds mid, start new subarray
                if current_sum + num > mid:
                    subarrays_needed += 1
                    current_sum = num
                else:
                    current_sum += num

            # STEP 3: Adjust search space based on feasibility
            #   - If we need more than k subarrays, mid is too small
            if subarrays_needed > k:
                left = mid + 1
            else:
                right = mid  # mid is feasible; try smaller

        # STEP 4: Return result
        #   - left == right is the minimal feasible max sum
        return left

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.splitArray([7,2,5,10,8], 2) == 18, "Test 1 failed"

    #  Test 2: Edge case - single element
    assert sol.splitArray([1], 1) == 1, "Test 2 failed"

    #  Test 3: Tricky/negative - k equals array length
    assert sol.splitArray([1,2,3,4,5], 5) == 5, "Test 3 failed"

    print(" All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `splitArray([7,2,5,10,8], k=2)` step by step.

Initial Setup:
- `nums = [7,2,5,10,8]`, `k = 2`
- `left = max(nums) = 10`
- `right = sum(nums) = 32`

Binary Search Iteration 1:
- `mid = (10 + 32) // 2 = 21`
- Validate `mid = 21`:
- Start: `subarrays_needed = 1`, `current_sum = 0`
- 7: `0+7  21` → `current_sum = 7`
- 2: `7+2=9  21` → `current_sum = 9`
- 5: `9+5=14  21` → `current_sum = 14`
- 10: `14+10=24 > 21` → new subarray!
→ `subarrays_needed = 2`, `current_sum = 10`
- 8: `10+8=18  21` → `current_sum = 18`
- Total subarrays = 2 k=2 → **feasible**
- Update: `right = 21`

Iteration 2:
- `mid = (10 + 21) // 2 = 15`
- Validate `mid = 15`:
- `7 → 7`
- `7+2=9`
- `9+5=14`
- `14+10=24 > 15` → new subarray (count=2), `current_sum=10`
- `10+8=18 > 15` → new subarray (count=3)
- Total = 3 > k=2 → **not feasible**
- Update: `left = 16`

Iteration 3:
- `left=16`, `right=21` → `mid = 18`
- Validate `mid=18`:
- `7→7`, `+2→9`, `+5→14`

- 14+10=24 > 18 → new subarray (count=2), current_sum=10
- 10+8=18 18 → ok
- Total = 2 2 → **feasible**
- Update: right = 18

Iteration 4:
- left=16, right=18 → mid=17
- Validate mid=17:
- 7,2,5 → 14
- 14+10=24 >17 → new subarray (count=2)
- 10+8=18 >17 → new subarray (count=3) → **not feasible**
- Update: left = 18

Now left == right == 18 → return **18**.

Final output: 18

Complexity Analysis

- **Time Complexity**: O(n log S)
  > Where n = len(nums) and S = sum(nums) - max(nums).
  > Binary search runs in O(log S), and each validation scans the array in O(n).
- **Space Complexity**: O(1)
  > Only a few integer variables used; no extra space proportional to input.

11. Koko Eating Bananas

Pattern: Binary Search on Answer

Problem Statement

Koko loves to eat bananas. There are n piles of bananas, the i-th pile has piles[i] bananas. The guards have gone and will come back in h hours.

Koko can decide her bananas-per-hour eating speed of k. Each hour, she chooses some pile of bananas and eats k bananas from that pile. If the pile has fewer than k bananas, she eats all of them instead and will not eat any more bananas during this hour.

Koko likes to eat slowly but still wants to finish eating all the bananas before the guards return.

Return the **minimum integer** `k` such that she can eat all the bananas within `h` hours.

Sample Input & Output

```
Input: piles = [3,6,7,11], h = 8
Output: 4
Explanation: At k=4, hours needed = ceil(3/4)+ceil(6/4)+ceil(7/4)+ceil(11/4) = 1+2+2+3 = 8.
```

```
Input: piles = [30,11,23,4,20], h = 5
Output: 30
Explanation: Must finish each pile in 1 hour → k   max(piles) = 30.
```

```
Input: piles = [1], h = 1
Output: 1
Explanation: Only one banana, one hour → k=1 is minimal.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
import math

class Solution:
    def minEatingSpeed(self, piles: List[int], h: int) -> int:
        # STEP 1: Initialize binary search bounds
        #   - Lower bound: 1 (must eat at least 1 banana/hour)
        #   - Upper bound: max(piles) (can finish any pile in 1 hour)
        left, right = 1, max(piles)

        # STEP 2: Binary search on possible k values
        #   - Invariant: answer is in [left, right]
```

```
        #   - For each mid = k, compute total hours needed
        while left < right:
            k = (left + right) // 2

            # Compute total hours required at speed k
            total_hours = 0
            for pile in piles:
                total_hours += math.ceil(pile / k)

            # STEP 3: Update search space
            #   - If total_hours <= h, k is feasible → try smaller k
            #   - Else, k too small → need larger k
            if total_hours <= h:
                right = k
            else:
                left = k + 1

        # STEP 4: Return result
        #   - left == right is minimal feasible k
        return left

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.minEatingSpeed([3,6,7,11], 8) == 4

    #  Test 2: Edge case - h equals number of piles
    assert sol.minEatingSpeed([30,11,23,4,20], 5) == 30

    #  Test 3: Tricky/negative - single pile, minimal h
    assert sol.minEatingSpeed([1], 1) == 1

    print(" All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `minEatingSpeed([3,6,7,11], 8)` step by step:

1. **Initialization**:
   - `piles = [3,6,7,11]`, `h = 8`
   - `left = 1`, `right = max(piles) = 11`
2. **First loop iteration** (`left=1`, `right=11`):
   - `k = (1 + 11) // 2 = 6`
   - Compute `total_hours`:
     - `ceil(3/6) = 1`
     - `ceil(6/6) = 1`
     - `ceil(7/6) = 2`
     - `ceil(11/6) = 2`
     - Total = `1+1+2+2 = 6` $\leq$ `8` $\rightarrow$ feasible
   - Update: `right = 6`
3. **Second iteration** (`left=1`, `right=6`):
   - `k = (1 + 6) // 2 = 3`
   - `total_hours`:
     - `ceil(3/3)=1`, `ceil(6/3)=2`, `ceil(7/3)=3`, `ceil(11/3)=4`
     - Total = `1+2+3+4 = 10 > 8` $\rightarrow$ not feasible
   - Update: `left = 3 + 1 = 4`
4. **Third iteration** (`left=4`, `right=6`):
   - `k = (4 + 6) // 2 = 5`
   - `total_hours`:
     - `ceil(3/5)=1`, `ceil(6/5)=2`, `ceil(7/5)=2`, `ceil(11/5)=3`

- Total = 8 8 → feasible

- Update: `right = 5`

5. **Fourth iteration** (`left=4`, `right=5`):
   - `k = (4 + 5) // 2 = 4`
   - `total_hours`:
     - `ceil(3/4)=1`, `ceil(6/4)=2`, `ceil(7/4)=2`, `ceil(11/4)=3`
     - Total = 8 8 → feasible
   - Update: `right = 4`
6. **Loop ends** (`left == right == 4`) → return 4.

Final Output: 4
Key Insight: We're not searching in the array—we're searching in the **space of possible answers** (k values), which is a classic "binary search on answer" pattern.

Complexity Analysis

- **Time Complexity**: `O(n log m)`

  `n = len(piles)`, `m = max(piles)`.
  Binary search runs in `O(log m)` iterations.
  Each iteration scans all piles → `O(n)`.

- **Space Complexity**: `O(1)`

  Only a few integer variables used—no extra space scaling with input.

12. Minimum Number of Days to Make m Bouquets

Pattern: Binary Search on Answer + Sliding Window / Greedy Validation

---

Problem Statement

You are given an integer array `bloomDay`, an integer `m`, and an integer `k`.

You want to make `m` bouquets. To make a bouquet, you need to use `k` **adjacent** flowers from the garden.

The garden consists of `n` flowers, the `i`th flower will bloom in `bloomDay[i]` and then can be used in **exactly one** bouquet.

Return *the minimum number of days you need to wait to be able to make `m` bouquets from the garden*. If it is impossible to make `m` bouquets, return `-1`.

Sample Input & Output

```
Input: bloomDay = [1,10,3,10,2], m = 3, k = 1
Output: 3
Explanation: We need 3 bouquets, each using 1 flower.
The 3 earliest bloom days are 1, 2, 3 → answer = 3.
```

```
Input: bloomDay = [1,10,2,9,3,8,4,7,5,6], m = 4, k = 2
Output: 9
Explanation: Need 4 bouquets of 2 adjacent flowers.
By day 9, we have enough adjacent bloomed flowers:
e.g., [2,9], [3,8], [4,7], [5,6] → all bloomed by day 9.
```

```
Input: bloomDay = [1,10,3,10,2], m = 3, k = 2
Output: -1
Explanation: Need 3×2 = 6 flowers, but only 5 exist → impossible.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def minDays(self, bloomDay: List[int], m: int, k: int) -> int:
        # STEP 1: Check feasibility - total flowers needed
        #   - If m * k > total flowers, impossible
        if m * k > len(bloomDay):
            return -1

        # Helper: Can we make m bouquets by 'day'?
        def canMake(day: int) -> bool:
            bouquets = 0      # Count of completed bouquets
            consecutive = 0   # Current streak of bloomed flowers

            for bd in bloomDay:
                if bd <= day:
                    consecutive += 1
                    # If we have k consecutive, form a bouquet
                    if consecutive == k:
                        bouquets += 1
                        consecutive = 0  # reset streak
                else:
                    consecutive = 0  # break streak

                # Early exit: already have enough bouquets
                if bouquets >= m:
                    return True
            return bouquets >= m

        # STEP 2: Binary search on answer (days)
        #   - Lower bound: min bloom day
        #   - Upper bound: max bloom day
        left, right = min(bloomDay), max(bloomDay)
        result = right

        while left <= right:
            mid = (left + right) // 2
            if canMake(mid):
                result = mid      # valid candidate
                right = mid - 1   # try smaller day
            else:
                left = mid + 1    # need more days
```

```
        return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.minDays([1,10,3,10,2], 3, 1) == 3

    #  Test 2: Tricky case with adjacency requirement
    assert sol.minDays([1,10,2,9,3,8,4,7,5,6], 4, 2) == 9

    #  Test 3: Impossible case (not enough flowers)
    assert sol.minDays([1,10,3,10,2], 3, 2) == -1

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 2**:
`bloomDay = [1,10,2,9,3,8,4,7,5,6]`, `m = 4`, `k = 2`

We'll simulate the binary search and the `canMake(9)` call.

Step 1: Feasibility Check

- `m * k = 4 * 2 = 8`
- `len(bloomDay) = 10` → 8 10 → **possible**, continue.

Step 2: Binary Search Bounds

- `left = min(bloomDay) = 1`
- `right = max(bloomDay) = 10`
- `result = 10` (initial upper bound)

Step 3: First Mid = (1 + 10) // 2 = 5

Call `canMake(5)`:

- Traverse `bloomDay`:
  - Day 1 5 → `consecutive = 1`
  - Day 10 > 5 → reset → `consecutive = 0`
  - Day 2 5 → `consecutive = 1`
  - Day 9 > 5 → reset → `0`
  - Day 3 5 → `1`
  - Day 8 > 5 → `0`
  - Day 4 5 → `1`
  - Day 7 > 5 → `0`
  - Day 5 5 → `1`
  - Day 6 > 5 → `0`
- Max consecutive = 1 → **no bouquet formed** → `bouquets = 0` →

→ `left = 6`

Step 4: Mid = (6 + 10) // 2 = 8

Call `canMake(8)`:

- Check each flower 8?
  - [1 , 10 , 2 , 9 , 3 , 8 , 4 , 7 , 5 , 6]
- Consecutive tracking:
  - 1 → streak=1
  - 10 → reset → 0
  - 2 → 1
  - 9 → reset → 0

  - $3 \rightarrow 1$
  - $8 \rightarrow 2 \rightarrow$ **bouquet #1**, reset $\rightarrow 0$
  - $4 \rightarrow 1$
  - $7 \rightarrow 2 \rightarrow$ **bouquet #2**, reset $\rightarrow 0$
  - $5 \rightarrow 1$
  - $6 \rightarrow 2 \rightarrow$ **bouquet #3**, reset $\rightarrow 0$

- Total bouquets $= 3 < 4 \rightarrow$

$\rightarrow$ `left = 9`

Step 5: Mid = (9 + 10) // 2 = 9

Call `canMake(9)`:

- Bloom status (9?):
  [1 ,10 ,2 ,9 ,3 ,8 ,4 ,7 ,5 ,6]
- Consecutive:
  - $1 \rightarrow 1$
  - $10 \rightarrow$ reset $\rightarrow 0$
  - $2 \rightarrow 1$
  - $9 \rightarrow 2 \rightarrow$ **bouquet #1**, reset
  - $3 \rightarrow 1$
  - $8 \rightarrow 2 \rightarrow$ **bouquet #2**, reset
  - $4 \rightarrow 1$
  - $7 \rightarrow 2 \rightarrow$ **bouquet #3**, reset
  - $5 \rightarrow 1$
  - $6 \rightarrow 2 \rightarrow$ **bouquet #4** $\rightarrow$ early exit!

$\rightarrow$ `canMake(9)` returns `True`
$\rightarrow$ `result = 9`, then `right = 8` $\rightarrow$ loop ends.

Final answer: **9**

Complexity Analysis

- **Time Complexity**: `O(n log(maxDay))`

  Binary search runs in `O(log(max(bloomDay)))`.
  Each `canMake` validation scans all `n` flowers → `O(n)`.
  Total: `O(n log D)` where `D = max(bloomDay) - min(bloomDay)`.

- **Space Complexity**: `O(1)`

  Only a few integer variables used (`bouquets`, `consecutive`, etc.).
  No extra data structures scaling with input.

13. Find K-th Smallest Pair Distance

Pattern: Binary Search on Answer + Sliding Window (Two Pointers)

Problem Statement

Given an integer array `nums` and an integer `k`, return the `k`-th smallest distance among all the pairs `(nums[i], nums[j])` where `0 <= i < j < nums.length`.
The distance of a pair `(A, B)` is defined as `|A - B|`.

Sample Input & Output

```
Input: nums = [1,3,1], k = 1
Output: 0
Explanation: All pairs: (1,3)→2, (1,1)→0, (3,1)→2. Sorted distances: [0,2,2]. 1st smallest is
```

```
Input: nums = [1,1,1], k = 2
Output: 0
Explanation: All distances are 0. So any k returns 0.
```

```
Input: nums = [1,6,1], k = 3
Output: 5
Explanation: Pairs: (1,6)→5, (1,1)→0, (6,1)→5 → sorted: [0,5,5]. 3rd smallest = 5.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def smallestDistancePair(self, nums: List[int], k: int) -> int:
        # STEP 1: Sort array to enable sliding window counting
        #   - Sorting ensures distances are non-decreasing in windows
        nums.sort()
        n = len(nums)

        # Helper: count pairs with distance <= max_dist
        def count_pairs_with_max_dist(max_dist: int) -> int:
            count = 0
            left = 0
            # STEP 2: Sliding window over sorted array
            #   - For each right, move left until nums[right] - nums[left] <= max_dist
            #   - Invariant: window [left, right] has all valid pairs ending at right
            for right in range(n):
                while nums[right] - nums[left] > max_dist:
                    left += 1
                # STEP 3: Add valid pairs ending at 'right'
                #   - All indices from left to right-1 form valid pairs with right
                count += right - left
            return count

        # STEP 4: Binary search on answer (distance)
        #   - Min possible distance = 0
        #   - Max possible distance = nums[-1] - nums[0]
        low, high = 0, nums[-1] - nums[0]
        while low < high:
            mid = (low + high) // 2
            if count_pairs_with_max_dist(mid) >= k:
                high = mid  # mid might be answer; keep it
            else:
                low = mid + 1  # too few pairs; need larger distance
        return low  # low == high is the k-th smallest distance

# ------------------- INLINE TESTS -------------------
```

```python
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.smallestDistancePair([1,3,1], 1) == 0

    #  Test 2: Edge case (all same)
    assert sol.smallestDistancePair([1,1,1], 2) == 0

    #  Test 3: Tricky/negative (larger k, asymmetric values)
    assert sol.smallestDistancePair([1,6,1], 3) == 5

    print("  All inline tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `smallestDistancePair([1,3,1], k=1)` step by step.

Step 0: Initial Setup
- Input: `nums = [1,3,1]`, `k = 1`
- After sorting: `nums = [1,1,3]`
- `n = 3`
- Binary search bounds: `low = 0`, `high = 3 - 1 = 2`

Step 1: First Binary Search Iteration (`low=0`, `high=2`)
- `mid = (0 + 2) // 2 = 1`
- Call `count_pairs_with_max_dist(1)`

Inside `count_pairs_with_max_dist(1)`:
- `count = 0`, `left = 0`
- `right = 0`: window [0,0] → `nums[0]-nums[0]=0  1` → `count += 0 - 0 = 0`
- `right = 1`: `nums[1]-nums[0] = 0  1` → `count += 1 - 0 = 1` → `count=1`
- `right = 2`: `nums[2]-nums[0] = 2 > 1` → move `left` to 1
- Now `nums[2]-nums[1] = 2 > 1` → move `left` to 2
- Now `left == right`, so `count += 2 - 2 = 0`
- Total `count = 1`

Back to binary search:
- `count = 1 >= k=1` → set `high = mid = 1`

Step 2: Second Binary Search Iteration (`low=0`, `high=1`)
- `mid = (0 + 1) // 2 = 0`
- Call `count_pairs_with_max_dist(0)`

Inside `count_pairs_with_max_dist(0)`:
- `right=0`: count += 0
- `right=1`: `1-1=0  0` → count += 1 → `count=1`
- `right=2`: `3-1=2 > 0` → move `left` to 1 → `3-1=2 > 0` → move `left` to 2 → count += 0
- Total `count = 1`

- `count = 1 >= 1` → set `high = 0`

Step 3: Loop ends (`low=0`, `high=0`)
- Return `low = 0`

Final output: `0` — matches expected.

Key insight: We never enumerate all pairs! We **count** how many pairs have distance X using a sliding window, then **binary search** on X until we find the smallest X with k pairs — which is the k-th smallest distance.

Complexity Analysis

- **Time Complexity**: `O(n log n + n log D)`
  - `O(n log n)` for sorting.
  - Binary search runs `O(log D)` times, where `D = max(nums) - min(nums)`.
  - Each `count_pairs_with_max_dist` is `O(n)` via two pointers.
  - Total: `O(n log n + n log D)`
- **Space Complexity**: `O(1)`
  - Only a few integer variables used. Sorting is in-place (Python's Timsort uses `O(n)` worst-case, but we consider auxiliary space → `O(1)` extra).

14. Find the Smallest Divisor Given a Threshold

Pattern: Binary Search on Answer

Problem Statement

Given an array of integers `nums` and an integer `threshold`, we will choose a positive integer `divisor`, divide all the array elements by it, and sum the division results (using ceiling division). Find the **smallest** such `divisor` so that the sum is **less than or equal to** `threshold`.

Each result of division is rounded **up** to the nearest integer greater than or equal to that element. (i.e., ceiling of the division).
It is guaranteed that there will be an answer.

Sample Input & Output

```
Input: nums = [1,2,5,9], threshold = 6
Output: 5
Explanation:
- divisor=5 → ceil(1/5)=1, ceil(2/5)=1, ceil(5/5)=1, ceil(9/5)=2 → sum=5   6
- divisor=4 → sum = 1+1+2+3 = 7 > 6
So 5 is the smallest valid divisor.
```

```
Input: nums = [2,3,5,7,11], threshold = 11
Output: 3
Explanation: divisor=3 gives sum = 1+1+2+3+4 = 11
```

```
Input: nums = [19], threshold = 5
Output: 4
Explanation: ceil(19/4) = 5   5  ; divisor=3 → ceil(19/3)=7 > 5
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
import math

class Solution:
    def smallestDivisor(self, nums: List[int], threshold: int) -> int:
        # STEP 1: Initialize binary search bounds
        #   - left = 1 (smallest possible divisor)
        #   - right = max(nums) (largest needed; beyond this, sum won't change)
        left, right = 1, max(nums)

        # STEP 2: Binary search on divisor value
        #   - We're searching for the smallest divisor where sum <= threshold
        #   - Invariant: answer always in [left, right]
        while left < right:
            mid = (left + right) // 2

            # Compute total sum using ceiling division with divisor = mid
            total = 0
            for num in nums:
                total += math.ceil(num / mid)

            # STEP 3: Update search space based on total vs threshold
            #   - If total <= threshold, mid is valid → try smaller divisor
            #   - Else, mid too small → need larger divisor
            if total <= threshold:
                right = mid  # mid could be the answer
            else:
                left = mid + 1  # mid is too small

        # STEP 4: Return result
        #   - left == right is the minimal valid divisor
        return left

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.smallestDivisor([1,2,5,9], 6) == 5
```

```
    #   Test 2: Edge case - single element
    assert sol.smallestDivisor([19], 5) == 4

    #   Test 3: Tricky/negative - large numbers, tight threshold
    assert sol.smallestDivisor([44,22,33,11,1], 5) == 44
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `smallestDivisor([1,2,5,9], 6)` step by step:

1. **Initialize**:
   - `nums = [1,2,5,9]`, `threshold = 6`
   - `left = 1`, `right = max(nums) = 9`
2. **First loop iteration** (`left=1`, `right=9`):
   - `mid = (1+9)//2 = 5`
   - Compute total:
     - `ceil(1/5) = 1`
     - `ceil(2/5) = 1`
     - `ceil(5/5) = 1`
     - `ceil(9/5) = 2`
     - `total = 1+1+1+2 = 5`
   - Since `5 <= 6` → valid → set `right = 5`
3. **Second iteration** (`left=1`, `right=5`):
   - `mid = (1+5)//2 = 3`
   - Total:

- `ceil(1/3)=1`, `ceil(2/3)=1`, `ceil(5/3)=2`, `ceil(9/3)=3`
- `total = 1+1+2+3 = 7 > 6` → invalid

- Set `left = 3 + 1 = 4`

4. **Third iteration** (`left=4`, `right=5`):
   - `mid = (4+5)//2 = 4`
   - Total:
     - `ceil(1/4)=1`, `ceil(2/4)=1`, `ceil(5/4)=2`, `ceil(9/4)=3`
     - `total = 7 > 6` → invalid
   - Set `left = 4 + 1 = 5`
5. **Loop ends** (`left=5`, `right=5`) → return `5`

Final output: `5` — matches expected.

Complexity Analysis

- **Time Complexity**: `O(n log M)`

  `n = len(nums)`, `M = max(nums)`.
  Binary search runs in `O(log M)` iterations.
  Each iteration computes sum in `O(n)` time.

- **Space Complexity**: `O(1)`

  Only a few integer variables used — no extra space scaling with input.

15. Ugly Number III

Pattern: Binary Search + Inclusion-Exclusion Principle

Problem Statement

Write a program to find the `n`-th ugly number.

Ugly numbers are positive integers which are divisible by `a`, `b`, or `c`.

Note:
- `1 <= n, a, b, c <= 10^9`
- `1 <= a * b * c <= 10^18` (to avoid overflow in LCM)

Sample Input & Output

```
Input: n = 3, a = 2, b = 3, c = 5
Output: 4
Explanation: The ugly numbers are 2, 3, 4, 5, 6, 8, 9, 10... → 3rd is 4.
```

```
Input: n = 4, a = 2, b = 3, c = 4
Output: 6
Explanation: Ugly numbers: 2, 3, 4, 6 → 4th is 6.
```

```
Input: n = 1000000000, a = 2, b = 217983653, c = 336916467
Output: 1999999984
Explanation: Large inputs require efficient counting via math, not brute force.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
import math

class Solution:
    def nthUglyNumber(self, n: int, a: int, b: int, c: int) -> int:
        # STEP 1: Helper to compute LCM of two numbers
        def lcm(x, y):
            return x * y // math.gcd(x, y)
```

```
        # Precompute pairwise and triple LCMs
        ab = lcm(a, b)
        ac = lcm(a, c)
        bc = lcm(b, c)
        abc = lcm(ab, c)

        # STEP 2: Count how many ugly numbers <= x
        def count_ugly(x):
            # Inclusion-Exclusion Principle:
            # |A  B  C| = |A| + |B| + |C|
            #              - |A B| - |A C| - |B C|
            #              + |A B C|
            return (x // a + x // b + x // c
                    - x // ab - x // ac - x // bc
                    + x // abc)

        # STEP 3: Binary search over answer space
        # Lower bound: 1
        # Upper bound: n * min(a, b, c) is safe but loose;
        # we use n * min(a,b,c)  2e9 → but worst-case n=1e9, min=1 → 1e9
        # However, ugly number could be as large as ~2e9, so set high = 2 * 10**9
        low, high = 1, 2 * 10**9

        while low < high:
            mid = (low + high) // 2
            if count_ugly(mid) < n:
                low = mid + 1
            else:
                high = mid

        # STEP 4: Return the smallest x such that count_ugly(x) >= n
        return low

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.nthUglyNumber(3, 2, 3, 5) == 4, "Test 1 failed"

    #  Test 2: Edge case - duplicates due to common multiples
    assert sol.nthUglyNumber(4, 2, 3, 4) == 6, "Test 2 failed"
```

```
    #   Test 3: Tricky/negative - large inputs
    assert sol.nthUglyNumber(1000000000, 2, 217983653, 336916467) == 1999999984, "Test 3 fai

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 1**: `n=3, a=2, b=3, c=5`

1. **Compute LCMs**:
   - `lcm(2,3) = 6`, `lcm(2,5)=10`, `lcm(3,5)=15`
   - `lcm(6,5) = 30` → so `ab=6, ac=10, bc=15, abc=30`
2. **Define `count_ugly(x)`**:
   For any `x`, it returns how many numbers `x` divisible by 2, 3, or 5.
3. **Binary Search Setup**:
   - `low = 1`, `high = 2_000_000_000`
4. **First Iteration**:
   - `mid = (1 + 2e9)//2` `1e9` → `count_ugly(1e9)` is huge (>3) → so `high = 1e9`

   … (search narrows down) …
5. **When `mid = 3`**:
   - `count_ugly(3) = 3//2 + 3//3 + 3//5 - 3//6 - 3//10 - 3//15 + 3//30`
     `= 1 + 1 + 0 - 0 - 0 - 0 + 0 = 2` → < 3 → so `low = 4`
6. **When `mid = 4`**:
   - `count_ugly(4) = 4//2 + 4//3 + 4//5 - ... = 2 + 1 + 0 - 0 - 0 - 0 + 0`
     `= 3`
   - Since `3 >= 3`, set `high = 4`

7. **Loop ends**: `low == high == 4` → return `4`

Final output: `4` — correct!

Complexity Analysis

- **Time Complexity**: `O(log(n * min(a, b, c)))`

  Binary search runs in `O(log(max_answer))`. Since answer `2e9`, it's ~31 iterations. Each `count_ugly` call does O(1) math ops.

- **Space Complexity**: `O(1)`

  Only a few integer variables and LCMs stored — no input-dependent structures.

16. Kth Smallest Number in Multiplication Table

Pattern: Binary Search on Answer

Problem Statement

Nearly everyone has used the multiplication table. The multiplication table of size `m x n` is an integer matrix `mat` where `mat[i][j] == i * j` (1-indexed).

Given three integers `m`, `n`, and `k`, return the `k`th smallest element in the `m x n` multiplication table.

Sample Input & Output

```
Input: m = 3, n = 3, k = 5
Output: 3
Explanation: The multiplication table is:
1 2 3
2 4 6
3 6 9
Sorted elements: [1,2,2,3,3,4,6,6,9] → 5th smallest is 3.
```

```
Input: m = 2, n = 3, k = 6
Output: 6
Explanation: Table: [1,2,3; 2,4,6] → sorted: [1,2,2,3,4,6]
```

```
Input: m = 1, n = 1, k = 1
Output: 1
Explanation: Only one element → trivial edge case.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def findKthNumber(self, m: int, n: int, k: int) -> int:
        # STEP 1: Initialize binary search bounds
        #   - Smallest possible value is 1
        #   - Largest possible value is m * n
        low, high = 1, m * n

        # STEP 2: Main binary search loop
        #   - We search for the smallest value 'x' such that
        #     count(x) >= k, where count(x) = # of elements <= x
        while low < high:
            mid = (low + high) // 2

            # Count how many numbers <= mid in table
            count = 0
            for i in range(1, m + 1):
                # For row i, max value is i * n
```

```
                # Numbers <= mid in this row: min(n, mid // i)
                count += min(n, mid // i)

            # STEP 3: Update search space
            #   - If count >= k, mid might be answer (or too big)
            #   - Else, mid is too small
            if count >= k:
                high = mid
            else:
                low = mid + 1

        # STEP 4: Return result
        #   - low == high is the minimal x with count(x) >= k
        return low

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.findKthNumber(3, 3, 5) == 3, "Test 1 failed"
    print("  Test 1 passed: (3,3,5) → 3")

    #  Test 2: Edge case - single cell
    assert sol.findKthNumber(1, 1, 1) == 1, "Test 2 failed"
    print("  Test 2 passed: (1,1,1) → 1")

    #  Test 3: Tricky/negative - large k
    assert sol.findKthNumber(2, 3, 6) == 6, "Test 3 failed"
    print("  Test 3 passed: (2,3,6) → 6")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `findKthNumber(3, 3, 5)` step by step:

1. **Initialize**:

- `low = 1`, `high = 9` (since 3×3=9)

2. **First iteration**:
   - `mid = (1 + 9) // 2 = 5`
   - Count numbers 5:
     - Row 1: `min(3, 5//1) = min(3,5) = 3`
     - Row 2: `min(3, 5//2) = min(3,2) = 2`
     - Row 3: `min(3, 5//3) = min(3,1) = 1`
     - Total `count = 3 + 2 + 1 = 6`
   - Since `6 >= 5`, set `high = 5`
3. **Second iteration**:
   - `low = 1`, `high = 5` → `mid = 3`
   - Count 3:
     - Row 1: `min(3, 3//1)=3`
     - Row 2: `min(3, 3//2)=1`
     - Row 3: `min(3, 3//3)=1`
     - `count = 5`
   - `5 >= 5` → `high = 3`
4. **Third iteration**:
   - `low = 1`, `high = 3` → `mid = 2`
   - Count 2:
     - Row 1: `min(3,2)=2`
     - Row 2: `min(3,1)=1`
     - Row 3: `min(3,0)=0`
     - `count = 3 < 5` → `low = 3`

5. **Loop ends**: `low == high == 3` → return `3`

Final Output: `3`
Key Insight: We never build the table! We use math (`mid // i`) to count entries `mid` efficiently.

Complexity Analysis

- **Time Complexity**: `O(m * log(m * n))`

  Binary search runs in `O(log(mn))`. Each iteration scans `m` rows and does `O(1)` work per row.

- **Space Complexity**: `O(1)`

  Only a few integer variables used — no extra space proportional to input.

17. Search in Rotated Sorted Array

Pattern: Binary Search (Modified)

Problem Statement

There is an integer array `nums` sorted in ascending order (with **distinct** values).

Prior to being passed to your function, `nums` is **possibly rotated** at an unknown pivot index `k` (`1 <= k < nums.length`) such that the resulting array is `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]` (0-indexed).

Given the array `nums` **after** the possible rotation and an integer `target`, return the **index** of `target` if it is in `nums`, or `-1` if it is not in `nums`.

You must write an algorithm with `O(log n)` runtime complexity.

Sample Input & Output

```
Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4
Explanation: 0 is at index 4 in the rotated array.
```

```
Input: nums = [4,5,6,7,0,1,2], target = 3
Output: -1
Explanation: 3 is not present in the array.
```

```
Input: nums = [1], target = 0
Output: -1
Explanation: Single-element array doesn't contain target.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def search(self, nums: List[int], target: int) -> int:
        # STEP 1: Initialize binary search bounds
        #   - left and right pointers cover full array
        left, right = 0, len(nums) - 1

        # STEP 2: Main loop - maintain O(log n) by halving search space
        #   - Invariant: target (if exists) is always in [left, right]
        while left <= right:
            mid = (left + right) // 2

            # Found target?
            if nums[mid] == target:
                return mid

            # STEP 3: Determine which half is sorted
            #   - In rotated sorted array, at least one half is always sorted
            if nums[left] <= nums[mid]:
```

```
                # Left half [left, mid] is sorted
                if nums[left] <= target < nums[mid]:
                    # Target in sorted left half → narrow right
                    right = mid - 1
                else:
                    # Target not in left → search right half
                    left = mid + 1
            else:
                # Right half [mid, right] is sorted
                if nums[mid] < target <= nums[right]:
                    # Target in sorted right half → narrow left
                    left = mid + 1
                else:
                    # Target not in right → search left half
                    right = mid - 1

        # STEP 4: Return result - target not found
        return -1

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.search([4,5,6,7,0,1,2], 0) == 4

    #  Test 2: Edge case - single element not found
    assert sol.search([1], 0) == -1

    #  Test 3: Tricky/negative - target absent in rotated array
    assert sol.search([4,5,6,7,0,1,2], 3) == -1

    print(" All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `nums = [4,5,6,7,0,1,2]`, `target = 0`.

Initial state:
- `left = 0`, `right = 6`
- Array: `[4,5,6,7,0,1,2]`

Iteration 1:
- `mid = (0 + 6) // 2 = 3` → `nums[3] = 7`
- `7 != 0` → continue
- Check if left half sorted: `nums[0] = 4 <= nums[3] = 7` → **yes**
- Is `target = 0` in `[4, 7)`? → `4 <= 0 < 7`? → **No**
- So discard left half → `left = mid + 1 = 4`

State: `left=4`, `right=6`

Iteration 2:
- `mid = (4 + 6) // 2 = 5` → `nums[5] = 1`
- `1 != 0` → continue
- Check if left half sorted: `nums[4] = 0 <= nums[5] = 1` → **yes**
- Is `0` in `[0, 1)`? → `0 <= 0 < 1` → **Yes!**
- So search left → `right = mid - 1 = 4`

State: `left=4`, `right=4`

Iteration 3:
- `mid = (4 + 4) // 2 = 4` → `nums[4] = 0`
- `0 == target` → **return 4**

Final output: `4`

Key insight: Even though the array is rotated, **one half is always sorted**, letting us apply binary search logic by checking which side could contain the target.

Complexity Analysis

- **Time Complexity**: `O(log n)`

  Each iteration halves the search space (`left` or `right` moves toward `mid`), just like standard binary search. Only constant-time checks per step.

- **Space Complexity**: `O(1)`

  Only a few integer variables (`left`, `right`, `mid`) are used. No recursion or extra data structures scale with input size.

18. Find Minimum in Rotated Sorted Array

Pattern: Binary Search on Rotated Sorted Array

Problem Statement

Suppose an array of length `n` sorted in ascending order is rotated between `1` and `n` times. For example, the array `nums = [0,1,2,4,5,6,7]` might become:
- `[4,5,6,7,0,1,2]` if rotated 4 times.
- `[0,1,2,4,5,6,7]` if rotated 7 times.

Given the sorted rotated array `nums` of **unique** elements, return the **minimum element** of this array.

You must solve it in **O(log n)** time.

Sample Input & Output

```
Input: nums = [3,4,5,1,2]
Output: 1
Explanation: The array was rotated 3 times; 1 is the smallest.
```

```
Input: nums = [4,5,6,7,0,1,2]
Output: 0
Explanation: Rotation point is between 7 and 0; 0 is min.
```

```
Input: nums = [2,1]
Output: 1
Explanation: Edge case with only two elements.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def findMin(self, nums: List[int]) -> int:
        # STEP 1: Initialize pointers
        #   - left and right bound the search space
        left, right = 0, len(nums) - 1

        # STEP 2: Main loop - binary search invariant
        #   - We maintain that the min is in [left, right]
        #   - Since array is rotated, one half is always sorted
        while left < right:
            mid = (left + right) // 2

            # STEP 3: Compare mid with right
            #   - If nums[mid] > nums[right], min is in right half
            #   - Else, min is in left half (including mid)
            if nums[mid] > nums[right]:
                left = mid + 1
            else:
                right = mid

        # STEP 4: Return result
        #   - When left == right, we've found the min
        return nums[left]

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.findMin([3,4,5,1,2]) == 1
```

```
    #   Test 2: Edge case - already sorted (no effective rotation)
    assert sol.findMin([0,1,2,4,5,6,7]) == 0

    #   Test 3: Tricky/negative - two elements, decreasing
    assert sol.findMin([2,1]) == 1

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `findMin([3,4,5,1,2])` step by step:

1. **Initialize**:
   - `left = 0`, `right = 4`
   - Array: `[3,4,5,1,2]`
2. **First loop iteration** (`left=0`, `right=4`):
   - `mid = (0+4)//2 = 2` → `nums[mid] = 5`
   - Compare `nums[mid] (5)` vs `nums[right] (2)`
   - Since `5 > 2`, the right half `[1,2]` contains the drop → min is there
   - Update: `left = mid + 1 = 3`
   - Now: `left=3`, `right=4`
3. **Second loop iteration** (`left=3`, `right=4`):
   - `mid = (3+4)//2 = 3` → `nums[mid] = 1`
   - Compare `1` vs `nums[right]=2`
   - Since `1 <= 2`, min is in left half (including mid)

- Update: `right = mid = 3`
- Now: `left=3`, `right=3`

4. **Loop ends** (`left == right`):
   - Return `nums[3] = 1`

Final Output: `1`
Key Insight: By comparing `mid` with `right`, we determine which half is **not** sorted—and the rotation point (hence min) must lie in the unsorted half.

Complexity Analysis

- **Time Complexity**: `O(log n)`

  We halve the search space each iteration (classic binary search).

- **Space Complexity**: `O(1)`

  Only using a few integer variables (`left`, `right`, `mid`). No recursion or extra data structures.

19. Find Minimum in Rotated Sorted Array II

Pattern: Binary Search (with Duplicates)

Problem Statement

Suppose an array of length `n` sorted in ascending order is rotated between `1` and `n` times. For example, the array `nums = [0,1,2,4,5,6,7]` might become:
- `[4,5,6,7,0,1,2]` if rotated 4 times.
- `[0,1,2,4,5,6,7]` if rotated 7 times.

Notice that rotating an array `[a[0], a[1], a[2], ..., a[n-1]]` 1 time results in the array `[a[n-1], a[0], a[1], a[2], ..., a[n-2]]`.

Given the sorted rotated array `nums` that may contain **duplicates**, return the **minimum element** of this array.

You must decrease the overall operation steps as much as possible.

Sample Input & Output

```
Input: [1,3,5]
Output: 1
Explanation: Array is not rotated; minimum is first element.
```

```
Input: [2,2,2,0,1]
Output: 0
Explanation: Rotated array with duplicates; minimum is 0.
```

```
Input: [3,3,1,3]
Output: 1
Explanation: Tricky case where nums[left] == nums[mid] == nums[right].
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def findMin(self, nums: List[int]) -> int:
        # STEP 1: Initialize pointers
        #   - left and right bound the search space
        left, right = 0, len(nums) - 1

        # STEP 2: Main binary search loop
        #   - Invariant: min element is always in [left, right]
        while left < right:
            mid = (left + right) // 2

            # STEP 3: Compare mid with right to decide direction
            if nums[mid] > nums[right]:
                # Minimum must be in right half (mid cannot be min)
                left = mid + 1
            elif nums[mid] < nums[right]:
```

```
                # Minimum is in left half (mid could be min)
                right = mid
            else:
                # nums[mid] == nums[right]: ambiguous due to duplicates
                # Cannot discard mid, but can safely shrink right
                right -= 1

        # STEP 4: Return result
        #   - When left == right, we've converged to min
        return nums[left]

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.findMin([1, 3, 5]) == 1

    #  Test 2: Edge case (all same)
    assert sol.findMin([2, 2, 2, 2]) == 2

    #  Test 3: Tricky/negative (duplicates obscure rotation)
    assert sol.findMin([3, 3, 1, 3]) == 1

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through `nums = [3, 3, 1, 3]` step by step.

Initial state:
- `left = 0`, `right = 3`
- `nums = [3, 3, 1, 3]`

Iteration 1:
- `left=0 < right=3` → enter loop
- `mid = (0 + 3) // 2 = 1`
- `nums[mid] = nums[1] = 3`
- `nums[right] = nums[3] = 3`
- Since `nums[mid] == nums[right]`, we do: `right -= 1` → `right = 2`
- **State**: `left=0`, `right=2`

Iteration 2:
- `left=0 < right=2` → continue
- `mid = (0 + 2) // 2 = 1`
- `nums[mid] = 3`, `nums[right] = nums[2] = 1`
- Now `3 > 1` → so `left = mid + 1 = 2`
- **State**: `left=2`, `right=2`

Loop ends because `left == right`.
Return `nums[2] = 1`.

Final output: `1`

Key insight: When `nums[mid] == nums[right]`, we can't tell which side the min is on, but we **can safely remove `right`** because even if it's the minimum, `mid` has the same value — so the min is still in `[left, right-1]`.

Complexity Analysis

- **Time Complexity**: `O(log n)` average, `O(n)` worst-case

  In the average case, binary search halves the space each time → `O(log n)`. Worst case occurs when all elements are equal (e.g., `[2,2,2,2]`), forcing us to decrement `right` one by one → `O(n)`.

- **Space Complexity**: `O(1)`

  Only uses a constant amount of extra space (`left`, `right`, `mid`). No recursion or auxiliary data structures.

20. Search a 2D Matrix

Pattern: Binary Search (Flattened 2D Array)

Problem Statement

Write an efficient algorithm that searches for a target value in an `m x n` integer matrix.
The matrix has the following properties:
- Integers in each row are sorted in ascending order from left to right.
- Integers in each column are sorted in ascending order from top to bottom.

Clarification: This version assumes the **entire matrix is sorted row-wise**, i.e., the first integer of each row is greater than the last integer of the previous row — making it equivalent to a sorted 1D array. (This matches LeetCode problem #74.)

Sample Input & Output

```
Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3
Output: true
Explanation: 3 is present in the first row.
```

```
Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13
Output: false
Explanation: 13 is not in the matrix.
```

```
Input: matrix = [[1]], target = 1
Output: true
Explanation: Single-element matrix contains the target.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        # STEP 1: Initialize structures
        #   - Treat matrix as a flattened sorted array of size m*n.
        #   - Use binary search on virtual indices [0, m*n - 1].
        if not matrix or not matrix[0]:
            return False

        m, n = len(matrix), len(matrix[0])
        left, right = 0, m * n - 1

        # STEP 2: Main loop / recursion
        #   - Maintain invariant: target is in [left, right] if present.
        #   - Convert 1D mid index to 2D (row, col) via divmod.
        while left <= right:
            mid = (left + right) // 2
            row, col = divmod(mid, n)
            mid_val = matrix[row][col]

            # STEP 3: Update state / bookkeeping
            #   - If mid_val == target, found it.
            #   - If too small, search right half; else left half.
            if mid_val == target:
                return True
            elif mid_val < target:
                left = mid + 1
            else:
                right = mid - 1

        # STEP 4: Return result
        #   - Loop ended without finding target → not present.
        return False

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.searchMatrix(
        [[1,3,5,7],[10,11,16,20],[23,30,34,60]], 3
```

```
    ) == True

    #  Test 2: Edge case
    assert sol.searchMatrix([[1]], 1) == True

    #  Test 3: Tricky/negative
    assert sol.searchMatrix(
        [[1,3,5,7],[10,11,16,20],[23,30,34,60]], 13
    ) == False

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `searchMatrix([[1,3,5,7],[10,11,16,20],[23,30,34,60]], 13)`.

Initial state:
- `matrix` has `m = 3` rows, `n = 4` columns → total 12 elements.
- `left = 0`, `right = 11`.

Step 1:
- `mid = (0 + 11) // 2 = 5`
- `row, col = divmod(5, 4) → (1, 1)`
- `mid_val = matrix[1][1] = 11`
- Since `11 < 13`, set `left = mid + 1 = 6`

State: `left=6`, `right=11`

Step 2:
- `mid = (6 + 11) // 2 = 8`
- `row, col = divmod(8, 4) → (2, 0)`
- `mid_val = matrix[2][0] = 23`
- Since `23 > 13`, set `right = mid - 1 = 7`

State: `left=6`, `right=7`

Step 3:
- `mid = (6 + 7) // 2 = 6`
- `row, col = divmod(6, 4) → (1, 2)`

- `mid_val = matrix[1][2] = 16`
- Since `16 > 13`, set `right = 5`

State: `left=6`, `right=5` → loop ends (`left > right`)

Final result: return `False` → 13 not found.

Complexity Analysis

- **Time Complexity**: `O(log(m * n))`

  We perform binary search over `m * n` virtual elements. Each step halves the search space → logarithmic time.

- **Space Complexity**: `O(1)`

  Only a few integer variables (`left`, `right`, `mid`, `row`, `col`) are used — no extra space scaling with input.

21. Search a 2D Matrix II

Pattern: Matrix Search / Divide and Eliminate

Problem Statement

Write an efficient algorithm that searches for a `target` value in an `m x n` integer matrix `matrix`. This matrix has the following properties: - Integers in each row are sorted in ascending order from left to right. - Integers in each column are sorted in ascending order from top to bottom.

Sample Input & Output

```
Input: matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]]
Output: true
Explanation: 5 exists at matrix[1][1].
```

```
Input: matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]]
Output: false
Explanation: 20 is not present in the matrix.
```

```
Input: matrix = [[-1,3]], target = 3
Output: true
Explanation: Single-row matrix; target is at the end.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        # STEP 1: Initialize structures
        #   - Start from top-right corner (row=0, col=last)
        #   - This position allows us to eliminate a row or column
        #     in each step based on comparison with target.
        if not matrix or not matrix[0]:
            return False

        rows, cols = len(matrix), len(matrix[0])
        row, col = 0, cols - 1  # top-right corner

        # STEP 2: Main loop / recursion
        #   - While within bounds, compare current value to target
        #   - If equal → found!
        #   - If current > target → eliminate column (move left)
        #   - If current < target → eliminate row (move down)
        while row < rows and col >= 0:
```

```
            current = matrix[row][col]
            if current == target:
                return True
            elif current > target:
                col -= 1  # eliminate current column
            else:
                row += 1  # eliminate current row

        # STEP 4: Return result
        #   - If loop ends without match, target not present
        return False

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    matrix1 = [
        [1,4,7,11,15],
        [2,5,8,12,19],
        [3,6,9,16,22],
        [10,13,14,17,24],
        [18,21,23,26,30]
    ]
    assert sol.searchMatrix(matrix1, 5) == True
    print(" Test 1 passed: target 5 found")

    #  Test 2: Edge case - target not present
    assert sol.searchMatrix(matrix1, 20) == False
    print(" Test 2 passed: target 20 not found")

    #  Test 3: Tricky/negative - single row, negative number
    matrix3 = [[-1, 3]]
    assert sol.searchMatrix(matrix3, 3) == True
    print(" Test 3 passed: single-row matrix with target at end")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1**: `target = 5` in the 5x5 matrix.

Initial state:
- `row = 0`, `col = 4` → `current = matrix[0][4] = 15`

Step 1:
- Compare `15` vs `5` → `15 > 5`
- So move **left**: `col = 3`
- Now `current = matrix[0][3] = 11`

Step 2:
- `11 > 5` → move left → `col = 2`
- `current = matrix[0][2] = 7`

Step 3:
- `7 > 5` → move left → `col = 1`
- `current = matrix[0][1] = 4`

Step 4:
- `4 < 5` → move **down**: `row = 1`
- `current = matrix[1][1] = 5`

Step 5:
- `5 == 5` → return `True`

Found in 5 steps!
Key insight: each step **eliminates** a full row or column, making search efficient.

Complexity Analysis

- **Time Complexity**: `O(m + n)`

  In worst case, we traverse from top-right to bottom-left:
  at most `m` downward moves and `n` leftward moves → total `m + n` steps.

- **Space Complexity**: `O(1)`

  Only a few variables (`row`, `col`, `current`) used — no extra data structures.

22. Median of Two Sorted Arrays

Pattern: Binary Search on Partition (Advanced Divide & Conquer)

Problem Statement

Given two sorted arrays `nums1` and `nums2` of size `m` and `n` respectively, return the **median** of the two sorted arrays.
The overall run time complexity should be `O(log (m+n))`.

Sample Input & Output

```
Input: nums1 = [1,3], nums2 = [2]
Output: 2.00000
Explanation: Merged array = [1,2,3], median is 2.
```

```
Input: nums1 = [1,2], nums2 = [3,4]
Output: 2.50000
Explanation: Merged array = [1,2,3,4], median = (2+3)/2 = 2.5.
```

```
Input: nums1 = [], nums2 = [1]
Output: 1.00000
Explanation: Edge case - one array empty; median is the only element.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def findMedianSortedArrays(
        self, nums1: List[int], nums2: List[int]
    ) -> float:
        # Ensure nums1 is the smaller array to minimize binary search space
        if len(nums1) > len(nums2):
            nums1, nums2 = nums2, nums1

        m, n = len(nums1), len(nums2)
        total = m + n
        half = total // 2

        # STEP 1: Binary search on partition index in nums1
        #   - We're searching for a split in nums1 such that combined left
        #     side (from both arrays) has exactly 'half' elements.
        left, right = 0, m

        while left <= right:
            # Partition index in nums1
            i = (left + right) // 2
            # Partition index in nums2 to balance total left size = half
            j = half - i

            # Handle boundaries: use -inf / +inf for empty sides
            nums1_left = nums1[i - 1] if i > 0 else float('-inf')
            nums1_right = nums1[i] if i < m else float('inf')
            nums2_left = nums2[j - 1] if j > 0 else float('-inf')
            nums2_right = nums2[j] if j < n else float('inf')

            # STEP 2: Check if partition is valid
            #   - Left max <= right min in both arrays → correct partition
            if nums1_left <= nums2_right and nums2_left <= nums1_right:
                # STEP 4: Compute median based on total length parity
                if total % 2 == 1:
                    return min(nums1_right, nums2_right)
                else:
                    left_max = max(nums1_left, nums2_left)
                    right_min = min(nums1_right, nums2_right)
                    return (left_max + right_min) / 2.0
```

```
            # STEP 3: Adjust binary search bounds
            #   - If nums1's left > nums2's right, move left in nums1
            elif nums1_left > nums2_right:
                right = i - 1
            else:
                left = i + 1

        # Should never reach here for valid input
        raise ValueError("Input arrays are not sorted or invalid.")

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.findMedianSortedArrays([1, 3], [2]) == 2.0

    #  Test 2: Even total length
    assert sol.findMedianSortedArrays([1, 2], [3, 4]) == 2.5

    #  Test 3: Edge case - one empty array
    assert sol.findMedianSortedArrays([], [1]) == 1.0
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1**: `nums1 = [1, 3]`, `nums2 = [2]`.

Initial Setup: - `nums1 = [1,3]` (m=2), `nums2 = [2]` (n=1) - Since `m > n`, we swap → now `nums1 = [2]`, `nums2 = [1,3]` - `total = 3`, `half = 1` - Binary search: `left = 0`, `right = 1` (length of new `nums1`)

Iteration 1: - `i = (0 + 1) // 2 = 0` → partition at start of `nums1` - `j = half - i = 1 - 0 = 1` - Values: - `nums1_left = -inf` (i=0 → no left) - `nums1_right = nums1[0] = 2` - `nums2_left = nums2[0] = 1` (j=1 → left has 1 element) - `nums2_right = inf` (j=1 = n → no right) - Check condition: - `nums1_left (-inf) <= nums2_right (inf)` → - `nums2_left (1) <= nums1_right (2)` → → **valid partition!** - Total length is odd → return `min(nums1_right, nums2_right) = min(2, inf) = 2`

Output: `2.0`

Key Insight: We never merge arrays. Instead, we *virtually* split both arrays so the left halves contain the smaller half of all elements. The median comes from boundary values around the split.

Complexity Analysis

- **Time Complexity**: `O(log(min(m, n)))`

  We perform binary search on the smaller array. Each step halves the search space → logarithmic in the size of the smaller input.

- **Space Complexity**: `O(1)`

  Only a constant number of variables are used — no recursion, no extra arrays.

23. Find Smallest Letter Greater Than Target

Pattern: Binary Search

Problem Statement

Given a characters array `letters` that is sorted in **non-decreasing order**, and a character `target`, return the **smallest character** in `letters` that is **strictly greater** than `target`.

Letters also wrap around. For example, if `target == 'z'` and `letters == ['a', 'b']`, the answer is `'a'`.

Sample Input & Output

```
Input: letters = ["c","f","j"], target = "a"
Output: "c"
Explanation: 'c' is the first letter > 'a'.
```

```
Input: letters = ["c","f","j"], target = "c"
Output: "f"
Explanation: 'f' is the smallest letter strictly greater than 'c'.
```

```
Input: letters = ["x","x","y","y"], target = "z"
Output: "x"
Explanation: Wrap-around: no letter > 'z', so return first letter.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def nextGreatestLetter(
        self, letters: List[str], target: str
    ) -> str:
        # STEP 1: Initialize binary search bounds
        #   - left = 0, right = len(letters) (exclusive upper bound)
        left, right = 0, len(letters)

        # STEP 2: Main loop - search for first letter > target
        #   - Invariant: answer is in [left, right)
        #   - We maintain that all letters < left are <= target
        while left < right:
            mid = (left + right) // 2
            if letters[mid] <= target:
                # Too small or equal → discard left half
                left = mid + 1
            else:
                # Candidate found → keep it and search left
```

```
                right = mid

        # STEP 3: Return result with wrap-around
        #   - If left == len(letters), all letters <= target
        #   - Wrap to first letter using modulo
        return letters[left % len(letters)]

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.nextGreatestLetter(
        ["c","f","j"], "a"
    ) == "c", "Test 1 failed"

    #  Test 2: Edge case - duplicate letters
    assert sol.nextGreatestLetter(
        ["c","f","j"], "c"
    ) == "f", "Test 2 failed"

    #  Test 3: Tricky/negative - wrap-around
    assert sol.nextGreatestLetter(
        ["x","x","y","y"], "z"
    ) == "x", "Test 3 failed"

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 3**:
`letters = ["x","x","y","y"]`, `target = "z"`

Initial state:
- `left = 0`, `right = 4` (length of list)

Iteration 1:
- `mid = (0 + 4) // 2 = 2`
- `letters[2] = "y"`
- Compare: `"y" <= "z"` → **True**
- So: `left = mid + 1 = 3`
- State: `left=3`, `right=4`

Iteration 2:
- `mid = (3 + 4) // 2 = 3`
- `letters[3] = "y"`
- `"y" <= "z"` → **True**
- So: `left = 3 + 1 = 4`
- Now `left == right` → loop ends

Final step:
- `left = 4`, `len(letters) = 4`
- `left % len(letters) = 4 % 4 = 0`
- Return `letters[0] = "x"`

Output: `"x"` — correct due to wrap-around.

Complexity Analysis

- **Time Complexity**: `O(log n)`

  Binary search halves the search space each iteration → log (n) steps.

- **Space Complexity**: `O(1)`

  Only uses a few integer variables (`left`, `right`, `mid`) — no extra data structures.

24. Maximum Profit in Job Scheduling

Pattern: Dynamic Programming + Binary Search (Weighted Interval Scheduling)

Problem Statement

We have `n` jobs, where every job is scheduled to be done from `startTime[i]` to `endTime[i]`, obtaining a profit of `profit[i]`.

You're given the `startTime`, `endTime` and `profit` arrays. Return the maximum profit you can take such that no two jobs overlap.

A job that starts at time `t` can only be scheduled if the previous job ends **at or before** `t`.

Sample Input & Output

```
Input: startTime = [1,2,3,3], endTime = [3,4,5,6], profit = [50,10,40,70]
Output: 120
Explanation: Choose jobs [1,3,6] → profit = 50 + 70 = 120 (job 1: [1,3], job 4: [3,6])
```

```
Input: startTime = [1,2,3,4,6], endTime = [3,5,10,6,9], profit = [20,20,100,70,60]
Output: 150
Explanation: Choose jobs [1,4,6] → [1,3] + [4,6] + [6,9] → 20 + 70 + 60 = 150
```

```
Input: startTime = [1], endTime = [2], profit = [10]
Output: 10
Explanation: Only one job → take it.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
import bisect

class Solution:
    def jobScheduling(
        self, startTime: List[int], endTime: List[int], profit: List[int]
    ) -> int:
```

```
        # STEP 1: Initialize structures
        #   - Combine jobs into tuples and sort by endTime.
        #   - This lets us use DP where dp[i] = max profit using first i jobs.
        jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1])
        n = len(jobs)
        end_times = [job[1] for job in jobs]  # for binary search
        dp = [0] * (n + 1)  # dp[0] = 0 (no jobs)

        # STEP 2: Main loop / recursion
        #   - For each job i (1-indexed in dp), decide: skip or take?
        for i in range(1, n + 1):
            start, end, prof = jobs[i - 1]

            # STEP 3: Update state / bookkeeping
            #   - Option 1: Skip job → dp[i] = dp[i-1]
            #   - Option 2: Take job → find latest non-overlapping job j
            #     using binary search on end_times.
            #   - Why here? Because jobs are sorted by end time,
            #     so all jobs before j are compatible.
            skip = dp[i - 1]
            # Find rightmost job with endTime <= start
            j = bisect.bisect_right(end_times, start, 0, i - 1)
            take = dp[j] + prof
            dp[i] = max(skip, take)

        # STEP 4: Return result
        #   - dp[n] holds max profit using all jobs considered.
        return dp[n]

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.jobScheduling([1,2,3,3], [3,4,5,6], [50,10,40,70]) == 120

    #  Test 2: Edge case - single job
    assert sol.jobScheduling([1], [2], [10]) == 10

    #  Test 3: Tricky/negative - overlapping high-profit vs chain
    assert sol.jobScheduling(
        [1,2,3,4,6], [3,5,10,6,9], [20,20,100,70,60]
```

```
    ) == 150

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**:
`startTime = [1,2,3,3]`, `endTime = [3,4,5,6]`, `profit = [50,10,40,70]`

Step 1: Combine and sort jobs by endTime
- Jobs: `[(1,3,50), (2,4,10), (3,5,40), (3,6,70)]`
- After sorting by `endTime`:
→ `jobs = [(1,3,50), (2,4,10), (3,5,40), (3,6,70)]`
- `end_times = [3, 4, 5, 6]`
- `dp = [0, 0, 0, 0, 0]` (length 5)

Step 2: Process i = 1 (job = (1,3,50))
- `start=1`, `end=3`, `prof=50`
- `skip = dp[0] = 0`
- Find `j`: rightmost job with `endTime <= 1` → none → `j = 0`
- `take = dp[0] + 50 = 50`
- `dp[1] = max(0, 50) = 50`
→ `dp = [0, 50, 0, 0, 0]`

Step 3: Process i = 2 (job = (2,4,10))
- `start=2`
- `skip = dp[1] = 50`
- Find job with `endTime <= 2` → none → `j = 0`
- `take = 0 + 10 = 10`
- `dp[2] = max(50, 10) = 50`
→ `dp = [0, 50, 50, 0, 0]`

Step 4: Process i = 3 (job = (3,5,40))
- `start=3`
- `skip = dp[2] = 50`
- Find job with `endTime <= 3` → job 0 (endTime=3) → `j = 1`
- `take = dp[1] + 40 = 50 + 40 = 90`

- `dp[3] = max(50, 90) = 90`
→ `dp = [0, 50, 50, 90, 0]`

Step 5: Process i = 4 (job = (3,6,70))
- `start=3`
- `skip = dp[3] = 90`
- Find job with `endTime <= 3` → job 0 → `j = 1`
- `take = dp[1] + 70 = 50 + 70 = 120`
- `dp[4] = max(90, 120) = 120`

Final Output: `dp[4] = 120`

Key insight: Even though job 3 (profit 40) looks good, skipping it to take job 4 (profit 70) after job 1 gives higher total.

Complexity Analysis

- **Time Complexity**: `O(n log n)`

  Sorting takes `O(n log n)`. Each of the `n` jobs triggers a binary search (`O(log n)`), so total `O(n log n)`.

- **Space Complexity**: `O(n)`

  We store `jobs`, `end_times`, and `dp` arrays — all linear in `n`.

25. Time Based Key-Value Store

Pattern: Arrays & Hashing + Binary Search

Problem Statement

Design a time-based key-value data structure that can store multiple values for the same key at different timestamps and retrieve the key's value at a certain timestamp.

Implement the `TimeMap` class:
- `TimeMap()` Initializes the object.
- `void set(String key, String value, int timestamp)` Stores the key `key` with the value `value` at the given time `timestamp`.
- `String get(String key, int timestamp)` Returns a value such that `set` was called previously, with `timestamp_prev <= timestamp`. If there are multiple such values, it returns the value associated with the largest `timestamp_prev`. If there is no such value, return `""`.

Sample Input & Output

```
Input:
["TimeMap", "set", "get", "get", "set", "get", "get"]
[[], ["foo", "bar", 1], ["foo", 1], ["foo", 3], ["foo", "bar2", 4], ["foo", 4], ["foo", 5]]

Output:
[null, null, "bar", "bar", null, "bar2", "bar2"]

Explanation:
- After set("foo", "bar", 1), get("foo", 1) → "bar"
- get("foo", 3) → no entry at 3, but 1  3 → "bar"
- After set("foo", "bar2", 4), get("foo", 4) → "bar2"
- get("foo", 5) → latest  5 is 4 → "bar2"
```

```
Input:
["TimeMap", "set", "set", "get", "get", "get"]
[[], ["love", "high", 10], ["love", "low", 20], ["love", 5], ["love", 10], ["love", 15]]

Output:
[null, null, null, "", "high", "high"]
```

```
Input:
["TimeMap", "get"]
[[], ["nonexistent", 1]]

Output:
[null, ""]
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List, Dict
import bisect

class TimeMap:
    def __init__(self):
        # STEP 1: Initialize structures
        #   - Use dict to map each key to a list of (timestamp, value)
        #   - Each list is kept sorted by timestamp (since set is
        #     called in increasing timestamp order per problem)
        self.store: Dict[str, List[tuple]] = {}

    def set(self, key: str, value: str, timestamp: int) -> None:
        # STEP 2: Append new (timestamp, value) to key's list
        #   - No need to sort; timestamps are strictly increasing
        if key not in self.store:
            self.store[key] = []
        self.store[key].append((timestamp, value))

    def get(self, key: str, timestamp: int) -> str:
        # STEP 3: Retrieve value for key at or before timestamp
        #   - If key missing → return ""
        #   - Use binary search to find rightmost timestamp  given
        if key not in self.store:
            return ""

        entries = self.store[key]
        # Extract timestamps for binary search
        timestamps = [ts for ts, _ in entries]
```

```
        # Find insertion point: index of first > timestamp
        idx = bisect.bisect_right(timestamps, timestamp)

        # STEP 4: Return result
        #   - If idx == 0 → no timestamp  given → return ""
        #   - Else return value at idx - 1
        if idx == 0:
            return ""
        return entries[idx - 1][1]

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    #  Test 1: Normal case
    tm1 = TimeMap()
    tm1.set("foo", "bar", 1)
    assert tm1.get("foo", 1) == "bar"
    assert tm1.get("foo", 3) == "bar"
    tm1.set("foo", "bar2", 4)
    assert tm1.get("foo", 4) == "bar2"
    assert tm1.get("foo", 5) == "bar2"
    print("  Test 1 passed")

    #  Test 2: Edge case - timestamp too early
    tm2 = TimeMap()
    tm2.set("love", "high", 10)
    tm2.set("love", "low", 20)
    assert tm2.get("love", 5) == ""
    assert tm2.get("love", 10) == "high"
    assert tm2.get("love", 15) == "high"
    print("  Test 2 passed")

    #  Test 3: Tricky/negative - nonexistent key
    tm3 = TimeMap()
    assert tm3.get("nonexistent", 1) == ""
    print("  Test 3 passed")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 1** step by step:

1. `tm1 = TimeMap()`
   - `self.store = {}` (empty dictionary)
2. `tm1.set("foo", "bar", 1)`
   - `"foo"` not in `store` → create new list
   - Append `(1, "bar")`
   - Now: `store = {"foo": [(1, "bar")]}`
3. `tm1.get("foo", 1)`
   - `"foo"` exists → `entries = [(1, "bar")]`
   - `timestamps = [1]`
   - `bisect_right([1], 1)` → returns `1` (insert after index 0)
   - `idx = 1` → not 0 → return `entries[0][1] = "bar"`
4. `tm1.get("foo", 3)`
   - `timestamps = [1]`
   - `bisect_right([1], 3)` → returns `1`
   - Return `entries[0][1] = "bar"`
5. `tm1.set("foo", "bar2", 4)`
   - Append `(4, "bar2")`
   - Now: `store = {"foo": [(1, "bar"), (4, "bar2")]}`
6. `tm1.get("foo", 4)`
   - `timestamps = [1, 4]`
   - `bisect_right([1,4], 4)` → returns `2`
   - Return `entries[1][1] = "bar2"`
7. `tm1.get("foo", 5)`

- `bisect_right([1,4], 5)` → returns `2`
- Return `entries[1][1] = "bar2"`

Final state: all assertions pass → prints “ Test 1 passed”.

Complexity Analysis

- **Time Complexity**:
  - `set`: `O(1)`
    > Appending to list is constant time.
  - `get`: `O(log n)` per key
    > Binary search (`bisect_right`) on list of size `n` (number of values for that key).
- **Space Complexity**: `O(n)`
  > Store every `(timestamp, value)` pair once, where `n` is total number of `set` calls.
  > Hash map + lists scale linearly with input.