Binary Search: Understanding and Application

Binary Search

Binary Search is conceptually straightforward. It splits the search space into two halves, keeping only the half that potentially contains the target, and discards the other half. This process reduces the search space by half at each step, changing the time complexity from linear (O(n)) to logarithmic (O(log n)). However, implementing a bug-free version can be challenging. Common issues include:

- **Loop exit condition:** Should we use `left < right` or `left <= right`?
- **Boundary initialization:** How to initialize `left` and `right`?
- **Boundary updates:** Should we use `left = mid`, `left = mid + 1`, `right = mid`, or `right = mid - 1`?

A common misconception is that binary search is only applicable for simple problems like finding a specific value in a sorted array. In fact, it can be applied to much more complex scenarios.

Generalized Binary Search Template

Binary search often focuses on the following task:

> Minimize (k), such that `condition(k)` is `True`.

Here's the template:

```
def binary_search(array) -> int:
    def condition(value) -> bool:
        pass  # Define the condition logic

    # Initialize boundaries for the search space
    # Define the search space
    left, right = min(search_space), max(search_space)
```

```
    # Continue until the search space is narrowed down to one element
    while left < right:
        # Calculate the middle index to prevent overflow
        mid = left + (right - left) // 2
        if condition(mid):  # If condition is met, shrink the right boundary
            right = mid
        else:  # If condition is not met, shrink the left boundary
            left = mid + 1
    return left  # Return the smallest k that satisfies the condition
```

Key Points

1. **Initialize boundaries:** Define `left` and `right` to include all possible values in the search space.
2. **Return value:** After exiting the loop, `left` is the minimal (k) satisfying `condition(k)`. Adjust return value as needed.
3. **Condition function:** This is the core logic and often the hardest part to define.

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 `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 = [5], target = 5
Output: 0
Explanation: Single-element array; target matches the only element.
```

```
Input: nums = [1, 3, 5], target = 2
Output: -1
Explanation: Target 2 is not in the array;
binary search correctly concludes absence.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def search(self, nums: List[int], target: int) -> int:
        # STEP 1: Initialize pointers to track search boundaries
        #   - left starts at 0 (beginning of array)
        #   - right starts at last valid index
        left = 0
        right = len(nums) - 1

        # STEP 2: Main loop - continue while search space is valid
        #   - Invariant: target (if present) must be in [left, right]
        #   - Loop ends when left > right → target not found
        while left <= right:
            # STEP 3: Compute mid safely to avoid overflow
            #   - Using left + (right - left) // 2 prevents int overflow
            #     (not critical in Python, but good practice)
            mid = left + (right - left) // 2
```

```python
            # STEP 4: Compare mid-value with target
            if nums[mid] == target:
                return mid  # Found! Return index immediately
            elif nums[mid] < target:
                left = mid + 1  # Target must be in right half
            else:
                right = mid - 1  # Target must be in left half

        # STEP 5: Return -1 if loop exits without finding target
        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 - single element match
    assert sol.search([5], 5) == 0

    #  Test 3: Tricky/negative - target not present
    assert sol.search([1, 3, 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

We'll trace `search([1, 3, 5], 2)` step by step.

Initial state:
- `nums = [1, 3, 5]`, `target = 2`
- `left = 0`, `right = 2`

Step 1: Enter `while left <= right` → `0 <= 2` → **true**
- Compute `mid = 0 + (2 - 0) // 2 = 1`
- `nums[mid] = nums[1] = 3`
- Compare: `3 == 2`? → No
- `3 < 2`? → No → go to `else`
- Update: `right = mid - 1 = 0`

State: `left = 0`, `right = 0`

Step 2: Loop condition: `0 <= 0` → **true**
- `mid = 0 + (0 - 0) // 2 = 0`
- `nums[0] = 1`
- `1 == 2`? → No
- `1 < 2`? → **Yes** → `left = mid + 1 = 1`

State: `left = 1`, `right = 0`

Step 3: Loop condition: `1 <= 0` → **false** → exit loop
- Return `-1`

Final output: `-1`
Why? The algorithm correctly narrowed the search until the interval became empty, proving `2` is not in `[1, 3, 5]`.

Complexity Analysis

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

  Each iteration cuts the search space in half. For `n` elements, at most `log (n)` steps are needed.

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

  Only a constant amount of extra space is used (`left`, `right`, `mid`), regardless of input size.

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, isBadVersion(5) → true,
isBadVersion(4) → true → so 4 is the first bad version.
```

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

```
Input: n = 10, bad = 2
Output: 2
Explanation: Version 1 is good, version 2 is the first bad.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

# Mock API - will be overridden in tests
def isBadVersion(version: int) -> bool:
    raise NotImplementedError("This is a mock API")

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

        # STEP 2: Main loop - binary search invariant:
        #   - At each step, [left, right] always contains
        #     the first bad version.
        #   - Loop ends when left == right → answer found.
        while left < right:
            # STEP 3: Compute mid without overflow
            #   - mid = left + (right - left) // 2 ensures
            #     safe integer arithmetic.
            mid = left + (right - left) // 2

            # STEP 4: Decision based on API result
            #   - If mid is bad, first bad is at mid or left
            #     → move right to mid (include mid).
            #   - Else, first bad is strictly right of mid
            #     → move left to mid + 1.
            if isBadVersion(mid):
                right = mid
            else:
                left = mid + 1

        # STEP 5: Return result
        #   - When loop exits, left == right == first bad
        return left

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

```python
# Test 1: Normal case
def isBadVersion(version: int) -> bool:
    return version >= 4
assert sol.firstBadVersion(5) == 4
print("✅ Test 1 passed: n=5, first bad=4")

# Test 2: Edge case - only one version, and it's bad
def isBadVersion(version: int) -> bool:
    return version >= 1
assert sol.firstBadVersion(1) == 1
print("✅ Test 2 passed: n=1, first bad=1")

# Test 3: Tricky case - early bad version
def isBadVersion(version: int) -> bool:
    return version >= 2
assert sol.firstBadVersion(10) == 2
print("✅ Test 3 passed: n=10, first bad=2")
```

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

Example Walkthrough

We'll walk through **Test 1**: `n = 5`, first bad version = 4.

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

Iteration 1:
- `left (1) < right (5)` → enter loop
- `mid = 1 + (5 - 1) // 2 = 1 + 2 = 3`
- Call `isBadVersion(3)` → returns `False`
- So, set `left = mid + 1 = 4`
- **State**: `left = 4`, `right = 5`

Iteration 2:
- `left (4) < right (5)` → enter loop
- `mid = 4 + (5 - 4) // 2 = 4 + 0 = 4`
- Call `isBadVersion(4)` → returns `True`

- So, set `right = mid = 4`
- **State**: `left = 4`, `right = 4`

Loop condition check:
- `left (4) < right (4)` → **false** → exit loop

Return: `left = 4` → correct answer.

Each step halves the search space while preserving the invariant:
> "The first bad version is always in `[left, right]`."

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 uses constant extra space (`left`, `right`, `mid`) — no recursion or auxiliary data structures.

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, such as `pow(x, 0.5)` or `x ** 0.5`.

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, so floor(2.828) = 2.

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

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def mySqrt(self, x: int) -> int:
        # STEP 1: Initialize boundaries
        #   - Search space: [0, x], but we use right = x + 1
        #     to ensure upper bound is exclusive in loop
        left, right = 0, x + 1

        # STEP 2: Main loop - binary search for largest k
        #   - Invariant: answer is in [left, right)
        #   - We find first k where k*k > x, then answer is k-1
        while left < right:
            mid = left + (right - left) // 2
            sq = mid * mid  # Avoid repeated computation

            # STEP 3: Update boundaries based on comparison
            if sq > x:
                right = mid  # Too big → search left half
            else:
                left = mid + 1  # Valid candidate → try larger
```

```
        # STEP 4: Return result
        #   - left is first value where k*k > x
        #   - So left - 1 is largest k with k*k <= x
        return left - 1

# ------------------- 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)}"

    #  Test 3: Tricky/negative (non-perfect square)
    assert sol.mySqrt(8) == 2, f"Expected 2, got {sol.mySqrt(8)}"

    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 `mySqrt(8)` step by step:

Initial State:
- `x = 8`
- `left = 0`, `right = 9` (since `x + 1 = 9`)

Iteration 1:
- Condition: `left (0) < right (9)` → enter loop
- `mid = 0 + (9 - 0) // 2 = 4`
- `sq = 4 * 4 = 16`
- `16 > 8` → set `right = mid = 4`
- **New state**: `left = 0`, `right = 4`

Iteration 2:
- Condition: `0 < 4` → continue
- `mid = 0 + (4 - 0) // 2 = 2`
- `sq = 2 * 2 = 4`
- `4 <= 8` → set `left = mid + 1 = 3`
- **New state**: `left = 3`, `right = 4`

Iteration 3:
- Condition: `3 < 4` → continue
- `mid = 3 + (4 - 3) // 2 = 3`
- `sq = 3 * 3 = 9`
- `9 > 8` → set `right = 3`
- **New state**: `left = 3`, `right = 3`

Loop Ends:
- Now `left == right` → exit loop
- Return `left - 1 = 3 - 1 = 2`

Final Output: `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 + 1), which is logarithmic in input size.

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

  Only uses a constant number of integer variables (`left`, `right`, `mid`, `sq`).
  No recursion or dynamic data 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 = 7
Output: 4
Explanation: 7 is larger than all elements → insert at end (index 4).
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def searchInsert(self, nums: List[int], target: int) -> int:
        # STEP 1: Initialize boundaries
        #   - left = 0, right = len(nums) (exclusive upper bound)
        #   - This ensures we cover insertion at the very end.
        left, right = 0, len(nums)

        # STEP 2: Main loop / recursion
        #   - Invariant: target must be in [left, right)
        #   - Loop continues while search space is non-empty.
        while left < right:
            # STEP 3: Update state / bookkeeping
            #   - Compute mid without overflow
            mid = left + (right - left) // 2
            #   - If nums[mid] >= target, insertion point is at mid
            #     or to the left → move right boundary to mid.
            if nums[mid] >= target:
                right = mid
            else:
                #   - If nums[mid] < target, insertion point is to
                #     the right → move left to mid + 1.
                left = mid + 1

        # STEP 4: Return result
        #   - When loop ends, left == right == insertion index.
        return left

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

    #  Test 1: Normal case - target found
    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

We'll trace `searchInsert([1, 3, 5, 6], 2)` step by step.

Initial state:
- `nums = [1, 3, 5, 6]`, `target = 2`
- `left = 0`, `right = 4` (since `len(nums) = 4`)

Step 1: Enter `while left < right` → `0 < 4` → **true**
- Compute `mid = 0 + (4 - 0) // 2 = 2`
- `nums[mid] = nums[2] = 5`
- Compare: `5 >= 2` → **true** → set `right = mid = 2`
- **State**: `left=0`, `right=2`

Step 2: Loop condition → `0 < 2` → **true**
- `mid = 0 + (2 - 0) // 2 = 1`
- `nums[1] = 3`
- `3 >= 2` → **true** → `right = 1`
- **State**: `left=0`, `right=1`

Step 3: Loop condition → `0 < 1` → **true**
- `mid = 0 + (1 - 0) // 2 = 0`
- `nums[0] = 1`
- `1 >= 2` → **false** → `left = mid + 1 = 1`
- **State**: `left=1`, `right=1`

Step 4: Loop condition → `1 < 1` → **false** → exit loop

Return: `left = 1` → correct insertion index.

Final output: `1`

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 a few integer variables (`left`, `right`, `mid`) — no extra space scaling with input.

5. 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 `D` 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 `D` days.

Sample Input & Output

```
Input: weights = [1,2,3,4,5,6,7,8,9,10], D = 5
Output: 15
Explanation: A ship capacity of 15 is the minimum to ship all
packages in 5 days:
  Day 1: 1+2+3+4+5 = 15
  Day 2: 6+7 = 13
  Day 3: 8
  Day 4: 9
  Day 5: 10
```

```
Input: weights = [3,2,2,4,1,4], D = 3
Output: 6
Explanation:
  Day 1: 3+2+2 = 7 → too much if capacity=6? Wait-actually:
  With capacity=6: Day1=3+2=5 (can't add next 2 → 5+2=7>6),
  so Day1=3+2, Day2=2+4, Day3=1+4 → works.
```

```
Input: weights = [1,2,3,1,1], D = 4
Output: 3
Explanation: Max single weight is 3,
and we can ship as [1+2], [3], [1], [1] → 4 days.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def shipWithinDays(self, weights: List[int], D: int) -> int:
        # STEP 1: Define feasibility helper
        #   - Checks if a given capacity can ship all weights in <= D days
        def feasible(capacity: int) -> bool:
            days = 1        # Start counting from day 1
            current_load = 0
            for weight in weights:
                current_load += weight
                # If current load exceeds capacity, start a new day
                if current_load > capacity:
                    current_load = weight  # This weight starts next day
                    days += 1
                    # Early exit if we exceed allowed days
                    if days > D:
                        return False
            return True

        # STEP 2: Set binary search bounds
        #   - Lower bound: heaviest single package (must fit)
        #   - Upper bound: total sum (ship everything in 1 day)
```

```
        left = max(weights)
        right = sum(weights)

        # STEP 3: Binary search for minimum feasible capacity
        #   - Invariant: answer is in [left, right]
        while left < right:
            mid = left + (right - left) // 2
            if feasible(mid):
                right = mid      # Try smaller capacity
            else:
                left = mid + 1   # Need larger capacity

        # STEP 4: Return left (smallest feasible capacity)
        return left

# ------------------- 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 - D equals number of packages
    #   → capacity must be at least max(weights)
    assert sol.shipWithinDays([1,2,3,1,1], 5) == 3

    #  Test 3: Tricky/negative - D = 1 (must take all)
    assert sol.shipWithinDays([1,2,3,4,5], 1) == 15
```

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

Example Walkthrough

We'll trace `weights = [1,2,3,4,5,6,7,8,9,10]`, `D = 5`.

Goal: Find smallest capacity so we can ship in 5 days.

Step 1: Set bounds

- `left = max(weights) = 10`
- `right = sum(weights) = 55`

Step 2: First binary search iteration
- `mid = 10 + (55-10)//2 = 32`
- Check `feasible(32)`: - Day 1: add 1→2→3→…→10 → total=55 → but 55>32? Yes → so we reset when exceeding. - Actually:
- Start day=1, load=0
- Add 1→2→3→4→5→6 → load=21
- Add 7 → 28
- Add 8 → 36 > 32 → so start new day: day=2, load=8
- Add 9 → 17
- Add 10 → 27 → done
- Total days = 2 → 5 → **feasible!**
- So set `right = 32`

Step 3: Continue binary search
- Now `left=10`, `right=32` → `mid=21`
- `feasible(21)`: - Day1: 1+2+3+4+5+6=21 → ok
- Day2: 7+8=15 → add 9? 15+9=24>21 → so Day2 ends at 7+8
→ Day3: 9 → add 10? 19 21 → Day3: 9+10
- Total days = 3 → feasible → `right = 21`

Step 4: Keep narrowing
Eventually, we test `mid=15`: - Day1: 1+2+3+4+5=15
- Day2: 6+7=13 → add 8? 21>15 → no → Day2 ends
- Day3: 8 → add 9? 17>15 → no → Day3=8
- Day4: 9
- Day5: 10 → total=5 days → feasible!

Now test `mid=14`: - Day1: 1+2+3+4=10 → add 5? 15>14 → so Day1=1-4
- Day2: 5+6=11 → add 7? 18>14 → Day2=5+6
- Day3: 7 → add 8? 15>14 → Day3=7
- Day4: 8 → add 9? 17>14 → Day4=8
- Day5: 9 → add 10? 19>14 → Day5=9
- Day6: 10 → **6 days > 5** → not feasible!

So answer is **15**.

Complexity Analysis

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

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

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

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

6. Split Array Largest Sum

Pattern: Binary Search on Answer + Greedy Feasibility Check

Problem Statement

Given an integer array `nums` and an integer `m`, split `nums` into `m` **non-empty** contiguous subarrays. Minimize the **largest sum** among these subarrays.

Return the **minimum possible value** of the largest subarray sum after splitting into exactly `m` subarrays.

Constraints: - `1 nums.length 1000 - 0 nums[i] 10 - 1 m min(50, nums.length)`

Sample Input & Output

```
Input: nums = [7,2,5,10,8], m = 2
Output: 18
Explanation: Split into [7,2,5] and [10,8]. Sums = 14 and 18 → max = 18.
```

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

```
Input: nums = [1], m = 1
Output: 1
Explanation: Only one element, one subarray → sum = 1.
```

LeetCode Editorial Solution + Inline Tests

```
from typing import List

class Solution:
    def splitArray(self, nums: List[int], m: int) -> int:
        # STEP 1: Define feasibility function
        #   - Checks if we can split nums into  m subarrays
        #     such that each subarray sum  threshold.
        def feasible(threshold: int) -> bool:
            count = 1      # At least one subarray
            total = 0      # Running sum of current subarray
            for num in nums:
                total += num
                # If current sum exceeds threshold,
                # start a new subarray with this num
                if total > threshold:
                    total = num
                    count += 1
                    # Early exit: too many subarrays
                    if count > m:
                        return False
            return True

        # STEP 2: Binary search over answer space
        #   - Lower bound: max(nums) (each subarray must hold
        #     at least one element → max element is minimum
        #     possible largest sum)
        #   - Upper bound: sum(nums) (one subarray holds all)
        left, right = max(nums), sum(nums)

        # STEP 3: Search for smallest feasible threshold
        while left < right:
            mid = left + (right - left) // 2
            if feasible(mid):
                right = mid   # Try smaller max sum
            else:
                left = mid + 1  # Need larger threshold

        # STEP 4: Return minimal feasible largest sum
        return left
```

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

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

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

    #  Test 3: Tricky case - many small numbers, m=3
    assert sol.splitArray([1,2,3,4,5], 3) == 6  # [1,2,3],[4],[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 `splitArray([7,2,5,10,8], m=2)` step by step.

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

Binary Search Iterations:

Iteration 1: - `left=10`, `right=32` → `mid = 10 + (32-10)//2 = 21` - Call `feasible(21)`: - `count=1`, `total=0` - Add 7 → total=7 21 → OK - Add 2 → total=9 21 → OK - Add 5 → total=14 21 → OK - Add 10 → total=24 >21 → **start new subarray** - `total=10`, `count=2` - Add 8 → total=18 21 → OK - Final `count=2` `m=2` → **feasible!** - So set `right = 21`

Iteration 2: - `left=10`, `right=21` → `mid = 10 + (21-10)//2 = 15` - `feasible(15)`: - 7+2+5 = 14 15 → OK - +10 → 24 >15 → new subarray (`count=2`, total=10) - +8 → 18 >15 → new subarray (`count=3`) - But `m=2`, so `count>m` → **not feasible** - Set `left = 15 + 1 = 16`

Iteration 3: - `left=16`, `right=21` → `mid = 18` - `feasible(18)`: - 7+2+5 = 14 18 - +10 → 24 >18 → new subarray (`count=2`, total=10) - +8 → 18 18 → OK - `count=2` → **feasible** - Set `right = 18`

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

Now `left == right == 18` → loop ends → return **18**.

Final answer: **18**

Complexity Analysis

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

  Binary search runs in `O(log(total_range))`, where `total_range = sum(nums) - max(nums)`.
  Each `feasible()` call scans all `n` elements → `O(n)`.
  Combined: `O(n log(sum))`.

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

  Only a few integer variables used (`count`, `total`, `left`, `right`, etc.).
  No extra space proportional to input size.

7. 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 speed 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 → speed >= max(piles) = 30.
```

```
Input: piles = [30,11,23,4,20], H = 6
Output: 23
Explanation: Speed 23 → [2,1,1,1,1] = 6 hours.
Speed 22 would need 2 for 30 → 7 hours > 6.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def minEatingSpeed(self, piles: List[int], H: int) -> int:
        # STEP 1: Initialize structures
        #   - checks if speed allows finishing in H hours
        #   - (pile - 1) // speed + 1 == ceil(pile / speed)
        def feasible(speed) -> bool:
            total_hours = 0
            for pile in piles:
                # Compute hours for this pile with ceiling division
                hours_for_pile = (pile - 1) // speed + 1
                total_hours += hours_for_pile
```

```
                # Early exit if already over H (minor optimization)
                if total_hours > H:
                    return False
            return total_hours <= H

        # STEP 2: Main loop / recursion
        #   - Binary search over possible speeds: [1, max(piles)]
        #   - Invariant: answer is always in [left, right]
        left, right = 1, max(piles)
        while left < right:
            # Avoid overflow; same as (left+right)//2
            mid = left + (right - left) // 2
            if feasible(mid):
                # STEP 3: Update state / bookkeeping
                #   - mid works → try smaller speeds (answer   mid)
                right = mid
            else:
                # mid too slow → must go faster (answer   mid + 1)
                left = mid + 1

        # STEP 4: Return result
        #   - left == right is the minimal feasible speed
        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 (must eat each in 1 hour)
    assert sol.minEatingSpeed([30,11,23,4,20], 5) == 30

    #  Test 3: Tricky/negative - just enough time to require optimal speed
    assert sol.minEatingSpeed([30,11,23,4,20], 6) == 23

    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 `minEatingSpeed([3,6,7,11], 8)` step by step.

Initial state:
- `piles = [3,6,7,11]`, `H = 8`
- `left = 1`, `right = 11` (max pile)

Iteration 1:
- `left=1`, `right=11` → `mid = 1 + (11-1)//2 = 6`
- Call `feasible(6)`: - pile=3 → (3-1)//6 + 1 = 0 + 1 = 1
- pile=6 → (6-1)//6 + 1 = 0 + 1 = 1
- pile=7 → (7-1)//6 + 1 = 1 + 1 = 2
- pile=11 → (11-1)//6 + 1 = 1 + 1 = 2
- Total = 1+1+2+2 = 6 8 → **feasible!**
- So set `right = mid = 6`

State: `left=1`, `right=6`

Iteration 2:
- `mid = 1 + (6-1)//2 = 3`
- `feasible(3)`: - 3 → (2//3)+1 = 1
- 6 → (5//3)+1 = 1+1 = 2
- 7 → (6//3)+1 = 2+1 = 3
- 11 → (10//3)+1 = 3+1 = 4
- Total = 1+2+3+4 = 10 > 8 → **not feasible**
- So set `left = 3 + 1 = 4`

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

Iteration 3:
- `mid = 4 + (6-4)//2 = 4 + 1 = 5`
- `feasible(5)`: - 3 → 1
- 6 → (5//5)+1 = 1+1 = 2
- 7 → (6//5)+1 = 1+1 = 2
- 11 → (10//5)+1 = 2+1 = 3
- Total = 1+2+2+3 = 8 8 → **feasible**
- Set `right = 5`

State: `left=4`, `right=5`

Iteration 4:
- `mid = 4 + (5-4)//2 = 4`
- `feasible(4)`: - 3 → 1
- 6 → (5//4)+1 = 1+1 = 2
- 7 → (6//4)+1 = 1+1 = 2
- 11 → (10//4)+1 = 2+1 = 3
- Total = 8 → **feasible**
- Set `right = 4`

Now: `left=4`, `right=4` → loop ends

Return: `4`

Final output: `4`

Key takeaway: We **binary search on the answer space** (speed), not the input array. The feasibility function uses ceiling division without floats.

Complexity Analysis

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

  `n = len(piles)`, `m = max(piles)`. Binary search runs `O(log m)` iterations. Each `feasible()` call scans all `n` piles → `O(n)` per check.

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

  Only a few integer variables used. No extra space scales with input (recursion stack is avoided via iterative binary search).

8. Minimum Number of Days to Make m Bouquets

Pattern: Binary Search on Answer + Greedy Feasibility Check

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 with 1 flower.
The earliest day we can pick 3 flowers is day 3
(flowers at indices 0, 2, and 4 have bloomed by then).
```

```
Input: bloomDay = [1,10,2,9,3,8,4,7,5,6], m = 4, k = 2
Output: 9
Explanation: Need 4 bouquets × 2 adjacent flowers = 8 total.
By day 9, we can form [2,9], [3,8], [4,7], [5,6] as adjacent groups.
```

```
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

```
from typing import List

class Solution:
    def minDays(self, bloomDay: List[int], m: int, k: int) -> int:
        # STEP 1: Check if it's even possible to form m bouquets
        #   - Each bouquet needs k flowers → total needed = m * k
        #   - If garden has fewer flowers, impossible.
        if len(bloomDay) < m * k:
            return -1

        # STEP 2: Define feasibility function
        #   - Given a candidate 'days', can we form  m bouquets?
        def feasible(days: int) -> bool:
            bouquets = 0      # Count of completed bouquets
            flowers = 0       # Current streak of adjacent bloomed flowers

            for bloom in bloomDay:
                if bloom > days:
                    # Flower not bloomed yet → break adjacency
                    flowers = 0
                else:
                    # Flower is bloomed → extend current streak
                    flowers += 1
                    if flowers == k:
                        # Completed one bouquet
                        bouquets += 1
                        flowers = 0  # Reset for next bouquet
                        if bouquets >= m:
                            return True  # Early exit
            return bouquets >= m

        # STEP 3: Binary search over answer space
        #   - Minimum possible day: 1
        #   - Maximum needed: max(bloomDay) (all flowers bloomed)
        left, right = 1, max(bloomDay)
        while left < right:
            mid = left + (right - left) // 2
            if feasible(mid):
                # If feasible at 'mid', try fewer days
                right = mid
```

```
            else:
                # Not enough bouquets → need more days
                left = mid + 1

        # STEP 4: Return minimal feasible day
        return left

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

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

    #  Test 2: Edge case - impossible due to insufficient flowers
    assert sol.minDays([1,10,3,10,2], 3, 2) == -1

    #  Test 3: Tricky case - requires adjacent grouping
    assert sol.minDays([1,10,2,9,3,8,4,7,5,6], 4, 2) == 9

    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**:
`bloomDay = [1,10,3,10,2]`, `m = 3`, `k = 1`

Goal: Find the **minimum day** such that we can pick **3 flowers** (since `k=1`, adjacency doesn't matter).

Step 1: Check feasibility

- Total flowers needed = `m * k = 3 * 1 = 3`
- Garden has 5 flowers → possible.

Step 2: Binary search bounds

- `left = 1`, `right = max(bloomDay) = 10`

Step 3: First iteration — `mid = 1 + (10-1)//2 = 5`

- Call `feasible(5)`:
  - Flower 0: `1  5` → `flowers = 1` → bouquet! (`bouquets=1`, reset)
  - Flower 1: `10 > 5` → reset → `flowers = 0`
  - Flower 2: `3  5` → `flowers = 1` → bouquet! (`bouquets=2`)
  - Flower 3: `10 > 5` → reset
  - Flower 4: `2  5` → `flowers = 1` → bouquet! (`bouquets=3`)
  - Returns `True` → set `right = 5`

Step 4: Next: `left=1`, `right=5` → `mid = 3`

- `feasible(3)`:
  - Flower 0: `1  3` → bouquet #1
  - Flower 1: `10 > 3` → reset
  - Flower 2: `3  3` → bouquet #2
  - Flower 3: `10 > 3` → reset
  - Flower 4: `2  3` → bouquet #3 → `True`
  - → `right = 3`

Step 5: Now `left=1`, `right=3` → `mid = 2`

- `feasible(2)`:
  - Flower 0: `1  2` → bouquet #1
  - Flower 1: `10 > 2` → reset
  - Flower 2: `3 > 2` → reset
  - Flower 3: `10 > 2` → reset
  - Flower 4: `2  2` → bouquet #2 → only $2 < 3$ → `False`
  - → `left = 3`

Step 6: Now `left == right == 3` → exit loop → return `3`

Final answer: **3**

Key Insight:
We're **not simulating day-by-day**. Instead, we **guess a day** and **greedily check** if enough **adjacent bloomed flowers** exist to form bouquets. Binary search finds the **smallest valid guess**.

Complexity Analysis

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

  Binary search runs in `O(log(max))`. Each `feasible()` call scans all `n` flowers → `O(n)`. Combined: `O(n log(max))`.

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

  Only a few integer variables used. No extra space proportional to input size.

9. Find 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 = 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] → 6th = 6
```

```
Input: m = 1, n = 1, k = 1
Output: 1
Explanation: Only one element: 1×1 = 1
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def findKthNumber(self, m: int, n: int, k: int) -> int:
        # STEP 1: Define helper to count numbers <= 'num'
        #   - For each row i (1 to m), max value in row is i*n
        #   - Count of values <= num in row i = min(num // i, n)
        def enough(num: int) -> bool:
            count = 0
            for i in range(1, m + 1):
                add = min(num // i, n)
                if add == 0:  # No more contributions from later rows
                    break
                count += add
            return count >= k  # True if at least k numbers <= num

        # STEP 2: Binary search over answer space [1, m*n]
        #   - Invariant: answer is in [left, right]
        #   - 'enough(mid)' tells us if mid is too large or not
```

```
        left, right = 1, m * n
        while left < right:
            mid = left + (right - left) // 2
            if enough(mid):
                right = mid  # mid might be answer; keep it
            else:
                left = mid + 1  # mid too small; discard it

        # STEP 3: Return converged value
        #   - Loop ends when left == right == smallest valid answer
        return left

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

    #  Test 1: Normal case
    assert sol.findKthNumber(3, 3, 5) == 3

    #  Test 2: Edge case - smallest table
    assert sol.findKthNumber(1, 1, 1) == 1

    #  Test 3: Tricky/negative - k equals total elements
    assert sol.findKthNumber(2, 3, 6) == 6

    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 `findKthNumber(3, 3, 5)` step by step.

Initial state:
- `m = 3`, `n = 3`, `k = 5`
- Search space: `left = 1`, `right = 9`

Iteration 1:
- `mid = 1 + (9 - 1) // 2 = 5`
- Call `enough(5)`: - Row 1 (`i=1`): `min(5//1, 3) = min(5,3) = 3` → `count = 3` - Row 2 (`i=2`): `min(5//2, 3) = min(2,3) = 2` → `count = 5` - Row 3 (`i=3`): `min(5//3, 3) = min(1,3) = 1` → `count = 6` - `6 >= 5` → returns `True` - Since `enough(5)` is `True`, set `right = 5`

State: `left = 1`, `right = 5`

Iteration 2:
- `mid = 1 + (5 - 1) // 2 = 3`
- Call `enough(3)`: - Row 1: `min(3//1, 3) = 3` → `count = 3` - Row 2: `min(3//2, 3) = 1` → `count = 4` - Row 3: `min(3//3, 3) = 1` → `count = 5` - `5 >= 5` → `True` - Set `right = 3`

State: `left = 1`, `right = 3`

Iteration 3:
- `mid = 1 + (3 - 1) // 2 = 2`
- Call `enough(2)`: - Row 1: `min(2//1, 3) = 2` → `count = 2` - Row 2: `min(2//2, 3) = 1` → `count = 3` - Row 3: `min(2//3, 3) = 0` → break (early exit) - `3 < 5` → returns `False` - Set `left = 2 + 1 = 3`

State: `left = 3`, `right = 3` → loop ends

Return: `3`

Final output: `3` — matches expected 5th smallest.

Complexity Analysis

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

  Binary search runs in `O(log(m·n))`. Each `enough()` call loops up to `m` rows. Early break helps in practice but worst-case still `O(m)`.

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

  Only a few integer variables used; no extra data structures scale with input.

10. Find K-th Smallest Pair Distance

Pattern: Binary Search on Answer + Sliding Window

Problem Statement

> Given an integer array `nums` and an integer `k`, return the *k-th smallest distance among all pairs* defined as `|nums[i] - nums[j]|` where `0 <= i < j < len(nums)`.

The distance of a pair `(i, j)` is the absolute difference between `nums[i]` and `nums[j]`.

Sample Input & Output

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

```
Input: nums = [1, 1, 1], k = 2
Output: 0
Explanation: All pairs have distance 0. Any k  3 returns 0.
```

```
Input: nums = [1, 6, 1], k = 3
Output: 5
Explanation: Pairs → (0,1):5, (0,2):0, (1,2):5 → distances [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 to enable sliding window over distances
        #   - Sorting ensures nums[j] - nums[i] is non-negative
        #   - Allows us to count pairs with distance <= D efficiently
        nums.sort()

        # STEP 2: Helper to check if  k pairs have distance  'dist'
        #   - Uses sliding window: for each j, find smallest i such that
        #     nums[j] - nums[i] <= dist → all indices from i to j-1 work
        def enough(dist: int) -> bool:
            count, i = 0, 0
            for j in range(len(nums)):
                # Shrink window from left until valid
                while nums[j] - nums[i] > dist:
                    i += 1
                # All pairs (i, j), (i+1, j), ..., (j-1, j) are valid
                count += j - i
            return count >= k

        # STEP 3: Binary search over possible distances
        #   - Min distance = 0, max = nums[-1] - nums[0]
        #   - We seek smallest distance D where enough(D) is True
        left, right = 0, nums[-1] - nums[0]
        while left < right:
            mid = left + (right - left) // 2
            if enough(mid):
                right = mid      # D might be smaller
            else:
                left = mid + 1   # Need larger D

        # STEP 4: Return found minimal valid distance
        return left

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

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

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

    #   Test 3: Tricky/negative - k at upper bound
    assert sol.smallestDistancePair([1, 6, 1], 3) == 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 `smallestDistancePair([1, 3, 1], k=1)` step by step.

1. **Initial state**:
   `nums = [1, 3, 1]`, `k = 1`
2. **After sorting**:
   `nums = [1, 1, 3]`
   → Now distances are monotonic as we move right.
3. **Binary search setup**:
   `left = 0`, `right = 3 - 1 = 2`
4. **First binary search iteration**:
   - `mid = 0 + (2 - 0) // 2 = 1`
   - Call `enough(1)`:
     - `i = 0`, `count = 0`
     - `j = 0`: window = [0,0] → `3-1=2` not relevant yet → `count += 0`
     - `j = 1`: `nums[1] - nums[0] = 0  1` → `count += 1 - 0 = 1`
     - `j = 2`: `nums[2] - nums[0] = 2 > 1` → increment `i` to 1
       Now `nums[2] - nums[1] = 2 > 1` → increment `i` to 2
       Now `i == j`, so `count += 2 - 2 = 0`
     - Total `count = 1` → `enough(1)` returns `True`
   - So set `right = 1`
5. **Second iteration**:

- `left = 0`, `right = 1` → `mid = 0`
- Call `enough(0)`:
  - `j=0`: count += 0
  - `j=1`: `1-1=0  0` → count += 1
  - `j=2`: `3-1=2 > 0` → move `i` to 1 → `3-1=2 > 0` → move `i` to 2 → count += 0
  - Total `count = 1` → `enough(0)` is `True`
- Set `right = 0`

6. **Loop ends** (`left == right == 0`) → return `0`

Final output: `0` — matches expected.

Key insight: **We never enumerate all pairs**. Instead, we *count* how many pairs have distance D using a sliding window, then binary search the minimal D that gives k pairs.

Complexity Analysis

- **Time Complexity**: `O(n log n + n log W)`
  - `O(n log n)` for sorting.
  - Binary search runs `O(log W)` times, where `W = max - min`.
  - Each `enough()` call is `O(n)` via sliding window.
  - Total: `O(n log n + n log W)`.
- **Space Complexity**: `O(1)`
  - Only a few extra variables (`i`, `j`, `count`, `left`, `right`, etc.).
  - Sorting is in-place (Python's Timsort uses `O(n)` worst-case, but we treat input as mutable per LeetCode norms).
  - No additional data structures scale with input beyond sorting. "'

11. 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`.

Return the `n`-th ugly number in ascending order.

Sample Input & Output

```
Input: n = 3, a = 2, b = 3, c = 5
Output: 4
Explanation: The first 3 ugly numbers are 2, 3, 4 (5 is the 4th).
```

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

```
Input: n = 1, a = 1, b = 1, c = 1
Output: 1
Explanation: Edge case - all divisors are 1; 1st ugly number is 1.
```

LeetCode Editorial Solution + Inline Tests

```python
from math import gcd
from typing import List

class Solution:
    def nthUglyNumber(self, n: int, a: int, b: int, c: int) -> int:
        # STEP 1: Precompute LCMs for inclusion-exclusion
        #   - ab, ac, bc: pairwise LCMs
        #   - abc: LCM of all three
        ab = a * b // gcd(a, b)
        ac = a * c // gcd(a, c)
        bc = b * c // gcd(b, c)
```

```
        abc = a * bc // gcd(a, bc)

        # STEP 2: Define helper to count ugly numbers  num
        #   - Uses inclusion-exclusion to avoid double-counting
        def enough(num: int) -> bool:
            count = (
                num // a + num // b + num // c
                - num // ab - num // ac - num // bc
                + num // abc
            )
            return count >= n

        # STEP 3: Binary search over answer space
        #   - Left bound: 1 (smallest positive)
        #   - Right bound: 2e9 (safe upper bound per constraints)
        left, right = 1, 2 * 10**9
        while left < right:
            mid = left + (right - left) // 2
            if enough(mid):
                right = mid      # mid might be answer; keep it
            else:
                left = mid + 1   # mid too small; discard it

        # STEP 4: Return smallest number with  n ugly numbers  it
        return left

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

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

    #  Test 2: Edge case (n=1)
    assert sol.nthUglyNumber(1, 1, 1, 1) == 1

    #  Test 3: Tricky case with overlapping multiples
    assert sol.nthUglyNumber(4, 2, 3, 4) == 6

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

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

instant feedback.

Example Walkthrough

We'll trace `nthUglyNumber(n=3, a=2, b=3, c=5)` step by step.

Step 1: Compute LCMs - `gcd(2,3)=1` → `ab = 2*3//1 = 6` - `gcd(2,5)=1` → `ac = 10` - `gcd(3,5)=1` → `bc = 15` - `gcd(2,15)=1` → `abc = 2*15//1 = 30`

Step 2: Binary search setup - `left = 1`, `right = 2_000_000_000`

Step 3: First iteration - `mid = 1 + (2e9 - 1)//2` `1e9` - Call `enough(1e9)`: - `1e9//2 = 500_000_000` - `1e9//3` `333_333_333` - `1e9//5 = 200_000_000` - Subtract overlaps: `1e9//6` `166_666_666`, etc. - Add back `1e9//30` `33_333_333` - Total 733 million 3 → `True` - So `right = 1e9`

... (search narrows down) ...

Step N: When mid = 4 - `enough(4)`: - `4//2 = 2` (2,4) - `4//3 = 1` (3) - `4//5 = 0` - Overlaps: `4//6=0`, `4//10=0`, `4//15=0`, `4//30=0` - Total = 2 + 1 + 0 = 3 3 → `True` - So `right = 4`

Step N+1: mid = 3 - `enough(3)`: - `3//2=1` (2), `3//3=1` (3), `3//5=0` - Total = 2 < 3 → `False` - So `left = 4`

Now `left == right == 4` → loop ends → return `4`.

Final output: `4`

Key takeaway: Binary search finds the **smallest** number where count n, which is exactly the n-th ugly number.

Complexity Analysis

- **Time Complexity**: `O(log(2×10 ))` `O(31)` → `O(1)`

  Binary search runs in logarithmic time over fixed range (up to 2e9).
  Each `enough()` call does O(1) arithmetic.

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

  Only a few integer variables stored; no input-dependent structures.

12. 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. Find the **smallest** such divisor that the sum is **less than or equal to** `threshold`.

Each division result is rounded **up** to the nearest integer (i.e., ceiling division).

It is guaranteed that there will be a valid 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  ; divisor=2 gives 1+2+3+4+6=16
```

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

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def smallestDivisor(self, nums: List[int], threshold: int) -> int:
        # STEP 1: Define condition function
        #   - For a candidate divisor, compute total sum of ceil(num/divisor)
        #   - Using (num - 1) // divisor + 1 avoids floating point & is faster
        def condition(divisor: int) -> bool:
            total = 0
            for num in nums:
                total += (num - 1) // divisor + 1
            return total <= threshold

        # STEP 2: Set binary search bounds
        #   - Smallest divisor is 1 (max sum)
        #   - Largest needed divisor is max(nums) (sum = len(nums))
        left, right = 1, max(nums)

        # STEP 3: Binary search for smallest valid divisor
        #   - Invariant: answer is in [left, right]
        #   - If mid works, try smaller (move right = mid)
        #   - Else, need larger divisor (move left = mid + 1)
        while left < right:
            mid = left + (right - left) // 2
            if condition(mid):
                right = mid
            else:
                left = mid + 1

        # STEP 4: Return left (== right), which is 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 threshold (answer = 1)
    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

We'll trace `smallestDivisor([1, 2, 5, 9], 6)` step by step.

Initial state:
- `nums = [1, 2, 5, 9]`, `threshold = 6`
- `left = 1`, `right = max(nums) = 9`

Iteration 1:
- `left=1`, `right=9` → `mid = 1 + (9-1)//2 = 5`
- Call `condition(5)`: - For 1: `(1-1)//5 + 1 = 0 + 1 = 1` - For 2: `(2-1)//5 + 1 = 0 + 1 = 1` - For 5: `(5-1)//5 + 1 = 0 + 1 = 1` - For 9: `(9-1)//5 + 1 = 8//5 + 1 = 1 + 1 = 2`
- Total = 1+1+1+2 = **5** **6** → **True** - Since condition is **True**, set `right = mid = 5`

State: `left=1`, `right=5`

Iteration 2:
- `mid = 1 + (5-1)//2 = 3`
- `condition(3)`: - 1 → 1, 2 → 1, 5 → (4//3)+1 = 1+1=2, 9 → (8//3)+1=2+1=3
- Total = 1+1+2+3 = **7** $>$ **6** → **False** - Set `left = mid + 1 = 4`

State: `left=4`, `right=5`

Iteration 3:
- `mid = 4 + (5-4)//2 = 4`
- `condition(4)`: - 1→1, 2→1, 5→(4//4)+1=1+1=2, 9→(8//4)+1=2+1=3
- Total = 1+1+2+3 = **7 > 6 → False** - Set `left = 4 + 1 = 5`

State: `left=5`, `right=5` → loop ends

Return: `5`

Final output: **5**

Key takeaway: We **binary search on the answer space** (divisors), not the array. The monotonic property is:
As divisor increases, total sum decreases.
This makes binary search valid.

Complexity Analysis

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

  `n = len(nums)`, `M = max(nums)`.
  Binary search runs in `O(log M)`, and each `condition` check scans all `n` elements.

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

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

13. Find Minimum in Rotated Sorted Array II

Pattern: Binary Search (Modified)

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[mid] == nums[right];
must shrink search space carefully.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def findMin(self, nums: List[int]) -> int:
        # STEP 1: Initialize pointers to cover entire array
        #   - 'left' starts at 0, 'right' at last index
        left, right = 0, len(nums) - 1

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

            # STEP 3: Compare mid with right to decide half
            #   - If nums[mid] > nums[right], min is in right half
            if nums[mid] > nums[right]:
                left = mid + 1
            #   - If nums[mid] < nums[right], min is in left half (incl. mid)
            elif nums[mid] < nums[right]:
                right = mid
            #   - If equal, we can't tell - safely shrink right by 1
            else:
                right -= 1

        # STEP 4: Return result
        #   - When loop ends, left == right → points to min
        return nums[left]

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

    #  Test 1: Normal case - no duplicates, rotated
    assert sol.findMin([4,5,6,7,0,1,2]) == 0, "Test 1 failed"

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

    #  Test 3: Tricky/negative - duplicates obscure rotation
    assert sol.findMin([3,3,1,3]) == 1, "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 walk through `findMin([3,3,1,3])` step by step.

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

Step 1: Enter `while left < right` → `0 < 3` → **true**
- Compute `mid = (0 + 3) // 2 = 1`
- `nums[mid] = nums[1] = 3`
- `nums[right] = nums[3] = 3`
- Since `3 == 3`, enter `else` → `right -= 1` → `right = 2`
- **State**: `left=0`, `right=2`

Step 2: Loop condition `0 < 2` → **true**
- `mid = (0 + 2) // 2 = 1`
- `nums[mid] = 3`, `nums[right] = nums[2] = 1`
- Now `3 > 1` → enter `if` → `left = mid + 1 = 2`
- **State**: `left=2`, `right=2`

Step 3: Loop condition `2 < 2` → **false** → exit loop
- Return `nums[left] = nums[2] = 1`

Final output: `1`

Key insight: When `nums[mid] == nums[right]`, we **cannot** know which half contains the min, so we safely reduce `right` by 1. This preserves correctness while handling duplicates.

Complexity Analysis

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

  In the average case (few duplicates), we halve the search space → `O(log n)`.
  In the worst case (all elements equal), we decrement `right` one by one → `O(n)`.

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

  Only uses a constant amount of extra space for pointers (`left`, `right`, `mid`).

14. Find Minimum in Rotated Sorted Array

Pattern: Binary Search

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 this problem in **O(log n)** time.

Sample Input & Output

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

```
Input: [2,1]
Output: 1
Explanation: Rotated once; minimum is at index 1.
```

```
Input: [0]
Output: 0
Explanation: Single-element edge case.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def findMin(self, nums: List[int]) -> int:
        # STEP 1: Initialize pointers to start and end of array.
        #   - We use binary search to eliminate half the array
        #     each iteration based on rotation property.
        left, right = 0, len(nums) - 1

        # STEP 2: Main loop - continue while search space > 1.
        #   - Invariant: min element is always in [left, right].
        #   - We compare mid with right to decide which half is
        #     sorted and where the pivot (min) must lie.
        while left < right:
            mid = (left + right) // 2

            # STEP 3: Update state based on comparison.
            #   - If nums[mid] > nums[right], the right half
            #     is unsorted → min must be in (mid, right].
            if nums[mid] > nums[right]:
                left = mid + 1
            #   - Else, left half (including mid) may contain min.
            #     So we move right to mid (not mid - 1!).
            else:
                right = mid

        # STEP 4: Return result.
        #   - When loop ends, left == right points to min.
        return nums[left]

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
```

```
    sol = Solution()

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

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

    #   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([4, 5, 6, 7, 0, 1, 2])` step by step.

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

Iteration 1:
- `left (0) < right (6)` → enter loop
- `mid = (0 + 6) // 2 = 3`
- `nums[mid] = nums[3] = 7`
- `nums[right] = nums[6] = 2`
- Compare: `7 > 2` → **true**
- So: `left = mid + 1 = 4`
- **New state**: `left = 4`, `right = 6`

Iteration 2:
- `4 < 6` → continue
- `mid = (4 + 6) // 2 = 5`
- `nums[5] = 1`, `nums[6] = 2`
- Compare: `1 > 2` → **false**
- So: `right = mid = 5`
- **New state**: `left = 4`, `right = 5`

Iteration 3:
- `4 < 5` → continue
- `mid = (4 + 5) // 2 = 4`
- `nums[4] = 0`, `nums[5] = 1`
- Compare: `0 > 1` → **false**
- So: `right = mid = 4`
- **New state**: `left = 4`, `right = 4`

Loop ends because `left == right`.
Return `nums[4] = 0`.

Final output: `0`

Key insight: By comparing `mid` with `right` (not `left`), we reliably detect which half contains the rotation point. The right side gives clearer signal because in a rotated array, the rightmost element is always **less than** the left part if rotation occurred.

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 constant extra space (`left`, `right`, `mid`). No recursion or auxiliary data structures.

15. 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 wrap around — if no such character exists (i.e., `target` is greater than or equal to all letters), return the **first letter** in the array.

Sample Input & Output

```
Input: letters = ["c","f","j"], target = "a"
Output: "c"
Explanation: 'a' is smaller than all letters;
wrap-around returns first letter.
```

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

```
Input: letters = ["c","f","j"], target = "j"
Output: "c"
Explanation: No letter > 'j', so wrap to 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
        #   - low = 0, high = last index
        #   - We'll search for first letter > target
        low, high = 0, len(letters) - 1

        # STEP 2: Handle wrap-around edge case upfront
        #   - If target >= last letter OR < first,
        #     answer must be letters[0] due to circularity
        if target >= letters[high] or target < letters[low]:
            return letters[0]

        # STEP 3: Perform binary search for smallest letter > target
        #   - Invariant: answer is in [low, high]
        #   - We converge low and high to the first valid index
        while low < high:
            mid = low + (high - low) // 2  # safe mid calculation

            # If mid letter <= target, it can't be answer
            # → search right half (mid+1 to high)
            if letters[mid] <= target:
                low = mid + 1
            else:
                # mid letter > target → candidate!
                # but maybe smaller one exists → keep mid in range
                high = mid

        # STEP 4: Return result
        #   - low == high, and points to smallest letter > target
        return letters[low]

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

    #  Test 1: Normal case - target in middle
    assert sol.nextGreatestLetter(["c", "f", "j"], "d") == "f"
```

```python
    #  Test 2: Edge case - target equals a letter
    assert sol.nextGreatestLetter(["c", "f", "j"], "c") == "f"

    #  Test 3: Tricky/negative - wrap-around needed
    assert sol.nextGreatestLetter(["c", "f", "j"], "j") == "c"

    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 `nextGreatestLetter(["c", "f", "j"], "d")` step by step:

1. **Initial Setup**
   - `letters = ["c", "f", "j"]`, `target = "d"`
   - `low = 0`, `high = 2`
   - Check wrap: `"d" >= "j"`? No. `"d" < "c"`? No. → Skip wrap return.
2. **First Loop Iteration** (`low=0`, `high=2`)
   - `mid = 0 + (2-0)//2 = 1`
   - `letters[1] = "f"`
   - Compare: `"f" <= "d"`? → **False**
   - So: `high = mid = 1`
   - State: `low=0`, `high=1`
3. **Second Loop Iteration** (`low=0`, `high=1`)
   - `mid = 0 + (1-0)//2 = 0`
   - `letters[0] = "c"`

- Compare: `"c" <= "d"`? → **True**
- So: `low = mid + 1 = 1`
- State: `low=1`, `high=1`

4. **Loop Ends** (`low == high`)
   - Return `letters[1] = "f"`

Final Output: `"f"`
Key Insight: Binary search finds the **first** element > target by narrowing the range while preserving the answer in `[low, high]`.

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 (`low`, `high`, `mid`) — no extra space scaling with input. "'

16. 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 at time 2-4 skipped to avoid overlap).
```

```
Input: startTime = [1,2,3,4,6], endTime = [3,5,10,6,9],
profit = [20,20,100,70,60]
Output: 150
Explanation: Choose jobs [1,3] and [6,9] → 20 + 60 = 80? No!
Better: [3,10] alone = 100, or [1,3]+[4,6]+[6,9] = 20+70+60=150.
```

```
Input: startTime = [1,1,1], endTime = [2,3,4], profit = [5,6,4]
Output: 6
Explanation: Only one job can be picked; max profit is 6.
```

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 list of (start, end, profit)
        #   - Sort by end time to enable DP with non-overlapping condition
        jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1])
        n = len(jobs)

        # dp[i] = max profit using first i jobs (i from 0 to n)
        dp = [0] * (n + 1)

        # STEP 2: Main loop / recursion
        #   - For each job i (1-indexed), decide: skip or take?
```

```
        for i in range(1, n + 1):
            start_i, end_i, profit_i = jobs[i - 1]

            # Find latest job that ends <= start_i (non-overlapping)
            # Use binary search on end times
            # Build list of end times for binary search
            # (We can extract this once outside loop for efficiency)
            # But for clarity, we build it inline here per iteration
            # → Actually, better to precompute end times
            # Let's refactor: precompute end times once
            pass  # placeholder

        # Actually, restructure for clarity and efficiency:
        end_times = [job[1] for job in jobs]  # sorted by construction
        dp[0] = 0

        for i in range(1, n + 1):
            start_i, end_i, profit_i = jobs[i - 1]

            # STEP 3: Update state / bookkeeping
            # Option 1: Skip current job → dp[i] = dp[i-1]
            # Option 2: Take current job → profit_i + dp[j]
            #   where j = largest index such that end_times[j-1] <= start_i

            # Binary search for rightmost job ending <= start_i
            j = bisect.bisect_right(end_times, start_i, 0, i - 1)
            # j is index in [0, i-1]; dp[j] is max profit up to job j

            profit_if_take = profit_i + dp[j]
            profit_if_skip = dp[i - 1]
            dp[i] = max(profit_if_take, profit_if_skip)

        # STEP 4: Return result
        #   - dp[n] holds max profit using all jobs
        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 1 failed"

    #  Test 2: Tricky case with multiple compatible jobs
    assert sol.jobScheduling(
        [1,2,3,4,6], [3,5,10,6,9], [20,20,100,70,60]
    ) == 150, "Test 2 failed"

    #  Test 3: Edge case - all jobs overlap
    assert sol.jobScheduling(
        [1,1,1], [2,3,4], [5,6,4]
    ) == 6, "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 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 end time
- Jobs before sort:
`(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]`

Step 2: Initialize DP array
- `dp = [0, 0, 0, 0, 0]` (length 5)

Step 3: Process each job (i from 1 to 4)

i = 1 → job = (1,3,50)
- `start_i = 1`
- Find `j = bisect_right(end_times, 1, 0, 0)` → search in empty range → `j = 0`
- `profit_if_take = 50 + dp[0] = 50`
- `profit_if_skip = dp[0] = 0`

- dp[1] = max(50, 0) = 50
→ dp = [0, 50, 0, 0, 0]

i = 2 → job = (2,4,10)
- start_i = 2
- Search in end_times[0:1] = [3] for value 2 → none → j = 0
- profit_if_take = 10 + 0 = 10
- profit_if_skip = dp[1] = 50
- dp[2] = max(10, 50) = 50
→ dp = [0, 50, 50, 0, 0]

i = 3 → job = (3,5,40)
- start_i = 3
- Search in end_times[0:2] = [3,4] for 3 → 3 qualifies → index = 0 → j = 1
(because bisect_right returns insertion point after last value → index 1)
- profit_if_take = 40 + dp[1] = 40 + 50 = 90
- profit_if_skip = dp[2] = 50
- dp[3] = max(90, 50) = 90
→ dp = [0, 50, 50, 90, 0]

i = 4 → job = (3,6,70)
- start_i = 3
- Search in end_times[0:3] = [3,4,5] for 3 → 3 → j = 1
- profit_if_take = 70 + dp[1] = 70 + 50 = 120
- profit_if_skip = dp[3] = 90
- dp[4] = max(120, 90) = 120
→ dp = [0, 50, 50, 90, 120]

Final Output: dp[4] = 120

Key insight: Even though job 4 starts at same time as job 3, we can pair it with job 1 (ends at 3), giving 50+70=120.

Complexity Analysis

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

  Sorting takes O(n log n). The loop runs n times, and each bisect_right is O(log n). Total: O(n log n).

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

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

17. Time Based Key-Value Store

Pattern: Binary Search + Hash Map (Timestamped Data)

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 setting ("foo", "bar", 1),
get("foo", 1) returns "bar".
- get("foo", 3) returns "bar" because no newer
value exists before timestamp 3.
- After setting ("foo", "bar2", 4),
get("foo", 4) and get("foo", 5) both return "bar2".
```

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

Explanation:
Key doesn't exist → return empty string.
```

```
Input: ["TimeMap", "set", "get"]
[[], ["a", "b", 100], ["a", 1]]
Output: [null, ""]

Explanation:
Timestamp 1 is earlier than the only stored timestamp (100) → no valid value.
```

LeetCode Editorial Solution + Inline Tests

```python
from collections import defaultdict
import bisect

class TimeMap:
    def __init__(self):
        # Map each key to a list of (value, timestamp) pairs.
        # List is kept in increasing timestamp order (by design).
        self.store = defaultdict(list)

    def set(self, key: str, value: str, timestamp: int) -> None:
        # Append new (value, timestamp) - timestamps are strictly
        # increasing per problem constraints, so list stays sorted.
        self.store[key].append((value, timestamp))

    def get(self, key: str, timestamp: int) -> str:
        # STEP 1: Check if key exists
        if key not in self.store:
            return ""

        # STEP 2: Retrieve sorted list of (value, ts) for key
        values = self.store[key]
```

```
        # STEP 3: Binary search on timestamps
        # Extract timestamps for bisect (only needed for search)
        timestamps = [ts for _, ts in values]

        # bisect_right returns insertion point to maintain order.
        # All values to the left have ts <= timestamp.
        i = bisect.bisect_right(timestamps, timestamp)

        # STEP 4: Return result
        # If i == 0, no timestamp <= given timestamp
        return values[i - 1][0] if i > 0 else ""

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

    #  Test 1: Normal case
    sol.set("foo", "bar", 1)
    assert sol.get("foo", 1) == "bar"
    assert sol.get("foo", 3) == "bar"

    sol.set("foo", "bar2", 4)
    assert sol.get("foo", 4) == "bar2"
    assert sol.get("foo", 5) == "bar2"

    #  Test 2: Edge case - key not present
    assert sol.get("missing", 10) == ""

    #  Test 3: Tricky/negative - timestamp too early
    sol.set("early", "val", 100)
    assert sol.get("early", 1) == ""

    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 `TimeMap` step-by-step using the first test sequence:

1. `TimeMap()`
   - `self.store = {}` (empty defaultdict)
2. `set("foo", "bar", 1)`
   - `self.store["foo"]` becomes `[("bar", 1)]`
3. `get("foo", 1)`
   - Key `"foo"` exists → `values = [("bar", 1)]`
   - `timestamps = [1]`
   - `bisect.bisect_right([1], 1)` → returns `1`
   - `i = 1 > 0` → return `values[0][0] = "bar"`
4. `get("foo", 3)`
   - Same `values = [("bar", 1)]`
   - `bisect_right([1], 3)` → returns `1` (since $3 > 1$)
   - Return `values[0][0] = "bar"`
5. `set("foo", "bar2", 4)`
   - Append → `self.store["foo"] = [("bar", 1), ("bar2", 4)]`
6. `get("foo", 4)`
   - `timestamps = [1, 4]`
   - `bisect_right([1,4], 4)` → returns `2`
   - Return `values[1][0] = "bar2"`
7. `get("foo", 5)`
   - `bisect_right([1,4], 5)` → returns `2`
   - Still returns `"bar2"`

Key Insight: Because `set` is always called with increasing timestamps (per problem), the list stays sorted — enabling binary search (`O(log n)`) instead of linear scan.

Complexity Analysis

- **Time Complexity**:
  - `set`: `O(1)` — appending to list
  - `get`: `O(log n)` — binary search on list of size `n` for that key

    `n` = number of entries for a given key. Total operations scale with calls.
- **Space Complexity**: `O(k * n)`

  `k` = number of unique keys, `n` = average entries per key. We store every `(value, timestamp)` pair.

18. Single Element in a Sorted Array

Pattern: Binary Search on Answer (Modified for Pair Structure)

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: All numbers except 2 appear twice; array is sorted.
```

```
Input: [3, 3, 7, 7, 10, 11, 11]
Output: 10
Explanation: The single element breaks the pairing pattern after index 4.
```

```
Input: [1]
Output: 1
Explanation: Edge case - only one element in array.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def singleNonDuplicate(self, nums: List[int]) -> int:
        # STEP 1: Initialize pointers to search space
        #   - We maintain [left, right] as valid range for answer
        left, right = 0, len(nums) - 1

        # STEP 2: Main loop - binary search while range has >1 element
        #   - Invariant: single element is always in [left, right]
        while left < right:
            mid = left + (right - left) // 2

            # STEP 3: Force mid to even index to align with pair starts
            #   - Pairs start at even indices before the single element
            if mid % 2 == 1:
                mid -= 1

            # STEP 4: Compare mid with its pair (mid+1)
            #   - If equal → left half is "normal", answer is right
            #   - If not → single element disrupted left side
            if nums[mid] == nums[mid + 1]:
                left = mid + 2   # Skip this intact pair
            else:
                right = mid      # Single element is at or before mid

        # STEP 5: Return result
```

```
        #   - Loop ends when left == right → that's the answer
        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([1, 1, 2, 2, 3]) == 3
```

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

Example Walkthrough

We'll trace `singleNonDuplicate([1, 1, 2, 3, 3, 4, 4, 8, 8])`.

Initial state:
- `nums = [1,1,2,3,3,4,4,8,8]`
- `left = 0`, `right = 8`

Iteration 1: - `left=0 < right=8` → enter loop
- `mid = 0 + (8-0)//2 = 4`
- `mid=4` is even → no change
- Compare `nums[4] == nums[5]` → `3 == 4`? False
- So: `right = mid = 4`
- **State**: `left=0`, `right=4`

Iteration 2: - `0 < 4` → continue
- `mid = 0 + (4-0)//2 = 2`
- `mid=2` is even → ok
- Compare `nums[2] == nums[3]` → `2 == 3`? False
- So: `right = 2`
- **State**: `left=0`, `right=2`

Iteration 3: - `0 < 2` → continue
- `mid = 0 + (2-0)//2 = 1`
- `mid=1` is odd → adjust: `mid = 0`
- Compare `nums[0] == nums[1]` → `1 == 1`? True
- So: `left = mid + 2 = 0 + 2 = 2`
- **State**: `left=2`, `right=2`

Loop ends: `left == right == 2`
→ Return `nums[2] = 2`

Final output: `2`

Key insight:
Before the single element, pairs start at **even** indices (0,2,4…).
After the single element, pairs start at **odd** indices.
Binary search exploits this parity shift.

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 — no recursion or extra arrays.

19. Median of Two Sorted Arrays

Pattern: Binary Search on Partition (Advanced Two Pointers)

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: Only one element; edge case with empty array.
```

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 the correct split point in nums1
        #   - Such that left partition has 'half' elements total
        left, right = 0, m

        while left <= right:
            # Partition nums1 at i → left1 = nums1[:i], right1 = nums1[i:]
            i = (left + right) // 2
            # Partition nums2 at j → j = half - i
            j = half - i

            # Handle edge cases with -inf / +inf
            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
            #   - Max of left <= min of right for both arrays
            if nums1_left <= nums2_right and nums2_left <= nums1_right:
                # STEP 4: Compute median based on even/odd total length
                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 is too big, move partition left
            elif nums1_left > nums2_right:
                right = i - 1
            else:
                left = i + 1

        # Should never reach here if inputs are valid
        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: Edge case - one empty array
    assert sol.findMedianSortedArrays([], [1]) == 1.0

    #  Test 3: Tricky case - even total, interleaved
    assert sol.findMedianSortedArrays([1, 2], [3, 4]) == 2.5
```

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

Example Walkthrough

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

1. **Initial Setup**
   - `nums1 = [1,3]` (m=2), `nums2 = [2]` (n=1)
   - Since `m > n`, we swap → `nums1 = [2]`, `nums2 = [1,3]`
   - Now `m=1`, `n=2`, `total = 3`, `half = 1`
   - Binary search: `left = 0`, `right = 1`
2. **First Loop Iteration**

- `i = (0 + 1) // 2 = 0` → partition `nums1` at 0 → left part empty
- `j = half - i = 1 - 0 = 1` → partition `nums2` at 1 → left = `[1]`
- Values:
  - `nums1_left = -inf` (i=0)
  - `nums1_right = 2` (i=0 < m=1)
  - `nums2_left = nums2[0] = 1` (j=1 > 0)
  - `nums2_right = nums2[1] = 3` (j=1 < n=2)
- Check: `-inf <= 3` and `1 <= 2` → **valid partition!**
- Total length = 3 (odd) → return `min(2, 3) = 2.0`

3. **Final Output**: `2.0`

The key insight: we never merge arrays. Instead, we **binary search the correct partition** such that all left elements all right elements. This achieves `O(log(min(m,n)))`.

Complexity Analysis

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

  We perform binary search on the smaller array. Each step is `O(1)`, and we halve the search space each time.

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

  Only a few variables are used (`i`, `j`, `left`, `right`, etc.). No extra arrays or recursion stack.‘m

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

Pattern: Binary Search (Modified for Bounds)

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 in the array.
```

```
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: Find first occurrence using modified binary search
        #   - We search for leftmost index where target appears
        first_idx = self.binary_search(nums, target, False)
```

```
        # STEP 2: Find last occurrence using modified binary search
        #   - We search for rightmost index where target appears
        last_idx = self.binary_search(nums, target, True)

        # STEP 3: Return result
        #   - If first_idx is -1, target not found → return [-1, -1]
        return [first_idx, last_idx] if first_idx != -1 else [-1, -1]

    def binary_search(
        self, nums: List[int], target: int, find_last: bool
    ) -> int:
        left, right = 0, len(nums) - 1
        candidate = -1  # Tracks valid index if target is found

        # STEP 1: Standard binary search loop
        while left <= right:
            mid = left + (right - left) // 2

            if nums[mid] == target:
                candidate = mid  # Record potential answer

                # STEP 2: Adjust search direction based on mode
                #   - For first occurrence: search left half
                #   - For last occurrence: search right half
                if find_last:
                    left = mid + 1
                else:
                    right = mid - 1

            elif nums[mid] < target:
                left = mid + 1  # Target is in right half
            else:
                right = mid - 1  # Target is in left half

        return candidate

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

    #  Test 1: Normal case - target appears multiple times
    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 case - empty array
assert sol.searchRange([], 0) == [-1, -1]
```

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

Example Walkthrough

We'll trace **Test 1**: `nums = [5,7,7,8,8,10]`, `target = 8`.

Step 1: Call `searchRange`

- `nums = [5,7,7,8,8,10]`, `target = 8`
- Calls `binary_search(nums, 8, False)` → find **first** occurrence.

Step 2: First Binary Search (`find_last=False`)

- `left=0`, `right=5`
- **Iteration 1**:
  - `mid = 0 + (5-0)//2 = 2` → `nums[2] = 7`
  - `7 < 8` → `left = 3`
- **Iteration 2**:
  - `mid = 3 + (5-3)//2 = 4` → `nums[4] = 8` → match!
  - `candidate = 4`
  - Since `find_last=False`, move **left**: `right = 4 - 1 = 3`
- **Iteration 3**:
  - `left=3`, `right=3` → `mid=3` → `nums[3]=8` → match!
  - `candidate = 3`

  - Move left: `right = 2`

- Loop ends (`left=3 > right=2`) → return `candidate = 3`

Step 3: Second Binary Search (`find_last=True`)

- `left=0`, `right=5`
- **Iteration 1**: `mid=2` → `7 < 8` → `left=3`
- **Iteration 2**: `mid=4` → `8 == 8` → `candidate=4`, move **right**: `left=5`
- **Iteration 3**: `left=5`, `right=5` → `mid=5` → `nums[5]=10 > 8` → `right=4`
- Loop ends → return `candidate = 4`

Step 4: Final Result

- `first_idx = 3`, `last_idx = 4` → return `[3, 4]`

The key idea: **reuse binary search twice** — once biased left, once biased right — to find bounds without scanning.

Complexity Analysis

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

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

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

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

21. 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]]`.
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: The array is rotated at index 4. Target 0 is at index 4.
```

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

```
Input: nums = [3,1], target = 1
Output: 1
Explanation: Rotated array with two elements; target is in right half.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def search(self, nums: List[int], target: int) -> int:
        # STEP 1: Initialize pointers for the binary search
        #   - left and right define the current search window
        left, right = 0, len(nums) - 1
```

```
        # STEP 2: Main loop / recursion
        #   - Continue while the search window is valid (left <= right)
        #   - Invariant: target, if present, is within [left, right]
        while left <= right:
            # Compute mid without overflow (though not needed in Python)
            mid = left + (right - left) // 2

            # If found, return index immediately
            if nums[mid] == target:
                return mid

            # STEP 3: Update state / bookkeeping
            #   - Determine which half is sorted (left or right)
            #   - Use sorted half to decide where target could be
            if nums[left] <= nums[mid]:  # Left half is sorted
                # Check if target lies in the sorted left half
                if nums[left] <= target < nums[mid]:
                    right = mid - 1  # Search left
                else:
                    left = mid + 1   # Search right
            else:  # Right half must be sorted
                # Check if target lies in the sorted right half
                if nums[mid] < target <= nums[right]:
                    left = mid + 1   # Search right
                else:
                    right = mid - 1  # Search left

        # STEP 4: Return result
        #   - If loop ends, 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 present
    assert sol.search([1], 0) == -1

    #  Test 3: Tricky/negative - small rotated array
```

```
    assert sol.search([3,1], 1) == 1

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

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

Example Walkthrough

We'll trace `sol.search([6, 7, 1, 2, 3, 4, 5], 3)` step by step.

Initial state:
- `nums = [6,7,1,2,3,4,5]`, `target = 3`
- `left = 0`, `right = 6`

Step 1: Enter loop (`0 <= 6` → true)
- `mid = 0 + (6 - 0) // 2 = 3`
- `nums[3] = 2` 3 → not found
- Check if left half sorted: `nums[0] = 6 <= nums[3] = 2`? → **False**

→ So **right half is sorted** (`[2,3,4,5]`)
- Is `target = 3` in `(2, 5]`? → Yes (`2 < 3 <= 5`)

→ Move `left = mid + 1 = 4`
- **New state**: `left=4`, `right=6`

Step 2: Loop (`4 <= 6` → true)
- `mid = 4 + (6-4)//2 = 5`
- `nums[5] = 4` 3
- Check left half: `nums[4]=3 <= nums[5]=4` → **True** (left half `[3,4]` sorted)
- Is `target=3` in `[3, 4)`? → Yes (`3 <= 3 < 4`)

→ Move `right = mid - 1 = 4`
- **New state**: `left=4`, `right=4`

Step 3: Loop (`4 <= 4` → true)
- `mid = 4 + 0 = 4`
- `nums[4] = 3` == target → **return 4**

Final output: `4`

Key insight: At each step, **one half is always sorted** due to rotation with distinct elements. We use that sorted half to eliminate the other half confidently.

Complexity Analysis

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

  Binary search halves the search space each iteration. Even with rotation, we discard half the array per step → logarithmic time.

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

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

Search a 2D Matrix

Pattern: Binary Search (Treated as Sorted 1D Array)

Problem Statement

You are given an `m x n` integer matrix `matrix` with the following two properties:
- Each row is sorted in non-decreasing order.
- The first integer of each row is greater than the last integer of the previous row.

Given an integer `target`, return `true` if `target` is in `matrix` or `false` otherwise.

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

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 any row.
```

```
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 sorted 1D array of size m*n
        #   - Use binary search on virtual indices [0, m*n)
        m, n = len(matrix), len(matrix[0])
        left, right = 0, m * n - 1

        # STEP 2: Main loop / recursion
        #   - Invariant: target is in [left, right] if present
        #   - Condition: when left > right, target not found
        while left <= right:
            mid = (left + right) // 2

            # Convert 1D index to 2D coordinates
            row = mid // n
            col = mid % n
            mid_val = matrix[row][col]
```

```
            # STEP 3: Update state / bookkeeping
            #   - Why here? We compare and narrow search space
            if mid_val == target:
                return True
            elif mid_val < target:
                left = mid + 1
            else:
                right = mid - 1

        # STEP 4: Return result
        #   - Handle edge cases / defaults
        return False

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

    #  Test 1: Normal case
    matrix1 = [[1,3,5,7],[10,11,16,20],[23,30,34,60]]
    assert sol.searchMatrix(matrix1, 3) == True
    print(" Test 1 passed")

    #  Test 2: Edge case
    matrix2 = [[1]]
    assert sol.searchMatrix(matrix2, 1) == True
    print(" Test 2 passed")

    #  Test 3: Tricky/negative
    matrix3 = [[1,3,5,7],[10,11,16,20],[23,30,34,60]]
    assert sol.searchMatrix(matrix3, 13) == False
    print(" Test 3 passed")
```

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

Example Walkthrough

We'll trace **Test 1**: `matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]]`, `target = 3`.

- **Initial state**:
  `m = 3`, `n = 4` → total elements = 12
  `left = 0`, `right = 11`
- **Iteration 1**:
  `mid = (0 + 11) // 2 = 5`
  `row = 5 // 4 = 1`, `col = 5 % 4 = 1` → `matrix[1][1] = 11`
  Since `11 > 3`, set `right = 5 - 1 = 4`
- **Iteration 2**:
  `mid = (0 + 4) // 2 = 2`
  `row = 2 // 4 = 0`, `col = 2 % 4 = 2` → `matrix[0][2] = 5`
  Since `5 > 3`, set `right = 2 - 1 = 1`
- **Iteration 3**:
  `mid = (0 + 1) // 2 = 0`
  `row = 0 // 4 = 0`, `col = 0 % 4 = 0` → `matrix[0][0] = 1`
  Since `1 < 3`, set `left = 0 + 1 = 1`
- **Iteration 4**:
  `left = 1`, `right = 1` → `mid = 1`
  `row = 1 // 4 = 0`, `col = 1 % 4 = 1` → `matrix[0][1] = 3`
  Match! Return `True`.

Final Output: `True`
Key Insight: The matrix's global sorted property lets us treat it as a 1D sorted array — enabling classic binary search.

Complexity Analysis

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

  We perform binary search over `m * n` 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 proportional to input.

23. Search a 2D Matrix II

Pattern: Matrix Traversal (Top-Right / Bottom-Left Elimination)

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],[2,5,8,12],[3,6,9,16],[10,13,14,17]], target = 5
Output: true
Explanation: 5 exists at matrix[1][1].
```

```
Input: matrix = [[1,4,7,11],[2,5,8,12],[3,6,9,16],[10,13,14,17]], target = 20
Output: false
Explanation: 20 is larger than all elements; not present.
```

```
Input: matrix = [], target = 1
Output: false
Explanation: Empty matrix - no elements to search.
```

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
        #   - Early exit if matrix is empty or has no rows
        if not matrix or len(matrix) == 0:
            return False

        # STEP 2: Main loop / recursion
        #   - Start from top-right corner: this position allows
        #     elimination of either a row or a column per step
        row = 0
        col = len(matrix[0]) - 1

        #   - Invariant: target (if exists) must lie in the
        #     submatrix from (row, 0) to (m-1, col)
        while col >= 0 and row < len(matrix):
            current = matrix[row][col]
            if current == target:
                # Found target → return immediately
                return True
            elif current > target:
                # Current too big → eliminate this column
                col -= 1
            else:
                # Current too small → eliminate this row
                row += 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
    matrix1 = [[1,4,7,11],
               [2,5,8,12],
               [3,6,9,16],
               [10,13,14,17]]
```

```
    assert sol.searchMatrix(matrix1, 5) == True
    print("  Test 1 passed: target 5 found")

    #   Test 2: Edge case - empty matrix
    assert sol.searchMatrix([], 1) == False
    print("  Test 2 passed: empty matrix handled")

    #   Test 3: Tricky/negative - target not present
    assert sol.searchMatrix(matrix1, 20) == False
    print("  Test 3 passed: target 20 correctly not found")
```

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

Example Walkthrough

Let's trace **Test 1** with `target = 5` and the given matrix:

Initial state:
- `row = 0`, `col = 3` → start at `matrix[0][3] = 11`

Step 1:
- Check `11 == 5`? No.
- `11 > 5` → move left: `col = 2`
- Now at `matrix[0][2] = 7`

Step 2:
- `7 == 5`? No.
- `7 > 5` → move left: `col = 1`
- Now at `matrix[0][1] = 4`

Step 3:
- `4 == 5`? No.
- `4 < 5` → move down: `row = 1`
- Now at `matrix[1][1] = 5`

Step 4:
- `5 == 5`? Yes! → return `True`

Why this works:
Starting from the **top-right**, every comparison lets us **discard a full row or column**: - If

current > target → everything below in this column is even larger → discard column. - If current < target → everything to the left in this row is smaller → discard row.

This mimics binary search logic but in 2D, leveraging sorted rows **and** columns.

Complexity Analysis

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

  At each step, we move either left or down. In worst case, we traverse from top-right to bottom-left: `m` rows + `n` columns → linear in dimensions.

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

  Only using two integer pointers (`row`, `col`) — no extra data structures or recursion stack.

24. Peak Index in a Mountain Array

Pattern: Binary Search

Problem Statement

An array `arr` is a **mountain array** if and only if: - `arr.length >= 3` - There exists some `i` with `0 < i < arr.length - 1` such that: - `arr[0] < arr[1] < ... < arr[i - 1] < arr[i]`, and - `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.

Sample Input & Output

```
Input: [0, 2, 5, 3, 1]
Output: 2
Explanation: arr[2] = 5 is greater than both neighbors (2 and 3),
so index 2 is the peak.
```

```
Input: [0, 10, 5, 2]
Output: 1
Explanation: Peak is at index 1 (value 10).
```

```
Input: [1, 2, 3, 4, 5, 4, 3, 2, 1]
Output: 4
Explanation: Strictly increasing until index 4,
then decreasing - peak at index 4.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def peakIndexInMountainArray(self, arr: List[int]) -> int:
        # STEP 1: Initialize pointers at both ends
        #   - We know the peak is not at the edges (by problem definition)
        left, right = 0, len(arr) - 1

        # STEP 2: Binary search loop
        #   - Invariant: peak is always in [left, right]
        #   - We compare arr[mid] with arr[mid + 1] to decide direction
        while left < right:
            mid = left + (right - left) // 2

            # STEP 3: Check slope direction
            #   - If ascending (arr[mid] < arr[mid+1]), peak is to the right
            if arr[mid] < arr[mid + 1]:
                left = mid + 1
            else:
                # If descending or flat (shouldn't be flat in valid input),
                # peak is at mid or to the left
```

```
                right = mid

        # STEP 4: Return left (== right), which is the peak index
        #   - Loop ends when search space collapses to one index
        return left

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

    #  Test 1: Normal case
    assert sol.peakIndexInMountainArray([0, 2, 5, 3, 1]) == 2

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

    #  Test 3: Tricky/negative - long ascending then steep drop
    assert sol.peakIndexInMountainArray([1, 2, 3, 4, 5, 4, 3, 2, 1]) == 4

    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 `arr = [0, 2, 5, 3, 1]` step by step:

1. **Initialize**:
   `left = 0`, `right = 4` (since `len(arr) = 5`)
2. **First loop iteration** (`left=0`, `right=4`):
   - `mid = 0 + (4 - 0) // 2 = 2`
   - Compare `arr[2] = 5` and `arr[3] = 3`
   - Since `5 > 3`, we are on the **descending** side → set `right = mid = 2`
   - State: `left=0`, `right=2`

3. **Second loop iteration** (`left=0`, `right=2`):
    - `mid = 0 + (2 - 0) // 2 = 1`
    - Compare `arr[1] = 2` and `arr[2] = 5`
    - Since `2 < 5`, we are on the **ascending** side → set `left = mid + 1 = 2`
    - State: `left=2`, `right=2`
4. **Loop condition check**: `left < right` → `2 < 2` is **False** → exit loop
5. **Return**: `left = 2` → correct peak index!

Key Insight:
We never check `arr[mid - 1]`, only `arr[mid + 1]`, which avoids index errors and keeps logic clean. The binary search narrows down by always moving **toward the rising slope**.

Complexity Analysis

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

  We halve the search space each iteration using binary search — classic logarithmic time.

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

  Only two pointers (`left`, `right`) and one `mid` variable — constant extra space.

25. 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] = -∞`.
You must solve it in `O(log n)` time.

Sample Input & Output

```
Input: [1, 2, 3, 1]
Output: 2
Explanation: nums[2] = 3 > nums[1] = 2 and nums[3] = 1 → peak at index 2.
```

```
Input: [1, 2, 1, 3, 5, 6, 4]
Output: 5 (or 1)
Explanation: Multiple peaks exist.
Index 5 → 6 > 5 and 4; index 1 → 2 > 1 and 1.
```

```
Input: [1]
Output: 0
Explanation: Single element is a peak by definition (neighbors are -∞).
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def findPeakElement(self, nums: List[int]) -> int:
        # STEP 1: Initialize pointers to cover full array
        #   - left = 0, right = last valid index
        left, right = 0, len(nums) - 1
```

```
        # STEP 2: Binary search loop
        #   - Invariant: a peak exists in [left, right]
        #   - We move toward the rising slope (guaranteed peak ahead)
        while left < right:
            mid = left + (right - left) // 2

            # STEP 3: Compare mid with its right neighbor
            #   - If nums[mid] < nums[mid+1], rising slope → peak on right
            if nums[mid] < nums[mid + 1]:
                left = mid + 1
            else:
                # Falling slope or flat → peak at mid or left
                right = mid

        # STEP 4: Converged to a peak index
        #   - Guaranteed by problem constraints and binary search logic
        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([5]) == 0

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

    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. **Initialization**
   - `left = 0`, `right = 3` (since `len(nums) = 4`)
   - Array: `[1, 2, 3, 1]`
2. **First loop iteration** (`left=0`, `right=3`)
   - `mid = 0 + (3 - 0) // 2 = 1`
   - Compare `nums[1] = 2` vs `nums[2] = 3` → `2 < 3` → rising slope
   - So move `left = mid + 1 = 2`
   - **State**: `left=2`, `right=3`
3. **Second loop iteration** (`left=2`, `right=3`)
   - `mid = 2 + (3 - 2) // 2 = 2`
   - Compare `nums[2] = 3` vs `nums[3] = 1` → `3 > 1` → falling slope
   - So move `right = mid = 2`
   - **State**: `left=2`, `right=2`
4. **Loop ends** (`left == right`)
   - Return `left = 2` → correct peak index.

Final output: `2`

Key insight: We don't need to check both neighbors. By always climbing the rising slope, we're guaranteed to hit a peak because boundaries are `-∞`.

Complexity Analysis

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

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

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

  Only uses two pointers (`left`, `right`) and one `mid` variable — constant extra space.

26. Two Sum (Sorted Array) – Find All Pairs That Sum to Target

Pattern: Two Pointers

Problem Statement

Given a **sorted array of distinct integers** and a **target sum**, find **all unique pairs** of numbers that add up to the target.
Return the list of pairs. Each pair should appear only once, and the solution must run in linear time.

Note: This is a variation of the classic Two Sum problem, optimized using the **Two Pointers** pattern due to the sorted input.

Sample Input & Output

```
Input: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], target = 10
Output: [(1, 9), (2, 8), (3, 7), (4, 6)]
Explanation: These are all unique pairs that sum to 10.
```

```
Input: [2, 7], target = 9
Output: [(2, 7)]
Explanation: Only one pair exists.
```

```
Input: [1, 2, 3], target = 10
Output: []
Explanation: No pair sums to 10.
```

LeetCode Editorial Solution + Inline Tests

```
from typing import List, Tuple

class Solution:
    def find_pairs(self, nums: List[int], target: int):
        # STEP 1: Initialize two pointers at start and end
        #   - Because array is sorted, we can adjust sum by moving pointers
        left = 0
        right = len(nums) - 1

        # List to collect valid pairs
        result = []

        # STEP 2: Main loop - continue while pointers haven't crossed
        #   - Invariant: all pairs between left and right are unexamined
        while left < right:
            current_sum = nums[left] + nums[right]

            # STEP 3: Compare current sum to target
            if current_sum == target:
                # Found a valid pair - add to result
                result.append((nums[left], nums[right]))
                # Move both pointers to find next potential pair
                left += 1
                right -= 1
            elif current_sum < target:
                # Sum too small → increase by moving left pointer right
                left += 1
            else:
                # Sum too large → decrease by moving right pointer left
                right -= 1

        # STEP 4: Return all found pairs
        #   - Handles empty result naturally if no pairs found
        return result

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

    #  Test 1: Normal case
```

```
    assert sol.find_pairs([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 10) == [
        (1, 9), (2, 8), (3, 7), (4, 6)
    ]

    #  Test 2: Edge case - only two elements that match
    assert sol.find_pairs([2, 7], 9) == [(2, 7)]

    #  Test 3: Tricky/negative - no valid pairs
    assert sol.find_pairs([1, 2, 3], 10) == []

    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 `find_pairs([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 10)` step by step.

Initial State: - `left = 0` → `nums[0] = 1` - `right = 9` → `nums[9] = 10` - `result = []`

Step 1:
- Compute `current_sum = 1 + 10 = 11`
- `11 > 10` → move `right` left → `right = 8`

State: `left=0`, `right=8`, `result=[]`

Step 2:
- `current_sum = 1 + 9 = 10` → match!
- Append `(1, 9)` to `result`
- Move both: `left = 1`, `right = 7`

State: `left=1`, `right=7`, `result=[(1,9)]`

Step 3:
- `current_sum = 2 + 8 = 10` → match!
- Append `(2, 8)`
- Move: `left = 2`, `right = 6`

State: `left=2`, `right=6`, `result=[(1,9), (2,8)]`

Step 4:
- `current_sum = 3 + 7 = 10` → match!
- Append `(3, 7)`
- Move: `left = 3`, `right = 5`

State: `left=3`, `right=5`, `result=[(1,9), (2,8), (3,7)]`

Step 5:
- `current_sum = 4 + 6 = 10` → match!
- Append `(4, 6)`
- Move: `left = 4`, `right = 4`

State: `left=4`, `right=4` → loop condition `left < right` is now **false**

Final Output: `[(1, 9), (2, 8), (3, 7), (4, 6)]`

Key Takeaway: Because the array is **sorted**, we can **adjust the sum predictably**—move left to increase, right to decrease—without missing any pairs.

Complexity Analysis

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

  We traverse the array at most once with two pointers moving inward. Each element is visited at most once.

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

  Only a constant amount of extra space is used (`left`, `right`, `current_sum`). The output list does not count toward auxiliary space in standard LeetCode conventions.