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

Given a sorted array of integers, `nums`, and an integer `target`, write an efficient algorithm to search for `target` in `nums`. If `target` exists, return its index. Otherwise, return `-1`.

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

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

`Output: 4`

`Input: nums = [-1, 0, 3, 5, 9, 12], target = 2`

`Output: -1`

```
from typing import List

class Solution:
    def search(self, nums: List[int], target: int) -> int:
        """
        Performs a binary search on a sorted list of integers to
        find the target value.

        Args:
        nums: A list of sorted integers.
```

```
        target: The integer value to search for in nums.

        Returns:
        The index of the target in nums if it exists, otherwise -1.
        """

        # Initialize the left and right pointers
        left = 0
        right = len(nums) - 1

        # Loop until the left pointer is greater than the right pointer
        while left <= right:
            # Calculate the midpoint of the current search range
            mid = left + (right - left) // 2

            # Check if the midpoint element is the target
            if nums[mid] == target:
                return mid  # Target found, return its index
            elif nums[mid] < target:
                # If the target is greater, ignore the left half
                left = mid + 1
            else:
                # If the target is smaller, ignore the right half
                right = mid - 1

        # Target is not found in the list
        return -1

# Sample input:
nums = [-1, 0, 3, 5, 9, 12]
target = 9

# Creating an instance of Solution and using the search method
solution = Solution()
result = solution.search(nums, target)

# Sample output:
print(result)  # Output: 4
```

2. First Bad Version

Problem

You are given (n) versions `[1, 2, ..., n]`. A function `isBadVersion(version)` is provided, returning whether a version is bad. Find the first bad version.

Example

`Input: n = 5, isBadVersion(3) = false, isBadVersion(4) = true, isBadVersion(5) = true`

`Output: 4`

Solution

The goal is to find the smallest (k) such that `isBadVersion(k)` is `True`. Using the API as the condition, the solution is:

```
class Solution:
    def firstBadVersion(self, n) -> int:
        # Initialize search space boundaries
        left, right = 1, n

        # Perform binary search
        while left < right:
            # Calculate the middle version
            mid = left + (right - left) // 2
            # If the mid version is bad, narrow the right boundary
            if isBadVersion(mid):
                right = mid
            else:  # Otherwise, narrow the left boundary
                left = mid + 1

        # Return the first bad version (minimum k satisfying isBadVersion(k))
        return left
```

```
# Test case
n = 5  # Total versions
first_bad_version = 4  # The first bad version
```

```
# Mocking the isBadVersion API for testing
def isBadVersion(version):
    return version >= first_bad_version

# Solution instance and execution
solution = Solution()
result = solution.firstBadVersion(n)
print(result)  # Expected output: 4
```

3. Sqrt(x)

Problem

Compute and return the integer part of the square root of (x).

Example

```
Input: x = 4
Output: 2

Input: x = 8
Output: 2
```

Solution

Search for the smallest (k) such that (k^2 > x). The result is (k - 1).

```
def mySqrt(x: int) -> int:
    # Initialize boundaries: include all possible values of k
    left, right = 0, x + 1

    # Perform binary search
    while left < right:
        # Calculate the middle value
        mid = left + (right - left) // 2
        # If mid squared is greater than x, shrink the right boundary
        if mid * mid > x:
            right = mid
        else:  # Otherwise, shrink the left boundary
            left = mid + 1
```

```
    # Return the largest k such that k^2 <= x
    return left - 1
```

```
# Test case
x = 8

# Solution execution
result = mySqrt(x)
# Output: 2 (because √8 = 2.828..., and only the integer part is returned)
print(result)
```

4. Search Insert Position

Problem

Given a sorted array and a target value, return the index of the target. If the target is not found, return the index where it should be inserted.

Example

```
Input: nums = [1,3,5,6], target = 5
Output: 2

Input: nums = [1,3,5,6], target = 2
Output: 1
```

Solution

Search for the smallest (k) such that `nums[k] >= target`.

```
class Solution:
    def searchInsert(self, nums: List[int], target: int) -> int:
        # Initialize boundaries
        left, right = 0, len(nums)

        # Perform binary search
        while left < right:
            # Calculate the middle index
            mid = left + (right - left) // 2
            # If nums[mid] meets or exceeds the target, shrink right boundary
```

```
            if nums[mid] >= target:
                right = mid
            else:  # Otherwise, shrink the left boundary
                left = mid + 1

        # Return the index where the target should be inserted
        return left
```

```
# Test case
nums = [1, 3, 5, 6]  # Sorted array
target = 5           # Target value

# Solution instance and execution
solution = Solution()
result = solution.searchInsert(nums, target)
print(result)  # Expected output: 2 (target is found at index 2)
```

5. Capacity to Ship Packages Within D Days

Problem Statement:

You are given a conveyor belt with packages of weights given in the array `weights`. The packages must be shipped from one port to another within `D` days in the given order.

The ship has a maximum weight capacity, and you want to determine the **minimum capacity** required so that all packages are shipped within `D` days.

Key Constraints: 1. Packages must be shipped in the order they appear in `weights`. 2. A ship cannot carry more than its weight capacity on any day.

Monotonicity Insight: If a ship can ship all packages within `D` days with a given capacity, it can also do so with any capacity greater than that.

Sample Input:

```
weights = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
D = 5
```

Sample Output:

```
Output: 15
```

Explanation:

- The minimum ship capacity required is 15.
- Example shipping plan:
  - Day 1: Packages [1, 2, 3, 4, 5] (total weight = 15)
  - Day 2: Packages [6, 7] (total weight = 13)
  - Day 3: Package [8] (total weight = 8)
  - Day 4: Package [9] (total weight = 9)
  - Day 5: Package [10] (total weight = 10)

```python
from typing import List

def shipWithinDays(weights: List[int], D: int) -> int:
    # Feasibility function: Can we ship within D days with the given capacity?
    def feasible(capacity) -> bool:
        days = 1  # Start with 1 day
        total = 0  # Current weight loaded onto the ship
        for weight in weights:
            total += weight
            if total > capacity:  # If overloaded, move to the next day
                total = weight
                days += 1
                if days > D:  # Exceeds allowed days
                    return False
        return True

    # Binary search bounds
    # Minimum capacity: heaviest package; maximum: sum of all weights
    left, right = max(weights), sum(weights)
    while left < right:
        mid = left + (right - left) // 2  # Middle capacity
        if feasible(mid):  # If feasible, try smaller capacity
            right = mid
        else:  # Otherwise, increase capacity
            left = mid + 1
    return left  # Minimum feasible capacity


weights = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
D = 5
print(shipWithinDays(weights, D))  # Output: 15
```

15

6. Split Array Largest Sum

Problem Statement:

You are given an array `nums` and an integer `m`. Split the array into `m` subarrays such that the **maximum sum** of the subarrays is minimized.

Key Constraints: 1. Each subarray must be continuous. 2. You need to find the optimal split that minimizes the largest sum among the subarrays.

Sample Input:

nums = [7, 2, 5, 10, 8]

m = 2

Sample Output:

Output: 18

Explanation:

- The array can be split into [7, 2, 5] and [10, 8].
- The largest sum among these subarrays is `18`, which is the minimum possible value.

```
def splitArray(nums: List[int], m: int) -> int:
    # Feasibility function: Can we split into m or
    # fewer subarrays with sums <= threshold?
    def feasible(threshold) -> bool:
        count = 1  # Start with 1 subarray
        total = 0  # Current subarray sum
        for num in nums:
            total += num
            if total > threshold:  # Start a new subarray
                total = num
                count += 1
                if count > m:  # Exceeds allowed subarrays
```

```
                    return False
        return True

    # Binary search bounds
    # Minimum sum: max element; maximum sum: sum of all elements
    left, right = max(nums), sum(nums)
    while left < right:
        mid = left + (right - left) // 2  # Middle threshold
        if feasible(mid):  # If feasible, try smaller maximum sum
            right = mid
        else:  # Otherwise, increase threshold
            left = mid + 1
    return left  # Minimum feasible maximum sum
```

```
nums = [7, 2, 5, 10, 8]
m = 2
print(splitArray(nums, m))  # Output: 18
```

```
18
```

7. Koko Eating Bananas

Problem Statement:

Koko is eating bananas from `N` piles. Each pile has a certain number of bananas. Koko eats bananas at a fixed speed (bananas per hour). She can eat from only one pile per hour. If there are fewer bananas left in a pile than her eating speed, she finishes that pile in one hour.

Find the **minimum eating speed** (bananas/hour) so that Koko can finish eating all the bananas within `H` hours.

Key Constraints: 1. The lower bound for the speed is `1`. 2. The upper bound for the speed is the size of the largest pile.

Monotonicity Insight: If Koko can eat all the bananas at a given speed, she can also eat them at any faster speed.

Sample Input:

```
piles = [30, 11, 23, 4, 20]
H = 6
```

Sample Output:

```
Output: 23
```

Explanation:

- At speed 23, Koko can finish all bananas in 6 hours:
  - Hour 1: Eats 23 bananas from pile [30], leaving 7.
  - Hour 2: Finishes pile [7].
  - Hour 3: Eats 11 bananas from pile [11].
  - Hour 4: Eats 23 bananas from pile [23].
  - Hour 5: Eats 20 bananas from pile [20].
  - Total time: 6 hours.

```
def minEatingSpeed(piles: List[int], H: int) -> int:
    # Feasibility function: Can Koko finish eating all bananas
    # within H hours at the given speed?
    def feasible(speed) -> bool:
        # Calculate the total hours needed at the given speed
        return sum((pile - 1) // speed + 1 for pile in piles) <= H

    # Binary search bounds
    # Minimum speed: 1 banana/hour; maximum speed: largest pile
    left, right = 1, max(piles)
    while left < right:
        mid = left + (right - left) // 2  # Middle speed
        if feasible(mid):  # If feasible, try smaller speed
            right = mid
        else:  # Otherwise, increase speed
            left = mid + 1
    return left  # Minimum feasible speed
```

```
piles = [30, 11, 23, 4, 20]
H = 6
print(minEatingSpeed(piles, H))  # Output: 23
```

23

The line:

```
sum((pile - 1) // speed + 1 for pile in piles) <= H
```

is a compact way of calculating how many hours it will take for Koko to eat all the bananas at a given eating speed (`speed`), and then checking if that total time is within the allowed hours (`H`).

Explanation of the Formula:

1. **For each pile:**
   ```
   (pile - 1) // speed + 1
   ```
   - `pile`: The number of bananas in the current pile.
   - `speed`: The number of bananas Koko can eat in one hour.
   - `(pile - 1) // speed`: This computes the integer division of `(pile - 1)` by `speed`. It effectively gives the number of full hours required if there were `pile - 1` bananas.
   - `+ 1`: Adds 1 more hour to account for the remainder, i.e., any leftover bananas in the pile that do not complete a full hour of eating.

   Together, this calculates the total number of hours needed for Koko to finish the pile at the current `speed`.

   Example:

   - If `pile = 30` and `speed = 23`:
     - `(30 - 1) // 23 + 1 = 29 // 23 + 1 = 1 + 1 = 2 hours`.
2. `sum((pile - 1) // speed + 1 for pile in piles)`: This calculates the total number of hours required to finish all piles at the current eating speed by summing the hours for each pile.
3. `<= H`: This checks if the total hours required is less than or equal to the allowed time `H`. If true, it means that the speed `speed` is feasible because Koko can finish all the bananas within `H` hours at this speed.

Why This Formula Works:

- It avoids the overhead of using `math.ceil(pile / speed)`, which could be slower for large inputs because it involves floating-point arithmetic.
- Instead, `(pile - 1) // speed + 1` computes the same result using integer arithmetic, which is faster and avoids precision issues.

Example Walkthrough:

Let's consider:

```
piles = [30, 11, 23, 4, 20]
speed = 10
```

1. For `pile = 30`: `(30 - 1) // 10 + 1 = 29 // 10 + 1 = 2 + 1 = 3 hours`.
2. For `pile = 11`: `(11 - 1) // 10 + 1 = 10 // 10 + 1 = 1 + 1 = 2 hours`.
3. For `pile = 23`: `(23 - 1) // 10 + 1 = 22 // 10 + 1 = 2 + 1 = 3 hours`.
4. For `pile = 4`: `(4 - 1) // 10 + 1 = 3 // 10 + 1 = 0 + 1 = 1 hour`.
5. For `pile = 20`: `(20 - 1) // 10 + 1 = 19 // 10 + 1 = 1 + 1 = 2 hours`.

Total Hours = 3 + 2 + 3 + 1 + 2 = 11 hours.

If `H = 11`, this speed is feasible. If `H < 11`, this speed is too slow, and we need to increase the speed.

8. Minimum Number of Days to Make `m` Bouquets

Problem Statement:

You are given an array `bloomDay` where each element represents the day a flower blooms. You need to make `m` bouquets, where each bouquet requires `k` adjacent flowers.

Return the **minimum number of days** needed to make the bouquets. If it is not possible, return `-1`.

Key Constraints: 1. Flowers in a bouquet must be adjacent. 2. The total number of flowers needed is `m * k`.

Monotonicity Insight: If we can make `m` bouquets after waiting for `d` days, we can also make them for any day greater than `d`.

Sample Input:

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

Sample Output:

```
Output: 3
```

Explanation:

- On day 3, we can make 3 bouquets:
  - Bouquet 1: Flower [1].
  - Bouquet 2: Flower [3].
  - Bouquet 3: Flower [2].
- Waiting for fewer days (e.g., 2) does not allow us to make 3 bouquets.

```python
def minDays(bloomDay: List[int], m: int, k: int) -> int:
    # Feasibility function: Can we make m bouquets
    # in the given number of days?
    def feasible(days) -> bool:
        # Count of bouquets made and current flowers collected
        bouquets, flowers = 0, 0
        for bloom in bloomDay:
            if bloom > days:  # If flower has not bloomed, reset count
                flowers = 0
            else:  # Otherwise, count this flower
                flowers += 1
                if flowers == k:  # If enough flowers for one bouquet
                    bouquets += 1
                    flowers = 0
        return bouquets >= m

    # If impossible to make m bouquets
    if len(bloomDay) < m * k:
        return -1

    # Binary search bounds
    # Minimum days: 1; maximum days: longest bloom time
    left, right = 1, max(bloomDay)
    while left < right:
        mid = left + (right - left) // 2  # Middle days
        if feasible(mid):  # If feasible, try fewer days
            right = mid
        else:  # Otherwise, increase days
            left = mid + 1
    return left  # Minimum feasible days
```

```
#### Test Case:
bloomDay = [1, 10, 3, 10, 2]
m = 3
k = 1
print(minDays(bloomDay, m, k))  # Output: 3
```

3

Problem Statement:

You are given: - `bloomDay`: An array where each value represents the day a flower will bloom. - `m`: The number of bouquets you need to make. - `k`: The number of adjacent flowers required for one bouquet.

The goal is to find the **minimum number of days** needed to make `m` bouquets. If it's impossible to make `m` bouquets, return `-1`.

Example Input:

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

Step 1: Check for Impossible Cases

```
if len(bloomDay) < m * k:
    return -1
```

- If the total number of flowers in `bloomDay` is less than the required `m * k` flowers, it is impossible to make `m` bouquets. Return `-1`.

In our case: - `len(bloomDay) = 5`, `m * k = 3 * 1 = 3`. - Since `5 >= 3`, it's possible to proceed.

Step 2: Define Binary Search Bounds

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

- `left`: The minimum number of days required is `1`.
- `right`: The maximum number of days required is `max(bloomDay)` because no flower will bloom before its bloom day.

In our case: - `left = 1` - `right = 10`

Step 3: Feasibility Function

```
def feasible(days) -> bool:
    bouquets, flowers = 0, 0
    for bloom in bloomDay:
        if bloom > days:
            flowers = 0  # Reset if flower hasn't bloomed yet
        else:
            flowers += 1
            if flowers == k:  # If enough flowers for one bouquet
                bouquets += 1
                flowers = 0  # Reset for the next bouquet
    return bouquets >= m
```

Logic: - For each day (`days`), determine if we can collect `k` consecutive flowers to form `m` bouquets. - If a flower hasn't bloomed by `days`, reset the `flowers` count. - If enough flowers (`k`) are collected for one bouquet, increment the `bouquets` count and reset `flowers`.

Step 4: Binary Search Logic

The binary search narrows down the minimum number of days required:

1. Calculate `mid` as the average of `left` and `right`.
2. Check if it's feasible to make `m` bouquets in `mid` days using the `feasible` function:

- If feasible, try fewer days by setting `right = mid`.
- If not feasible, try more days by setting `left = mid + 1`.

While Loop:

```
while left < right:
    mid = left + (right - left) // 2
    if feasible(mid):
        right = mid
    else:
        left = mid + 1
```

Walkthrough with Example

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

1. **Initial Bounds**:
   - `left = 1`, `right = 10`.
2. **Iteration 1**:
   - `mid = (1 + 10) // 2 = 5`.
   - Check if it's feasible in `5` days:
     - Flowers bloomed: `[1, _, 3, _, 2]` (bloomed on days 5).
     - Bouquets: `[1], [3], [2]` → **3 bouquets formed**.
   - `feasible(5) = True`.
   - Update bounds: `right = 5`.
3. **Iteration 2**:
   - `mid = (1 + 5) // 2 = 3`.
   - Check if it's feasible in `3` days:
     - Flowers bloomed: `[1, _, 3, _, _]` (bloomed on days 3).
     - Bouquets: `[1], [3]` → **2 bouquets formed**.
   - `feasible(3) = False`.
   - Update bounds: `left = 4`.
4. **Iteration 3**:

- `mid = (4 + 5) // 2 = 4`.
- Check if it's feasible in `4` days:
  - Flowers bloomed: `[1, _, 3, _, 2]` (bloomed on days 4).
  - Bouquets: `[1], [3], [2]` → **3 bouquets formed**.
- `feasible(4) = True`.
- Update bounds: `right = 4`.

5. **End Condition**:
   - `left == right == 4`.

Output: `4`.

Conclusion

The minimum number of days required to make `3` bouquets is `4`.

9. Kth Smallest Number in Multiplication Table

Problem Statement

You are given a multiplication table of size `m x n`. Find the k-th smallest number in this table. Instead of explicitly constructing the table, the goal is to use binary search and a condition function to determine the k-th smallest number efficiently.

Explanation with Test Case

- **Input**: `m = 3`, `n = 3`, `k = 5`
  Multiplication Table:

  ```
  1 2 3
  2 4 6
  3 6 9
  ```

  The sorted order of numbers: `[1, 2, 2, 3, 3, 4, 6, 6, 9]`. The 5th smallest number is `3`.

- **Output**: `3`

```
#### Code with Explanation

def findKthNumber(m: int, n: int, k: int) -> int:
    def enough(num) -> bool:
        # checks if there are at least k numbers <= num in the table
        count = 0
        for val in range(1, m + 1):
            # Count numbers in the current row that are <= num
            add = min(num // val, n)
            if add == 0:  # Early exit optimization
                break
            count += add
        return count >= k  # Return True if we have enough numbers

    # Binary search boundaries: smallest value = 1, largest value = m * n
    left, right = 1, n * m
    while left < right:
        mid = left + (right - left) // 2
        if enough(mid):
            right = mid  # Narrow down to left half
        else:
            left = mid + 1  # Narrow down to right half
    return left

# Test case
m, n, k = 3, 3, 5
print(findKthNumber(m, n, k))  # Output: 3
```

3

10. Find K-th Smallest Pair Distance

Problem Statement

Given an array of integers, find the k-th smallest distance between all pairs of elements. The distance of a pair `(A, B)` is defined as `abs(A - B)`.

Explanation with Test Case

- **Input**: `nums = [1, 3, 1]`, `k = 1`
  All Distances: `[(1, 1) -> 0, (1, 3) -> 2, (3, 1) -> 2]`

The 1st smallest distance is `0`.

- **Output**: `0`

```python
from typing import List

def smallestDistancePair(nums: List[int], k: int) -> int:
    # Define a helper function to determine if there are at least 'k' pairs
    # with a distance <= given distance
    def enough(distance) -> bool:
        # Initialize the count of valid pairs and left pointer `i`
        count, i = 0, 0
        # Iterate over `nums` with right pointer `j`
        for j in range(len(nums)):
            # Move the left pointer `i` to maintain a window
            # where nums[j] - nums[i] <= distance
            while nums[j] - nums[i] > distance:
                i += 1
            # All pairs (i, i+1), ..., (i, j-1), (i, j)
            # have a distance <= given `distance`
            count += j - i
        # Return True if the count of such pairs is at least `k`,
        # False otherwise
        return count >= k

    # Sort the array to facilitate the sliding window technique
    nums.sort()
    # Initial binary search range based on the max distance
    left, right = 0, nums[-1] - nums[0]

    # Perform binary search over the distance values
    while left < right:
        mid = left + (right - left) // 2  # Calculate the middle distance
        # Check if there's enough pairs with max distance <= mid
        if enough(mid):
            # If yes, try smaller distances; move `right` to `mid`
            right = mid
        else:
            # If not, try larger distances; move `left` past `mid`
            left = mid + 1
    # `left` now holds the smallest distance for which
    # there are at least `k` pairs
    return left
```

```
# Test case
nums = [1, 3, 1]
k = 1
print(smallestDistancePair(nums, k))  # Output: 0
```

0

11. Ugly Number III

Problem Statement

Find the n-th ugly number that is divisible by any of the numbers `a`, `b`, or `c`. Use the inclusion-exclusion principle to count numbers.

Explanation with Test Case

- **Input**: `n = 3`, `a = 2`, `b = 3`, `c = 5`
  Ugly Numbers: `[2, 3, 4, 5, 6, ...]`. The 3rd ugly number is `4`.
- **Output**: `4`

```
import math

def nthUglyNumber(n: int, a: int, b: int, c: int) -> int:
    def enough(num) -> bool:
        # Count numbers divisible by a, b, or c using inclusion-exclusion
        total = (
            (num // a) + (num // b) + (num // c) -
            (num // ab) - (num // ac) - (num // bc) + (num // abc)
        )
        return total >= n

    # Calculate least common multiples
    ab = a * b // math.gcd(a, b)
    ac = a * c // math.gcd(a, c)
    bc = b * c // math.gcd(b, c)
    abc = a * bc // math.gcd(a, bc)

    left, right = 1, 2 * 10**9  # Search space
    while left < right:
```

```
        mid = left + (right - left) // 2
        if enough(mid):
            right = mid  # Narrow down to left half
        else:
            left = mid + 1  # Narrow down to right half
    return left

# Test case
n, a, b, c = 3, 2, 3, 5
print(nthUglyNumber(n, a, b, c))  # Output: 4
```

4

12. Find the Smallest Divisor Given a Threshold

Problem Statement

Find the smallest divisor such that dividing each element in the array by the divisor and summing up the results is less than or equal to a given threshold.

Explanation with Test Case

- **Input**: `nums = [1, 2, 5, 9]`, `threshold = 6`
  Divisors tested:
  - `Divisor = 5`: Result = `1 + 1 + 1 + 2 = 5` (valid).

- **Output**: 5 `Explanation: We can get a sum to 17 (1+2+5+9) if the divisor is 1.If the divisor is 4 we can get a sum to 7 (1+1+2+3) and if the divisor is 5 the sum will be 5 (1+1+1+2).`

```
from typing import List

def smallestDivisor(nums: List[int], threshold: int) -> int:
    def condition(divisor) -> bool:
        # Calculate the sum of ceil(num / divisor) for each num in nums
        # (equivalent to (num - 1) // divisor + 1)
        # and check if the total sum is within the threshold.
        total = sum((num - 1) // divisor + 1 for num in nums)
        return total <= threshold
```

```
    # Set the search range between the smallest possible divisor (1) and
    # the maximum value in nums (as a start for the largest possible divisor).
    left, right = 1, max(nums)

    # Perform binary search to find the smallest valid divisor
    while left < right:
        # Calculate the midpoint in the current search range
        mid = left + (right - left) // 2

        # Check if the current midpoint satisfies the condition
        if condition(mid):
            # If true, it means the current divisor is valid
            # We can check to find if there's a smaller valid divisor
            right = mid  # Narrow search to left half
        else:
            # If false, it means the divisor is too small
            # We need to look in the right half
            left = mid + 1

    # When the loop exits, 'left' should point to the smallest divisor
    # that satisfies the condition. This is our answer.
    return left

# Test case
nums = [1, 2, 5, 9]
threshold = 6
print(smallestDivisor(nums, threshold))  # Output: 5
```

5

13. Find Minimum in Rotated Sorted Array II

Problem Statement

You are given an integer array `nums` that is sorted in non-decreasing order. The array is rotated at an unknown pivot and **may contain duplicates**. Find and return the minimum element in the array.

Input

- `nums` (list[int]): A rotated sorted array with possible duplicates.

Output

- `int`: The minimum element in the array.

Example

Input

```
nums = [2, 2, 2, 0, 1]
```

Output

```
0
```

```
def findMin(nums):
    left, right = 0, len(nums) - 1

    while left < right:
        mid = (left + right) // 2

        # If the middle element is greater than the rightmost element,
        # the smallest value must be in the right half.
        if nums[mid] > nums[right]:
            left = mid + 1
        # If the middle element is less than the rightmost element,
        # the smallest value could be at mid or in the left half.
        elif nums[mid] < nums[right]:
            right = mid
        # If nums[mid] == nums[right], we cannot determine the direction;
        # reduce the search space from the right.
        else:
            right -= 1

    # When left equals right, the smallest value is found.
    return nums[left]
```

```
nums = [2, 2, 2, 0, 1]
print(findMin(nums))
```

```
0
```

14. Find Minimum in Rotated Sorted Array

Problem Statement

You are given an integer array `nums` sorted in ascending order but rotated at some pivot. The array does **not contain duplicates**. Find and return the minimum element in the array.

Input

- `nums` (list[int]): A rotated sorted array without duplicates.

Output

- `int`: The minimum element in the array.

Example

Input

```
nums = [4, 5, 6, 7, 0, 1, 2]
```

Output

```
0
```

```
def findMin(nums):
    left, right = 0, len(nums) - 1

    while left < right:
        mid = (left + right) // 2

        # If the middle element is greater than the rightmost element,
        # this indicates the smallest value is to the right of mid.
        if nums[mid] > nums[right]:
            left = mid + 1
        # If the middle element is less than or equal to the rightmost element
        # the smallest value could be at mid or in the left of mid.
        else:
            right = mid
```

```
    # At the end of the loop, left == right, pointing to the smallest element.
    return nums[left]

nums = [4, 5, 6, 7, 0, 1, 2]
print(findMin(nums))  # Output: 0
```

0