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:

```python
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)
```

```python
    # 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`

```python
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.
```

```python
        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
```

```
4
```

Time Complexity: (O(log n)) (binary search halves the search space each iteration).

Space Complexity: (O(1)) (constant space is used).‘

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:

```python
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
```

```python
# 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
```

```
4
```

Time Complexity: O(log n) - The binary search reduces the search space by half at each step.
Space Complexity: O(1) - No additional space is used other than a few variables.

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).

```python
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
```

```python
# Test case
x = 8

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

```
2
```

Time Complexity: O(log(x)) - Binary search reduces the range by half in each step.
Space Complexity: O(1) - Constant space is used as no extra data structures are required.

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`.

```python
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
```

```python
# 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)
```

```
2
```

Time Complexity: O(log n), as the algorithm performs a binary search.
Space Complexity: O(1), as it uses constant extra space.

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
```

Time Complexity: O(n · log(sum(weights) - max(weights))), where n is the number of weights.
Space Complexity: O(1), since no additional space proportional to the input size is used.

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
```

Time Complexity: O(n * log(s)), where n is the number of elements in `nums` and s = sum(nums) - max(nums), due to binary search and the feasibility check.
Space Complexity: O(1), as the algorithm uses constant extra space.

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.

Time Complexity: $O(n \times \log(\max(\text{piles})))$, where n is the number of piles, and $\log(\max(\text{piles}))$ comes from the binary search on speeds.
Space Complexity: $O(1)$, as the algorithm uses a constant amount of extra space.

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
```

```python
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:

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

Step 1: Check for Impossible Cases

```python
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`.

Time Complexity: O(n × log(max(bloomDay))), where n is the length of `bloomDay` and max(bloomDay) is the range of binary search.
Space Complexity: O(1), as no extra space proportional to input size is used.

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`

```python
#### 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
```

Time Complexity: O(m * log(m * n)), where binary search runs in O(log(m * n)) and `enough` function takes O(m) time for each mid.
Space Complexity: O(1), as the solution uses constant extra space.

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`
```

```python
            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
```

Time Complexity: O(n log n + n log d), where n is the size of the array and d is the difference between the maximum and minimum elements in the sorted array. Sorting takes O(n log n), and the binary search with the sliding window runs in O(n log d).

Space Complexity: O(1), as the algorithm uses constant extra space apart from the input array.

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`

```python
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)
```

```python
        )
        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
```

Time Complexity: O(log(2 * 10^9)) due to the binary search on the range [1, 2 * 10^9].

Space Complexity: O(1) since only a constant amount of extra space is used for variables.

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).

```python
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
```

Time Complexity: O(n * log(max(nums)))
Space Complexity: O(1)

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
```

```python
        # 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]
```

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

```
0
```

Time complexity: O(log n) — The search space is halved in each iteration.

Space complexity: O(1) — The algorithm uses only a constant amount of extra space.

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

Time Complexity: O(log n) - The algorithm performs a binary search, reducing the search space by half at each step.

Space Complexity: O(1) - The algorithm only uses a constant amount of extra space.

15. Find Smallest Letter Greater Than Target

Problem Statement

You are given an array of characters `letters` that is sorted in non-decreasing order, and a character `target`. Your task is to find the smallest character in the `letters` array that is lexicographically greater than the given `target`.

The `letters` array is circular, meaning that if there is no character in the array that is greater than `target`, you should return the first character of the array.

Constraints:

- `letters` has a length in the range `[2, 10^4]`.
- `letters` contains only lowercase English letters.
- `target` is a lowercase English letter.

Example

- **Input:** "'python letters = ["c", "f", "j"] target = "a"

`Output: "c"`

Explanation: Since the letters "c", "f", and "j" are greater than "a" and the list is circular, the smallest letter greater than "a" is "c".

```
def nextGreatestLetter(letters, target):
    # Initialize binary search bounds
    low, high = 0, len(letters) - 1

    # Handle the circular case
    if target >= letters[high] or target < letters[low]:
        return letters[0]

    # Perform binary search
    while low < high:
          # Avoid overflow by calculating mid this way
        mid = low + (high - low) // 2

        # If letters[mid] is less than or equal to target,
        # search in the right half
        if letters[mid] <= target:
```

```python
            low = mid + 1
        else:
            # Otherwise, search in the left half
            high = mid

    # At this point, low points to the smallest letter greater than target
    return letters[low]

# Sample Test Cases
print(nextGreatestLetter(["c", "f", "j"], "a"))  # Output: "c"
print(nextGreatestLetter(["c", "f", "j"], "c"))  # Output: "f"
print(nextGreatestLetter(["c", "f", "j"], "d"))  # Output: "f"
print(nextGreatestLetter(["d", "h", "l", "p", "z"], "o"))  # Output: "p"
```

```
c
f
f
p
```

Time Complexity: O(log N) where N is the number of letters, due to the binary search.
Space Complexity: O(1) as the solution only uses a constant amount of extra space.

16. Maximum Profit in Job Scheduling

You are given `n` jobs, where each job is represented by three integers:
- **`startTime[i]`**: The starting time of the job.
- **`endTime[i]`**: The ending time of the job.
- **`profit[i]`**: The profit earned by completing the job.

Goal

Return the **maximum profit** you can earn such that no two jobs overlap.

Problem Constraints

- A job `(i)` ends **before** the start of another job `(j)` if `endTime[i] <= startTime[j]`.

Input

```
startTime = [1, 2, 3, 3]
endTime = [3, 4, 5, 6]
profit = [50, 10, 40, 70]
```

Output

```
120
```

Explanation

- Select the **1st job** `(start=1, end=3, profit=50)` and the **4th job** `(start=3, end=6, profit=70)`.

- Combined profit: `50 + 70 = 120`.
- No other combination gives a higher profit.

Approach

To solve the problem, use **Dynamic Programming (DP)** with **Binary Search Optimization**.

Steps:

1. **Sort Jobs by `endTime`**:
   - Sorting allows efficient binary search for the last non-conflicting job.
2. **Binary Search to Find Non-Conflicting Jobs**:
   - Use `bisect_right` to quickly find the last job whose `endTime` is less than or equal to the current job's `startTime`.
3. **Dynamic Programming Transition**:
   - Maintain a DP array `dp` where `dp[i]` stores the maximum profit achievable considering jobs from `0` to `i`.
   - For each job:
     - Either **skip** the job: `dp[i] = dp[i-1]`.

    - Or **take** the job: `dp[i] = profit[i] + dp[j]` (where `j` is the index of the last non-conflicting job).

4. **Result**:
    - The final result is the maximum profit stored in the last element of the DP array.

```
from bisect import bisect_right

def jobScheduling(startTime, endTime, profit):
    # Combine all jobs into a single list and sort by endTime
    jobs = sorted(zip(endTime, startTime, profit))

    # DP array to store the maximum profit at each step (endTime, max_profit)
    dp = [(0, 0)]  # Initialize with a dummy job (endTime=0, profit=0)

    for end, start, prof in jobs:
        # Use binary search to find the last job that does not conflict
        i = bisect_right(dp, (start, float('inf')))

        # Calculate profit if we take this job
        curr_profit = dp[i-1][1] + prof

        # If taking this job is better than the current max profit, update DP
        if curr_profit > dp[-1][1]:
            dp.append((end, curr_profit))

    # The maximum profit is stored in the last element of DP
    return dp[-1][1]

# Test Case
startTime = [1, 2, 3, 3]
endTime = [3, 4, 5, 6]
profit = [50, 10, 40, 70]

# Output: 120
print(jobScheduling(startTime, endTime, profit))
```

```
120
```

Explanation of the Code

1. **Sorting Jobs**:

- The jobs list is created as (endTime, startTime, profit) tuples and sorted by endTime.

2. **Binary Search**:

   - bisect_right(dp, (start, float('inf'))) finds the index of the last non-conflicting job for the current job.

3. **Dynamic Programming**:

   - dp[i-1][1]: Maximum profit up to the last non-conflicting job.
   - curr_profit: Profit from taking the current job.
   - Compare curr_profit with the current max profit (dp[-1][1]) and update DP if it's greater.

4. **Final Result**:

   - The last element of dp, dp[-1][1], stores the maximum profit.

Example Walkthrough

Given the input:

```
startTime = [1, 2, 3, 3]
endTime = [3, 4, 5, 6]
profit = [50, 10, 40, 70]
```

1. **Sort Jobs**:

   ```
   jobs = [(3, 1, 50), (4, 2, 10), (5, 3, 40), (6, 3, 70)]
   ```

2. **Iterate Through Jobs**:

   - Initialize dp = [(0, 0)].
   - **Job 1** (end=3, start=1, profit=50):
     - No conflicting jobs (i=1), curr_profit = 0 + 50 = 50.
     - Update dp = [(0, 0), (3, 50)].
   - **Job 2** (end=4, start=2, profit=10):
     - No conflicting jobs (i=1), curr_profit = 0 + 10 = 10.
     - Max profit remains 50. No update.
   - **Job 3** (end=5, start=3, profit=40):
     - Last non-conflicting job is Job 1 (i=2), curr_profit = 50 + 40 = 90.

    - Update `dp = [(0, 0), (3, 50), (5, 90)]`.

- **Job 4** `(end=6, start=3, profit=70)`:
    - Last non-conflicting job is Job 1 (`i=2`), `curr_profit = 50 + 70 = 120`.
    - Update `dp = [(0, 0), (3, 50), (5, 90), (6, 120)]`.

3. **Result**:
    - Maximum profit is `dp[-1][1] = 120`.

Output

```
120
```

Time Complexity: O(N log N), where N is the number of jobs, due to sorting the jobs and performing binary search on the DP array.

Space Complexity: O(N), for storing the DP array which holds up to N+1 elements.

17. Time Based Key-Value Store

Problem Statement

Design a **time-based key-value store** that supports the following operations:

1. `set(string key, string value, int timestamp)`

- Stores the **key-value pair** along with the given `timestamp`.

2. `get(string key, int timestamp)`

- Returns the **value** associated with the `key` at the **latest timestamp** the given `timestamp`.
- If there is **no such timestamp**, return an **empty string** (`""`).

Constraints

1. **Timestamps are strictly increasing** for all `set` calls for a specific key.
2. Multiple values can exist for the same key, each associated with a different timestamp.

Example

Sample Input

```
timeMap.set("foo", "bar", 1)
timeMap.get("foo", 1)
timeMap.get("foo", 3)
timeMap.set("foo", "bar2", 4)
timeMap.get("foo", 4)
timeMap.get("foo", 5)
```

Sample Output

```
[null, "bar", "bar", null, "bar2", "bar2"]
```

Explanation

1. **`set("foo", "bar", 1)`**:
   - Stores the value `"bar"` for the key `"foo"` at timestamp `1`.
   - Output: `null` (since `set` does not return a value).
2. **`get("foo", 1)`**:
   - Finds the exact value stored at timestamp `1`.
   - Returns: `"bar"`.
3. **`get("foo", 3)`**:
   - Finds the latest value stored at or before timestamp `3`.
   - Returns: `"bar"` (value at timestamp `1` is the closest).
4. **`set("foo", "bar2", 4)`**:
   - Updates the value for `"foo"` to `"bar2"` at timestamp `4`.
   - Output: `null`.
5. **`get("foo", 4)`**:
   - Finds the exact value stored at timestamp `4`.
   - Returns: `"bar2"`.
6. **`get("foo", 5)`**:
   - Finds the latest value stored at or before timestamp `5`.
   - Returns: `"bar2"` (value at timestamp `4` is the closest).

Detailed Workflow

Operation	Key	Timestamp	Action	Output
`set("foo", "bar", 1)`	`"foo"`	`1`	Store the key-value pair: `{ "foo": [(1, "bar")] }`	`null`
`get("foo", 1)`	`"foo"`	`1`	Find value at timestamp `1`: `"bar"`	`"bar"`
`get("foo", 3)`	`"foo"`	`3`	Find the latest value `3`: `"bar"` (from timestamp `1`)	`"bar"`
`set("foo", "bar2", 4)`	`"foo"`	`4`	Update key-value pair: `{ "foo": [(1, "bar"), (4, "bar2")] }`	`null`
`get("foo", 4)`	`"foo"`	`4`	Find value at timestamp `4`: `"bar2"`	`"bar2"`
`get("foo", 5)`	`"foo"`	`5`	Find the latest value `5`: `"bar2"` (from timestamp `4`)	`"bar2"`

Key Takeaways

1. **Storage Structure**:
   - Store the key-value pairs in a dictionary where:
     - Key: The string `key`.
     - Value: A list of `(timestamp, value)` pairs in **sorted order**.
2. **Efficient Retrieval**:
   - Use binary search to efficiently retrieve the latest value the given timestamp, as timestamps are strictly increasing.
3. **Output Rules**:
   - `set`: Always returns `null` as it only updates the store.
   - `get`: Returns the value or an empty string if no valid timestamp exists.

```python
from collections import defaultdict
import bisect

class TimeMap:

    def __init__(self):
        # Initialize a dictionary to hold all key-value-time mapping
        self.store = defaultdict(list)
```

```
    def set(self, key: str, value: str, timestamp: int) -> None:
        """
        Store the given value and timestamp under the dictionary key.
        """
        # Append (value, timestamp) tuple to the list for this key
        self.store[key].append((value, timestamp))
        # print(self.store)

    def get(self, key: str, timestamp: int) -> str:
        """
        Retrieve the latest value for the given key with
        timestamp <= given timestamp.
        """
        # Check if key exists in the storage.
        if key not in self.store:
            return ""

        # Retrieve the list of (value, timestamp) pairs for this key
        values = self.store[key]

        # Extract timestamps only for binary search
        timestamps = [time for val, time in values]

        # Use bisect_right to find the position where timestamp would fit
        i = bisect.bisect_right(timestamps, timestamp)

        # If i is 0, no timestamps are less than or equal to 'timestamp'
        if i == 0:
            return ""

        # The last valid timestamp less than or
        # equal to 'timestamp' is at index i-1
        return values[i - 1][0]

# Example usage:

# Initialize the time map
timeMap = TimeMap()

# Set operations
 # [null] Expected as set() doesn't return anything
timeMap.set("foo", "bar", 1)
```

```python
# Get operations
# ["bar"] Expected output, value at timestamp 1
print(timeMap.get("foo", 1))
# ["bar"] Expected output, latest value at timestamp <= 3
print(timeMap.get("foo", 3))

# Set operation with a higher timestamp
# [null] Expected as set() doesn't return anything
timeMap.set("foo", "bar2", 4)

# Get operations
# ["bar2"] Expected output, value at timestamp 4
print(timeMap.get("foo", 4))
# ["bar2"] Expected output, latest value at timestamp <= 5
print(timeMap.get("foo", 5))
```

```
bar
bar
bar2
bar2
```

Time Complexity:
- `set()` operation: O(1)
- `get()` operation: O(log N), where N is the number of timestamped entries for the key (due to binary search).

Space Complexity:
- O(N), where N is the total number of entries (key-value-timestamp pairs) stored across all keys.

18. Single Element in a Sorted Array

Problem Statement

The task is to find a unique element in a sorted array where every element appears **exactly twice**, except for one single element that appears **only once**.

Key Points:

1. The array is sorted in **non-decreasing order**.
2. Every element appears exactly twice, except for one element.

Example

Input:

```
[1, 1, 2, 3, 3, 4, 4, 8, 8]
```

Output:

```
2
```

Explanation:

- Each number appears exactly twice except for `2`, which appears only once.

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

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

        # Ensure mid is even for pair comparisons
        if mid % 2 == 1:
            mid -= 1

        # Check pairs
        if nums[mid] == nums[mid + 1]:
            # Single element is after mid
            left = mid + 2
        else:
            # Single element is before or at mid
            right = mid

    # Single element is at left position
    return nums[left]
```

```
# Sample Test Case
print(singleNonDuplicate([1, 1, 2, 3, 3, 4, 4, 8, 8]))  # Output: 2
```

```
2
```

Explanation of the Code:

1. **Initialize Pointers**:
   - `left` is set to the start of the array.
   - `right` is set to the end of the array.
2. **Binary Search**:
   - Narrow down the search using binary search, which runs in **O(log n)** time.
   - **Calculate the Midpoint (`mid`)**:
     - Use `mid = left + (right - left) // 2` to find the middle index.
     - If `mid` is odd, adjust it to be even (`mid -= 1`) to facilitate pair comparisons.
   - **Check Pairs**:
     - If `nums[mid] == nums[mid + 1]`, it means the single element lies **after mid**. Move `left` to `mid + 2`.
     - Otherwise, the single element lies **before or at mid**. Move `right` to `mid`.
3. **Return the Single Element**:
   - When the loop ends, `left` will point to the single element.

Sample Test Cases

Test Case 1:

Input:

```
[3, 3, 7, 7, 10, 11, 11]
```

Output:

```
10
```

Explanation:

- The pairs are complete for `3`, `7`, and `11`.
- The mismatch happens at index 4 (0-based), where `10` is not paired with another `10`.
- The algorithm efficiently narrows down to `10`.

Execution:

```
print(singleNonDuplicate([3, 3, 7, 7, 10, 11, 11]))  # Output: 10
```

How the Algorithm Works:

Iteration Example:

Input:

```
[1, 1, 2, 3, 3, 4, 4, 8, 8]
```

1. **Initial State**:
   - `left = 0`, `right = 8`.
2. **First Iteration**:
   - `mid = 4` (even index).
   - `nums[mid] = 3`, `nums[mid + 1] = 4` (mismatch).
   - Move `right = mid = 4`.
3. **Second Iteration**:
   - `mid = 2` (even index).
   - `nums[mid] = 2`, `nums[mid + 1] = 3` (mismatch).
   - Move `right = mid = 2`.
4. **Third Iteration**:
   - `mid = 0` (even index).
   - `nums[mid] = 1`, `nums[mid + 1] = 1` (match).
   - Move `left = mid + 2 = 2`.
5. **End of Loop**:
   - `left = 2`.
   - Single element is `nums[left] = 2`.

Time Complexity: O(log n), as we are performing binary search. **Space Complexity**: O(1), as we are using only a constant amount of extra space.

19. Median of Two Sorted Arrays

Problem Statement

You are given two sorted arrays, `nums1` and `nums2`, of sizes `m` and `n` respectively. Your task is to find the median of these two arrays. The overall run-time complexity should be **O(log (m+n))**.

Example Inputs and Outputs

Example 1

Input:

```
nums1 = [1, 3]
nums2 = [2]
```

Output:

```
2.0
```

Example 2

Input:

```
nums1 = [1, 2]
nums2 = [3, 4]
```

Output:

```
2.5
```

Solution Explanation

To achieve the required time complexity of **O(log (m+n))**, we use a **binary search approach**. The key idea is to partition the arrays such that: - The left side of the partition contains elements less than or equal to the elements on the right side. - The total number of elements on the left and right sides is balanced.

Steps to Solve:

1. **Partition the Arrays:**
   - Partition `nums1` and `nums2` at indices `i` and `j`, respectively, so that:
     - Left partition of `nums1`: `nums1[0:i]`
     - Right partition of `nums1`: `nums1[i:]`
     - Left partition of `nums2`: `nums2[0:j]`
     - Right partition of `nums2`: `nums2[j:]`
   - Ensure the left partition contains elements less than or equal to those in the right partition.

2. **Use Binary Search:**
   - Perform binary search on the smaller array to find the correct partition.
   - Use the relationship: $[j = \frac{(m+n+1)}{2} - i]$ to calculate `j` based on `i`.

3. **Check Partition Validity:**
   - Ensure:
     - `max(nums1[i-1], nums2[j-1]) <= min(nums1[i], nums2[j])`
   - Adjust `i` (and thus `j`) using binary search.

4. **Calculate the Median:**
   - If the total number of elements is odd: $[\text{median} = \max(\text{left partition})]$
   - If even: $[\text{median} = \frac{\max(\text{left partition}) + \min(\text{right partition})}{2}]$

```python
def findMedianSortedArrays(nums1, nums2):
    # Ensure nums1 is the smaller array for efficient binary search
    if len(nums1) > len(nums2):
        nums1, nums2 = nums2, nums1

    x, y = len(nums1), len(nums2)
    low, high = 0, x

    while low <= high:
        # Partition the arrays
        partitionX = (low + high) // 2
        partitionY = (x + y + 1) // 2 - partitionX

        # Edge cases for out-of-bound partitions
        maxX = float('-inf') if partitionX == 0 else nums1[partitionX - 1]
        minX = float('inf') if partitionX == x else nums1[partitionX]
```

```python
        maxY = float('-inf') if partitionY == 0 else nums2[partitionY - 1]
        minY = float('inf') if partitionY == y else nums2[partitionY]

        # Check if we have found the correct partition
        if maxX <= minY and maxY <= minX:
            # Odd total number of elements
            if (x + y) % 2 == 1:
                return max(maxX, maxY)
            # Even total number of elements
            else:
                return (max(maxX, maxY) + min(minX, minY)) / 2
        elif maxX > minY:  # Move partitionX to the left
            high = partitionX - 1
        else:  # Move partitionX to the right
            low = partitionX + 1
# Test Case 1
nums1 = [1, 3]
nums2 = [2]
print(findMedianSortedArrays(nums1, nums2))  # Output: 2.0

# Test Case 2
nums1 = [1, 2]
nums2 = [3, 4]
print(findMedianSortedArrays(nums1, nums2))  # Output: 2.5
```

```
2
2.5
```

Time Complexity: O(log(min(n, m))) where n and m are the lengths of nums1 and nums2, as binary search is applied on the smaller array.

Space Complexity: O(1), since the algorithm uses a constant amount of extra space.

Function: `findMedianSortedArrays(nums1, nums2)`

This function finds the median of two sorted arrays using **binary search** for optimal performance. Here's a step-by-step breakdown:

Step 1: Ensure `nums1` is the smaller array

```
if len(nums1) > len(nums2):
    nums1, nums2 = nums2, nums1
```

- **Purpose**: Always perform binary search on the smaller array (`nums1`) to minimize time complexity.

Step 2: Initialize variables

```
x, y = len(nums1), len(nums2)
low, high = 0, x
```

- `x`, `y`: Store the lengths of `nums1` and `nums2`, respectively.
- `low`, `high`: Define the search bounds for binary search within `nums1`.

Step 3: Perform binary search

```
while low <= high:
```

- Loop to adjust partitions until the correct median is found.

Step 3.1: Calculate partitions

```
partitionX = (low + high) // 2
partitionY = (x + y + 1) // 2 - partitionX
```

- `partitionX`: Index for partitioning `nums1`.
- `partitionY`: Corresponding partition index in `nums2` based on the total number of elements.

Step 3.2: Handle edge cases for boundaries

```
maxX = float('-inf') if partitionX == 0 else nums1[partitionX - 1]
minX = float('inf') if partitionX == x else nums1[partitionX]

maxY = float('-inf') if partitionY == 0 else nums2[partitionY - 1]
minY = float('inf') if partitionY == y else nums2[partitionY]
```

- Use `-∞` (negative infinity) and `+∞` (positive infinity) to handle out-of-bounds conditions:
  - `maxX`, `minX`: Values around the partition in `nums1`.
  - `maxY`, `minY`: Values around the partition in `nums2`.

Step 3.3: Check for the correct partition

```
if maxX <= minY and maxY <= minX:
```

- Condition: The correct partition is found when:
  - `maxX`   `minY` **and** `maxY`   `minX`.

Step 4: Calculate the median

Case 1: Odd total number of elements

```
if (x + y) % 2 == 1:
    return max(maxX, maxY)
```

- Median is the **maximum of the left partitions** (`maxX`, `maxY`).

Case 2: Even total number of elements

```
else:
    return (max(maxX, maxY) + min(minX, minY)) / 2
```

- Median is the **average of the maximum of the left partitions and the minimum of the right partitions**.

Step 5: Adjust partitions

Move `partitionX` left:

```
elif maxX > minY:
    high = partitionX - 1
```

- Shift `partitionX` to the left.

Move `partitionX` right:

```
else:
    low = partitionX + 1
```

- Shift `partitionX` to the right.

Example Test Cases

Test Case 1

```
nums1 = [1, 3]
nums2 = [2]
print(findMedianSortedArrays(nums1, nums2))  # Output: 2.0
```

- Merged array: `[1, 2, 3]`
- Median: `2.0`.

Test Case 2

```
nums1 = [1, 2]
nums2 = [3, 4]
print(findMedianSortedArrays(nums1, nums2))  # Output: 2.5
```

- Merged array: `[1, 2, 3, 4]`
- Median: `(2 + 3) / 2 = 2.5`.

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

You are given an array of integers `nums` sorted in **non-decreasing order**. Your task is to find the **starting** and **ending position** of a given target value `target` in the array.

If the target is not present in the array, return `[-1, -1]`.

Example 1

Input:

```
nums = [5,7,7,8,8,10]
target = 8
```

Output:

```
[3, 4]
```

Explanation: The target value `8` occurs at indices `3` and `4`. So the starting position is `3`, and the ending position is `4`.

Example 2

Input:

```
nums = [5,7,7,8,8,10]
target = 6
```

Output:

```
[-1, -1]
```

Explanation: The target value `6` does not exist in the array. Therefore, the output is `[-1, -1]`.

Example 3

Input:

```
nums = []
target = 0
```

Output:

```
[-1, -1]
```

Explanation: The array is empty, so there is no occurrence of the target value `0`.

Approach and Explanation

The goal is to design an **efficient algorithm** with a runtime complexity of ($O(log n)$). Since the array is sorted, we can use **binary search** to locate the target value efficiently. Here's how:

Steps:

1. **Find the first occurrence** of the target:
   - Use binary search to locate the leftmost position of the target.
   - If the target is found, move the search range leftwards to check for earlier occurrences.
2. **Find the last occurrence** of the target:
   - Use binary search again, but this time move the search range rightwards to find the last occurrence.
3. **Combine the results**:
   - If the target exists, the first and last indices will be returned.
   - If the target doesn't exist, return `[-1, -1]`.

Binary Search Logic

1. **Initialization**:
   - Set `left` to the beginning of the array and `right` to the end of the array.
2. **Middle Calculation**:
   - Compute the middle index as `mid = left + (right - left) // 2`.
3. **Adjust the Search Range**:

- If `nums[mid] == target`:
  - Save `mid` as a potential answer.
  - Adjust the search range based on whether you're finding the first or last occurrence.
- If `nums[mid] < target`, move the search right (`left = mid + 1`).
- If `nums[mid] > target`, move the search left (`right = mid - 1`).

4. **End Condition**:

   - The loop ends when `left > right`.

```python
from typing import List

class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        # Find the first occurrence of the target
        first_idx = self.binary_search(nums, target, False)
        # Find the last occurrence of the target
        last_idx = self.binary_search(nums, target, True)

        # If the first and last occurrences are 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

        # Perform binary search
        while left <= right:
            mid = left + (right - left) // 2

            if nums[mid] == target:
                # When the target is found, record the candidate
                candidate = mid
                # Adjust search for either first or last position
                if find_last:
                    left = mid + 1  # Move right to find last occurrence
                else:
                    right = mid - 1  # Move left to find first occurrence
            elif nums[mid] < target:
                left = mid + 1  # Move right to look for target
            else:
                right = mid - 1  # Move left to look for target
```

```python
        return candidate

# Example usage:
solver = Solution()

# Example 1
result = solver.searchRange([5,7,7,8,8,10], 8)
print(result)  # Output: [3, 4]

# Example 2
result = solver.searchRange([5,7,7,8,8,10], 6)
print(result)  # Output: [-1, -1]

# Example 3
result = solver.searchRange([], 0)
print(result)  # Output: [-1, -1]
```

```
[3, 4]
[-1, -1]
[-1, -1]
```

Time Complexity: O(log n) for each binary search, so overall O(log n) for finding both the first and last occurrence.

Space Complexity: O(1) as we are using only a constant amount of extra space.

21. Search in Rotated Sorted Array

You are given: - An integer array `nums` sorted in ascending order (with distinct values). - An integer `target`.

The array `nums` is **rotated** at some unknown pivot (e.g., `[0,1,2,4,5,6,7]` might become `[4,5,6,7,0,1,2]`).

Your task: - Find the index of `target` in `nums`. - If `target` is not in `nums`, return `-1`.

The solution **must** have a runtime complexity of **O(log n)**.

Example Input and Output

Example 1

Input:
`nums = [4,5,6,7,0,1,2], target = 0`
Output:
`4`
Explanation: The target `0` is found at index `4`.

Example 2

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

Example 3

Input:
`nums = [1], target = 0`
Output:
`-1`
Explanation: The target `0` is not present in the single-element array.

Explanation and Approach

The goal is to perform the search in **O(log n)** time complexity. This suggests using a **modified binary search** due to the sorted and rotated nature of the array.

Step-by-Step Approach

1. **Initialize Pointers**:
    - Set `left` to the first index (`0`).
    - Set `right` to the last index (`len(nums) - 1`).
2. **Iterative Search**:
    - Use a `while` loop: `while left <= right`.

- Calculate the midpoint:
  `mid = (left + right) // 2`.

3. **Check Midpoint**:
   - If `nums[mid] == target`, return `mid` (we found the target).

4. **Determine the Sorted Half**:
   - **Left Half is Sorted**:
     - If `nums[left] <= nums[mid]`, then the **left half** is sorted.
     - Check if the `target` lies in this range (`nums[left] <= target < nums[mid]`):
       * If **yes**, move `right` pointer to `mid - 1`.
       * Otherwise, move `left` pointer to `mid + 1`.
   - **Right Half is Sorted**:
     - Otherwise, the **right half** is sorted.
     - Check if the `target` lies in this range (`nums[mid] < target <= nums[right]`):
       * If **yes**, move `left` pointer to `mid + 1`.
       * Otherwise, move `right` pointer to `mid - 1`.

5. **Repeat Until Found**:
   - Continue the loop until `left > right`.

6. **Target Not Found**:
   - If the loop ends without finding the target, return `-1`.

```python
def search(nums, target):
    # Initialize pointers for the binary search
    left, right = 0, len(nums) - 1

    # Continue searching while there is a valid range
    while left <= right:
        # Compute the mid-point index of the current search range
        mid =  left + (right - left) // 2

        # If the mid element matches the target, return its index
        if nums[mid] == target:
            return mid

        # Determine if the left half is properly sorted
        if nums[left] <= nums[mid]:  # Left half is sorted
            # Check if the target is within the sorted left half
            if nums[left] <= target < nums[mid]:
```

```
                right = mid - 1  # Narrow the search to the left half
            else:
                left = mid + 1  # Narrow the search to the right half
        else:  # Right half must be sorted
            # Check if the target is within the sorted right half
            if nums[mid] < target <= nums[right]:
                left = mid + 1  # Narrow the search to the right half
            else:
                right = mid - 1  # Narrow the search to the left half

    # Return -1 if the target is not found in the array
    return -1


# Test case
# Simple rotated array example
nums = [6, 7, 1, 2, 3, 4, 5]
target = 3
# Explanation: The target 3 is positioned at index 4 in the array.
# The function should return 4.
print(search(nums, target))  # Expected output: 4

# Another test case
nums = [8, 9, 10, 0, 1, 2, 3, 4, 5, 6, 7]
target = 9
# Explanation: In this rotated array, the target 9 is located at index 1.
# The function should return 1.
print(search(nums, target))  # Expected output: 1
```

```
4
1
```

Time Complexity: O(log n), because we are performing a binary search on a rotated sorted array.

Space Complexity: O(1), as we are using only a constant amount of extra space.

22. Search a 2D Matrix

Write an efficient algorithm to search for a value in an **(m times n)** matrix. The matrix has the following properties:

1. Integers in each row are sorted from left to right.
2. The first integer of each row is greater than the last integer of the previous row.

Example

Input:

```
matrix = [
  [1, 3, 5, 7],
  [10, 11, 16, 20],
  [23, 30, 34, 60]
]
target = 3
```

Output:

```
true
```

Concept:

- The matrix can be visualized as a single sorted 1D array because of the properties:
  - Rows are sorted.
  - The first element of each row is greater than the last element of the previous row.

```python
from typing import List

class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        # Check if the matrix is empty or the first row is empty
        if not matrix or len(matrix) == 0 or len(matrix[0]) == 0:
            return False

        # Start from the top-right corner of the matrix
        row, col = 0, len(matrix[0]) - 1

        # Loop until either the row or column goes out of bounds
        while col >= 0 and row < len(matrix):
            current_value = matrix[row][col]

            # If the target is found, return True
```

```
            if current_value == target:
                return True

            # If the current value is less than the target,
            # move down to the next row
            elif current_value < target:
                row += 1

            # If the current value is greater than the target,
            # move left to the previous column
            else:
                col -= 1

        # Return False if the target is not found
        return False

# Sample test case
if __name__ == "__main__":
    matrix = [
        [1, 3, 5, 7],
        [10, 11, 16, 20],
        [23, 30, 34, 60]
    ]
    target = 13

    solution = Solution()
    result = solution.searchMatrix(matrix, target)
    print("Test Result:", result)  # Expected output: False
```

```
Test Result: False
```

Time Complexity: O(m + n), where m is the number of rows and n is the number of columns in the matrix, as we can move at most m steps down and n steps left.

Space Complexity: O(1), as the algorithm uses constant extra space.

23. Search a 2D Matrix II

You are given an `m x n` matrix with the following properties:

1. Each row is sorted in **non-decreasing order**.

2. Each column is sorted in **non-decreasing order**.

The task is to write a function that efficiently determines if a target value exists in the matrix.

Example

Input

```
matrix = [
  [1, 4, 7, 11, 15],
  [2, 5, 8, 12, 19],
  [3, 6, 9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]
target = 5
```

Output

```
True
```

Explanation:

In the given matrix, the number `5` is present at position `(1, 1)` (row 1, column 1 in 0-based indexing). Thus, the output is `True`.

```python
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        # Check if the matrix is empty
        if not matrix or len(matrix) == 0:
            return False

        # Start from the top-right corner of the matrix
        row = 0
        col = len(matrix[0]) - 1

        # Traverse the matrix
        while col >= 0 and row < len(matrix):
            if matrix[row][col] == target:
```

```python
                # If the element is found, return True
                return True
            elif matrix[row][col] > target:
                # Move left if the current element is greater than the target
                col -= 1
            else:
                # Move down if the current element is less than the target
                row += 1

        # If the element is not found, return False
        return False
```

```python
matrix = [
    [1, 4, 7, 11, 15],
    [2, 5, 8, 12, 19],
    [3, 6, 9, 16, 22],
    [10, 13, 14, 17, 24],
    [18, 21, 23, 26, 30]
]
target = 5
```

```python
solution = Solution()
result = solution.searchMatrix(matrix, target)
print(result)  # Output: True
```

```
True
```

Time Complexity: O(m + n), where m is the number of rows and n is the number of columns in the matrix.
Space Complexity: O(1), as we are using only a constant amount of extra space.

24. Peak Index in a Mountain Array

Problem Statement

An array `arr` is a **mountain array** if the following properties hold:

1. `arr.length >= 3`
2. There exists some index `i` with `0 < i < arr.length - 1` such that:

- `arr[0] < arr[1] < ... < arr[i - 1] < arr[i]`
- `arr[i] > arr[i + 1] > ... > arr[arr.length - 1]`

Given a mountain array `arr`, return the index `i` such that: - `arr[0] < arr[1] < ... < arr[i - 1] < arr[i] > arr[i + 1] > ... > arr[arr.length - 1]`.

You must solve the problem in **O(log(arr.length))** time complexity.

Example 1

Input:
`arr = [0, 1, 0]`

Output:
`1`

Example 2

Input:
`arr = [0, 2, 1, 0]`

Output:
`1`

Example 3

Input:
`arr = [0, 10, 5, 2]`

Output:
`1`

```python
class Solution:
    def peakIndexInMountainArray(self, arr: List[int]) -> int:
        left, right = 0, len(arr) - 1

        # Perform a binary search
        while left < right:
            # Calculate the mid index
            mid = left + (right - left) // 2

            # If the element at mid is less than the element at mid + 1,
            # it means we are in the ascending part of the mountain,
```

```
            # so we move the left pointer to mid + 1.
            if arr[mid] < arr[mid + 1]:
                left = mid + 1
            else:
                # If arr[mid] >= arr[mid + 1], it means we are in the
                # descending part or at the peak of the mountain.
                # So we move the right pointer to mid.
                right = mid

        # When the loop ends, left will be pointing at the peak index.
        return left

arr = [0, 2, 5, 3, 1]
solution = Solution()
result = solution.peakIndexInMountainArray(arr)
print("Peak index is:", result)
```

```
Peak index is: 2
```

Time Complexity: O(log n), because we perform a binary search, halving the search space at each step.
Space Complexity: O(1), as we are using only a constant amount of extra space.

25. Find Peak Element

You are given an integer array `nums` where `nums[i]` `nums[i+1]` for all valid `i`. A peak element is an element that is strictly greater than its neighbors.

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

You may imagine `nums[-1] = -∞` and `nums[n] = -∞` where `n` is the length of the array. This means that the first and last elements in the array are treated as having one imaginary neighbor that is `-∞`.

You must write a solution with `O(log n)` time complexity.

Example 1:

Input:

```
nums = [1, 2, 3, 1]
```

Output:

```
2
```

Explanation: - `nums[2] = 3` is a peak element because it is greater than its neighbors `nums[1] = 2` and `nums[3] = 1`.

Example 2:

Input:

```
nums = [1, 2, 1, 3, 5, 6, 4]
```

Output:

```
5
```

Explanation: - `nums[1] = 2` is a peak element because it is greater than its neighbors `nums[0] = 1` and `nums[2] = 1`. - `nums[5] = 6` is also a peak element because it is greater than its neighbors `nums[4] = 5` and `nums[6] = 4`.

You can return either `1` or `5`.

Constraints:

1. `1 <= nums.length <= 1000`
2. `-2^31 <= nums[i] <= 2^31 - 1`
3. `nums[i]   nums[i + 1]` for all valid `i`.

Follow-Up:

- Can you implement a solution with `O(log n)` time complexity?

```python
from typing import List

class Solution:
    def findPeakElement(self, nums: List[int]) -> int:
        # Initialize left and right pointers
        left, right = 0, len(nums) - 1

        # Perform binary search
        while left < right:
            # Calculate the middle index
            mid = left + (right - left) // 2

            # Check if the mid element is less than the next element
            if nums[mid] < nums[mid + 1]:
                # If true, it means the peak element is on the right side
                left = mid + 1
            else:
                # If false, it means the peak element is on the left side or
                # it is the mid itself
                right = mid

        # When left == right, we have found the peak element
        return left

# Test the function with a test case
if __name__ == "__main__":
    solution = Solution()
    nums = [1, 2, 3, 1]
    peak_index = solution.findPeakElement(nums)
    print(f"The peak index is: {peak_index}")
```

```
The peak index is: 2
```

Time Complexity: O(log n) — The binary search reduces the problem size by half in each iteration.

Space Complexity: O(1) — Only a constant amount of extra space is used for the pointers.

Finding pairs

```
# Function to find all pairs in the array that sum up to a target value
def pair(array):
    # Define the target sum
    target = 10

    # Initialize two pointers
    left = 0  # Start pointer at the beginning of the array
    right = len(array) - 1  # End pointer at the last element of the array

    # List to store the resulting pairs
    output = []

    # Iterate until the two pointers meet
    while left < right:
        # Calculate the sum of the elements at the two pointers
        current_sum = array[left] + array[right]

        # If the sum matches the target, add the pair to the output
        if current_sum == target:
            output.append((array[left], array[right]))
            # Move both pointers inward
            left += 1
            right -= 1
        # If the sum is less than the target, move the left pointer
        # to increase the sum
        elif current_sum < target:
            left += 1
        # If the sum is greater than the target, move the right pointer
        # to decrease the sum
        else:
            right -= 1

    # Return the list of pairs
    return output

# Input array
array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Call the function and store the result
```

```
result = pair(array)

# Print the result
print(result)  # Expected Output: [(1, 9), (2, 8), (3, 7), (4, 6)]
```

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

Explanation:

1. **Input**:
   - The input `array` is assumed to be sorted in ascending order. If the input is unsorted, sort it first using `array.sort()`.
2. **Target**:
   - The `target` is the sum we want to find pairs for. In this case, it's 10.
3. **Two Pointers**:
   - `left` starts at the beginning (index 0).
   - `right` starts at the end (last index).
   - The loop continues until the two pointers meet (`left < right`).
4. **Logic**:
   - Compare the sum of the two pointers' values with the target:
     - If equal, add the pair to the output and move both pointers inward.
     - If less than the target, increment `left` to consider a larger number.
     - If greater than the target, decrement `right` to consider a smaller number.
5. **Output**:
   - The function returns a list of tuples, each representing a pair whose sum equals the target.

Output:

When you run this code, it will output:

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

This solution is efficient with (O(n)) time complexity and requires no additional space beyond the output list.

Time Complexity: O(n), where n is the number of elements in the array, because we iterate through the array once using two pointers.

Space Complexity: O(k), where k is the number of pairs found, as we store the pairs in the output list.