CODING INTERVIEW PREPARATION

Day 4: Binary Search & Trees (Part 1)

Complete Study Guide with Multiple Solutions

TABLE OF CONTENTS

Page

DAY 4: BINARY SEARCH & TREES (PART 1)

Key Concepts for Today:

- **Binary Search** algorithm and its variations
- **Rotated Array Search** techniques
- **2D Matrix Search** strategies
- **Search Space Reduction** principles
- **Binary Search on Answer** patterns
- **Time-based Data Structures** with search capabilities

Problems Index:

1. **Binary Search** (Easy)
2. **Search a 2D Matrix** (Medium)
3. **Koko Eating Bananas** (Medium)
4. **Find Minimum in Rotated Sorted Array** (Medium)
5. **Search in Rotated Sorted Array** (Medium)
6. **Time Based Key-Value Store** (Medium)
7. **Median of Two Sorted Arrays** (Hard)

CORE CONCEPTS EXPLANATION

Binary Search Fundamentals

Definition: A search algorithm that finds the position of a target value in a sorted array by repeatedly dividing the search interval in half.

Core Algorithm:

```
int BinarySearch(int[] arr, int target)
{
    int left = 0, right = arr.Length - 1;

    while (left <= right)
    {
        int mid = left + (right - left) / 2; // Avoid overflow

        if (arr[mid] == target) return mid;
        else if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }

    return -1; // Not found
}
```

Key Properties:

- **Time Complexity: O(log n)** - eliminates half the search space each iteration
- **Space Complexity: O(1)** for iterative, **O(log n)** for recursive
- **Prerequisite:** Array must be sorted
- **Invariant:** Target, if exists, is always within `[left, right]` range

Binary Search Variations

1. Find First/Last Occurrence:

- Modify standard binary search to continue searching after finding target
- Useful for duplicates and range queries

2. Binary Search on Answer:

- Search for optimal value in a range of possible answers
- Used when direct calculation is difficult but verification is easy
- Examples: Koko eating bananas, capacity problems

3. Rotated Array Search:

- Modified binary search for arrays rotated at unknown pivot
- Key insight: At least one half is always sorted

Search Space Analysis

Linear Search Space:

- Traditional arrays, lists
- Search through contiguous elements

2D Matrix Search:

- Row-wise and column-wise sorted matrices
- Treat as flattened 1D array or use elimination techniques

Answer Space Search:

- When searching for optimal value rather than element
- Define min/max bounds and binary search on possible answers

Common Binary Search Patterns

Pattern 1: Exact Match

```
while (left <= right) {
    int mid = left + (right - left) / 2;
    if (arr[mid] == target) return mid;
    // Adjust left or right
}
```

Pattern 2: Find First/Last Position

```
while (left < right) {
    int mid = left + (right - left) / 2;
    if (condition) right = mid; // or left = mid + 1
```

```
    else left = mid + 1; // or right = mid
}
```

Pattern 3: Binary Search on Answer

```
while (left < right) {
    int mid = left + (right - left) / 2;
    if (canAchieve(mid)) right = mid;
    else left = mid + 1;
}
```

Why These Concepts Matter in Interviews

Algorithmic Thinking:

- Demonstrates understanding of divide-and-conquer strategy
- Shows ability to optimize from O(n) to O(log n)
- Tests edge case handling and boundary management

Problem-Solving Patterns:

- Many complex problems reduce to binary search variants
- Understanding when to apply binary search vs linear search
- Ability to modify algorithm for specific constraints

Real-world Applications:

- **Database indexing**: B-trees and binary search trees
- **System design**: Load balancing, resource allocation
- **Machine Learning**: Hyperparameter optimization
- **Graphics**: Collision detection, ray tracing

PROBLEM 1: BINARY SEARCH

Problem Statement:

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

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

Test Cases:

```
Example 1:
Input: nums = [-1,0,3,5,9,12], target = 9
Output: 4
Explanation: 9 exists in nums and its index is 4
```

```
Example 2:
Input: nums = [-1,0,3,5,9,12], target = 2
Output: -1
Explanation: 2 does not exist in nums so return -1

Example 3:
Input: nums = [5], target = 5
Output: 0
Explanation: Single element array with matching target
```

Interview Questions & Answers:

1. **Q:** "How do you avoid integer overflow when calculating mid?"
   A: Use `mid = left + (right - left) / 2` instead of `mid = (left + right) / 2` to prevent overflow when left and right are large.
2. **Q:** "What's the difference between `left <= right` and `left < right` in the loop condition?"
   A: `left <= right` is for exact search (can return -1), `left < right` is for finding positions/boundaries (always returns valid index).
3. **Q:** "How do you handle duplicate elements?"
   A: Standard binary search returns any occurrence. For first/last occurrence, modify the algorithm to continue searching after finding target.
4. **Q:** "What if the array is empty?"
   A: Return -1 immediately since target cannot exist in empty array.
5. **Q:** "Can you implement this recursively?"
   A: Yes, but iterative is preferred due to O(1) space complexity vs O(log n) for recursive.

Approach 1: Linear Search

Idea: Iterate through array from left to right until target is found or end is reached.

Time Complexity: O(n) - potentially scan entire array
Space Complexity: O(1) - no extra space needed

```
using System;

public class Solution
{
    public int Search(int[] nums, int target)
    {
        // Linear search through the array
        for (int i = 0; i < nums.Length; i++)
        {
            if (nums[i] == target)
            {
                return i;
            }

            // Early termination if we've passed the target
```

```
            // (since array is sorted)
            if (nums[i] > target)
            {
                return -1;
            }
        }

        return -1; // Target not found
    }
}
```

Approach 2: Recursive Binary Search

Explanation: Implement binary search using recursion for educational purposes, though iterative is more efficient.

Time Complexity: O(log n) - divide search space by half each time
Space Complexity: O(log n) - recursion call stack

```
using System;

public class Solution
{
    public int Search(int[] nums, int target)
    {
        return BinarySearchRecursive(nums, target, 0, nums.Length - 1);
    }

    private int BinarySearchRecursive(int[] nums, int target, int left, int right)
    {
        // Base case: search space exhausted
        if (left > right) return -1;

        // Calculate middle index
        int mid = left + (right - left) / 2;

        if (nums[mid] == target)
        {
            return mid;
        }
        else if (nums[mid] < target)
        {
            // Search right half
            return BinarySearchRecursive(nums, target, mid + 1, right);
        }
        else
        {
            // Search left half
            return BinarySearchRecursive(nums, target, left, mid - 1);
        }
    }
}
```

Approach 3: Optimal - Iterative Binary Search

Detailed Reasoning: Iterative binary search is optimal because it achieves O(log n) time complexity with O(1) space complexity. This is the standard implementation used in production systems.

Time Complexity: O(log n) - eliminate half of remaining elements each iteration
Space Complexity: O(1) - only using a few variables

```
using System;

public class Solution
{
    /// <summary>
    /// Performs binary search on sorted array
    /// Time: O(log n), Space: O(1)
    /// </summary>
    public int Search(int[] nums, int target)
    {
        // Handle edge case
        if (nums == null || nums.Length == 0) return -1;

        int left = 0;
        int right = nums.Length - 1;

        while (left <= right)
        {
            // Calculate mid point, avoiding integer overflow
            int mid = left + (right - left) / 2;

            if (nums[mid] == target)
            {
                return mid; // Found target
            }
            else if (nums[mid] < target)
            {
                // Target is in right half
                left = mid + 1;
            }
            else
            {
                // Target is in left half
                right = mid - 1;
            }
        }

        // Target not found
        return -1;
    }
}
```

Key Takeaways:

- **Overflow prevention**: Use `left + (right - left) / 2` for mid calculation
- **Loop invariant**: Target is always within `[left, right]` if it exists
- **Termination condition**: `left <= right` for exact search problems

PROBLEM 2: SEARCH A 2D MATRIX

Problem Statement:

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

- Each row is sorted in non-decreasing order.
- The first integer of each row is greater than the last integer of the previous row.

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

You must write a solution in **O(log(m * n))** time complexity.

Test Cases:

```
Example 1:
Input: matrix = [[1,4,7,11],[2,5,8,12],[3,6,9,16],[10,13,14,17]], target = 5
Output: true
Explanation: 5 is found at position (1,1)

Example 2:
Input: matrix = [[1,4,7,11],[2,5,8,12],[3,6,9,16],[10,13,14,17]], target = 13
Output: true
Explanation: 13 is found at position (3,1)

Example 3:
Input: matrix = [[1,4,7,11],[2,5,8,12],[3,6,9,16],[10,13,14,17]], target = 20
Output: false
Explanation: 20 is not in the matrix
```

Interview Questions & Answers:

1. **Q:** "How do you convert 2D coordinates to 1D index and vice versa?"
   A: For m×n matrix: `1D_index = row * n + col`, `row = 1D_index / n`, `col = 1D_index % n`.
2. **Q:** "Why can we treat this as a 1D sorted array?"
   A: Because first element of each row > last element of previous row, the entire matrix forms one sorted sequence when read row by row.
3. **Q:** "What if the matrix doesn't have the special property (each row start > previous row end)?"
   A: Then we'd need different approach like starting from top-right corner and eliminating rows/columns.

4. **Q:** "How do you handle empty matrix or invalid dimensions?"
   A: Check for null matrix, zero rows, or zero columns and return false immediately.
5. **Q:** "Can you solve this by searching for the correct row first?"
   A: Yes, binary search for row, then binary search within that row, but treating as 1D is simpler and equally efficient.

Approach 1: Linear Search

Idea: Search through each row linearly, or search through entire matrix element by element.

Time Complexity: O(m × n) - potentially check every element
Space Complexity: O(1) - no extra space needed

```
using System;

public class Solution
{
    public bool SearchMatrix(int[][] matrix, int target)
    {
        if (matrix == null || matrix.Length == 0 || matrix[0].Length == 0)
            return false;

        int m = matrix.Length;
        int n = matrix[0].Length;

        // Search through each element
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < n; j++)
            {
                if (matrix[i][j] == target)
                {
                    return true;
                }

                // Early termination if current element > target
                if (matrix[i][j] > target)
                {
                    return false;
                }
            }
        }

        return false;
    }
}
```

Approach 2: Binary Search on Rows, Then Columns

Explanation: First use binary search to find the correct row, then binary search within that row.

Time Complexity: O(log m + log n) = O(log(m×n)) - two binary searches
Space Complexity: O(1) - only using variables

```
using System;

public class Solution
{
    public bool SearchMatrix(int[][] matrix, int target)
    {
        if (matrix == null || matrix.Length == 0 || matrix[0].Length == 0)
            return false;

        int m = matrix.Length;
        int n = matrix[0].Length;

        // Binary search for the correct row
        int top = 0, bottom = m - 1;
        int targetRow = -1;

        while (top <= bottom)
        {
            int mid = top + (bottom - top) / 2;

            if (matrix[mid][0] <= target && target <= matrix[mid][n - 1])
            {
                targetRow = mid;
                break;
            }
            else if (matrix[mid][0] > target)
            {
                bottom = mid - 1;
            }
            else
            {
                top = mid + 1;
            }
        }

        if (targetRow == -1) return false;

        // Binary search within the found row
        int left = 0, right = n - 1;

        while (left <= right)
        {
            int mid = left + (right - left) / 2;

            if (matrix[targetRow][mid] == target)
            {
                return true;
            }
```

```
            else if (matrix[targetRow][mid] < target)
            {
                left = mid + 1;
            }
            else
            {
                right = mid - 1;
            }
        }

        return false;
    }
}
```

Approach 3: Optimal - Treat as 1D Sorted Array

Detailed Reasoning: Since the matrix has the special property where each row's first element is greater than the previous row's last element, we can treat the entire matrix as one sorted 1D array and apply standard binary search with coordinate conversion.

Time Complexity: O(log(m × n)) - single binary search on virtual 1D array
Space Complexity: O(1) - only using index variables

```
using System;

public class Solution
{
    /// <summary>
    /// Searches 2D matrix by treating it as flattened 1D sorted array
    /// Time: O(log(m*n)), Space: O(1)
    /// </summary>
    public bool SearchMatrix(int[][] matrix, int target)
    {
        // Input validation
        if (matrix == null || matrix.Length == 0 || matrix[0].Length == 0)
            return false;

        int m = matrix.Length;    // Number of rows
        int n = matrix[0].Length; // Number of columns

        // Binary search on virtual 1D array of size m*n
        int left = 0;
        int right = m * n - 1;

        while (left <= right)
        {
            int mid = left + (right - left) / 2;

            // Convert 1D index to 2D coordinates
            int midRow = mid / n;
            int midCol = mid % n;
            int midValue = matrix[midRow][midCol];

            if (midValue == target)
```

```
            {
                return true;
            }
            else if (midValue < target)
            {
                left = mid + 1;
            }
            else
            {
                right = mid - 1;
            }
        }

        return false;
    }
}
```

Key Takeaways:

- **Coordinate conversion**: Master 1D ↔ 2D index transformations
- **Matrix properties**: Exploit special properties to simplify algorithms
- **Virtual flattening**: Treat 2D structure as 1D when properties allow

PROBLEM 3: KOKO EATING BANANAS

Problem Statement:

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

Koko can decide her bananas-per-hour eating speed of `k`. Each hour, she chooses some pile of bananas and eats `k` bananas from that pile. If the pile has less than `k` bananas, she eats all of them for that hour and won't eat any more bananas during that hour.

Koko likes to eat slowly but wants to finish all the bananas before the guards come back. Return the minimum integer `k` such that she can eat all the bananas within `h` hours.

Test Cases:

```
Example 1:
Input: piles = [3,6,7,11], h = 8
Output: 4
Explanation: If Koko eats at speed 4, she takes ceil(3/4) + ceil(6/4) + ceil(7/4) + ceil(

Example 2:
Input: piles = [30,11,23,4,20], h = 5
Output: 30
Explanation: If Koko eats at speed 30, she can finish each pile in 1 hour, totaling 5 hou

Example 3:
Input: piles = [30,11,23,4,20], h = 6
```

```
Output: 23
Explanation: With speed 23, she takes ceil(30/23) + ceil(11/23) + ceil(23/23) + ceil(4/23
```

Interview Questions & Answers:

1. **Q:** "How do you determine the search space for eating speed?"
   A: Minimum speed is 1 (must eat at least 1 banana per hour), maximum is the largest pile (can finish any pile in 1 hour).
2. **Q:** "How do you calculate hours needed for a given eating speed?"
   A: For each pile, use `ceil(pile_size / eating_speed)` which equals `(pile_size + speed - 1) / speed` using integer division.
3. **Q:** "Why is this a binary search problem?"
   A: We're searching for the minimum speed in a range where we can verify if a speed works, but calculating directly is complex.
4. **Q:** "What if h is less than the number of piles?"
   A: Impossible to finish since Koko needs at least 1 hour per pile. Problem constraints typically prevent this.
5. **Q:** "How do you handle the ceiling division without floating point?"
   A: Use `(numerator + denominator - 1) / denominator` for ceiling division with integers.

Approach 1: Linear Search Through All Speeds

Idea: Try every possible eating speed from 1 to max pile size until we find one that works.

Time Complexity: O(n × max(piles)) - check each speed, calculate time for each
Space Complexity: O(1) - no extra space needed

```
using System;
using System.Linq;

public class Solution
{
    public int MinEatingSpeed(int[] piles, int h)
    {
        int maxPile = piles.Max();

        // Try each possible eating speed
        for (int speed = 1; speed <= maxPile; speed++)
        {
            if (CanFinishInTime(piles, h, speed))
            {
                return speed;
            }
        }

        return maxPile; // Fallback
    }

    private bool CanFinishInTime(int[] piles, int h, int speed)
```

```
    {
        int totalHours = 0;

        foreach (int pile in piles)
        {
            // Calculate hours needed for this pile
            totalHours += (pile + speed - 1) / speed; // Ceiling division

            // Early termination if already exceeds limit
            if (totalHours > h) return false;
        }

        return totalHours <= h;
    }
}
```

Approach 2: Binary Search with Optimization

Explanation: Use binary search on the answer space, but with some optimizations like early termination.

Time Complexity: O(n × log(max(piles))) - binary search with time calculation
Space Complexity: O(1) - only using variables

```
using System;
using System.Linq;

public class Solution
{
    public int MinEatingSpeed(int[] piles, int h)
    {
        int left = 1;
        int right = piles.Max();
        int result = right;

        while (left <= right)
        {
            int mid = left + (right - left) / 2;

            if (CanFinishInTime(piles, h, mid))
            {
                result = mid; // This speed works, try slower
                right = mid - 1;
            }
            else
            {
                left = mid + 1; // This speed too slow, try faster
            }
        }

        return result;
    }

    private bool CanFinishInTime(int[] piles, int h, int speed)
```

```
    {
        long totalHours = 0; // Use long to prevent overflow

        foreach (int pile in piles)
        {
            totalHours += (pile + speed - 1) / speed;

            // Early termination
            if (totalHours > h) return false;
        }

        return true;
    }
}
```

Approach 3: Optimal - Binary Search on Answer

Detailed Reasoning: This is a classic "binary search on answer" problem. We can't directly calculate the minimum speed, but we can verify if a given speed works. Binary search finds the minimum valid speed efficiently.

Time Complexity: O(n × log(max(piles))) - log search space × linear verification
Space Complexity: O(1) - constant extra space

```
using System;
using System.Linq;

public class Solution
{
    /// <summary>
    /// Finds minimum eating speed using binary search on answer
    /// Time: O(n * log(max(piles))), Space: O(1)
    /// </summary>
    public int MinEatingSpeed(int[] piles, int h)
    {
        // Define search space: minimum speed = 1, maximum = largest pile
        int left = 1;
        int right = piles.Max();

        while (left < right)
        {
            int mid = left + (right - left) / 2;

            // Check if this eating speed allows finishing in time
            if (CanFinishInTime(piles, h, mid))
            {
                // This speed works, try to find a slower one
                right = mid;
            }
            else
            {
                // This speed is too slow, need faster speed
                left = mid + 1;
            }
```

```
        }

        return left; // left == right, minimum valid speed
    }

    /// <summary>
    /// Calculates if Koko can finish all bananas with given speed in h hours
    /// </summary>
    private bool CanFinishInTime(int[] piles, int h, int speed)
    {
        long hoursNeeded = 0;

        foreach (int pile in piles)
        {
            // Ceiling division: how many hours needed for this pile
            hoursNeeded += (long)(pile + speed - 1) / speed;

            // Early termination if already exceeded time limit
            if (hoursNeeded > h) return false;
        }

        return hoursNeeded <= h;
    }
}
```

Key Takeaways:

- **Binary search on answer**: When direct calculation is hard but verification is easy
- **Ceiling division trick**: `(a + b - 1) / b` for integer ceiling division
- **Search space bounds**: Minimum meaningful value to maximum possible value

PROBLEM 4: FIND MINIMUM IN ROTATED SORTED ARRAY

Problem Statement:

Suppose an array of length `n` sorted in ascending order is rotated between `1` and `n` times. For example, the array `nums =` might become:

- `` if it was rotated `4` times.
- `` if it was rotated `7` times.

Given the array `nums` after the possible rotation, return the minimum element of this array. You must write an algorithm that runs in **O(log n)** time.

Test Cases:

```
Example 1:
Input: nums = [3,4,5,1,2]
Output: 1
Explanation: The original array was [1,2,3,4,5] rotated 3 times.
```

```
Example 2:
Input: nums = [4,5,6,7,0,1,2]
Output: 0
Explanation: The original array was [0,1,2,4,5,6,7] rotated 4 times.

Example 3:
Input: nums = [11,13,15,17]
Output: 11
Explanation: The original array was [11,13,15,17] and was not rotated.
```

Interview Questions & Answers:

1. **Q:** "How do you determine which half contains the minimum?"
   A: Compare middle element with rightmost element. If mid > right, minimum is in right half. Otherwise, minimum is in left half (including mid).
2. **Q:** "What if the array is not rotated?"
   A: The algorithm still works correctly - it will find the first element as the minimum.
3. **Q:** "Why compare with right element instead of left element?"
   A: Comparing with right helps distinguish which part is sorted. Left comparison can be ambiguous in rotated arrays.
4. **Q:** "What about duplicate elements?"
   A: This problem assumes no duplicates. With duplicates, worst case becomes O(n) when we can't determine which half to eliminate.
5. **Q:** "How is this different from regular binary search?"
   A: We're not searching for a specific target, but for the inflection point where the rotation occurred.

Approach 1: Linear Search

Idea: Scan through array to find the point where next element is smaller than current element.

Time Complexity: O(n) - potentially scan entire array
Space Complexity: O(1) - no extra space needed

```
using System;

public class Solution
{
    public int FindMin(int[] nums)
    {
        int min = nums[0];

        // Find minimum by checking each element
        for (int i = 1; i < nums.Length; i++)
        {
            min = Math.Min(min, nums[i]);

            // If we find a drop, we've found the rotation point
            if (nums[i] < nums[i - 1])
```

```
            {
                return nums[i];
            }
        }

        return min; // Array wasn't rotated
    }
}
```

Approach 2: Find Rotation Point

Explanation: Look for the "break point" where a larger element is followed by a smaller element.

Time Complexity: O(n) in worst case - linear scan
Space Complexity: O(1) - only using variables

```
using System;

public class Solution
{
    public int FindMin(int[] nums)
    {
        // Handle single element
        if (nums.Length == 1) return nums[0];

        // Check if array is not rotated
        if (nums[0] < nums[nums.Length - 1]) return nums[0];

        // Find the rotation point
        for (int i = 0; i < nums.Length - 1; i++)
        {
            // Found the break point
            if (nums[i] > nums[i + 1])
            {
                return nums[i + 1];
            }
        }

        // This shouldn't happen with valid input
        return nums[0];
    }
}
```

Approach 3: Optimal - Binary Search

Detailed Reasoning: Use binary search to find the minimum element. The key insight is that in a rotated sorted array, at least one half is always sorted. By comparing the middle element with the rightmost element, we can determine which half contains the minimum.

Time Complexity: O(log n) - eliminate half of search space each iteration
Space Complexity: O(1) - only using pointer variables

```
using System;

public class Solution
{
    /// <summary>
    /// Finds minimum element in rotated sorted array using binary search
    /// Time: O(log n), Space: O(1)
    /// </summary>
    public int FindMin(int[] nums)
    {
        int left = 0;
        int right = nums.Length - 1;

        while (left < right)
        {
            int mid = left + (right - left) / 2;

            // Compare mid with right to determine which half is sorted
            if (nums[mid] > nums[right])
            {
                // Right half is not sorted, minimum must be in right half
                // Since nums[mid] > nums[right], mid cannot be minimum
                left = mid + 1;
            }
            else
            {
                // Left half is not sorted, minimum must be in left half
                // nums[mid] might be the minimum, so include it in search
                right = mid;
            }
        }

        // left == right, pointing to minimum element
        return nums[left];
    }
}
```

Key Takeaways:

- **Rotation detection**: Compare mid with rightmost element to determine sorted half
- **Include vs exclude**: When mid might be answer, use `right = mid`, not `right = mid - 1`
- **Loop termination**: Use `left < right` when searching for position/minimum

PROBLEM 5: SEARCH IN ROTATED SORTED ARRAY

Problem Statement:

There is an integer array `nums` sorted in ascending order (with distinct values). Prior to being passed to your function, `nums` is possibly rotated at an unknown pivot index `k` (1 <= k < nums.length) such that the resulting array is `[nums[k], nums[k+1], ..., nums[n-1], nums, nums, ..., nums[k-1]]` (0-indexed).

Given the array `nums` after the possible rotation and an integer `target`, return the index of `target` if it is in `nums`, or `-1` if it is not in `nums`. You must write an algorithm with **O(log n)** runtime complexity.

Test Cases:

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

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

Example 3:
Input: nums = [1], target = 0
Output: -1
Explanation: Single element array, target not found.
```

Interview Questions & Answers:

1. **Q:** "How do you determine which half of the rotated array to search?"
   A: First determine which half is sorted by comparing mid with left/right. Then check if target is in the sorted half's range.
2. **Q:** "What if both halves appear sorted?"
   A: If nums[left] <= nums[mid] <= nums[right], the array isn't rotated, use regular binary search.
3. **Q:** "How do you handle the case where target equals the middle element?"
   A: Return mid immediately - this is the same as regular binary search.
4. **Q:** "What's the difference between this and finding minimum in rotated array?"
   A: Here we're searching for a specific target, so we need to determine which half to search based on target's relationship to the sorted portion.
5. **Q:** "Can you solve this by first finding the rotation point?"
   A: Yes, but it's more complex and requires two binary searches. Single binary search is more elegant.

Approach 1: Linear Search

Idea: Simply scan through array linearly to find the target.

Time Complexity: O(n) - potentially scan entire array
Space Complexity: O(1) - no extra space needed

```
using System;
```

```
public class Solution
{
    public int Search(int[] nums, int target)
    {
        // Linear search through the array
        for (int i = 0; i < nums.Length; i++)
        {
            if (nums[i] == target)
            {
                return i;
            }
        }

        return -1; // Target not found
    }
}
```

Approach 2: Find Pivot Then Binary Search

Explanation: First find the rotation point (minimum element), then determine which part to search and apply regular binary search.

Time Complexity: O(log n) - two binary searches
Space Complexity: O(1) - only using variables

```
using System;

public class Solution
{
    public int Search(int[] nums, int target)
    {
        if (nums.Length == 0) return -1;

        // Find the pivot (minimum element index)
        int pivot = FindPivot(nums);

        // Determine which part to search
        if (target >= nums[0])
        {
            // Search left part (from 0 to pivot-1)
            return BinarySearch(nums, target, 0, pivot - 1);
        }
        else
        {
            // Search right part (from pivot to end)
            return BinarySearch(nums, target, pivot, nums.Length - 1);
        }
    }

    private int FindPivot(int[] nums)
    {
        int left = 0, right = nums.Length - 1;

        while (left < right)
```

```
        {
            int mid = left + (right - left) / 2;

            if (nums[mid] > nums[right])
            {
                left = mid + 1;
            }
            else
            {
                right = mid;
            }
        }

        return left;
    }

    private int BinarySearch(int[] nums, int target, int left, int right)
    {
        while (left <= right)
        {
            int mid = left + (right - left) / 2;

            if (nums[mid] == target) return mid;
            else if (nums[mid] < target) left = mid + 1;
            else right = mid - 1;
        }

        return -1;
    }
}
```

Approach 3: Optimal - One-Pass Binary Search

Detailed Reasoning: Use modified binary search that handles rotation in single pass. At each step, determine which half is sorted, then check if target is in that sorted half's range.

Time Complexity: O(log n) - single binary search pass
Space Complexity: O(1) - only using pointer variables

```
using System;

public class Solution
{
    /// <summary>
    /// Searches target in rotated sorted array using one-pass binary search
    /// Time: O(log n), Space: O(1)
    /// </summary>
    public int Search(int[] nums, int target)
    {
        if (nums == null || nums.Length == 0) return -1;

        int left = 0;
        int right = nums.Length - 1;
```

```
        while (left <= right)
        {
            int mid = left + (right - left) / 2;

            // Found target
            if (nums[mid] == target)
            {
                return mid;
            }

            // Determine which half is sorted
            if (nums[left] <= nums[mid])
            {
                // Left half is sorted
                if (nums[left] <= target && target < nums[mid])
                {
                    // Target is in sorted left half
                    right = mid - 1;
                }
                else
                {
                    // Target is in right half
                    left = mid + 1;
                }
            }
            else
            {
                // Right half is sorted
                if (nums[mid] < target && target <= nums[right])
                {
                    // Target is in sorted right half
                    left = mid + 1;
                }
                else
                {
                    // Target is in left half
                    right = mid - 1;
                }
            }
        }

        return -1; // Target not found
    }
}
```

Key Takeaways:

- **Sorted half identification**: Use `nums[left] <= nums[mid]` to determine which half is sorted
- **Range checking**: Verify target is within the range of the sorted half
- **Single pass efficiency**: Avoid multiple binary searches by handling rotation logic within one search

PROBLEM 6: TIME BASED KEY-VALUE STORE

Problem Statement:

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

Implement the `TimeMap` class:

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

Test Cases:

```
Example 1:
Input: ["TimeMap", "set", "get", "get", "set", "get", "get"]
       [[], ["foo", "bar", 1], ["foo", 1], ["foo", 3], ["foo", "bar2", 4], ["foo", 4], ["
Output: [null, null, "bar", "bar", null, "bar2", "bar2"]
Explanation:
TimeMap timeMap = new TimeMap();
timeMap.set("foo", "bar", 1);  // store key="foo", value="bar" at timestamp=1
timeMap.get("foo", 1);         // return "bar"
timeMap.get("foo", 3);         // return "bar", since no value at timestamp 3, return clo
timeMap.set("foo", "bar2", 4); // store key="foo", value="bar2" at timestamp=4
timeMap.get("foo", 4);         // return "bar2"
timeMap.get("foo", 5);         // return "bar2"

Example 2:
Input: ["TimeMap", "set", "set", "get", "get", "get"]
       [[], ["love", "high", 10], ["love", "low", 20], ["love", 15], ["love", 10], ["love
Output: [null, null, null, "high", "high", "low"]

Example 3:
Input: ["TimeMap", "get"]
       [[], ["missing_key", 1]]
Output: [null, ""]
Explanation: Key doesn't exist, return empty string.
```

Interview Questions & Answers:

1. **Q:** "How do you handle multiple values for the same key efficiently?"
   A: Store a list of (timestamp, value) pairs for each key, and since timestamps are increasing, use binary search to find the correct value.
2. **Q:** "What if set operations don't come in timestamp order?"
   A: Problem guarantees timestamps are increasing for set operations, but if not, we'd need

to maintain sorted order.

3. **Q:** "How do you find the largest timestamp <= target efficiently?"
   A: Use binary search to find the rightmost position where timestamp <= target.
4. **Q:** "What data structures are best for this problem?"
   A: HashMap for key-to-list mapping, ArrayList for timestamp-value pairs, binary search for efficient retrieval.
5. **Q:** "How would you optimize for memory if timestamps are very sparse?"
   A: Could use more complex data structures like segment trees or skip lists, but for most cases, sorted list with binary search is sufficient.

Approach 1: HashMap with Linear Search

Idea: Store list of (timestamp, value) pairs for each key, search linearly for best timestamp.

Time Complexity: O(1) for set, **O(n)** for get where n is number of values for key
Space Complexity: O(n) - store all timestamp-value pairs

```
using System;
using System.Collections.Generic;

public class TimeMap
{
    private Dictionary<string, List<(int timestamp, string value)>> store;

    public TimeMap()
    {
        store = new Dictionary<string, List<(int, string)>>();
    }

    public void Set(string key, string value, int timestamp)
    {
        if (!store.ContainsKey(key))
        {
            store[key] = new List<(int, string)>();
        }

        store[key].Add((timestamp, value));
    }

    public string Get(string key, int timestamp)
    {
        if (!store.ContainsKey(key)) return "";

        var values = store[key];
        string result = "";

        // Linear search for best timestamp
        foreach (var (ts, val) in values)
        {
            if (ts <= timestamp)
            {
                result = val; // Keep updating with valid timestamps
```

```
            }
            else
            {
                break; // Since timestamps are increasing
            }
        }

        return result;
    }
}
```

Approach 2: HashMap with Binary Search (Custom)

Explanation: Use binary search to find the position of largest timestamp <= target timestamp.

Time Complexity: **O(1)** for set, **O(log n)** for get
Space Complexity: **O(n)** - store all timestamp-value pairs

```
using System;
using System.Collections.Generic;

public class TimeMap
{
    private Dictionary<string, List<(int timestamp, string value)>> store;

    public TimeMap()
    {
        store = new Dictionary<string, List<(int, string)>>();
    }

    public void Set(string key, string value, int timestamp)
    {
        if (!store.ContainsKey(key))
        {
            store[key] = new List<(int, string)>();
        }

        store[key].Add((timestamp, value));
    }

    public string Get(string key, int timestamp)
    {
        if (!store.ContainsKey(key)) return "";

        var values = store[key];
        int index = BinarySearchFloor(values, timestamp);

        return index >= 0 ? values[index].value : "";
    }

    // Find largest index where timestamp <= target
    private int BinarySearchFloor(List<(int timestamp, string value)> values, int target)
    {
        int left = 0, right = values.Count - 1;
```

```
        int result = -1;

        while (left <= right)
        {
            int mid = left + (right - left) / 2;

            if (values[mid].timestamp <= target)
            {
                result = mid;
                left = mid + 1; // Look for larger valid timestamp
            }
            else
            {
                right = mid - 1;
            }
        }

        return result;
    }
}
```

Approach 3: Optimal - HashMap with Built-in Binary Search

Detailed Reasoning: Use .NET's built-in binary search capabilities for cleaner, more efficient code. Since timestamps are guaranteed to be increasing in set operations, we can use simple binary search.

Time Complexity: O(1) for set, **O(log n)** for get
Space Complexity: O(n) - store all timestamp-value pairs

```
using System;
using System.Collections.Generic;

/// <summary>
/// Time-based key-value store using HashMap and binary search
/// Set: O(1), Get: O(log n), Space: O(n)
/// </summary>
public class TimeMap
{
    private Dictionary<string, List<(int timestamp, string value)>> data;

    public TimeMap()
    {
        data = new Dictionary<string, List<(int, string)>>();
    }

    public void Set(string key, string value, int timestamp)
    {
        // Initialize list for new keys
        if (!data.ContainsKey(key))
        {
            data[key] = new List<(int, string)>();
        }
```

```
        // Add new timestamp-value pair
        // Problem guarantees timestamps are increasing
        data[key].Add((timestamp, value));
    }

    public string Get(string key, int timestamp)
    {
        // Return empty string for non-existent keys
        if (!data.ContainsKey(key) || data[key].Count == 0)
        {
            return "";
        }

        var entries = data[key];

        // Binary search for the rightmost entry with timestamp <= target
        int left = 0, right = entries.Count - 1;
        string result = "";

        while (left <= right)
        {
            int mid = left + (right - left) / 2;

            if (entries[mid].timestamp <= timestamp)
            {
                // This timestamp is valid, save the value
                result = entries[mid].value;
                left = mid + 1; // Look for a more recent valid timestamp
            }
            else
            {
                // This timestamp is too large
                right = mid - 1;
            }
        }

        return result;
    }
}
```

Key Takeaways:

- **Binary search on time**: Efficient retrieval with logarithmic complexity
- **Floor function**: Finding largest value ≤ target using binary search
- **Data structure design**: Combine HashMap and sorted lists for optimal performance

PROBLEM 7: MEDIAN OF TWO SORTED ARRAYS

Problem Statement:

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

Test Cases:

```
Example 1:
Input: nums1 = [1,3], nums2 = [2]
Output: 2.00000
Explanation: merged array = [1,2,3] and median is 2

Example 2:
Input: nums1 = [1,2], nums2 = [3,4]
Output: 2.50000
Explanation: merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5

Example 3:
Input: nums1 = [0,0], nums2 = [0,0]
Output: 0.00000
Explanation: merged array = [0,0,0,0] and median is (0 + 0) / 2 = 0
```

Interview Questions & Answers:

1. **Q:** "How do you find median without actually merging the arrays?"
   A: Use binary search to find the correct partition point where left half has equal elements to right half and max(left) <= min(right).
2. **Q:** "What if arrays have very different sizes?"
   A: Always binary search on the smaller array to ensure logarithmic complexity and avoid index out of bounds issues.
3. **Q:** "How do you handle even vs odd total length?"
   A: For odd length, median is the middle element. For even length, median is average of two middle elements.
4. **Q:** "What if one array is empty?"
   A: Return median of the non-empty array. Handle this edge case before main algorithm.
5. **Q:** "Why is this harder than regular binary search?"
   A: We're not searching for an element, but for a partition point that satisfies the median property across two arrays.

Approach 1: Merge and Find Median

Idea: Merge both arrays into single sorted array, then find median.

Time Complexity: O(m + n) - merge both arrays
Space Complexity: O(m + n) - store merged array

```
using System;
using System.Collections.Generic;
```

```
public class Solution
{
    public double FindMedianSortedArrays(int[] nums1, int[] nums2)
    {
        var merged = new List<int>();
        int i = 0, j = 0;

        // Merge both arrays
        while (i < nums1.Length && j < nums2.Length)
        {
            if (nums1[i] <= nums2[j])
            {
                merged.Add(nums1[i++]);
            }
            else
            {
                merged.Add(nums2[j++]);
            }
        }

        // Add remaining elements
        while (i < nums1.Length) merged.Add(nums1[i++]);
        while (j < nums2.Length) merged.Add(nums2[j++]);

        // Find median
        int n = merged.Count;
        if (n % 2 == 1)
        {
            return merged[n / 2];
        }
        else
        {
            return (merged[n / 2 - 1] + merged[n / 2]) / 2.0;
        }
    }
}
```

Approach 2: Two Pointers Without Extra Space

Explanation: Use two pointers to find median elements without storing the entire merged array.

Time Complexity: O(m + n) - traverse to median position
Space Complexity: O(1) - only using variables

```
using System;

public class Solution
{
    public double FindMedianSortedArrays(int[] nums1, int[] nums2)
    {
        int total = nums1.Length + nums2.Length;
        int half = total / 2;
```

```
        int i = 0, j = 0;
        int prev = 0, curr = 0;

        // Find the median element(s)
        for (int k = 0; k <= half; k++)
        {
            prev = curr;

            if (i >= nums1.Length)
            {
                curr = nums2[j++];
            }
            else if (j >= nums2.Length)
            {
                curr = nums1[i++];
            }
            else if (nums1[i] <= nums2[j])
            {
                curr = nums1[i++];
            }
            else
            {
                curr = nums2[j++];
            }
        }

        // Return median
        if (total % 2 == 1)
        {
            return curr;
        }
        else
        {
            return (prev + curr) / 2.0;
        }
    }
}
```

Approach 3: Optimal - Binary Search

Detailed Reasoning: Use binary search to find the correct partition point. The key insight is that we need to partition both arrays such that all elements in the left partitions are ≤ all elements in the right partitions, and the partitions have equal size (or differ by 1).

Time Complexity: O(log(min(m, n))) - binary search on smaller array
Space Complexity: O(1) - only using variables

```
using System;

public class Solution
{
    /// <summary>
    /// Finds median of two sorted arrays using binary search
    /// Time: O(log(min(m,n))), Space: O(1)
```

```
    /// </summary>
    public double FindMedianSortedArrays(int[] nums1, int[] nums2)
    {
        // Ensure nums1 is the smaller array for efficiency
        if (nums1.Length > nums2.Length)
        {
            return FindMedianSortedArrays(nums2, nums1);
        }

        int m = nums1.Length, n = nums2.Length;
        int left = 0, right = m;

        while (left <= right)
        {
            // Partition nums1 at partitionX
            int partitionX = (left + right) / 2;

            // Partition nums2 such that left partition has half the total elements
            int partitionY = (m + n + 1) / 2 - partitionX;

            // Handle edge cases for partition boundaries
            int maxLeftX = (partitionX == 0) ? int.MinValue : nums1[partitionX - 1];
            int minRightX = (partitionX == m) ? int.MaxValue : nums1[partitionX];

            int maxLeftY = (partitionY == 0) ? int.MinValue : nums2[partitionY - 1];
            int minRightY = (partitionY == n) ? int.MaxValue : nums2[partitionY];

            // Check if we found the correct partition
            if (maxLeftX <= minRightY && maxLeftY <= minRightX)
            {
                // Found correct partition
                if ((m + n) % 2 == 0)
                {
                    // Even total length: average of two middle elements
                    return (Math.Max(maxLeftX, maxLeftY) + Math.Min(minRightX, minRightY)
                }
                else
                {
                    // Odd total length: maximum of left partition
                    return Math.Max(maxLeftX, maxLeftY);
                }
            }
            else if (maxLeftX > minRightY)
            {
                // partitionX is too large, search left half
                right = partitionX - 1;
            }
            else
            {
                // partitionX is too small, search right half
                left = partitionX + 1;
            }
        }

        // Should never reach here with valid input
        throw new ArgumentException("Input arrays are not sorted");
```

```
    }
}
```

Key Takeaways:

- **Binary search on partition**: Search for correct split point, not element
- **Partition property**: Left elements ≤ right elements across both arrays
- **Edge case handling**: Use int.MinValue/MaxValue for boundary conditions

DAY 4 SUMMARY

Summary of Concepts Covered

Binary Search Fundamentals:

- **Standard binary search**: O(log n) search in sorted arrays
- **Overflow prevention**: Use `left + (right - left) / 2` for mid calculation
- **Loop variants**: `left <= right` for exact search, `left < right` for position finding
- **Boundary handling**: Careful with inclusive/exclusive bounds

Binary Search Variations:

- **Search in 2D matrix**: Treat as flattened 1D array or search row then column
- **Binary search on answer**: When verification is easier than direct calculation
- **Rotated array search**: Handle rotation by determining which half is sorted
- **Finding minimum/maximum**: Modified binary search for optimization problems

Advanced Applications:

- **Time-based data structures**: Combine HashMap with binary search for efficiency
- **Partition-based algorithms**: Binary search to find optimal partition points
- **Multi-array problems**: Extend binary search concepts to multiple sorted arrays

Time & Space Complexity Summary

Problem Type	Brute Force	Optimized	Key Technique
Binary Search	**O(n)** linear	**O(log n)**	Standard binary search
2D Matrix Search	**O(m×n)** scan	**O(log(m×n))**	Flatten or binary search twice
Eating Bananas	**O(n×max)** linear	**O(n×log(max))**	Binary search on answer
Rotated Min	**O(n)** linear	**O(log n)**	Modified binary search
Rotated Search	**O(n)** linear	**O(log n)**	One-pass modified binary search
Time-based Store	**O(n)** per get	**O(log n)** per get	HashMap + binary search

Problem Type	Brute Force	Optimized	Key Technique
Median Arrays	**O(m+n)** merge	**O(log(min(m,n)))**	Binary search on partition

Pattern Recognition Guide

Use Standard Binary Search When:

- Array is sorted and searching for specific element
- Need O(log n) improvement over linear search
- Working with indexed data structures

Use Binary Search on Answer When:

- Direct calculation is complex but verification is simple
- Searching for optimal value in continuous range
- Problem asks for "minimum/maximum such that condition holds"

Use Modified Binary Search When:

- Data has special properties (rotated, partially sorted)
- Need to find positions rather than exact values
- Working with multiple arrays or dimensions

Common Binary Search Patterns

Pattern 1: Exact Match

```
while (left <= right) {
    int mid = left + (right - left) / 2;
    if (arr[mid] == target) return mid;
    else if (arr[mid] < target) left = mid + 1;
    else right = mid - 1;
}
return -1;
```

Pattern 2: Find Boundary

```
while (left < right) {
    int mid = left + (right - left) / 2;
    if (condition(mid)) right = mid;
    else left = mid + 1;
}
return left;
```

Pattern 3: Search on Answer

```
while (left < right) {
    int mid = left + (right - left) / 2;
    if (canAchieve(mid)) right = mid;
    else left = mid + 1;
}
return left;
```

Review Tips

- **Master overflow prevention**: Always use `left + (right - left) / 2`
- **Understand loop termination**: Know when to use `<=` vs `<` in conditions
- **Practice coordinate conversion**: 1D ↔ 2D transformations for matrix problems
- **Visualize partitions**: Draw diagrams for complex binary search problems

How to Practice Mock Interviews

1. **Problem Analysis** (5 min): Identify if problem needs binary search and which variant
2. **Approach Discussion** (5 min): Explain search space, termination condition, and complexity
3. **Implementation** (20 min): Code while explaining binary search logic and edge cases
4. **Optimization** (5 min): Discuss alternative approaches and trade-offs
5. **Testing** (10 min): Walk through examples, especially edge cases and boundary conditions

Common Mistakes to Avoid

- **Integer overflow**: Always use safe mid calculation
- **Infinite loops**: Ensure search space reduces each iteration
- **Boundary errors**: Off-by-one errors in array indexing
- **Wrong termination**: Using incorrect loop condition for problem type
- **Edge case handling**: Empty arrays, single elements, all same elements
- **Partition logic**: Incorrect partition calculations in advanced problems

Advanced Topics for Further Study

- **Binary Search Trees**: In-order traversal, balancing, operations
- **Ternary Search**: For unimodal functions
- **Exponential Search**: For unbounded arrays
- **Interpolation Search**: For uniformly distributed data
- **Fractional Cascading**: For multiple sorted arrays
- **Parallel Binary Search**: For multiple queries

Real-world Applications

- **Database Systems**: B-tree indexing, query optimization
- **Operating Systems**: Memory allocation, process scheduling
- **Graphics**: Ray tracing, collision detection
- **Machine Learning**: Hyperparameter tuning, gradient descent
- **Networking**: Load balancing, resource allocation
- **Finance**: Option pricing, risk analysis

This completes Day 4: Binary Search & Trees (Part 1) with comprehensive coverage of all 7 problems, following the same detailed structure and educational approach as previous days.