CODING INTERVIEW PREPARATION

Day 7: Dynamic Programming (Part 1) & Review

Complete Study Guide with Multiple Solutions

TABLE OF CONTENTS

Page

- Day 7 Overview 3
- Key Concepts 4
- **Problems:**
  - Problem 1: Climbing Stairs 5-7
  - Problem 2: Coin Change 8-10
  - Problem 3: Longest Increasing Subsequence 11-13
  - Problem 4: House Robber 14-16
  - Problem 5: House Robber II 17-19
  - Problem 6: Unique Paths 20-22
  - Problem 7: Jump Game 23-25
- Day 7 Summary 26

DAY 7: DYNAMIC PROGRAMMING (PART 1) & REVIEW

Key Concepts for Today:

- **Fibonacci Pattern** and sequence-based DP
- **Knapsack Basics** and optimization problems
- **Greedy vs DP** decision-making strategies
- **Memoization vs Tabulation** implementation approaches
- **State Transition** and recurrence relations
- **Space Optimization** techniques for DP solutions

Problems Index:

1. **Climbing Stairs** (Easy)
2. **Coin Change** (Medium)
3. **Longest Increasing Subsequence** (Medium)
4. **House Robber** (Medium)
5. **House Robber II** (Medium)
6. **Unique Paths** (Medium)
7. **Jump Game** (Medium)

CORE CONCEPTS EXPLANATION

Dynamic Programming Fundamentals

Definition: Dynamic Programming is an algorithmic paradigm that solves complex problems by breaking them down into simpler subproblems and storing the results to avoid redundant calculations.

Key Properties:

1. **Overlapping Subproblems**: The same subproblems are solved multiple times
2. **Optimal Substructure**: The optimal solution can be constructed from optimal solutions of subproblems

DP Approaches:

1. Top-Down (Memoization):

```
int DP(int n, int[] memo)
{
    if (base_case) return base_value;
    if (memo[n] != -1) return memo[n];

    memo[n] = recurrence_relation(DP(n-1, memo), DP(n-2, memo));
    return memo[n];
}
```

2. Bottom-Up (Tabulation):

```
int DP(int n)
{
    int[] dp = new int[n+1];
    dp[^0] = base_value_0;
    dp[^1] = base_value_1;

    for (int i = 2; i <= n; i++)
    {
        dp[i] = recurrence_relation(dp[i-1], dp[i-2]);
    }
    return dp[n];
}
```

Common DP Patterns

1. Fibonacci Sequence Pattern:

- Problems where current state depends on previous states
- Examples: Climbing stairs, house robber, decode ways

2. Knapsack Pattern:

- Optimization problems with constraints
- Examples: Coin change, partition problems, subset sum

3. Grid Path Pattern:

- Counting or optimizing paths in 2D grids
- Examples: Unique paths, minimum path sum

4. Subsequence Pattern:

- Finding optimal subsequences in arrays/strings
- Examples: LIS, LCS, edit distance

Greedy vs Dynamic Programming

Greedy Algorithm:

- Makes locally optimal choices at each step
- Works when local optimum leads to global optimum
- Examples: Activity selection, fractional knapsack

Dynamic Programming:

- Considers all possible choices and their consequences
- Required when greedy approach fails to find global optimum
- Examples: 0/1 knapsack, coin change with arbitrary denominations

When to Use DP:

- Problem has overlapping subproblems
- Optimal substructure exists
- Brute force solution has exponential time complexity
- Need to find optimal value rather than just feasibility

PROBLEM 1: CLIMBING STAIRS

Problem Statement:

You are climbing a staircase. It takes `n` steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Test Cases:

```
Example 1:
Input: n = 2
Output: 2
Explanation: There are two ways to climb: (1 step + 1 step) or (2 steps)

Example 2:
Input: n = 3
Output: 3
Explanation: There are three ways: (1+1+1), (1+2), or (2+1)

Example 3:
Input: n = 5
Output: 8
Explanation: This follows Fibonacci sequence: F(6) = 8
```

Interview Questions & Answers:

1. **Q:** "Is this problem related to Fibonacci numbers?"
   A: Yes! The number of ways to climb n stairs equals F(n+1), where F is the Fibonacci sequence.
2. **Q:** "What if you could climb 1, 2, or 3 steps at a time?"
   A: The recurrence would become f(n) = f(n-1) + f(n-2) + f(n-3), similar to tribonacci sequence.
3. **Q:** "Can you solve this with constant space?"
   A: Yes, since we only need the previous two values, we can use two variables instead of an array.
4. **Q:** "What's the base case reasoning?"
   A: f(1) = 1 (one way), f(2) = 2 (two ways: 1+1 or 2), then f(n) = f(n-1) + f(n-2).
5. **Q:** "How would you handle very large n values?"
   A: Use modular arithmetic to prevent integer overflow, or consider matrix exponentiation for O(log n) solution.

Approach 1: Recursive (Brute Force)

Idea: Count all possible ways by trying both 1-step and 2-step choices recursively.

Time Complexity: O(2^n) - exponential due to repeated subproblems
Space Complexity: O(n) - recursion call stack

```
using System;

public class Solution
{
    public int ClimbStairs(int n)
    {
        // Base cases
        if (n <= 0) return 0;
        if (n == 1) return 1;
        if (n == 2) return 2;

        // Recursive case: can reach step n from step n-1 or n-2
        return ClimbStairs(n - 1) + ClimbStairs(n - 2);
    }
}
```

Approach 2: Top-Down DP (Memoization)

Explanation: Use recursion but cache results to avoid recomputing the same subproblems.

Time Complexity: O(n) - each subproblem computed once
Space Complexity: O(n) - memoization array plus recursion stack

```
using System;

public class Solution
{
    private int[] memo;

    public int ClimbStairs(int n)
    {
        memo = new int[n + 1];
        Array.Fill(memo, -1);
        return ClimbHelper(n);
    }

    private int ClimbHelper(int n)
    {
        // Base cases
        if (n <= 0) return 0;
        if (n == 1) return 1;
        if (n == 2) return 2;

        // Check memo
        if (memo[n] != -1) return memo[n];

        // Compute and store result
        memo[n] = ClimbHelper(n - 1) + ClimbHelper(n - 2);
        return memo[n];
    }
}
```

Approach 3: Optimal - Bottom-Up DP with Space Optimization

Detailed Reasoning: Since we only need the previous two values to compute the current value, we can optimize space from O(n) to O(1). This is the most efficient approach for this problem.

Time Complexity: O(n) - single pass through values
Space Complexity: O(1) - only using two variables

```
using System;

public class Solution
{
    /// <summary>
    /// Counts ways to climb stairs using space-optimized DP
    /// Time: O(n), Space: O(1)
    /// </summary>
    public int ClimbStairs(int n)
    {
        // Handle edge cases
        if (n <= 0) return 0;
        if (n == 1) return 1;
        if (n == 2) return 2;

        // Space-optimized DP: only keep track of previous two values
        int prev2 = 1; // f(1)
        int prev1 = 2; // f(2)
        int current = 0;

        // Build solution bottom-up
        for (int i = 3; i <= n; i++)
        {
            current = prev1 + prev2;
            prev2 = prev1;
            prev1 = current;
        }

        return current;
    }
}
```

Key Takeaways:

- **Fibonacci pattern**: Classic DP sequence where f(n) = f(n-1) + f(n-2)
- **Space optimization**: Reduce O(n) space to O(1) when only recent values are needed
- **Bottom-up efficiency**: Iterative approach avoids recursion overhead

PROBLEM 2: COIN CHANGE

Problem Statement:

You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money. Return the fewest number of coins that you need to make up that amount. If that amount cannot be made up by any combination of the coins, return `-1`.

You may assume that you have an infinite number of each kind of coin.

Test Cases:

```
Example 1:
Input: coins = [1,2,5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1 (3 coins)

Example 2:
Input: coins = [^2], amount = 3
Output: -1
Explanation: Cannot make amount 3 with only coins of denomination 2

Example 3:
Input: coins = [^1], amount = 0
Output: 0
Explanation: No coins needed for amount 0
```

Interview Questions & Answers:

1. **Q:** "Can we use a greedy approach for this problem?"
   A: No, greedy doesn't work for arbitrary coin systems. For example, coins and amount 6: greedy gives 4+1+1 (3 coins) but optimal is 3+3 (2 coins).
2. **Q:** "Is this an unbounded knapsack problem?"
   A: Yes, it's a variant where we want to minimize the number of items (coins) used to achieve exactly the target weight (amount).
3. **Q:** "How do you handle the case when amount cannot be formed?"
   A: Initialize DP array with amount+1 (impossible value), then check if final answer is still impossible.
4. **Q:** "What if coins array is empty?"
   A: Return -1 for any positive amount, 0 for amount 0.
5. **Q:** "Can you optimize space complexity?"
   A: No significant optimization possible since we need to build up from 0 to amount, requiring O(amount) space.

Approach 1: Recursive with Backtracking

Idea: Try all possible combinations of coins recursively and find the minimum count.

Time Complexity: O(a^c) - where a is amount, c is number of coin types (exponential)
Space Complexity: O(a) - recursion depth

```
using System;

public class Solution
{
    private int minCoins = int.MaxValue;

    public int CoinChange(int[] coins, int amount)
    {
        if (amount == 0) return 0;

        minCoins = int.MaxValue;
        Backtrack(coins, amount, 0);

        return minCoins == int.MaxValue ? -1 : minCoins;
    }

    private void Backtrack(int[] coins, int remaining, int count)
    {
        // Base cases
        if (remaining == 0)
        {
            minCoins = Math.Min(minCoins, count);
            return;
        }
        if (remaining < 0) return;

        // Try each coin
        foreach (int coin in coins)
        {
            Backtrack(coins, remaining - coin, count + 1);
        }
    }
}
```

Approach 2: Top-Down DP (Memoization)

Explanation: Use memoization to avoid recomputing the minimum coins needed for the same remaining amount.

Time Complexity: O(amount × coins.Length) - each subproblem solved once
Space Complexity: O(amount) - memoization table plus recursion stack

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

public class Solution
```

```
{
    private Dictionary<int, int> memo = new Dictionary<int, int>();

    public int CoinChange(int[] coins, int amount)
    {
        memo.Clear();
        int result = CoinChangeHelper(coins, amount);
        return result == int.MaxValue ? -1 : result;
    }

    private int CoinChangeHelper(int[] coins, int amount)
    {
        // Base cases
        if (amount == 0) return 0;
        if (amount < 0) return int.MaxValue;

        // Check memo
        if (memo.ContainsKey(amount)) return memo[amount];

        int minCoins = int.MaxValue;

        // Try each coin
        foreach (int coin in coins)
        {
            int subResult = CoinChangeHelper(coins, amount - coin);
            if (subResult != int.MaxValue)
            {
                minCoins = Math.Min(minCoins, subResult + 1);
            }
        }

        memo[amount] = minCoins;
        return minCoins;
    }
}
```

Approach 3: Optimal - Bottom-Up DP

Detailed Reasoning: Build solution iteratively from amount 0 to target amount. For each amount, try all coins and take the minimum. This approach is optimal because it systematically builds the solution and has clear space and time bounds.

Time Complexity: O(amount × coins.Length) - for each amount, try each coin
Space Complexity: O(amount) - DP array

```
using System;

public class Solution
{
    /// <summary>
    /// Finds minimum coins needed using bottom-up DP
    /// Time: O(amount * coins.Length), Space: O(amount)
    /// </summary>
    public int CoinChange(int[] coins, int amount)
    {
        // DP array where dp[i] = minimum coins needed for amount i
        int[] dp = new int[amount + 1];

        // Initialize with impossible value (amount + 1)
        Array.Fill(dp, amount + 1);

        // Base case: 0 coins needed for amount 0
        dp[^0] = 0;

        // Build solution for each amount from 1 to target
        for (int currentAmount = 1; currentAmount <= amount; currentAmount++)
        {
            // Try each coin denomination
            foreach (int coin in coins)
            {
                // If this coin can be used (doesn't exceed current amount)
                if (coin <= currentAmount)
                {
                    // Update minimum coins if using this coin gives better result
                    dp[currentAmount] = Math.Min(dp[currentAmount],
                                                 dp[currentAmount - coin] + 1);
                }
            }
        }

        // Return result: -1 if impossible, otherwise minimum coins needed
        return dp[amount] > amount ? -1 : dp[amount];
    }
}
```

Key Takeaways:

- **Unbounded knapsack pattern**: Can use each coin unlimited times
- **Greedy fails**: Need DP for arbitrary coin denominations
- **Bottom-up clarity**: Build solution systematically from smaller to larger amounts

PROBLEM 3: LONGEST INCREASING SUBSEQUENCE

Problem Statement:

Given an integer array `nums`, return the length of the longest strictly increasing subsequence.

Test Cases:

```
Example 1:
Input: nums = [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is [2,3,7,18], length = 4

Example 2:
Input: nums = [0,1,0,3,2,3]
```

```
Output: 4
Explanation: The longest increasing subsequence is [0,1,2,3], length = 4

Example 3:
Input: nums = [7,7,7,7,7,7,7]
Output: 1
Explanation: All elements are equal, so longest increasing subsequence has length 1
```

Interview Questions & Answers:

1. **Q:** "What's the difference between subsequence and subarray?"
   A: Subsequence doesn't need to be contiguous, subarray must be. We can skip elements in subsequence.
2. **Q:** "Can the subsequence have equal elements?"
   A: No, the problem asks for strictly increasing, so all elements must be different and in increasing order.
3. **Q:** "Is there an O(n log n) solution?"
   A: Yes, using binary search with patience sorting algorithm or maintaining sorted array of tail elements.
4. **Q:** "How would you return the actual subsequence, not just length?"
   A: Maintain a parent array to reconstruct the path, or use backtracking from the optimal solution.
5. **Q:** "What if array is empty or has one element?"
   A: Empty array returns 0, single element returns 1.

Approach 1: Brute Force - Generate All Subsequences

Idea: Generate all possible subsequences and check which ones are increasing, then return the length of the longest.

Time Complexity: O(2^n × n) - 2^n subsequences, O(n) to check each
Space Complexity: O(n) - recursion stack

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

public class Solution
{
    private int maxLength = 0;

    public int LengthOfLIS(int[] nums)
    {
        if (nums.Length == 0) return 0;

        maxLength = 1;
        var currentSubseq = new List<int>();

        GenerateSubsequences(nums, 0, currentSubseq);
        return maxLength;
    }

    private void GenerateSubsequences(int[] nums, int index, List<int> current)
    {
        // Check if current subsequence is increasing and update max
        if (IsIncreasing(current))
        {
            maxLength = Math.Max(maxLength, current.Count);
        }

        // Generate all subsequences starting from current index
        for (int i = index; i < nums.Length; i++)
        {
            current.Add(nums[i]);
            GenerateSubsequences(nums, i + 1, current);
            current.RemoveAt(current.Count - 1); // Backtrack
        }
    }

    private bool IsIncreasing(List<int> subseq)
    {
        for (int i = 1; i < subseq.Count; i++)
        {
            if (subseq[i] <= subseq[i - 1]) return false;
        }
        return true;
    }
}
```

Approach 2: Dynamic Programming O(n^2)

Explanation: For each element, find the longest increasing subsequence ending at that element by checking all previous elements.

Time Complexity: O(n^2) - nested loops through array
Space Complexity: O(n) - DP array

```
using System;

public class Solution
{
    public int LengthOfLIS(int[] nums)
    {
        if (nums.Length == 0) return 0;

        int n = nums.Length;
        int[] dp = new int[n];
        Array.Fill(dp, 1); // Each element forms subsequence of length 1

        int maxLength = 1;

        // For each element, check all previous elements
        for (int i = 1; i < n; i++)
        {
            for (int j = 0; j < i; j++)
            {
                // If current element is greater than previous element
                if (nums[i] > nums[j])
                {
                    dp[i] = Math.Max(dp[i], dp[j] + 1);
                }
            }
            maxLength = Math.Max(maxLength, dp[i]);
        }

        return maxLength;
    }
}
```

Approach 3: Optimal - Binary Search O(n log n)

Detailed Reasoning: Maintain an array where `tails[i]` is the smallest tail element of all increasing subsequences of length `i+1`. Use binary search to find position for each new element. This is optimal for large arrays.

Time Complexity: O(n log n) - n elements, log n binary search for each
Space Complexity: O(n) - tails array

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

public class Solution
{
    /// <summary>
    /// Finds length of LIS using binary search approach
    /// Time: O(n log n), Space: O(n)
    /// </summary>
    public int LengthOfLIS(int[] nums)
    {
        if (nums.Length == 0) return 0;

        // tails[i] = smallest tail of all increasing subsequences of length i+1
        var tails = new List<int>();

        foreach (int num in nums)
        {
            // Binary search for the position to insert/replace
            int left = 0, right = tails.Count;

            while (left < right)
            {
                int mid = left + (right - left) / 2;
                if (tails[mid] < num)
                {
                    left = mid + 1;
                }
                else
                {
                    right = mid;
                }
            }

            // If left == tails.Count, we're extending the sequence
            if (left == tails.Count)
            {
                tails.Add(num);
            }
            else
            {
                // Replace the element at position left
                tails[left] = num;
            }
        }

        return tails.Count;
    }
}
```

Key Takeaways:

- **Subsequence vs subarray**: Subsequence allows skipping elements
- **DP state definition**: dp[i] = length of LIS ending at index i
- **Binary search optimization**: Reduces O(n^2) to O(n log n) for large inputs

PROBLEM 4: HOUSE ROBBER

Problem Statement:

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array `nums` representing the amount of money of each house, return the maximum amount of money you can rob tonight **without alerting the police**.

Test Cases:

```
Example 1:
Input: nums = [1,2,3,1]
Output: 4
Explanation: Rob house 0 (money = 1) and house 2 (money = 3). Total = 1 + 3 = 4.

Example 2:
Input: nums = [2,7,9,3,1]
Output: 12
Explanation: Rob house 0 (money = 2), house 2 (money = 9) and house 4 (money = 1). Total

Example 3:
```

```
Input: nums = [5,1,2,9]
Output: 11
Explanation: Rob house 0 (money = 5) and house 3 (money = 9). Total = 5 + 9 = 14. Wait, t

Actually, let me be more careful: [5,1,2,9] indices 0,1,2,3. Adjacent pairs are (0,1), (1
- Rob 0,2: 5+2=7
- Rob 0,3: 5+9=14
- Rob 1,3: 1+9=10
- Rob just 3: 9
- Rob just 0: 5
So maximum should be 14, but expected is 11. Let me double-check the problem...

Oh wait, I think I made an error. Let me re-examine: if we have houses [5,1,2,9] at posit
- We can rob 0 and 2 (not adjacent): 5+2=7
- We can rob 0 and 3 (not adjacent): 5+9=14
- We can rob 1 and 3 (not adjacent): 1+9=10
- We can rob just 0: 5
- We can rob just 1: 1
- We can rob just 2: 2
- We can rob just 3: 9

The maximum is indeed 14, not 11. I think there might be an error in this example. Let me
```

Let me fix the example:

```
Example 3:
Input: nums = [2,1,1,9]
Output: 11
Explanation: Rob house 0 (money = 2) and house 3 (money = 9). Total = 2 + 9 = 11.
```

Interview Questions & Answers:

1. **Q:** "What if there are only 1 or 2 houses?"
   A: With 1 house, rob it. With 2 houses, rob the one with more money since they're adjacent.
2. **Q:** "Is this a 0/1 knapsack problem?"
   A: Not exactly. It's more like a linear DP problem where each house presents a choice: rob it (and skip previous) or don't rob it.
3. **Q:** "Can you solve this with constant space?"
   A: Yes, since we only need the previous two states (rob/don't rob previous house), we can use two variables.
4. **Q:** "What if houses were arranged in a circle?"
   A: That's House Robber II - you can't rob both first and last house since they'd be adjacent.
5. **Q:** "How would you track which houses were actually robbed?"
   A: Maintain a separate array to track decisions, or use backtracking from the optimal solution.

Approach 1: Recursive with All Choices

Idea: For each house, recursively try both choices (rob it or skip it) and return the maximum.

Time Complexity: O(2^n) - two choices for each house
Space Complexity: O(n) - recursion depth

```
using System;

public class Solution
{
    public int Rob(int[] nums)
    {
        if (nums.Length == 0) return 0;
        return RobFrom(nums, 0);
    }

    private int RobFrom(int[] nums, int index)
    {
        // Base case: no more houses
        if (index >= nums.Length) return 0;

        // Choice 1: Rob current house (skip next house)
        int robCurrent = nums[index] + RobFrom(nums, index + 2);

        // Choice 2: Skip current house (can consider next house)
        int skipCurrent = RobFrom(nums, index + 1);

        // Return maximum of both choices
        return Math.Max(robCurrent, skipCurrent);
    }
}
```

Approach 2: Top-Down DP (Memoization)

Explanation: Cache the results of subproblems to avoid recomputation in the recursive approach.

Time Complexity: O(n) - each house computed once
Space Complexity: O(n) - memoization array plus recursion stack

```
using System;

public class Solution
{
    private int[] memo;

    public int Rob(int[] nums)
    {
        if (nums.Length == 0) return 0;

        memo = new int[nums.Length];
        Array.Fill(memo, -1);
        return RobFrom(nums, 0);
    }

    private int RobFrom(int[] nums, int index)
    {
        // Base case
        if (index >= nums.Length) return 0;

        // Check memo
        if (memo[index] != -1) return memo[index];

        // Compute result
        int robCurrent = nums[index] + RobFrom(nums, index + 2);
        int skipCurrent = RobFrom(nums, index + 1);

        memo[index] = Math.Max(robCurrent, skipCurrent);
        return memo[index];
    }
}
```

Approach 3: Optimal - Bottom-Up DP with Space Optimization

Detailed Reasoning: Use bottom-up DP but optimize space since we only need the results from the previous two positions. This gives us the classic DP recurrence with O(1) space.

Time Complexity: O(n) - single pass through houses
Space Complexity: O(1) - only using two variables

```
using System;

public class Solution
{
    /// <summary>
    /// Finds maximum money that can be robbed using space-optimized DP
    /// Time: O(n), Space: O(1)
    /// </summary>
    public int Rob(int[] nums)
    {
        if (nums.Length == 0) return 0;
        if (nums.Length == 1) return nums[^0];

        // DP state:
        // prev2 = max money up to house i-2
        // prev1 = max money up to house i-1
        int prev2 = nums[^0];           // Rob first house
        int prev1 = Math.Max(nums[^0], nums[^1]); // Max of first two houses

        // For each house starting from index 2
        for (int i = 2; i < nums.Length; i++)
        {
            // Current max = max(rob current house + prev2, don't rob current house)
            int current = Math.Max(nums[i] + prev2, prev1);
            // Update for next iteration
            prev2 = prev1;
            prev1 = current;
        }

        return prev1;
    }
}
```

Key Takeaways:

- **Linear DP pattern**: Each position depends on non-adjacent previous positions
- **State transition**: rob[i] = max(rob[i-1], rob[i-2] + nums[i])
- **Space optimization**: Only need previous two states, not entire array

PROBLEM 5: HOUSE ROBBER II

Problem Statement:

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are **arranged in a circle**. That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have security systems connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array `nums` representing the amount of money of each house, return the maximum amount of money you can rob tonight **without alerting the police**.

Test Cases:

```
Example 1:
Input: nums = [2,3,2]
Output: 3
Explanation: You cannot rob house 0 and 2 (they are adjacent due to circular arrangement)

Example 2:
Input: nums = [1,2,3,1]
Output: 4
Explanation: Rob house 0 and house 2 (money = 1 + 3 = 4), or rob house 1 and house 3 (mor

Example 3:
Input: nums = [1,2,3]
Output: 3
Explanation: Rob house 1 (money = 2) or house 2 (money = 3). Choose house 2.
```

Interview Questions & Answers:

1. **Q:** "How does the circular arrangement change the problem?"
   A: The first and last houses are now adjacent, so we can't rob both. We need to consider two cases: rob first house (can't rob last) or rob last house (can't rob first).
2. **Q:** "Can you reuse the House Robber I solution?"
   A: Yes! Run House Robber I on two subarrays: [0...n-2] (exclude last) and [1...n-1] (exclude first), then take the maximum.
3. **Q:** "What if there's only one house?"
   A: Rob it, since there's no adjacency constraint with only one house.
4. **Q:** "What if there are only two houses?"
   A: Rob the one with more money, since they're adjacent in a circle.
5. **Q:** "Could you solve this in a single pass?"
   A: No, because of the circular constraint, we need to consider the two scenarios separately.

Approach 1: Brute Force - Try All Valid Combinations

Idea: Generate all possible combinations that don't include adjacent houses in the circular arrangement.

Time Complexity: O(2^n) - exponential combinations
Space Complexity: O(n) - recursion depth

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

public class Solution
{
    private int maxMoney = 0;

    public int Rob(int[] nums)
    {
        if (nums.Length == 0) return 0;
        if (nums.Length == 1) return nums[^0];
        if (nums.Length == 2) return Math.Max(nums[^0], nums[^1]);

        maxMoney = 0;
        var robbed = new bool[nums.Length];

        Backtrack(nums, 0, 0, robbed);
        return maxMoney;
    }

    private void Backtrack(int[] nums, int index, int currentMoney, bool[] robbed)
    {
        if (index >= nums.Length)
        {
            // Check if first and last are both robbed (invalid in circular)
            if (robbed[^0] && robbed[nums.Length - 1]) return;
            maxMoney = Math.Max(maxMoney, currentMoney);
            return;
        }

        // Choice 1: Don't rob current house
        Backtrack(nums, index + 1, currentMoney, robbed);

        // Choice 2: Rob current house (if not adjacent to previously robbed)
        bool canRob = true;
        if (index > 0 && robbed[index - 1]) canRob = false;

        if (canRob)
        {
            robbed[index] = true;
            Backtrack(nums, index + 1, currentMoney + nums[index], robbed);
            robbed[index] = false;
        }
    }
}
```

Approach 2: Two Linear Problems

Explanation: Split into two cases: include first house (exclude last) or include last house (exclude first). Solve each as linear House Robber problem.

Time Complexity: O(n) - two passes through array
Space Complexity: O(n) - DP arrays for both cases

```
using System;

public class Solution
{
    public int Rob(int[] nums)
    {
        if (nums.Length == 0) return 0;
        if (nums.Length == 1) return nums[^0];
        if (nums.Length == 2) return Math.Max(nums[^0], nums[^1]);

        // Case 1: Rob houses 0 to n-2 (exclude last house)
        int case1 = RobLinear(nums, 0, nums.Length - 2);

        // Case 2: Rob houses 1 to n-1 (exclude first house)
        int case2 = RobLinear(nums, 1, nums.Length - 1);

        return Math.Max(case1, case2);
    }

    private int RobLinear(int[] nums, int start, int end)
    {
        int[] dp = new int[end - start + 1];
        dp[^0] = nums[start];
        if (dp.Length == 1) return dp[^0];

        dp[^1] = Math.Max(nums[start], nums[start + 1]);
        for (int i = 2; i < dp.Length; i++)
        {
            int currentIndex = start + i;
            dp[i] = Math.Max(dp[i - 1], dp[i - 2] + nums[currentIndex]);
        }

        return dp[dp.Length - 1];
    }
}
```

Approach 3: Optimal - Space-Optimized Two Cases

Detailed Reasoning: Use the space-optimized House Robber solution for both cases (excluding first or last house). This gives us O(n) time with O(1) space complexity.

Time Complexity: O(n) - two linear passes
Space Complexity: O(1) - only using variables

```
using System;

public class Solution
{
    /// <summary>
    /// Solves circular house robber using two linear cases
    /// Time: O(n), Space: O(1)
    /// </summary>
    public int Rob(int[] nums)
    {
        if (nums.Length == 0) return 0;
        if (nums.Length == 1) return nums[^0];
        if (nums.Length == 2) return Math.Max(nums[^0], nums[^1]);

        // Since houses are in circle, first and last are adjacent
        // Case 1: Rob first house, can't rob last house (rob houses 0 to n-2)
        int robFirst = RobLinear(nums, 0, nums.Length - 2);

        // Case 2: Don't rob first house, can rob last house (rob houses 1 to n-1)
        int robLast = RobLinear(nums, 1, nums.Length - 1);

        return Math.Max(robFirst, robLast);
    }

    /// <summary>
    /// Linear house robber with space optimization
    /// </summary>
    private int RobLinear(int[] nums, int start, int end)
    {
        if (start == end) return nums[start];

        int prev2 = nums[start];
        int prev1 = Math.Max(nums[start], nums[start + 1]);

        for (int i = start + 2; i <= end; i++)
        {
            int current = Math.Max(prev1, prev2 + nums[i]);
            prev2 = prev1;
            prev1 = current;
        }

        return prev1;
    }
}
```

Key Takeaways:

- **Circular constraint**: Break into linear subproblems by excluding boundary elements
- **Case analysis**: Consider mutually exclusive scenarios (rob first vs rob last)
- **Code reuse**: Leverage existing linear solution for circular variant

PROBLEM 6: UNIQUE PATHS

Problem Statement:

There is a robot on an `m x n` grid. The robot is initially located at the **top-left corner** (i.e., `grid`). The robot tries to move to the **bottom-right corner** (i.e., `grid[m - 1][n - 1]`). The robot can only move either down or right at any point in time.

Given the two integers `m` and `n`, return the number of possible unique paths that the robot can take to reach the bottom-right corner.

Test Cases:

```
Example 1:
Input: m = 3, n = 7
Output: 28
Explanation: There are 28 unique paths from top-left to bottom-right.

Example 2:
Input: m = 3, n = 2
Output: 3
Explanation: From top-left corner, there are 3 ways to reach bottom-right:
1. Right -> Down -> Down
2. Down -> Right -> Down
3. Down -> Down -> Right

Example 3:
Input: m = 1, n = 1
Output: 1
Explanation: Only one cell, robot is already at destination.
```

Interview Questions & Answers:

1. **Q:** "Is this a combinatorics problem?"
   A: Yes! We need exactly (m-1) down moves and (n-1) right moves. Total paths = C(m+n-2, m-1) = C(m+n-2, n-1).
2. **Q:** "Can you solve this without DP?"
   A: Yes, using combinatorics formula, but DP is more intuitive and easily extensible to variants with obstacles.
3. **Q:** "How would you handle obstacles in the grid?"
   A: That's a variation - set dp[i][j] = 0 for cells with obstacles, otherwise same recurrence relation.
4. **Q:** "Can you optimize space complexity?"
   A: Yes, since each cell only depends on the cell above and to the left, we can use O(min(m,n)) space with rolling array.
5. **Q:** "What if robot could also move diagonally?"
   A: Add diagonal movement: dp[i][j] = dp[i-1][j] + dp[i][j-1] + dp[i-1][j-1].

Approach 1: Recursive (Brute Force)

Idea: From each cell, recursively count paths going right and paths going down.

Time Complexity: O(2^(m+n)) - exponential branching
Space Complexity: O(m+n) - recursion depth

```
using System;

public class Solution
{
    public int UniquePaths(int m, int n)
    {
        return CountPaths(0, 0, m, n);
    }

    private int CountPaths(int row, int col, int m, int n)
    {
        // Base case: reached destination
        if (row == m - 1 && col == n - 1) return 1;

        // Base case: out of bounds
        if (row >= m || col >= n) return 0;

        // Recursive case: sum paths going right and down
        int rightPaths = CountPaths(row, col + 1, m, n);
        int downPaths = CountPaths(row + 1, col, m, n);

        return rightPaths + downPaths;
    }
}
```

Approach 2: Top-Down DP (Memoization)

Explanation: Cache the results for each (row, col) position to avoid recomputing the same subproblems.

Time Complexity: O(m × n) - each cell computed once
Space Complexity: O(m × n) - memoization table plus recursion stack

```
using System;

public class Solution
{
    private int[,] memo;

    public int UniquePaths(int m, int n)
    {
        memo = new int[m, n];

        // Initialize memo with -1 (uncomputed)
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < n; j++)
            {
                memo[i, j] = -1;
            }
        }

        return CountPaths(0, 0, m, n);
    }

    private int CountPaths(int row, int col, int m, int n)
    {
        // Base cases
        if (row == m - 1 && col == n - 1) return 1;
        if (row >= m || col >= n) return 0;

        // Check memo
        if (memo[row, col] != -1) return memo[row, col];

        // Compute and store result
        int rightPaths = CountPaths(row, col + 1, m, n);
        int downPaths = CountPaths(row + 1, col, m, n);

        memo[row, col] = rightPaths + downPaths;
        return memo[row, col];
    }
}
```

Approach 3: Optimal - Bottom-Up DP with Space Optimization

Detailed Reasoning: Use bottom-up DP but optimize space since each row only depends on the previous row. We can use a 1D array and update it in place.

Time Complexity: O(m × n) - fill each cell once
Space Complexity: O(min(m, n)) - use smaller dimension for space optimization

```
using System;

public class Solution
{
    /// <summary>
    /// Counts unique paths using space-optimized bottom-up DP
    /// Time: O(m * n), Space: O(min(m, n))
    /// </summary>
    public int UniquePaths(int m, int n)
    {
        // Optimize space by using smaller dimension
        if (m < n)
        {
            int temp = m;
            m = n;
            n = temp;
        }

        // dp[j] represents number of paths to reach current row, column j
        int[] dp = new int[n];

        // Initialize first row: only one way to reach each cell in first row
        Array.Fill(dp, 1);

        // Fill remaining rows
        for (int i = 1; i < m; i++)
        {
            for (int j = 1; j < n; j++)
            {
                // Current cell = paths from above + paths from left
                // dp[j] already contains paths from above (previous iteration)
                // dp[j-1] contains paths from left (current iteration)
                dp[j] = dp[j] + dp[j - 1];
            }
        }

        return dp[n - 1];
    }
}
```

Alternative: Mathematical Solution

Mathematical Reasoning: This is equivalent to choosing (m-1) positions for down moves out of (m+n-2) total moves.

Time Complexity: O(min(m, n)) - computing combination
Space Complexity: O(1) - only using variables

```
using System;

public class Solution
{
    /// <summary>
    /// Counts unique paths using combinatorics formula
    /// Time: O(min(m, n)), Space: O(1)
    /// </summary>
    public int UniquePathsMath(int m, int n)
    {
        // Total moves needed: (m-1) down + (n-1) right = m+n-2
        // Choose (m-1) positions for down moves: C(m+n-2, m-1)

        int totalMoves = m + n - 2;
        int downMoves = m - 1;

        // Compute C(totalMoves, downMoves) efficiently
        long result = 1;

        // Use smaller of downMoves or (totalMoves - downMoves) for efficiency
        int k = Math.Min(downMoves, totalMoves - downMoves);

        for (int i = 0; i < k; i++)
        {
            result = result * (totalMoves - i) / (i + 1);
        }

        return (int)result;
    }
}
```

Key Takeaways:

- **Grid DP pattern**: dp[i][j] = dp[i-1][j] + dp[i][j-1]
- **Space optimization**: Use 1D array when only previous row/column needed
- **Mathematical insight**: Grid paths equivalent to combinatorial selection

PROBLEM 7: JUMP GAME

Problem Statement:

You are given an integer array `nums`. You are initially positioned at the array's **first index**, and each element in the array represents your maximum jump length at that position.

Return `true` if you can reach the last index, or `false` otherwise.

Test Cases:

```
Example 1:
Input: nums = [2,3,1,1,4]
Output: true
Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.

Example 2:
Input: nums = [3,2,1,0,4]
Output: false
Explanation: You will always arrive at index 3. Its maximum jump length is 0, which makes

Example 3:
Input: nums = [^0]
Output: true
Explanation: Already at the last index.
```

Interview Questions & Answers:

1. **Q:** "Is this a greedy problem or DP problem?"
   A: Both approaches work! Greedy is more efficient O(n), DP is O(n^2) but more intuitive for understanding reachability.
2. **Q:** "What does 'maximum jump length' mean?"
   A: From position i, you can jump to any position from i+1 to i+nums[i] (inclusive), not exactly nums[i] steps.
3. **Q:** "Can you jump backwards?"
   A: No, you can only move forward in the array.
4. **Q:** "What if an element is 0?"
   A: You can't move forward from that position. If you must pass through it to reach the end, the answer is false.
5. **Q:** "How would you modify this to find the minimum number of jumps?"
   A: That's "Jump Game II" - use BFS or greedy approach with jump counting.

Approach 1: Recursive with Backtracking

Idea: From each position, try all possible jump lengths and see if any path reaches the end.

Time Complexity: O(2^n) - exponential branching for each position
Space Complexity: O(n) - recursion depth

```
using System;

public class Solution
{
    public bool CanJump(int[] nums)
    {
        return CanJumpFromPosition(0, nums);
    }

    private bool CanJumpFromPosition(int position, int[] nums)
    {
        // Base case: reached the last index
        if (position >= nums.Length - 1) return true;

        // Try all possible jump lengths from current position
        int furthestJump = Math.Min(position + nums[position], nums.Length - 1);

        for (int nextPosition = position + 1; nextPosition <= furthestJump; nextPosition+
        {
            if (CanJumpFromPosition(nextPosition, nums))
            {
                return true;
            }
        }

        return false; // No valid path found from this position
    }
}
```

Approach 2: Dynamic Programming

Explanation: Use DP where dp[i] represents whether we can reach index i. Build solution bottom-up.

Time Complexity: $O(n^2)$ - for each position, check all reachable positions
Space Complexity: O(n) - DP array

```
using System;

public class Solution
{
    public bool CanJump(int[] nums)
    {
        if (nums.Length <= 1) return true;

        bool[] dp = new bool[nums.Length];
        dp[^0] = true; // Can always reach starting position

        for (int i = 0; i < nums.Length; i++)
        {
            if (!dp[i]) continue; // Can't reach position i, skip

            // From position i, mark all reachable positions as true
            int furthest = Math.Min(i + nums[i], nums.Length - 1);
            for (int j = i + 1; j <= furthest; j++)
            {
                dp[j] = true;

                // Early termination: if we can reach the end, return true
                if (j == nums.Length - 1) return true;
            }
        }

        return dp[nums.Length - 1];
    }
}
```

Approach 3: Optimal - Greedy

Detailed Reasoning: Track the furthest position reachable so far. If current position is beyond furthest reachable, return false. If furthest reachable covers the last index, return true. This greedy approach is optimal because we only care about reachability, not the specific path.

Time Complexity: O(n) - single pass through array
Space Complexity: O(1) - only using variables

```
using System;

public class Solution
{
    /// <summary>
    /// Determines if last index is reachable using greedy approach
    /// Time: O(n), Space: O(1)
    /// </summary>
    public bool CanJump(int[] nums)
    {
        if (nums.Length <= 1) return true;

        int furthestReachable = 0;

        for (int i = 0; i < nums.Length; i++)
        {
            // If current position is beyond what we can reach, return false
            if (i > furthestReachable) return false;

            // Update furthest reachable position
            furthestReachable = Math.Max(furthestReachable, i + nums[i]);

            // Early termination: if we can reach or exceed last index
            if (furthestReachable >= nums.Length - 1) return true;
        }

        return false;
    }
}
```

Alternative Greedy Approach - Backwards

Reasoning: Start from the end and work backwards. Track the leftmost position that can reach the end.

```
using System;

public class Solution
{
    /// <summary>
    /// Backwards greedy approach
    /// Time: O(n), Space: O(1)
    /// </summary>
    public bool CanJumpBackwards(int[] nums)
    {
        int lastGoodPosition = nums.Length - 1;

        // Work backwards from second-to-last position
        for (int i = nums.Length - 2; i >= 0; i--)
        {
            // If current position can reach the last good position
            if (i + nums[i] >= lastGoodPosition)
            {
                lastGoodPosition = i;
            }
        }

        // Check if we can reach the last good position from start
        return lastGoodPosition == 0;
    }
}
```

Key Takeaways:

- **Greedy optimization**: Track furthest reachable position instead of all paths
- **Early termination**: Return as soon as end is reachable
- **Multiple valid approaches**: DP for understanding, greedy for efficiency

DAY 7 SUMMARY

Summary of Concepts Covered

Dynamic Programming Fundamentals:

- **Overlapping subproblems**: Same calculations repeated multiple times
- **Optimal substructure**: Optimal solution contains optimal solutions to subproblems
- **Memoization vs Tabulation**: Top-down caching vs bottom-up table building
- **Space optimization**: Reducing space complexity when only recent states are needed

Common DP Patterns:

- **Fibonacci sequence**: Linear recurrence relations (Climbing Stairs, House Robber)
- **Knapsack variants**: Optimization with constraints (Coin Change, subset problems)
- **Grid pathfinding**: 2D DP for counting or optimizing paths (Unique Paths)
- **Sequence problems**: Finding optimal subsequences (LIS, LCS)

Decision Making Strategies:

- **Greedy vs DP**: When local optimum leads to global vs when it doesn't
- **State transition**: How to move from one state to another
- **Base cases**: Proper initialization for recursive and iterative solutions

Time & Space Complexity Summary

Problem Type	Brute Force	DP Solution	Optimized	Key Technique
Climbing Stairs	**$O(2^n)$** recursive	**$O(n)$** time/space	**$O(n)$** time, **$O(1)$** space	Fibonacci with space optimization
Coin Change	**$O(a^c)$** exponential	**O(amount × coins)**	**O(amount × coins)**	Unbounded knapsack DP
Longest Increasing Subsequence	**$O(2^n \times n)$**	**$O(n^2)$**	**$O(n \log n)$**	Binary search optimization
House Robber	**$O(2^n)$** recursive	**$O(n)$** time/space	**$O(n)$** time, **$O(1)$** space	Linear DP with optimization
House Robber II	**$O(2^n)$** circular	**$O(n)$** two cases	**$O(n)$** time, **$O(1)$** space	Reduce to linear cases
Unique Paths	**O(2^(m+n))**	**$O(m \times n)$**	**O(min(m,n))** space	Grid DP with rolling array
Jump Game	**$O(2^n)$** recursive	**$O(n^2)$** DP	**$O(n)$** greedy	Furthest reachable tracking

Pattern Recognition Guide

Use Fibonacci Pattern When:

- Problem involves sequences where current depends on previous values
- Linear recurrence relation exists
- Examples: Climbing stairs, decode ways, house robber

Use Knapsack Pattern When:

- Optimization problem with constraints
- Items can be included/excluded or used multiple times
- Examples: Coin change, subset sum, partition problems

Use Grid DP When:

- 2D problem space (matrix, grid)
- Path counting or optimization in 2D space
- Examples: Unique paths, minimum path sum, edit distance

Use Greedy When:

- Local optimal choice leads to global optimum
- Problem has greedy choice property
- Examples: Jump game, activity selection, fractional knapsack

DP Implementation Strategies

Top-Down (Memoization):

```
int DP(params, memo) {
    if (base_case) return base_value;
    if (memo.Contains(params)) return memo[params];

    int result = recurrence_relation();
    memo[params] = result;
    return result;
}
```

Bottom-Up (Tabulation):

```
int DP(n) {
    int[] dp = new int[n + 1];
    dp[^0] = base_case_0;
    dp[^1] = base_case_1;

    for (int i = 2; i <= n; i++) {
        dp[i] = recurrence_relation(dp[i-1], dp[i-2]);
    }
    return dp[n];
}
```

Space Optimization:

```
int DPOptimized(n) {
    int prev2 = base_case_0;
    int prev1 = base_case_1;

    for (int i = 2; i <= n; i++) {
        int current = recurrence_relation(prev1, prev2);
        prev2 = prev1;
        prev1 = current;
    }
    return prev1;
}
```

Review Tips

- **Identify overlapping subproblems**: Look for repeated calculations in brute force
- **Define state clearly**: What does dp[i] represent?
- **Find recurrence relation**: How does current state relate to previous states?
- **Handle base cases**: Initialize properly for edge cases
- **Consider space optimization**: Can you reduce space complexity?

How to Practice Mock Interviews

1. **Problem Analysis** (5 min): Identify DP pattern and overlapping subproblems
2. **State Definition** (3 min): Define what each DP state represents
3. **Recurrence Relation** (5 min): Derive the transition formula
4. **Implementation** (20 min): Code while explaining approach and optimizations
5. **Complexity Analysis** (2 min): Time and space complexity discussion
6. **Testing** (10 min): Walk through examples and edge cases

Common Mistakes to Avoid

- **Incorrect base cases**: Not handling edge cases properly (empty arrays, single elements)
- **Off-by-one errors**: Array indexing mistakes in DP table initialization
- **State definition confusion**: Unclear what each DP state represents
- **Recurrence relation errors**: Wrong transition formula between states
- **Space optimization bugs**: Incorrect variable updates in optimized versions
- **Integer overflow**: Not considering overflow in problems with large numbers

Advanced DP Topics for Further Study

- **Multi-dimensional DP**: Problems with multiple parameters
- **State compression**: Bit manipulation in DP states
- **Tree DP**: Dynamic programming on tree structures
- **Digit DP**: Problems involving digit constraints
- **Probability DP**: Expected value and probability calculations
- **Game theory DP**: Minimax and optimal strategy problems

Real-world Applications

- **Resource allocation**: Knapsack variants in operations research
- **Bioinformatics**: Sequence alignment using edit distance
- **Finance**: Portfolio optimization and risk management

- **Computer graphics**: Optimal path finding and rendering
- **Machine learning**: Viterbi algorithm, sequence labeling
- **Compiler optimization**: Code generation and optimization strategies

DP Mastery Checklist

- [ ] Understand when to use DP vs greedy vs divide-and-conquer
- [ ] Master both top-down and bottom-up approaches
- [ ] Practice space optimization techniques
- [ ] Recognize common DP patterns (Fibonacci, knapsack, grid, sequence)
- [ ] Handle edge cases and boundary conditions properly
- [ ] Analyze time and space complexity accurately
- [ ] Debug DP solutions systematically
- [ ] Optimize solutions when possible

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

⁂