**AI Prompts to Test Edge Cases**

**Question 1.** Find Missing Numbers in Array: Description: Given an unsorted integer array nums of size n containing numbers from 1 to n, find all the numbers that are missing from the array. Examples: • Input: [4, 3, 2, 7, 8, 2, 3, 1], Output: [5, 6] • Input: [1, 1], Output: [2]

**Below is the code for above question ,**

static void Main(string[] args) { // Question 1: Find Missing Numbers in Array Console.WriteLine("Question 1:"); int[] nums1 = { 4, 3, 2, 7, 8, 2, 3, 1 }; IList<int> missingNumbers = FindMissingNumbers(nums1); Console.WriteLine(string.Join(",", missingNumbers)); }

public static IList<int> FindMissingNumbers(int[] nums) { try { // Write your code here List<int> result = new List<int>(); // Create a list to store missing numbers // Iterate through the input array for (int i = 0; i < nums.Length; i++) { // Find the absolute value index (adjusting for zero-based index) int val = Math.Abs(nums[i]) - 1; // If the value at this index is positive, negate it to mark it as seen if (nums[val] > 0) { nums[val] = -nums[val]; } } // Check which indices are still positive (missing numbers) for (int i = 0; i < nums.Length; i++) { if (nums[i] > 0) { result.Add(i + 1); // Add the missing number (i + 1) to the result } } return result; // Return the list of missing numbers } catch (Exception) { throw; } }

**Prompt :does it cover all edge cases ?**

**Potential Edge Cases & Handling:**

| Edge Case | Covered? | Notes |
| --- | --- | --- |
| Empty array ([]) | ❌ No | Will throw IndexOutOfRangeException when trying to access nums[val]. |
| Array with all elements present ([1,2,3,4]) | ✅ Yes | Will return an empty list, as expected. |
| Array with all elements the same ([1,1,1,1]) | ✅ Yes | Will correctly identify missing numbers: [2,3,4]. |
| Very large arrays | ✅ Yes | Performance remains O(n), no extra space beyond result list. |
| Array with invalid values (e.g., out of 1 to n range) | ❌ No | Assumes all values are in 1 to n; out-of-range values may cause IndexOutOfRangeException. |

**❗️ Recommendations for Robustness:**

1. **Handle empty array: Add a quick check:**

**if (nums == null || nums.Length == 0) return new List<int>();**

1. **Validate input values: Optionally, check if values fall within the range 1 to n:**

**if (nums[i] < 1 || nums[i] > nums.Length)**

**throw new ArgumentException("Input values must be in the range 1 to n");**

**Question 2 Sort Array by Parity:**

Description: Given an integer array nums, move all even integers to the beginning of the array followed by all odd integers. Return the array in-place. Examples: • Input: [3, 1, 2, 4], Output: [2, 4, 3, 1] • Input: [0, 1, 2], Output: [0, 2, 1]

**Below is the code for above question**

public static int[] SortArrayByParity(int[] nums) { try { // Write your code here int[] result = new int[nums.Length]; // Array to hold sorted numbers int evenIndex = 0; // Start index for even numbers int oddIndex = nums.Length - 1; // Start index for odd numbers // Loop through the input array to classify even and odd numbers foreach (int num in nums) { if (num % 2 == 0) // Check if the number is even { result[evenIndex++] = num; // Place even number at the front } else { result[oddIndex--] = num; // Place odd number at the back } } return result; // Return the array sorted by parity } catch (Exception) { throw; } }

**Prompt: Does it handle all edge cases?**

Your current solution for "Sort Array by Parity" works correctly and handles most edge cases — but not all, especially with regard to the “in-place” requirement mentioned in the problem description.

✅ Let’s break it down:

🔧 Functionality Review:

**Your implementation:**

* Uses a result[] array to hold the final reordered numbers.
* Uses two pointers: one for even numbers (evenIndex) and one for odd numbers (oddIndex).
* Places even numbers at the front, odd numbers at the back.

**✅ Correctness Check:**

Let’s validate with the provided examples:

Input: [3, 1, 2, 4]

Output: [2, 4, 3, 1] → ✅ Yes, valid output.

Input: [0, 1, 2]

Output: [0, 2, 1] → ✅ Yes, valid output.

Your logic is sound and works as intended in terms of parity sorting.

🔍 **Edge Case Analysis:**

| Edge Case | Handled? | Notes |
| --- | --- | --- |
| Empty array [] | ✅ Yes | Returns an empty array. |
| Single element [0] or [1] | ✅ Yes | Works as expected. |
| All even numbers [2, 4, 6] | ✅ Yes | Returns the same or reordered even list. |
| All odd numbers [1, 3, 5] | ✅ Yes | Returns the same or reversed odd list. |
| Mix of negative even/odd numbers | ⚠️ *Maybe* | Your implementation works numerically, but if input allows negatives, you should clarify the parity definition (-2 % 2 == 0 still works). |
| Already sorted by parity [2, 4, 1, 3] | ✅ Yes | Still returns valid output. |

**❗️ In-Place Requirement Not Met**

Your current solution does not perform in-place sorting because it uses a separate array (result[]) to store the output.

In-place means you should rearrange the original array (nums) without allocating a new one. This is critical if the interviewer, assignment, or coding challenge specifically asks for it.

✅ Suggested Fix for True In-Place Version:

public static int[] SortArrayByParity(int[] nums)

{

try

{

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

while (left < right)

{

if (nums[left] % 2 > nums[right] % 2)

{

// Swap nums[left] and nums[right]

int temp = nums[left];

nums[left] = nums[right];

nums[right] = temp;

}

if (nums[left] % 2 == 0) left++;

if (nums[right] % 2 == 1) right--;

}

return nums;

}

catch (Exception)

{

throw;

}

}

**This modifies the input array directly and satisfies the "in-place" requirement.**

**Question 3:** Two Sum (Find Two Numbers that Add to Target): Description: Given an array of integers nums and an integer target, return the indices of the two numbers such that they add up to the target.

Examples:

Input: nums = [2, 7, 11, 15], target = 9, Output: [0, 1]

• Input: nums = [3, 2, 4], target = 6, Output: [1, 2]

**Below is the code for the above question**

public static int[] TwoSum(int[] nums, int target) { try { // Write your code here Dictionary<int, int> map = new Dictionary<int, int>(); // Create a dictionary to store number and index // Iterate through the array to find two numbers that add up to the target for (int i = 0; i < nums.Length; i++) { int complement = target - nums[i]; // Calculate the complement // Check if the complement exists in the dictionary if (map.ContainsKey(complement)) { return new int[] { map[complement], i }; // Return the indices of the two numbers } map[nums[i]] = i; // Add the number and its index to the dictionary } return new int[0]; // Return an empty array if no solution is found } catch (Exception) { throw; } }

**does it cover all edge cases?**

Your implementation of the **Two Sum** problem is excellent and uses a **hash map (dictionary)** to efficiently find the two numbers that add up to the target in **O(n)** time. Let’s go through it carefully to ensure it covers **all edge cases**.

**✅ What the code does well:**

* Uses a dictionary to store numbers and their indices.
* For each number, checks if the complement (target - num) exists in the map.
* Returns the correct indices once a valid pair is found.
* Efficient: **O(n)** time and **O(n)** space.
* Matches example test cases.

**✅ Example Test Cases:**

**Example 1:**

csharp

CopyEdit

nums = [2, 7, 11, 15], target = 9

Output: [0, 1] ✅

**Example 2:**

csharp

CopyEdit

nums = [3, 2, 4], target = 6

Output: [1, 2] ✅

**🔍 Edge Case Analysis:**

| **Edge Case** | **Covered?** | **Explanation** |
| --- | --- | --- |
| ✅ Valid input with one solution | ✅ Yes | Standard case handled correctly |
| ❌ No solution exists | ✅ Yes | Returns empty array [] |
| 🔁 Duplicate numbers | ✅ Yes | Example: [3, 3], target 6 → will return [0, 1] |
| 🟰 Same number used twice? | ✅ Yes | Logic ensures indices are **different** |
| 🔢 Negative numbers | ✅ Yes | Works fine, e.g., [-1, -2, 3], target 1 |
| ❌ Empty input array | ✅ Yes | Will not enter the loop, returns [] |
| ❌ Array of size 1 | ✅ Yes | Loop will not find a pair, returns [] |

**Question 4.** Find Maximum Product of Three Numbers:

Description: Given an integer array nums, find three numbers whose product is the maximum and return the product.

Examples: • Input: [1, 2, 3], Output: 6 • Input: [1, 2, 3, 4], Output: 24

**Below is the code for above question**

public static int MaximumProduct(int[] nums) { try { // Write your code here Array.Sort(nums); // Sort the array int n = nums.Length; // Get the length of the array // Calculate the maximum product of the last three numbers or the product of the first two (smallest) and the last (largest) return Math.Max(nums[n - 1] \* nums[n - 2] \* nums[n - 3], nums[0] \* nums[1] \* nums[n - 1]); } catch (Exception) { throw; } }

**Prompt: does it handles edge cases?**

Your implementation of "Find Maximum Product of Three Numbers" is correct, efficient, and handles edge cases well in most practical scenarios. Let’s walk through its strengths, limitations, and edge case coverage.

✅ What the code does right:

This handles the two common scenarios:

1. Top 3 positive numbers for maximum product.
2. Two large negative numbers and one large positive, which can result in a higher product.

**🔍 Edge Case Coverage:**

| Edge Case | Covered? | Explanation |
| --- | --- | --- |
| ✅ Minimum length (3 elements) | ✅ Yes | Sorting and max logic works fine. |
| ✅ Negative numbers | ✅ Yes | Considers both nums[0]\*nums[1]\*nums[n-1] and largest 3. |
| ✅ Mix of negatives and positives | ✅ Yes | Example: [-10, -10, 5, 2] → Max product is 500. |
| ✅ All negative numbers | ✅ Yes | For [-5, -4, -3, -2], top 3 largest negative numbers yield the product. |
| ✅ Contains zero(s) | ✅ Yes | Won’t affect unless zero is among top 3 contributors to product. |
| ❌ Array length < 3 | ⚠️ No | Code will throw an IndexOutOfRangeException. Should guard against this. |

**❗️ Improvement: Input Validation**

You can add a validation to ensure the input array has at least 3 elements:

if (nums == null || nums.Length < 3)

throw new ArgumentException("Input array must have at least 3 integers.");

**Question 5 : Decimal to Binary Conversion: Description:**

Write a function that converts a decimal number to its binary equivalent. Examples: • Input: 42, Output: 101010 • Input: 10, Output: 1010

**Below is the code for above question**

public static string DecimalToBinary(int decimalNumber) { try { // Write your code here if (decimalNumber == 0) return "0"; // Special case for zero string binary = ""; // Initialize binary string // Convert decimal to binary by repeatedly dividing by 2 while (decimalNumber > 0) { binary = (decimalNumber % 2) + binary; // Prepend the remainder (0 or 1) decimalNumber /= 2; // Divide the number by 2 } return binary; // Return the binary representation } catch (Exception) { throw; } }

**does it handles edge cases?**

Your implementation of the Decimal to Binary Conversion function is clean, correct, and handles the conversion logic well. Let’s evaluate whether it covers all relevant edge cases and see if there are any improvements or additions you could make.

**✅ What the Code Does Well**

if (decimalNumber == 0) return "0";

* This correctly handles the special case for 0.

while (decimalNumber > 0)

{

binary = (decimalNumber % 2) + binary;

decimalNumber /= 2;

}

* This converts any positive integer to its binary representation using repeated division.

**🔍 Edge Case Analysis**

| Edge Case | Handled? | Notes |
| --- | --- | --- |
| decimalNumber == 0 | ✅ Yes | Explicitly handled |
| Positive integer (e.g. 10, 42) | ✅ Yes | Works as expected |
| Large integers (e.g. 1000000) | ✅ Yes | Handled correctly unless integer overflows (not an issue here) |
| ❌ Negative numbers | ❌ No | Will result in an infinite loop or incorrect result since condition is while (decimalNumber > 0) |
| ❌ Null input | ❌ N/A | Not applicable — int is a value type (can't be null) |

❗️ Potential Issue: Negative Inputs

The function doesn't handle negative numbers. If someone calls:

csharp

CopyEdit

DecimalToBinary(-5)

* The loop will never execute, and the function will return "" (empty string), which is misleading.

**So handled negative numbers as well .**

**Question 6:** Find Minimum in Rotated Sorted Array: Description: Given a sorted array that has been rotated, find the minimum element. Examples: • Input: [3, 4, 5, 1, 2], Output: 1 • Input: [4, 5, 6, 7, 0, 1, 2], Output: 0

**below is the code for above question**

public static int FindMin(int[] nums) { try { // Write your code here int left = 0; // Start index int right = nums.Length - 1; // End index // Use binary search to find the minimum element in the rotated sorted array while (left < right) { int mid = (left + right) / 2; // Find the middle index // If the middle element is greater than the rightmost element, the minimum is to the right if (nums[mid] > nums[right]) { left = mid + 1; // Move left index to mid + 1 } else { right = mid; // Move right index to mid } } return nums[left]; // Return the minimum element } catch (Exception) { throw; } }

**Prompt : Does it covers all edge cases ?**

Your implementation of **"Find Minimum in Rotated Sorted Array"** is **correct**, efficient, and well-structured using a **binary search** approach. Let's evaluate its **correctness** and check whether it handles **all edge cases**.

**✅ What the Code Does Well:**

* Uses **binary search** to efficiently find the minimum in **O(log n)** time.
* Compares nums[mid] with nums[right] to determine whether the minimum lies to the left or right half.
* The loop exits when left == right, at which point nums[left] is the **minimum**.

**🧪 Example Test Cases**

| **Input** | **Output** | **Works?** |
| --- | --- | --- |
| [3, 4, 5, 1, 2] | 1 | ✅ |
| [4, 5, 6, 7, 0, 1, 2] | 0 | ✅ |

**🔍 Edge Case Analysis**

| **Edge Case** | **Covered?** | **Notes** |
| --- | --- | --- |
| Already sorted array (no rotation) e.g. [1, 2, 3, 4, 5] | ✅ Yes | Minimum is the first element |
| Fully rotated array (rotation equal to array length) e.g. [1, 2, 3, 4, 5] | ✅ Yes | Behaves like unrotated |
| Small array (length = 1) e.g. [10] | ✅ Yes | left == right == 0 |
| Array with 2 elements, rotated e.g. [2, 1] | ✅ Yes | Binary search still applies |
| Large array | ✅ Yes | Binary search ensures O(log n) |
| ❌ Array with **duplicates** (e.g. [2, 2, 2, 0, 1]) | ❌ No | Problem assumes **distinct elements** — this logic fails or needs to be adapted |
| ❌ Empty array | ❌ No | Will throw IndexOutOfRangeException |

**❗️ Improvements Suggested:**

**1. ✅ Guard for empty input:**

if (nums == null || nums.Length == 0)

throw new ArgumentException("Input array must not be empty.");

**So handled empty array and array with duplicates.**

**Question 7: Palindrome Number Description:**

Given an integer x, return true if x is a palindrome, and false otherwise. A palindrome is a number that reads the same forward and backward. Examples: • Input: 121, Output: true • Input: 10, Output: false (Explanation: Reads 01 from right to left. Therefore, it is not a palindrome.)

**Below is the code for the above question**

public static bool IsPalindrome(int x) { try { // Write your code here if (x < 0) return false; // Negative numbers are not palindromes int original = x; // Store the original number for comparison int reversed = 0; // Initialize the reversed number // Reverse the digits of the number while (x > 0) { int digit = x % 10; // Get the last digit reversed = reversed \* 10 + digit; // Append digit to the reversed number x /= 10; // Remove the last digit from x } return original == reversed; // Check if the original number and reversed number are equal } catch (Exception) { throw; } }

**Does it cover all edge cases**

Your implementation of the **Palindrome Number** check is solid and **handles most edge cases correctly**. It reverses the number and compares it to the original, which is a valid and widely used approach.

Let’s analyze it in depth:

**✅ What the code does right:**

* Handles **negative numbers** by returning false immediately.
* Reverses the digits of the number accurately.
* Compares the original and reversed values to determine if it's a palindrome.

**🧪 Edge Case Coverage:**

| **Edge Case** | **Handled?** | **Notes** |
| --- | --- | --- |
| x = 121 | ✅ Yes | Reverses to 121 |
| x = 10 | ✅ Yes | Reverses to 01 → false |
| x = -121 | ✅ Yes | Negative → false |
| x = 0 | ✅ Yes | Palindrome (0 == 0) |
| x = 1 | ✅ Yes | Palindrome |
| Very large numbers | ✅ Yes | As long as int doesn’t overflow |
| x = 1000021 | ✅ Yes | Not a palindrome, returns false |

**Question 8**: Fibonacci Number Description:

The Fibonacci numbers, commonly denoted F(n), form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is, • F(0) = 0, F(1) = 1 • F(n) = F(n - 1) + F(n - 2), for n > 1 Given n, calculate F(n). Examples: • Input: 2, Output: 1 • Input: 3, Output: 2 • Input: 4, Output: 3 Constraints: • 0 <= n <= 30

**Below is the code for the above question**

public static int Fibonacci(int n) { try { // Write your code here if (n == 0) return 0; // Base case for Fibonacci if (n == 1) return 1; // Base case for Fibonacci int a = 0, b = 1; // Initialize the first two Fibonacci numbers // Calculate Fibonacci iteratively for (int i = 2; i <= n; i++) { int temp = a + b; // Calculate the next Fibonacci number a = b; // Update a to the next number b = temp; // Update b to the newly calculated number } return b; } catch (Exception) { throw; } }

**does it handle all edge cases?**

Yes, your implementation of the **Fibonacci number calculator** is **correct**, efficient, and handles **all relevant edge cases** for the given constraints.

**✅ What the code does well:**

* Uses **iterative approach** (not recursive), which avoids:
  + Stack overflow
  + Redundant calculations
* Handles the **base cases** (n = 0 and n = 1) explicitly.
* Works efficiently for all values in the **constraint range 0 <= n <= 30**.

**🔍 Edge Case Coverage:**

| **Edge Case** | **Covered?** | **Notes** |
| --- | --- | --- |
| n = 0 | ✅ Yes | Returns 0 |
| n = 1 | ✅ Yes | Returns 1 |
| n = 2, 3, 4, … | ✅ Yes | Returns correct Fibonacci value |
| n = 30 | ✅ Yes | Result is 832040, well within int range |
| ❌ n < 0 | ❌ No | Not checked; currently would run loop from 2 to n, which breaks logic |
| ❌ n > 30 | ❌ Not needed | As per constraints, you don’t need to support beyond n = 30, but values up to ~46 still fit in int |

**🛡️ Optional Improvement: Input Validation**

If you want to make your function more robust (beyond constraints), add a guard clause:

if (n < 0)

throw new ArgumentException("Input must be a non-negative integer.");

So included the constraints exception.