**Starting Prompt:**

* Help me solving the coding problem in C#. Don’t change the method declarations — just write your logic inside as it is (same parameters, same return types).
* Add inline comments wherever it is necessary, so that it will be easy to understand
* Able to handle errors through try and catch blocks
* Your solution should handle all kinds of test cases — like empty inputs, invalid values, negatives, and duplicates.
* Make sure the algorithm you provide is efficient and covers the time and space complexity constraints.
* Implement right data structures or techniques wherever needed like hash maps, sorting, binary search according to the problem.

**Question 1: Find Missing Numbers in Array**

* **Prompt Used:**   
  Find missing numbers in an unsorted integer array nums of size n containing numbers from 1 to n, find all the numbers that are missing from the array.
* **Response Received:**  
  Copilot suggested using a boolean array to keep track of which numbers from 1 to n are in the input. For each number, if it’s valid, we mark it as found. Then we just go through the list and pick out the numbers that weren’t marked those are the missing ones.
* **Implementation Details:**  
  Implemented a boolean array of size n + 1 to track valid numbers. Implemented loop through the array and mark true for each valid number (1 to n). At the end, collected all indices that are still false.
* **Edge Cases Covered:**  
  Empty array, duplicates, negatives, values greater than n, and all values present.

**Question 2: Sort Array by Parity**

* **Prompt Used:**  
  Solve Sort Array by Parity, move all even integers to the beginning of the array followed by all odd integers.
* **Response Received:**  
  Copilot recommended a two-pointer solution and confirmed that parity (even/odd) logic applies to both positive and negative numbers, including zero.
* **Implementation Details:**  
  Used two-pointer approach. First initialized one pointer at start and one at end. Swapped elements based on parity, moved left pointer for even, right pointer for odd.
* **Edge Cases Covered:**  
  Negative numbers, 0, all even or all odd, mixed inputs, and already sorted parity.

**Question 3: Two Sum**

* **Prompt Used:**  
  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.
* **Response Received:**  
  Copilot suggested using a dictionary for complement checks and allowed duplicates as long as indices differ.
* **Implementation Details:**  
  Iterated through the array, checking if the complement of each number exists in the dictionary. If found, returned both indices. Otherwise, stored the number and its index.
* **Edge Cases Covered:**  
  Duplicates (like [3,3]), empty array, single element, target not reachable, negatives.

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

* **Prompt Used:**  
  Find maximum product of three numbers in integer array
* **Response Received:** Copilot suggested sorting the array and checking two possible products: the product of the three biggest numbers, and the product of the two smallest (possibly negative) numbers with the largest one. The higher of the two is our answer.
* **Implementation Details:**  
  Sorted the array and calculated both arr[n-1] \* arr[n-2] \* arr[n-3] and arr[0] \* arr[1] \* arr[n-1]. Returned the greater of the two values.
* **Edge Cases Covered:**  
  Less than 3 elements, zeros, all negatives, mixed signs.

**Question 5: Decimal to Binary**

* **Prompt Used:**  
  how to convert decimal to binary representation
* **Response Received:**  
  Copilot recommended dividing the number by 2 repeatedly and storing the remainder each time. It build the binary string by adding each remainder to the front. It also mentioned that if the input is 0, we just return "0".
* **Implementation Details:**  
  Used a loop to divide the number by 2, storing each remainder at the front of a string. Special case: returned "0" if the input was 0.
* **Edge Cases Covered:**  
  Zero, one, large numbers.

**Question 6: Find Minimum in Rotated Sorted Array**

* **Prompt Used:**Given a sorted array that has been rotated, find the minimum element.
* **Response Received:**  
  Copilot explained that we can use binary search by comparing the middle element with the rightmost one to decide which half of the array contains the minimum, and continue this process until we narrow it down to the smallest element.
* **Implementation Details:**  
  Used binary search: compared mid and right values to adjust the search window. Continued until left and right pointers converged on the minimum.
* **Edge Cases Covered:**  
  Array with one element, not rotated, rotated at different positions, small arrays.

**Question 7: Palindrome Number**

* **Prompt Used:**  
  c# Program to check if number is palindrome with overflow and negative cases
* **Response Received:**  
  Copilot suggested reversing the number by taking digits from the end and building a new number. Before reversing, we check for negatives. At the end, we compare the reversed number to the original to see if it’s a palindrome.
* **Implementation Details:**  
  Checked if the number is negative. Reversed the digits by extracting the last digit and building the reverse using multiplication and addition.
* **Edge Cases Covered:**  
  Negative numbers, 0, overflow risk, single-digit numbers.

**Question 8: Fibonacci Number**

* **Prompt Used:**  
  Implement fibonacci in c# with input constraint between 0 and 30
* **Response Received:**  
  Copilot suggested using a loop instead of recursion to calculate the Fibonacci number. We start with 0 and 1, and add them up until we reach the nth number. It also recommended checking if the input is within the valid range (0 to 30).
* **Implementation Details:**  
  Used a loop to build the Fibonacci series up to the given number. Returned 0 for n=0 and 1 for n=1. Used iteration for efficiency.
* **Edge Cases Covered:**  
  n = -1, n = 31, n = 0, n = 1, upper bound check.

**Adjustments:**

In all the problems, I incorporated try-catch blocks to catch errors in a safe manner and made sure code is processed for edge cases like empty inputs, negatives, and inputs outside of expected ranges. I tried with invalid inputs, accepted duplicates when necessary, and used appropriate data structures like dictionaries and boolean arrays so that everything remained efficient. For overflow scenarios with potential limit or special scenarios such as reversing a number or calculating the Fibonacci, I included a safety check. I also added comments where ever necessary

using System;

using System.Collections.Generic;

namespace Assignment\_2

{

class Program

{

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));

// Question 2: Sort Array by Parity

Console.WriteLine("Question 2:");

int[] nums2 = { 3, 1, 2, 4 };

int[] sortedArray = SortArrayByParity(nums2);

Console.WriteLine(string.Join(",", sortedArray));

// Question 3: Two Sum

Console.WriteLine("Question 3:");

int[] nums3 = { 2, 7, 11, 15 };

int target = 9;

int[] indices = TwoSum(nums3, target);

Console.WriteLine(string.Join(",", indices));

// Question 4: Find Maximum Product of Three Numbers

Console.WriteLine("Question 4:");

int[] nums4 = { 1, 2, 3, 4 };

int maxProduct = MaximumProduct(nums4);

Console.WriteLine(maxProduct);

// Question 5: Decimal to Binary Conversion

Console.WriteLine("Question 5:");

int decimalNumber = 42;

string binary = DecimalToBinary(decimalNumber);

Console.WriteLine(binary);

// Question 6: Find Minimum in Rotated Sorted Array

Console.WriteLine("Question 6:");

int[] nums5 = { 3, 4, 5, 1, 2 };

int minElement = FindMin(nums5);

Console.WriteLine(minElement);

// Question 7: Palindrome Number

Console.WriteLine("Question 7:");

int palindromeNumber = 121;

bool isPalindrome = IsPalindrome(palindromeNumber);

Console.WriteLine(isPalindrome);

// Question 8: Fibonacci Number

Console.WriteLine("Question 8:");

int n = 4;

int fibonacciNumber = Fibonacci(n);

Console.WriteLine(fibonacciNumber);

}

// Question 1: Find Missing Numbers in Array

public static IList<int> FindMissingNumbers(int[] nums)

{

try

{

// length of the input array

int n = nums.Length;

// Create a boolean array to track presence of numbers from 1 to n,Default value=false, means number is not present yet

bool[] present = new bool[n];

// Mark numbers that appear in the input array

for (int i = 0; i < n; i++)

{

int val = nums[i];

// Only consider numbers in valid range [1,n].So, Numbers < 1 or > n are ignored as they don't affect missing numbers in range

if (val >= 1 && val <= n)

{

// Mark this number as present

present[val - 1] = true;

}

}

//Find all missing numbers from 1 to n

List<int> result = new List<int>();

for (int i = 0; i < n; i++)

{

// If number wasn't marked as present, it's missing

if (!present[i])

{

// Add the missing number to result

result.Add(i + 1);

}

}

// Return the list of missing numbers

return result;

}

catch (Exception)

{

throw;

}

}

// Edge Cases:

// Input: [] → Output: [] Reason: Empty array gives us nothing to check, so nothing’s missing

// Input: [-1, 2, -3] → Output: [1, 3] Reason: Negative numbers don’t count, so all 1-3 are missing

// Input: [-1, 4] → Output: [1, 2] Reason: -1 is ignored (too small), 4 is ignored (too big for n=2), so 1 and 2 are missing

// Input: [1, 2, 3] → Output: [] Reason: All numbers from 1 to 3 are here, so no gaps

// Input: [4, 5, 6] → Output: [1, 2, 3] Reason: All numbers are > n (3), so 1-3 are missing

// Input: [2, 2, 2] → Output: [1, 3] Reason: Only 2 is present, duplicates don’t change that

// Input: [1] → Output: [] Reason: Just 1 fits n=1, no numbers missing

// Question 2: Sort Array by Parity

// This method moves all even numbers to the beginning of the array and places odd numbers after them.

// We use two pointers: one to find even numbers, and one to place them at the front by swapping positions.

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

{

try

{

int left = 0, right = 0; // Both pointers start from the beginning

// Loop through the array using the right pointer

while (right < nums.Length)

{

if (nums[right] % 2 == 0) // If the number is even

{

// Swap the even number to the front, position of left pointer

int temp = nums[left];

nums[left] = nums[right];

nums[right] = temp;

left++; // Move left pointer forward for the next even number

}

right++; // Move to the next number

}

return nums; // Return the updated array

}

catch (Exception)

{

throw;

}

}

//Edge Cases

// Input: [] → Output: [] Reason: Empty array gives us nothing to check, so nothing’s missing.

// Input: [-2, -3, -4, -1] → Output: [-2, -4, -3, -1] Reason: Negative numbers — evens and odds still apply with negatives.

// Input: [0] → Output: [0] Reason: 0 is an even number, should stay in front if it's the only element.

// Input: [5, 6, 7, 0, 8, 2, 9] → Output: [6, 0, 8, 2, 7, 5, 9] Reason: Mixed values with 0 and unsorted elements — tests correct grouping of evens to front.

// Input: [-5, 4, -2, 3, 0, -1, 6] → Output: [4, -2, 0, 6, -5, 3, -1] Reason: Mix of negative and positive numbers — evenness applies regardless of sign. All even numbers (positive or negative, including 0) should be grouped at the front.

// Input: [2, 4, 1, 3] → Output: [2, 4, 1, 3] Reason: Already sorted by parity, function should leave it valid.

// Input: [1, 3, 5, 7] → Output: [1, 3, 5, 7] Reason: All odd numbers, no change needed but still valid.

// Question 3: Two Sum

// This method returns indices of the two numbers in the array that add up to the given target.So, We use a dictionary to store numbers we've seen and their indices.

// For each number, we check if the complement (target - number) is already in the dictionary. If yes, we return the pair of indices, else no- we add the number and its index to the dictionary for future checks.

public static int[] TwoSum(int[] nums, int target)

{

try

{

// Dictionary to store each number and its index

Dictionary<int, int> map = new Dictionary<int, int>();

for (int i = 0; i < nums.Length; i++)

{

int complement = target - nums[i]; // Calculate the complement

// Check if the complement already exists in the map

if (map.ContainsKey(complement))

{

// If yes, return the pair of indices

return new int[] { map[complement], i };

}

// Otherwise, add the current number and its index to the map.This supports finding future complements

map[nums[i]] = i;

}

// If no pair is found, return an empty array

return new int[] { };

}

catch (Exception)

{

throw;

}

}

// Edge Cases:

// Input: [] → Output: [] Reason: Empty array — no elements to form a pair.

// Input: [0], target = 0 → Output: [] Reason: Single element 0 — cannot form 0 + 0 with only one index.

// Input: [0, 0], target = 0 → Output: [0, 1] Reason: 0 + 0 = 0 — valid pair using two different indices.

// Input: [3, 3], target = 6 → Output: [0, 1] Reason: Duplicate numbers — allowed if different indices.

// Input: [1, 2, 3], target = 7 → Output: [] Reason: No two numbers add up to target.

// Input: [-3, 4, 3, 90], target = 0 → Output: [0, 2] Reason: Valid pair using negative and positive.

// Question 4: Find Maximum Product of Three Numbers

// This method finds the maximum product of any three numbers in the input array.It handles both positive and negative numbers

public static int MaximumProduct(int[] nums)

{

try

{ // Validate input: at least 3 elements are required

if (nums.Length < 3)

{

Console.WriteLine("Array must contain at least three numbers.");

return -1; // Return -1 to indicate invalid input

}

Array.Sort(nums); // Sort the array in ascending order

int n = nums.Length;

// Product of the three largest numbers (last 3 in sorted array)

int product1 = nums[n - 1] \* nums[n - 2] \* nums[n - 3];

// Product of two smallest numbers (can be negative) and the largest

int product2 = nums[0] \* nums[1] \* nums[n - 1];

// Return the maximum Product

return Math.Max(product1, product2);

}

catch (Exception)

{

throw;

}

}

// Edge Cases:

// Input: [0, 0, 0] → Output: 0 // Reason: All zeros, product = 0

// Input: [0, 0] → Output: Array must contain at least three numbers.-1 // Reason: only 2 numbers are Present

// Input: [1, 2, 3, 4] → Output: 24 // Reason: Top 3 numbers are 2, 3, 4 → 2\*3\*4 = 24

// Input: [-4, -3, -2, -1, 60] → Output: 720 // Reason: -4 \* -3 = 12; 12 \* 60 = 720

// Question 5: Decimal to Binary Conversion

// This method converts a non-negative decimal number into its binary string representation.

public static string DecimalToBinary(int decimalNumber)

{

try

{

// if binary of 0 is "0"

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

string binary = "";

// Repeatedly divide the number by 2 and add the remainder to the front of the result.

while (decimalNumber > 0)

{

int remainder = decimalNumber % 2;

binary = remainder + binary; // Add to the front

decimalNumber /= 2;

}

return binary;

}

catch (Exception)

{

throw;

}

}

// Edge Cases

// Input: 0 → Output: "0" // Reason: Special case — 0 in binary is 0

// Input: 1 → Output: "1" // Reason: 1 in binary is 1

// Input: 42 → Output: "101010" // Reason: 42 in binary is 101010

// Question 6: Find Minimum in Rotated Sorted Array

// This method uses binary search to find the minimum element in a rotated sorted array.

public static int FindMin(int[] nums)

{

try

{

int left = 0;

int right = nums.Length - 1;

// Binary search loop

while (left < right)

{

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

if (nums[mid] > nums[right])

{

// Minimum is in the right half (excluding mid)

left = mid + 1;

}

else

{

// Minimum is in the left half (including mid)

right = mid;

}

}

// When left == right, we have found the minimum

return nums[left];

}

catch (Exception)

{

throw;

}

}

// Edge Cases:

// Input: [1] → Output: 1 // Reason: Only one element

// Input: [4, 5, 6, 7, 0, 1, 2] → Output: 0

// Input: [11, 13, 15, 17] → Output: 11

// Input: [2, 1] → Output: 1

// Question 7: Palindrome Number

// This method returns true if the input number is a palindrome (reads the same forward and backward)

public static bool IsPalindrome(int x)

{

try

{

// Negative numbers are not palindromes

if (x < 0) return false;

int original = x; // Store the original value for comparison later

int reversed = 0;

while (x > 0)

{

int digit = x % 10; // Extract the last digit

// check for overflow before multiplying

if (reversed > (int.MaxValue - digit) / 10)

return false;

reversed = reversed \* 10 + digit; // Build the reversed number

x /= 10; // Remove the last digit from the input number

}

// If the original number equals the reversed one, it's a palindrome

return original == reversed;

}

catch (Exception)

{

throw;

}

}

// Edge Cases:

// Input: 121 → Output: true Reason: Reversed = 121, same as input

// Input: 10 → Output: false Reason: Reversed = 01, not equal to input

// Input: -121 → Output: false Reason: Negative numbers aren't palindromes

// Input: 0 → Output: true Reason: Single-digit numbers are always palindromes

// Question 8: Fibonacci Number

// This method returns the nth Fibonacci number using an iterative approach.

// It validates the input to ensure it is between 0 and 30. Otherwise, it throws and catches an ArgumentOutOfRangeException and returns -1.

public static int Fibonacci(int n)

{

try

{

// Validate input range

// Only allow values from 0 to 30

if (n < 0 || n > 30)

throw new ArgumentOutOfRangeException("n", "Input must be between 0 and 30.");

// Fibonacci(0) = 0

if (n == 0) return 0;

// Fibonacci(1) = 1

if (n == 1) return 1;

// Initialize the first two Fibonacci numbers

int a = 0, b = 1;

// Iteratively compute Fibonacci from 2 to n

for (int i = 2; i <= n; i++)

{

int sum = a + b; // Next Fibonacci number

a = b;

b = sum;

}

return b; // Return the nth Fibonacci number

}

catch (Exception ex)

{

// Catch and log the exception, then return -1 to indicate an error

Console.WriteLine($"ArgumentOutOfRangeException: {ex.Message}");

return -1;

}

}

// Edge Cases:

// Input: -1 → Output: -1 Reason: Invalid input; less than 0 → throws and catches ArgumentOutOfRangeException

// Input: 31 → Output: -1 Reason: Invalid input; greater than 30 → throws and catches ArgumentOutOfRangeException

// Input: 0 → Output: 0 Reason: Base case, first Fibonacci number

// Input: 1 → Output: 1 Reason: Base case, second Fibonacci number

// Input: 2 → Output: 1

// Input: 30 → Output: 832040 Reason: Upper boundary 30th Fibonacci number

}

}