**PROMPTS DOCUMENT**

**Initial Solution 1:**

// Question 1: Find Missing Numbers in Array

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

{

try

{

int n = nums.Length;

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

bool[] ispresent = new bool[n + 1];

// Checking the presence of each number in the array

foreach (int num in nums)

{

if (num > 0 && num <= n)

{

ispresent[num] = true;

}

}

// Finding the missing numbers

for (int i = 1; i <= n; i++)

{

if (!ispresent[i])

{

missingNumbers.Add(i);

}

}

return missingNumbers;

}

catch (Exception)

{

throw;

}

}

**Copilot Prompt 1:**

How can I improve #FindMissingNumbers? Suggest code for any improvements

**Copilot Response:**

To improve the FindMissingNumbers method, we can focus on enhancing its performance and readability. One way to optimize the method is by using the input array itself to mark the presence of numbers, which can reduce the space complexity from O(n) to O(1). Here's the improved version of the method:

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

{

try

{

int n = nums.Length;

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

// Mark the presence of each number in the array

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

{

int index = Math.Abs(nums[i]) - 1;

if (index < n && nums[index] > 0)

{

nums[index] = -nums[index];

}

}

// Find the missing numbers

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

{

if (nums[i] > 0)

{

missingNumbers.Add(i + 1);

}

}

return missingNumbers;

}

catch (Exception)

{

throw;

}

}

**Implementation Details:**

1. Space Optimization: Instead of using an additional boolean array to track the presence of numbers, we use the input array itself. This reduces the space complexity from O(n) to O(1).
2. Marking Presence: We mark the presence of each number by negating the value at the corresponding index.
3. Finding Missing Numbers: We identify missing numbers by checking for positive values in the array, which indicates that the index+1 is missing.

**Adjustments:**

No adjustments were needed. This was applied as-is as it reduced my code’s space complexity.

**Copilot Prompt 2:**

How can I improve both Space and Time Complexity for #FindMissingNumbers

**Copilot Response:**

To improve both the space and time complexity of the FindMissingNumbers method, we can use the input array itself to mark the presence of numbers. This approach reduces the space complexity to O(1) while maintaining the time complexity at O(n). Here's the improved version of the method:

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

{

try

{

int n = nums.Length;

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

// Mark the presence of each number in the array

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

{

int index = Math.Abs(nums[i]) - 1;

if (index < n && nums[index] > 0)

{

nums[index] = -nums[index];

}

}

// Find the missing numbers

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

{

if (nums[i] > 0)

{

missingNumbers.Add(i + 1);

}

}

return missingNumbers;

}

catch (Exception)

{

throw;

}

}

**Implementation Details:**

Gave the same code suggestion again so no changes were made.

**Adjustments:**

No adjustments were needed.

**Copilot Prompt 3:**

What are some edge cases I should consider when testing the FindMissingNumbers method?

**Copilot Response:**

When testing the FindMissingNumbers method, consider the following edge cases to ensure the method handles various scenarios correctly:

1. Empty Array:
   * Input: []
   * Expected Output: []
2. Array with All Numbers Present:
   * Input: [1, 2, 3, 4, 5]
   * Expected Output: []
3. Array with All Numbers Missing:
   * Input: [6, 7, 8, 9, 10] (assuming n = 5)
   * Expected Output: [1, 2, 3, 4, 5]
4. Array with Duplicates:
   * Input: [4, 3, 2, 7, 8, 2, 3, 1]
   * Expected Output: [5, 6]
5. Array with Negative Numbers:
   * Input: [-1, -2, -3, -4, -5]
   * Expected Output: [1, 2, 3, 4, 5] (assuming n = 5)
6. Array with Numbers Greater than n:
   * Input: [1, 2, 3, 10, 11] (assuming n = 5)
   * Expected Output: [4, 5]
7. Array with Mixed Valid and Invalid Numbers:
   * Input: [1, -1, 2, 0, 3, 6] (assuming n = 5)
   * Expected Output: [4, 5]
8. Single Element Array:
   * Input: [1]
   * Expected Output: []
   * Input: [2] (assuming n = 1)
   * Expected Output: [1]
9. Array with Repeated Single Element:
   * Input: [1, 1, 1, 1]
   * Expected Output: [2, 3, 4] (assuming n = 4)
10. Large Array:
    * Input: [1, 2, 3, ..., 999999] (with one number missing)
    * Expected Output: [missing\_number]

By considering these edge cases, you can ensure that the FindMissingNumbers method is robust and handles a wide range of input scenarios correctly.

**Implementation Details:**

Tested the method manually as per the edge cases suggested.

**Adjustments:**

Adjusted the code to handle exceptions.

**Gave Question 1 with the solution to ChatGPT to verify:**

**ChatGPT Prompt:**

Is my solution correct for question 1 and handles all the reasonable corner cases? <Attached my Code>

**ChatGPT Response:**

**🔍 Summary of Edge Cases & Handling**

| **Case** | **Example Input** | **Expected Output** | **Handled?** |
| --- | --- | --- | --- |
| Standard case | {4, 3, 2, 7, 8, 2, 3, 1} | [5, 6] | ✅ |
| No missing numbers | {1, 2, 3, 4, 5} | [] | ✅ |
| All elements are the same | {3, 3, 3, 3} | [1, 2, 4] | ✅ |
| Duplicates | {1, 1} | [2] | ✅ |
| Sorted array | {1, 2, 3, 4, 5, 6} | [] | ✅ |
| Missing first or last element | {2, 3, 4, 5} | [1] | ✅ |
| Empty array | {} | [] | ✅ |
| Negative numbers | {-1, -2, -3, -4} | **Not Supported** | ❌ (Needs validation) |
| Large n | 1 to 99998, missing 56790 | [56790] | ✅ |

**🚀 Final Thoughts**

* Your **original approach** was very close, but the **handling of duplicates** needed fixing.
* The **corrected version works for all major edge cases**.
* If **negative numbers** or **invalid values** must be handled, an extra validation step is required.

**Implementation Suggested:**

Case 8: Negative Numbers (Invalid Input)

Input:

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

Expected Output: Not Supported  
✅ How It Works:

* Problem: The logic assumes all numbers are positive (ranging from 1 to n).
* Fix: Add a validation check before processing:

if (nums[i] < 1 || nums[i] > n) continue;

* This ensures that negative numbers and out-of-range values are ignored.

**Adjustments:**

Not Implemented because the question mentions *integer array nums of size n containing numbers from 1 to n*

**Initial Solution 2:**

// Question 2: Sort Array by Parity

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

{

try

{

int left = 0;

int right = nums.Length - 1;

while (left < right)

{

while (left < right && nums[left] % 2 == 0)

{

left++;

}

while (left < right && nums[right] % 2 != 0)

{

right--;

}

if (left < right)

{

(nums[left], nums[right]) = (nums[right], nums[left]);

left++;

right--;

}

}

return nums; // Ensure the method returns the sorted array

}

catch (Exception)

{

throw;

}

}

**Copilot Prompt:**

How can I improve #SortArrayByParity? Suggest code for any improvements

**Copilot Response:**

To improve the SortArrayByParity method, we can focus on the following aspects:

1. Performance: The current implementation is already efficient with a time complexity of O(n). However, we can make it more concise.
2. Readability: Simplify the code to make it more readable.
3. Error Handling: The current try-catch block is not adding much value. We can remove it to make the code cleaner.

Here is the improved version of the SortArrayByParity method:

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

{

int left = 0;

int right = nums.Length - 1;

while (left < right)

{

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

{

(nums[left], nums[right]) = (nums[right], nums[left]);

}

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

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

}

return nums;

}

**Implementation suggested:**

1. Removed the try-catch block: Since there are no specific exceptions to handle, removing it makes the code cleaner.
2. Simplified the while loop conditions: Combined the conditions to swap elements and increment/decrement pointers in a more concise manner.
3. Returned the sorted array: Added a return statement to return the sorted array.

This version maintains the same O(n) time complexity but is more concise and readable.

**Adjustments made**: Did not remove try and catch block but modified the code as per suggestion.

**Copilot Prompt 2:**

A screenshot of a computer

AI-generated content may be incorrect.

**Copilot Response:**

Here are the unit tests for these scenarios using the **Xunit** framework:

using System;

using Xunit;

namespace Assignment\_2.Tests

{

public class ProgramTests

{

[Fact]

public void SortArrayByParity\_EmptyArray\_ReturnsEmptyArray()

{

int[] nums = { };

int[] result = Program.SortArrayByParity(nums);

Assert.Equal(nums, result);

}

[Fact]

public void SortArrayByParity\_AllEvenNumbers\_ReturnsSameArray()

{

int[] nums = { 2, 4, 6, 8 };

int[] result = Program.SortArrayByParity(nums);

Assert.Equal(nums, result);

}

[Fact]

public void SortArrayByParity\_AllOddNumbers\_ReturnsSameArray()

{

int[] nums = { 1, 3, 5, 7 };

int[] result = Program.SortArrayByParity(nums);

Assert.Equal(nums, result);

}

[Fact]

public void SortArrayByParity\_MixedNumbers\_ReturnsSortedArray()

{

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

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

int[] result = Program.SortArrayByParity(nums);

Assert.Equal(expected, result);

}

[Fact]

public void SortArrayByParity\_SingleEvenNumber\_ReturnsSameArray()

{

int[] nums = { 2 };

int[] result = Program.SortArrayByParity(nums);

Assert.Equal(nums, result);

}

[Fact]

public void SortArrayByParity\_SingleOddNumber\_ReturnsSameArray()

{

int[] nums = { 1 };

int[] result = Program.SortArrayByParity(nums);

Assert.Equal(nums, result);

}

[Fact]

public void SortArrayByParity\_TwoElements\_ReturnsSortedArray()

{

int[] nums = { 1, 2 };

int[] expected = { 2, 1 };

int[] result = Program.SortArrayByParity(nums);

Assert.Equal(expected, result);

}

[Fact]

public void SortArrayByParity\_NegativeNumbers\_ReturnsSortedArray()

{

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

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

int[] result = Program.SortArrayByParity(nums);

Assert.Equal(expected, result);

}

[Fact]

public void SortArrayByParity\_DuplicateValues\_ReturnsSortedArray()

{

int[] nums = { 2, 3, 2, 3 };

int[] expected = { 2, 2, 3, 3 };

int[] result = Program.SortArrayByParity(nums);

Assert.Equal(expected, result);

}

[Fact]

public void SortArrayByParity\_LargeAndSmallIntegers\_ReturnsSortedArray()

{

int[] nums = { int.MaxValue, int.MinValue, 0, 1 };

int[] expected = { int.MinValue, 0, int.MaxValue, 1 };

int[] result = Program.SortArrayByParity(nums);

Assert.Equal(expected, result);

}

}

}

These tests cover all the reasonable corner cases and ensure that the SortArrayByParity method handles them correctly.

**Not a very helpful response so asked ChatGPT:**

**ChatGPT Prompt:** Is my code for question 2 correct and handles all reasonable corner cases? <Attached code>

**ChatGPT Response:**

Your solution is mostly correct and follows the **two-pointer approach**, which is an efficient way to sort an array by parity in **O(n) time complexity**. However, let's analyze its correctness and whether it handles all reasonable corner cases.

**⚠️ Potential Issues / Edge Cases to Consider:**

| **Corner Case** | **Does Code Handle It?** | **Fix (if needed)** |
| --- | --- | --- |
| **Empty Array ([])** | ✅ Yes, returns an empty array. | No fix needed. |
| **All Even Numbers ([2, 4, 6])** | ✅ Yes, no swaps needed, returns the same array. | No fix needed. |
| **All Odd Numbers ([1, 3, 5])** | ✅ Yes, no swaps needed, returns the same array. | No fix needed. |
| **Alternating Even and Odd ([1, 2, 3, 4])** | ✅ Yes, swaps correctly to [2, 4, 3, 1]. | No fix needed. |
| **Already Sorted ([2, 4, 1, 3])** | ✅ Yes, leaves the array unchanged. | No fix needed. |
| **Single Element ([2] or [1])** | ✅ Yes, returns the same array. | No fix needed. |
| **Large Input (1000000 elements)** | ✅ Yes, O(n) complexity makes it scalable. | No fix needed. |

**🚀 Final Verdict: Your Code is Correct!**

* It efficiently separates even and odd numbers.
* It handles all reasonable edge cases correctly.
* **No modifications are needed.** 🎯

**Implementation suggested:**

None

**Adjustments made**:

None

**Initial Response 3:**

// Question 3: Two Sum

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

{

try

{

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

{

for (int j = i + 1; j < nums.Length; j++)

{

if (nums[i] + nums[j] == target)

{

return new int[] { i, j };

}

}

}

return new int[0]; // No solution found

}

catch (Exception)

{

throw;

}

}

**Copilot Prompt:**

How can I improve #TwoSum? Suggest code for any improvements

**Copilot Response:**

To improve the TwoSum method, you can use a hash map (dictionary in C#) to achieve a time complexity of O(n) instead of the current O(n^2). This approach allows you to find the two indices in a single pass through the array.

Here is the improved code:

// Question 3: Two Sum

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

{

try

{

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

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

{

int complement = target - nums[i];

if (numDict.ContainsKey(complement))

{

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

}

if (!numDict.ContainsKey(nums[i]))

{

numDict[nums[i]] = i;

}

}

return new int[0]; // No solution found

}

catch (Exception)

{

throw;

}

}

**Implementation suggested:**

1. Dictionary Usage: Introduced a Dictionary<int, int> to store the numbers and their indices.
2. Single Pass: The method now iterates through the array only once.
3. Complement Check: For each number, it calculates the complement (target - nums[i]) and checks if it exists in the dictionary.
4. Index Storage: If the complement is not found, it stores the current number and its index in the dictionary.

This approach significantly improves the performance of the TwoSum method.

**Adjustments made**: Modified the code directly as per suggestion.

**ChatGPT Prompt:**

Is my code for question 3 correct and handles all the reasonable corner cases:<Attached code>

**ChatGPT Response:**

Yes, your code for **Question 3 (Two Sum)** is **correct** and handles **all reasonable corner cases** well. Let's break it down to be sure and see how it behaves with edge scenarios:

**🔍 Reasonable Corner Cases & Your Code’s Behavior**

| **Test Case** | **Description** | **Behavior** |
| --- | --- | --- |
| nums = [2, 7, 11, 15], target = 9 | Normal case | ✅ Returns [0,1] |
| nums = [3, 2, 4], target = 6 | Requires skipping an earlier value | ✅ Returns [1,2] |
| nums = [3, 3], target = 6 | Duplicate numbers | ✅ Returns [0,1] |
| nums = [1, 2, 3], target = 7 | No match | ✅ Returns [] |
| nums = [], target = 0 | Empty array | ✅ Returns [] |
| nums = [5], target = 5 | Single element | ✅ Returns [] |
| nums = [1, 2, 3, 4, 4], target = 8 | Duplicate potential pair | ✅ Returns [3,4] |
| nums = [0, 4, 3, 0], target = 0 | Pair of zeros | ✅ Returns [0,3] |

**Implementation suggested:**

None

**Adjustments made**:

None

**Initial Solution 4:**

public static int MaximumProduct(int[] nums)

{

try

{

Array.Sort(nums);

int n = nums.Length;

return Math.Max(nums[0] \* nums[1] \* nums[n - 1], nums[n - 1] \* nums[n - 2] \* nums[n - 3]);

}

catch (Exception)

{

throw;

}

}

**Copilot Prompt:**

How can I improve #MaximumProduct? Suggest code for any improvements

**Copilot Response:**

To improve the MaximumProduct method, we can avoid sorting the entire array, which has a time complexity of O(n log n). Instead, we can find the three largest and two smallest numbers in a single pass through the array, which has a time complexity of O(n). This approach will improve the performance of the method.

Here is the improved code:

public static int MaximumProduct(int[] nums)

{

try

{

int max1 = int.MinValue, max2 = int.MinValue, max3 = int.MinValue;

int min1 = int.MaxValue, min2 = int.MaxValue;

foreach (int num in nums)

{

if (num > max1)

{

max3 = max2;

max2 = max1;

max1 = num;

}

else if (num > max2)

{

max3 = max2;

max2 = num;

}

else if (num > max3)

{

max3 = num;

}

if (num < min1)

{

min2 = min1;

min1 = num;

}

else if (num < min2)

{

min2 = num;

}

}

return Math.Max(max1 \* max2 \* max3, min1 \* min2 \* max1);

}

catch (Exception)

{

throw;

}

}

**Implementation suggested:**

1. Avoided Sorting: Instead of sorting the array, we find the three largest and two smallest numbers in a single pass.
2. Single Pass: The method now iterates through the array only once, reducing the time complexity to O(n).

This approach ensures that the method is more efficient, especially for large arrays.

**Adjustments made**:

Modified the code directly as per suggestion.

**ChatGPT Prompt:**

Is my code correct for question 4 and handles all reasonable corner cases? <Attached code>

**ChatGPT Response:**

Yes, your code for **Question 4: Maximum Product of Three Numbers** is **correct** and efficiently handles **all reasonable corner cases**. Let’s break it down and validate it against typical scenarios and edge cases.

**🔍 Corner Case Analysis**

| **Test Case** | **Description** | **Output** | **Handled?** |
| --- | --- | --- | --- |
| [1, 2, 3] | Simple increasing | 6 | ✅ |
| [1, 2, 3, 4] | More than 3 numbers | 24 | ✅ |
| [-10, -10, 5, 2] | Negatives for higher product | 500 | ✅ |
| [-1, -2, -3, -4] | All negatives | -6 (product of largest three negatives) | ✅ |
| [0, -1, 3, 100, -70, 50] | Mixed signs and zero | 350000 | ✅ |
| [1, 2, 3, -1000, -900] | Large negatives | 2700000 | ✅ |
| [0, 0, 0] | All zeros | 0 | ✅ |
| [1, 2] | Less than 3 elements | ❌ Throws unexpected result | ❌ **(Needs fix)** |
| [int.MaxValue, int.MinValue, 1, 2, 3] | Extremes | Correct large product | ✅ |

**Implementation suggested:**

Your code assumes that the input array always has at least **three elements**. If not, it'll give incorrect results or behave unpredictably.

Add a check at the start:

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

throw new ArgumentException("Input array must contain at least three integers.");

**Adjustments made**:

Modified the code as per suggestion.

**Initial Solution 5:**

public static string DecimalToBinary(int decimalNumber)

{

try

{

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

string binary = "";

while (decimalNumber > 0)

{

binary = (decimalNumber % 2) + binary;

decimalNumber /= 2;

}

return binary;

}

catch (Exception)

{

throw;

}

}

**Copilot Prompt:**

How can I improve #DecimalToBinary? Suggest code for any improvements

**Copilot Response:**

To improve the DecimalToBinary method, we can consider the following enhancements:

1. Use StringBuilder: Using StringBuilder instead of string concatenation can improve performance, especially for larger numbers.
2. Remove try-catch block: Since there are no specific exceptions being handled, the try-catch block can be removed to simplify the code.

Here is the improved version of the DecimalToBinary method:

public static string DecimalToBinary(int decimalNumber)

{

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

var binary = new System.Text.StringBuilder();

while (decimalNumber > 0)

{

binary.Insert(0, decimalNumber % 2);

decimalNumber /= 2;

}

return binary.ToString();

}

**Implementation suggested:**

1. Replaced string concatenation with StringBuilder to improve performance.
2. Removed the unnecessary try-catch block for cleaner code.

**Adjustments made**:

Did not remove try and catch block but modified the code as per suggestion.

**ChatGPT Prompt:**

Is my solution for question 5 better and handles all reasonable corner cases <Attached code>

**ChatGPT Response:**

Yes, your solution for **Question 5: Decimal to Binary** is **correct**, efficient, and handles **all reasonable corner cases** very well. Here's a quick breakdown:

**🔍 Reasonable Corner Cases Checked:**

| **Input** | **Expected Output** | **Handled?** |
| --- | --- | --- |
| 0 | "0" | ✅ Yes |
| 1 | "1" | ✅ Yes |
| 2 | "10" | ✅ Yes |
| 255 | "11111111" | ✅ Yes |
| **Negative input** | Not handled | ⚠️ No |

**Implementation suggested:**

To fully handle **all edge cases**, including **negative integers**, consider either:

* Throwing an exception for negative inputs, **or**
* Extending the function to support **two's complement** binary representation.

**Improved Version with Error Handling for Negatives:**

public static string DecimalToBinary(int decimalNumber)

{

if (decimalNumber < 0)

throw new ArgumentException("Negative numbers are not supported.");

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

var binary = new System.Text.StringBuilder();

while (decimalNumber > 0)

{

binary.Insert(0, decimalNumber % 2);

decimalNumber /= 2;

}

return binary.ToString();

}

Or you could extend it to return signed binary (e.g., two’s complement), if required.

**Adjustments made**:

Did not remove try and catch block but modified the code as per suggestion.

**Initial Solution 6:**

public static int FindMin(int[] nums)

{

try

{

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 nums[left];

}

catch (Exception)

{

throw;

}

}

**Copilot Prompt:**

How can I improve #FindMin? Suggest code for any improvements

**Copilot Response:**

To improve the FindMin method, we can focus on the following aspects:

1. Code Readability: Add comments to explain the logic.
2. Exception Handling: Provide more specific exception handling.
3. Edge Case Handling: Ensure the method handles edge cases properly.

Here is the improved version of the FindMin method:

// Question 6: Find Minimum in Rotated Sorted Array

public static int FindMin(int[] nums)

{

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

{

throw new ArgumentException("Input array cannot be null or empty.");

}

try

{

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

// Binary search to find the minimum element

while (left < right)

{

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

// If mid element is greater than the rightmost element,

// the minimum is in the right half

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

{

left = mid + 1;

}

// Otherwise, the minimum is in the left half (including mid)

else

{

right = mid;

}

}

// The left pointer will point to the minimum element

return nums[left];

}

catch (Exception ex)

{

throw new InvalidOperationException("An error occurred while finding the minimum element.", ex);

}

}

**Implementation suggested:**

1. Null or Empty Check: Added a check to handle null or empty input arrays.
2. Comments: Added comments to explain the logic of the binary search.
3. Specific Exception Handling: Changed the generic exception handling to provide more context in case of an error.

These improvements enhance the readability, robustness, and maintainability of the FindMin method.

**Adjustments**:

Modified the code directly as per suggestion.

**ChatGPT Prompt:**

Is my solution for question 6 correct and handles all reasonable corner cases: <Attached Code>

**ChatGPT Response:**

Yes, your solution for **Question 6: Find Minimum in Rotated Sorted Array** is **correct**, clean, and handles **all reasonable corner cases** very well.

**🔍 Corner Cases Covered:**

| **Input** | **Handled?** | **Notes** |
| --- | --- | --- |
| [3,4,5,1,2] | ✅ Yes | Rotated sorted array |
| [1,2,3,4,5] | ✅ Yes | Not rotated |
| [2,1] | ✅ Yes | Small input, rotated |
| [1] | ✅ Yes | Single element |
| [] | ✅ Yes | Throws ArgumentException |
| null | ✅ Yes | Throws ArgumentException |
| [2,2,2,0,1] | ❌ No | This has **duplicates**, your code assumes all elements are **distinct** |

**Implementation suggested:**

The current implementation does not handle duplicates in the array. You'd need an extra condition:

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

{

right--; // Skip the duplicate

}

**Adjustments**:

Modified into my existing code as per suggestion.

**Initial Solution 7:**

public static bool IsPalindrome(int x)

{

try

{

if (x < 0) return false;

int reversed = 0, original = x;

while (x > 0)

{

reversed = reversed \* 10 + x % 10;

x /= 10;

}

return original == reversed;

}

catch (Exception)

{

throw;

}

}

**Copilot Prompt:**

How can I improve #IsPalindrome? Suggest code for any improvements

**Copilot Response:**

To improve the IsPalindrome method, we can focus on the following aspects:

1. Avoid integer overflow: The current implementation may cause an overflow when reversing the integer. We can handle this by using a different approach.
2. Simplify logic: We can simplify the logic by comparing the first half of the number with the reversed second half.

Here is the improved version of the IsPalindrome method:

public static bool IsPalindrome(int x)

{

try

{

// Negative numbers are not palindromes

if (x < 0) return false;

// Numbers ending in 0 (except 0 itself) are not palindromes

if (x != 0 && x % 10 == 0) return false;

int reversed = 0;

while (x > reversed)

{

reversed = reversed \* 10 + x % 10;

x /= 10;

}

// Check if the original number is equal to the reversed number

// or if the original number without the last digit is equal to the reversed number

return x == reversed || x == reversed / 10;

}

catch (Exception)

{

throw;

}

}

**Implementation suggested:**

1. Avoid integer overflow: Instead of reversing the entire number, we reverse only half of it and compare it with the other half.
2. Simplify logic: The logic is simplified by comparing the first half of the number with the reversed second half. This avoids the need to handle large numbers and potential overflow issues.

This approach ensures that the method is more efficient and less prone to errors.

**Adjustments**:

Modified code as per suggestion.

**ChatGPT Prompt:**

Is my solution to question 7 correct and handle all the reasonable corner cases? <Attached code>

**ChatGPT Response:**

Your solution for **Question 7: Check if an Integer is a Palindrome** is ✅ **correct** and efficiently handles all **reasonable corner cases**. Here's a breakdown of what your code does

**Corner Case Handling**

| **Case** | **Description** | **Handled?** |
| --- | --- | --- |
| x < 0 | Negative numbers can't be palindromes | ✅ Yes |
| x == 0 | 0 is a valid palindrome | ✅ Yes |
| x % 10 == 0 | Numbers ending in zero (like 10, 100) can't be palindromes unless the number itself is zero | ✅ Yes |
| Even-digit numbers | e.g., 1221 — check if x == reversed | ✅ Yes |
| Odd-digit numbers | e.g., 12321 — check if x == reversed / 10 | ✅ Yes |
| Large numbers | Avoids overflow by reversing only half | ✅ Yes |
| Single digit numbers | All are palindromes | ✅ Yes |

**Implementation suggested:**

None

**Adjustments made**:

None

**Initial solution 8:**

public static int Fibonacci(int n)

{

try

{

if (n == 0) return 0;

if (n == 1) return 1;

int prev1 = 0, prev2 = 1, result = 0;

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

{

result = prev1 + prev2;

prev1 = prev2;

prev2 = result;

}

return result;

}

catch (Exception)

{

throw;

}

}

**Copilot Prompt:**

How can I improve #Fibonacci? Suggest code for any improvements

**Copilot Response:**

To improve the Fibonacci method, we can consider the following enhancements:

1. Avoid Exception Handling for Control Flow: The current implementation uses a try-catch block, which is unnecessary for this simple method. Removing it will improve performance and readability.
2. Use a More Descriptive Variable Name: Rename result to current for better readability.
3. Optimize for Edge Cases: Handle edge cases directly without additional checks inside the loop.

Here is the improved version of the Fibonacci method:

// Question 8: Fibonacci Number

public static int Fibonacci(int n)

{

if (n < 0)

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

if (n == 0) return 0;

if (n == 1) return 1;

int prev1 = 0, prev2 = 1, current = 0;

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

{

current = prev1 + prev2;

prev1 = prev2;

prev2 = current;

}

return current;

}

**Implementation suggested:**

1. Removed the try-catch block: Exception handling is not necessary for this simple method.
2. Added input validation: Added a check to ensure the input is a non-negative integer.
3. Renamed result to current: Improved variable naming for better readability.

These changes make the method more efficient and easier to understand.

**Adjustments made**:

Added the input validation but did not reduce three but only two variables prev1 and prev2.

Also did not remove try and catch blocks.

**ChatGPT Prompt:**

Is my solution to question 8 correct and handles all reasonable corner cases? <Attached Code>

**ChatGPT Response:**

Yes, your solution for **Question 8: Fibonacci Number** is ✅ **correct** and handles all **reasonable corner cases**. Let’s break it down to confirm its accuracy and robustness:

**Corner Case Handling**

| **Case** | **Description** | **Handled?** |
| --- | --- | --- |
| n < 0 | Throws an appropriate exception | ✅ Yes |
| n = 0 | Should return 0 | ✅ Yes |
| n = 1 | Should return 1 | ✅ Yes |
| n = 2 and above | Should return correct result | ✅ Yes |
| Large n (e.g., 40) | Efficient due to iterative approach | ✅ Yes |
| Max n near int limit | Will work correctly until result overflows int | ✅ **Partially** (see note below) |

**Implementation suggested:**

For n > 46, the result **overflows** the int type since Fibonacci(47) = **2,971,215,073** which exceeds int.MaxValue (2,147,483,647).

If you want to support larger values, consider changing the return type to long:

public static long Fibonacci(int n)

**Adjustments made**:

Did not implement the suggestion as n has the constraint 0 <= n <= 30

At last, updated all the methods catch blocks.

**Copilot Prompt:**

Suggest any improvements for my code.

**Copilot Response:**

Here are some improvements for your code:

1. Use var for local variable declarations: This can make the code more concise and readable.
2. Remove redundant null checks: Some methods have redundant null checks that can be removed.
3. Simplify exception handling: Instead of catching general exceptions, catch specific exceptions where possible.
4. Use StringBuilder for string concatenation: This is more efficient for multiple string concatenations.
5. Remove unnecessary comments: Some comments are redundant and can be removed to make the code cleaner.

**Implementation:**

Not implemented because it would disturb the assignment template.

**Adjustments made**:

None, because did not implement.