ISM6225 Assignment 2

Alvaro Montoya Ruiz

Invalid Date

# Problem 1

For Solving problem 1 ,I first used an implementation of the counting sort methodology , in which a temporary array is made to evalaute if the number is present on the original array I used input validation and I made used of the second for loop to identify which numbers are missing in the temporary array, later return them on the list

try {   
 int n = nums.Length;  
 List<int> missingNumbers = new List<int>();   
 bool[] present = new bool[n + 1];  
 // Ensure all numbers are positive and within the valid range  
 for (int i = 0; i < nums.Length; i++)  
 {  
 if (nums[i] <= 0 || nums[i] > n)  
 {  
 throw;  
 }  
 present[nums[i]] = true;  
 }  
 // Find missing numbers  
 for (int i = 1; i <= n; i++)  
 {  
 if (!present[i])  
 {  
 missingNumbers.Add(i);  
 }  
 }  
  
 return missingNumbers;  
 }  
 catch (Exception)  
 {  
 throw;  
 }

I made use of copilot to help me identify any edge cases to consider and also provide guidelines on how to improve my solution time complexity run. For the prompt I made the following

## Prompt

How can I improve my input validation to consider all edge cases?

int n = nums.Length;  
 List<int> missingNumbers = new List<int>();   
 bool[] present = new bool[n + 1];  
 // Ensure all numbers are positive and within the valid range  
 for (int i = 0; i < nums.Length; i++)  
 {  
 if (nums[i] <= 0 || nums[i] > n)  
 {  
 throw new ArgumentException("Array must only contain positive integers within the range 1 to n.");  
 }  
 present[nums[i]] = true;  
 }

## Response:

1. Handle empty arrays.
2. Handle arrays with duplicate values.
3. Ensure the array contains only positive integers within the valid range.

Also, you may want to include messages as follow : throw new ArgumentException(“Array must only contain positive integers within the range 1 to n.”);

## Prompt

How can I improve my solution time complexity for the following code?

// Find missing numbers  
 for (int i = 1; i <= n; i++)  
 {  
 if (!present[i])  
 {  
 missingNumbers.Add(i);  
 }  
 }  
  
 return missingNumbers;

## Response

The current time complexity of the code snippet is O(n), which is already optimal for finding missing numbers in an array. However, if you want to improve the overall efficiency of the FindMissingNumbers method, you can use a different approach that leverages a hash set to achieve the same result with a similar time complexity but potentially better performance in practice.

## Implementation fo AI solution

try  
{ //Copilot Provided me with a better solution to handle the edge case of null or empty array.  
 if (nums == null || nums.Length == 0)  
 {  
 return new List<int>(); // Return an empty list if the input array is null or empty  
 }  
  
 int n = nums.Length;  
 List<int> missingNumbers = new List<int>();  
 bool[] present = new bool[n + 1];  
  
 // Edge case Identified, array could have either neagtive numbers of numbers greater than n  
 for (int i = 0; i < nums.Length; i++)  
 {  
 if (nums[i] <= 0 || nums[i] > n)  
 {  
 // Copilot improved the code for throwing an ArgumentException. Original solution was throwing an Exception.   
 throw new ArgumentException("Array must only contain positive integers within the range 1 to n.");  
 }  
 present[nums[i]] = true;  
 }  
  
 // Find missing numbers  
 for (int i = 1; i <= n; i++)  
 {  
 if (!present[i])  
 {  
 missingNumbers.Add(i);  
 }  
 }  
  
 return missingNumbers;  
}  
catch (Exception ex)  
{  
 throw new Exception("An error occurred while finding missing numbers.", ex);  
}

## Adjustments

The adjustments I made with the implementation were:

1. returning the empty list. Implementation only used a simple return statement and since function needs to return a list I needed to modified it
2. The ArgumentExpection String for the Catch part I needed to write it.

# Problem 2

For Solving problem 2 ,I first used an implementation of a search and swap algorithm. It looks for even numbers on the left of the array and odd numbers from the end of the array. After, it performs the swap if it identified that an odd number was before an even number in the array

try  
 {  
 int left = 0, right = nums.Length - 1;  
 while (left < right)  
 {  
 // If left is even, move to the next element  
 if (nums[left] % 2 == 0)  
 {  
 left++;  
 }  
 // If right is odd, move to the previous element  
 else if (nums[right] % 2 == 1)  
 {  
 right--;  
 }  
  
 // Swap when left is odd and right is even  
 else  
 {  
 (nums[left], nums[right]) = (nums[right], nums[left]);  
 left++;  
 right--;  
 }  
 }  
 return nums;  
 }  
 catch (Exception)  
 {  
 throw;  
 }

When I runned the code, the order of the numbers was changed. Since we needed to maintain the order as they were presented on the array, I asked copilot for help on implementing the correct aproach.

I made use of copilot to help me identify any edge cases to consider and also provide guidelines on how to improve my solution time complexity run. For the prompt I made the following

## Prompt

How can I improve my input validation to consider all edge cases?

## Response

To improve the input validation and handle all edge cases, you can add checks for null or empty arrays, and ensure that the array contains only integers.

## Prompt

How can I improve my code so that it ensures the order of the numbers remains from the original input?

## Response

To ensure the order of the numbers is respected from your input, follow these steps: 1. First, create a temporary array that will store the result array. 2. Traverse your original array and make a first pass to find all even numbers and add them to the temporary array. 3. Later, do a second pass to add all odd numbers to the temporary array

## Implementation

try  
{ //Copilot Provided me with a better solution to handle the edge case of null or empty array.  
 if (nums == null || nums.Length == 0)  
 {  
 return new int[0]; // Return an empty array if the input array is null or empty  
 }  
  
 int[] result = new int[nums.Length];  
 int index = 0;  
  
 // Copilot approach is now to look for even numbers first and then odd numbers.  
 for (int i = 0; i < nums.Length; i++)  
 {  
 if (nums[i] % 2 == 0)  
 {  
 result[index++] = nums[i];  
 }  
 }  
  
 // Second pass: add all odd numbers  
 for (int i = 0; i < nums.Length; i++)  
 {  
 if (nums[i] % 2 != 0)  
 {  
 result[index++] = nums[i];  
 }  
 }  
  
 return result;  
}  
catch (Exception ex)  
{  
 //Copilot improved the code for throwing an exception with a more descriptive message.  
 throw new Exception("An error occurred while sorting the array by parity.", ex);  
}

## Adjustments

The adjustments I made with the implementation were:

1. include the input validation for null scenarios or empty
2. include an index variable to keep track at what position to beginning storing the odd numbers into result array
3. The ArgumentExpection String for the Catch part I needed to write it.

# Problem 3

For Solving problem 3 ,I first used an implementation of nested arrays to search for 2 numbers whose addition will be equal to the target.

try  
{  
 // Handle edge case: array is null or empty  
 if (nums == null)  
 {  
 throw;  
 }  
 // Iterate through the array to find the two numbers that add up to the target  
 for (int i = 0; i < nums.Length - 1; i++)  
 {  
 for (int j = i + 1; j < nums.Length; j++)  
 {  
 if (nums[i] + nums[j] == target)  
 {  
 return new int[] { i, j };  
 }  
 }  
 }  
  
}

Since the assignment looks for a O (n) solution I had to use advance data structures such as either lists or Hash maps (Dictionaries). As Dictionaries work better for searching operations since they use the key and value concept. I decided to modify my first implementation.

I made use of copilot to help me identify any edge cases to consider and also provide guidelines on how to improve my solution time complexity run. For the prompt I made the following

## Prompt

How can I improve my input validation to consider all edge cases?

// Handle edge case: array is null or empty  
 if (nums == null)  
 {  
 throw;  
 }

## Response

if (nums == null || nums.Length < 2)  
 {  
 throw new ArgumentException("Input array must contain at least two elements.");  
 }

## Implementation

try  
{ // copilot provided me with a better solution to handle the edge case , array has less than 2 elements or null or empty array.  
 if (nums == null || nums.Length < 2)  
 {  
 throw new ArgumentException("Input array must contain at least two elements.");  
 }  
  
 Dictionary<int, int> dict1 = new Dictionary<int, int>();  
  
 for (int i = 0; i < nums.Length; i++)  
 {  
 int number2 = target - nums[i];  
  
 if (!dict1.ContainsKey(nums[i]))  
 {  
 dict1[nums[i]] = i;  
 }  
  
  
  
   
 if (dict1.ContainsKey(number2))  
 {  
 return new int[] { dict1[number2], i };  
 }  
  
   
 }  
 // if no solution found, throw an exception  
 throw new InvalidOperationException("No two sum solution found.");  
}  
catch (Exception ex)  
{  
 throw new Exception("An error occurred while finding the two sum solution.", ex);  
}

## Adjustments

1. include the input validation for arrays with less than 2 elements.
2. made use of ContainsKey in order to do a linear search for the second number.
3. provide if statement to add numbers to the dictionary if there are not found on it.
4. The ArgumentExpection String for the Catch part I needed to write it.

# Problem 4

For solving problem 4 ,I first used an implementation of nested arrays to sort the array. For this multiplication of 3 numbers there could be the case 2 negatives numbers and 1 positive number might lead a larger result than all positives.

try  
 {  
   
 // Implementing Bubble Sort to sort the array  
 for (int i = 0; i < nums.Length - 1; i++)  
 {  
 for (int j = 0; j < nums.Length - i - 1; j++)  
 {  
 if (nums[j] > nums[j + 1])  
 {  
 // Swap nums[j] and nums[j + 1]  
 int temp = nums[j];  
 nums[j] = nums[j + 1];  
 nums[j + 1] = temp;  
 }  
 }  
 }  
  
 int n = nums.Length;  
 int product1 = nums[n - 1] \* nums[n - 2] \* nums[n - 3];  
 int product2 = nums[0] \* nums[1] \* nums[n - 1];  
  
 return Math.Max(product1, product2);  
 }  
 catch (Exception)  
 {  
 throw;  
 }

Since the assignment looks for a O (n) solution I had to use advance data structures such as either lists, Hash maps (Dictionaries). or other approches to lead linear time complexity. I asked copilot for assistance on it and also to check how could I improved my input validation to check all posible edge cases

## Prompt

How can I improve my input validation to consider all edge cases?

## Response

Your array must contain at least 3 numbers to perform the multiplication. I would also check for null values on the array.

if (nums == null || nums.Length < 3)  
 throw new ArgumentException("Array must contain at least three numbers.");

## Prompt

How can I perform this multiplication , accounting that 2 negatives values with a positive value could lead to a max value. I need to maintain a linear time complexity for it.

## Response

You will need to create 3 max variables for the case your multiplication involves the highest 3 positive numbers, and you will also need to create 2 min variables in case your multiplication involves 2 negative numbers and 1 positive number. After updating the variables, you will have to decide which multiplication has a higher value, either max1 \* max2 \* max3 or max1 \* min1 \* min2.

int max1 = int.MinValue, max2 = int.MinValue, max3 = int.MinValue;  
int min1 = int.MaxValue, min2 = int.MaxValue;  
  
foreach (int num in nums)  
{  
 // Update max1, max2, max3  
 if (num > max1)  
 {  
 max3 = max2;  
 max2 = max1;  
 max1 = num;  
 }  
 else if (num > max2)  
 {  
 max3 = max2;  
 max2 = num;  
 }  
 else if (num > max3)  
 {  
 max3 = num;  
 }  
  
 // Update min1, min2  
 if (num < min1)  
 {  
 min2 = min1;  
 min1 = num;  
 }  
 else if (num < min2)  
 {  
 min2 = num;  
 }  
}

## Implementation

try  
{  
 if (nums == null || nums.Length < 3)  
 throw new ArgumentException("Array must contain at least three numbers.");  
 //Copilot solution offers a O(n) solution to find the maximum product of 3 numbers. by using MinValue and MaxValue  
 //other approaches would need to sort the array and then find the maximum product of 3 numbers. and in some edge cases also consider the product of 2 negative numbers and 1 positive number.  
 //Copilot solution create max and min variables for testing the edge cases of negative numbers.  
 int max1 = int.MinValue, max2 = int.MinValue, max3 = int.MinValue;  
 int min1 = int.MaxValue, min2 = int.MaxValue;  
  
 foreach (int num in nums)  
 {  
 // Update max1, max2, max3  
 if (num > max1)  
 {  
 max3 = max2;  
 max2 = max1;  
 max1 = num;  
 }  
 else if (num > max2)  
 {  
 max3 = max2;  
 max2 = num;  
 }  
 else if (num > max3)  
 {  
 max3 = num;  
 }  
  
 // Update min1, min2  
 if (num < min1)  
 {  
 min2 = min1;  
 min1 = num;  
 }  
 else if (num < min2)  
 {  
 min2 = num;  
 }  
 }  
  
 //Copilot solution offers the option of having a product of 2 negative numbers and 1 positive number.  
 // and a product of 3 positive numbers.  
 int product1 = max1 \* max2 \* max3;  
 int product2 = min1 \* min2 \* max1;  
  
 return Math.Max(product1, product2);  
}  
catch (Exception)  
{  
 throw;  
}

## Adjustments

1. include the input validation for arrays with less than 3 elements.
2. create product1 and product2
3. use the Math.Max() to evaluate which has a higher value

# Problem 5

For Solving problem 5 ,I first used an implementation where only positive inputs where converted to binary representation.

try  
 {  
 if (decimalNumber == 0)  
 return "0";  
  
 if (decimalNumber < 0)  
 throw new ArgumentException("Negative numbers are not supported.");  
  
 string binary = "";  
 while (decimalNumber > 0)  
 {  
 binary = (decimalNumber % 2) + binary;  
 decimalNumber /= 2;  
 }  
  
 return binary;  
 }  
 catch (Exception)  
 {  
 throw   
 }

Since the assignment ask to approach all edge cases. I asked copilot on the steps to implement a converter using twos complement for negative numbers

## Prompt

Improve my solution to performs a correct representation of negative decimals into binary following the steps :

1. Identify the decimal to be either negative or positive.
2. if negative convert the decimal to its absolute value to binary
3. invert the bits (flip 0s to 1 and 1s to 0s)
4. add 1 to the result (1 in the least significant bit)
5. Perform checking using the carry bit

## Response

bool isNegative = decimalNumber < 0;  
 int absValue = Math.Abs(decimalNumber);  
  
 // Convert absolute value to binary since either positive or negative numbers are represented in binary  
 string binary = "";  
 while (absValue > 0)  
 {  
 binary = (absValue % 2) + binary;  
 absValue /= 2;  
 }  
  
  
 int bits = binary.Length;  
  
 // If the number is negative, add bit for the sign (two's complement)  
 if (isNegative)  
 {  
 bits++; // Add 1 more bit for the two's complement representation  
 }  
  
 // Copilot proposed to used the padding to add the missing bits to the binary number.  
 binary = binary.PadLeft(bits, '0');  
  
 if (isNegative)  
 {  
 // Invert bits  
 char[] invertedBinary = binary.ToCharArray();  
 for (int i = 0; i < invertedBinary.Length; i++)  
 {  
 invertedBinary[i] = invertedBinary[i] == '0' ? '1' : '0';  
 }  
  
 // Add 1 to the inverted binary (two's complement)  
 bool carry = true;  
 for (int i = invertedBinary.Length - 1; i >= 0; i--)  
 {  
 if (carry)  
 {  
 if (invertedBinary[i] == '0')  
 {  
 invertedBinary[i] = '1';  
 carry = false;  
 }  
 else  
 {  
 invertedBinary[i] = '0';  
 }  
 }  
 }  
 }

## Implementation

try  
 {  
 if (decimalNumber == 0)  
 return "0";  
  
 bool isNegative = decimalNumber < 0;  
 int absValue = Math.Abs(decimalNumber);  
  
 // Convert absolute value to binary since either positive or negative numbers are represented in binary  
 string binary = "";  
 while (absValue > 0)  
 {  
 binary = (absValue % 2) + binary;  
 absValue /= 2;  
 }  
   
 int bits = binary.Length;  
  
 // If the number is negative, add bit for the sign (two's complement)  
 if (isNegative)  
 {  
 bits++; // Add 1 more bit for the two's complement representation  
 }  
 // Copilot proposed to used the padding to add the missing bits to the binary number.  
 binary = binary.PadLeft(bits, '0');  
  
 if (isNegative)  
 {  
 // Invert bits  
 char[] invertedBinary = binary.ToCharArray();  
 for (int i = 0; i < invertedBinary.Length; i++)  
 {  
 invertedBinary[i] = invertedBinary[i] == '0' ? '1' : '0';  
 }  
  
 // Add 1 to the inverted binary (two's complement)  
 bool carry = true;  
 for (int i = invertedBinary.Length - 1; i >= 0; i--)  
 {  
 if (carry)  
 {  
 if (invertedBinary[i] == '0')  
 {  
 invertedBinary[i] = '1';  
 carry = false;  
 }  
 else  
 {  
 invertedBinary[i] = '0';  
 }  
 }  
 }  
  
 binary = new string(invertedBinary);  
 }  
  
 return binary;  
 }  
 catch (Exception ex)  
 {  
 //Copilot proposed to throw an exception with a more descriptive message.  
 throw new Exception("An error occurred while converting decimal to binary.", ex);  
 }

## Adjustments

1. include the input validation for number = 0
2. convert invertedBinary (char array) into Binary (string)
3. The ArgumentExpection String for the Catch part I needed to write it.

# Problem 6

For Solving problem 6 ,I made used of a binary search implementation , which leads to a time complexity of O(n).

try  
 {  
 //Using binary search approach to find the minimum element in the rotated array.  
 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 Convert.ToInt32(nums[left]);  
 }  
 catch (Exception)  
 {  
 throw;  
 }

Since the assignment ask to approach all edge cases. I asked copilot on the steps to account for all possible edge cases. Considering that null and empty arrays could be a possibility

## Prompt

Provide me steps to check if my int array is either empty or has null values.

## Response

// Handle edge cases  
 if (nums == null || nums.Length == 0)  
 {  
 throw new ArgumentException("Input array must not be null or empty.");  
 }  
 if (nums.Length == 1)  
 {  
 return nums[0];  
 }

## Implementation

try  
 {  
 // Handle edge cases  
 if (nums == null || nums.Length == 0)  
 {  
 throw new ArgumentException("Input array must not be null or empty.");  
 }  
 if (nums.Length == 1)  
 {  
 return nums[0];  
 }  
  
 // Using binary search approach to find the minimum element in the rotated sorted array.  
 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 ex)  
 {  
 throw new Exception("An error occurred while finding the minimum element in the rotated sorted array.", ex);  
 }

## Adjustments

1. include the input validation on the implementation
2. The ArgumentExpection String for the Catch part I needed to write it.

# Problem 7

For Solving problem 7 ,I used a standard implementation for searching for palindrome. For strings we will try to get each character. However, for integers I needed to extract the complete value of each individual digit representation on the decimal system.

try  
 {  
 if (decimalNumber == 0)  
 return "0";  
  
 if (decimalNumber < 0)  
 throw new ArgumentException("Negative numbers are not supported.");  
  
 string binary = "";  
 while (decimalNumber > 0)  
 {  
 binary = (decimalNumber % 2) + binary;  
 decimalNumber /= 2;  
 }  
  
 return binary;  
 }  
 catch (Exception)  
 {  
 throw   
 }

Since the assignment ask to approach all edge cases. I asked copilot to verify if negative numbers could be considered palindromes

## Prompt

Can negative integers be considered palindromes?

## Response

Negative integers cannot be palindromes because the negative sign - at the beginning of the number makes it impossible for the number to read the same backward as forward. Therefore, the current implementation of returning false for negative numbers is correct.

## Implementation

// Use standard approach to check if the number is a palindrome.  
 try  
 {  
 if (x < 0) return false;   
  
 int original = x, reversed = 0;  
  
 while (x > 0)  
 {   
 //Compute digit of number by performing mod 10  
 int digit = x % 10;  
  
 //build the reversed number by multiplying by 10 and adding the digit  
 reversed = reversed \* 10 + digit;  
 x /= 10;  
 }  
  
 return original == reversed;  
 }  
 catch (Exception)  
 {  
 throw;  
 }

## Adjustments

No adjustements required for this implementation

# Problem 8

For Solving problem 8 ,I used an standard implementation of fibonacci series recursion. I looked to implement a dictionary ( hash table ) to reduce time complexity.However, this will need the modification of the function int Fibonacci(int n)

No Copilot promting was used since the question included the variable constraint of 0 to 30.

## Implementation

// Standard finobacci series calculation using recursion.  
 try  
 { // Using the constraint indicated on the instructions document  
 if (n < 0 || n > 30)  
 throw new ArgumentOutOfRangeException("n needs to be in range 0 <= n <= 30");  
  
 // Base cases  
 if (n == 0) return 0;  
 if (n == 1) return 1;  
  
 //Recursive case   
 return Fibonacci(n - 1) + Fibonacci(n - 2);  
 }  
 catch (Exception ex)  
 {  
 throw new Exception("Error at time of Fibonacci calculation", ex);  
 }