# Q.1

## Prompt

For function FindMissingNumbers, 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.

## Response

Here is the implementation for the [FindMissingNumbers](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) function. It uses a marking approach to identify the missing numbers efficiently:

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

{

    try

    {

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

        // Mark the indices corresponding to the numbers in the array

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

        {

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

            if (nums[index] > 0)

            {

                nums[index] = -nums[index];

            }

        }

        // Find the indices that are still positive, which correspond to missing numbers

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

        {

            if (nums[i] > 0)

            {

                missingNumbers.Add(i + 1);

            }

        }

        return missingNumbers;

    }

    catch (Exception)

    {

        throw;

    }

}

**Explanation:**

1. **Marking Indices**: Iterate through the array and for each number [nums[i]](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html), mark the index Math.Abs(nums[i]) - 1 as negative to indicate that the number i + 1 exists in the array.
2. **Finding Missing Numbers**: After marking, any index that remains positive corresponds to a missing number.
3. **Time Complexity**: O(n) since we iterate through the array twice.
4. **Space Complexity**: O(1) as the marking is done in-place.

## Implementation and Adjustments

Added the code to the program.cs file.

## Prompt

Now I want to discuss the edge cases. Would the current algorithm work well with these scenarios:

1. Empty array

2. Array has duplicates

3. Array is given out of bounds. for example, if a member of the array is larger than n or is negative.

## Response

Let's analyze the edge cases for the [FindMissingNumbers](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) function:

**1. Empty Array**

* **Scenario**: If the input array [nums](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) is empty ([nums.Length == 0](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)).
* **Behavior**: The function will skip both loops since there are no elements to process. The [missingNumbers](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) list will remain empty and will be returned as is.
* **Result**: The function works correctly for this case and returns an empty list.

**2. Array with Duplicates**

* **Scenario**: If the array contains duplicate numbers (e.g., {4, 3, 2, 7, 8, 2, 3, 1}).
* **Behavior**: The algorithm handles duplicates correctly because it uses the absolute value of each number to mark indices. Even if a number appears multiple times, the corresponding index will only be marked once.
* **Result**: The function works correctly for this case and identifies the missing numbers accurately.

**3. Array with Out-of-Bounds Values**

* **Scenario**: If the array contains numbers that are:
  + Greater than [n](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) (e.g., {1, 2, 3, 10} for [n = 4](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)).
  + Negative numbers (e.g., {1, -2, 3, 4}).
* **Behavior**:
  + **Greater than**[n](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html): The algorithm will attempt to access an index that is out of bounds ([nums[index]](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) where [index >= nums.Length](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)), causing an IndexOutOfRangeException.
  + **Negative numbers**: The algorithm uses [Math.Abs(nums[i]) - 1](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) to calculate the index. If the number is negative, it will still calculate an index, but this behavior is undefined for values that are not in the range [1, n].
* **Result**: The function does **not** handle out-of-bounds values correctly and will throw an exception or produce incorrect results.

**Suggested Fix for Out-of-Bounds Values**

To handle out-of-bounds values, you can add a validation step at the beginning of the function to ensure all numbers are within the range [1, n].

        // Validate input: Ensure all numbers are within the range [1, nums.Length]

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

        {

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

            {

                throw new ArgumentException("Array contains out-of-bounds values.");

            }

        }

## Implementation and Adjustments

The two first scenarios work well and there is no need for anything to change. However the third scenario was not considered. So I had to add the suggested code by Copilot to the beginning of the function. This function checks if all array members are within the range needed and then continues to solve the problem. This ensures termination of the program if the input array has out-of-bounds members.

However the code given by copilot throws an exception and terminates the program which is not intended. I want to run the whole program and get the output so I can submit the HW. That is why I changed the code to:

                // Validate input: Ensure all numbers are within the range [1, nums.Length]

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

                {

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

                    {

                        // print the error message

                        Console.WriteLine($"Error: Array contains out-of-bounds value {nums[i]}.");

                        return missingNumbers;

                    }

                }

This ensures that if the array is invalidated an error message will appear, an empty array is returned and shown in the terminal and the program will continue to run for other functions.

## Prompt

Now let’s discuss efficiency. Are there any more efficient algorithm in regards to time complexity and storage complexity that can be used?

## Response

The current implementation of [FindMissingNumbers](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) is already efficient in terms of **time complexity** and **space complexity**:

* **Time Complexity**: **O(n)** — The algorithm iterates through the array twice, which is optimal for this problem.
* **Space Complexity**: **O(1)** — The algorithm modifies the input array in place and uses only a small amount of additional space for the result list.

Given these complexities, the current algorithm is already the most efficient in terms of both time and space. However, let's explore alternative approaches and their trade-offs:

**1. Using a HashSet**

* **Algorithm**:
  + Add all numbers from 1 to [n](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) to a HashSet.
  + Iterate through the input array and remove each number from the HashSet.
  + The remaining numbers in the HashSet are the missing numbers.
* **Time Complexity**: **O(n)** — Adding and removing elements from a HashSet is O(1) on average.
* **Space Complexity**: **O(n)** — The HashSet requires additional space proportional to the size of the input array.
* **Trade-Off**: This approach is simpler to implement but uses more memory compared to the in-place marking approach.

**2. Using a Counting Array**

* **Algorithm**:
  + Create a counting array of size [n](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) initialized to 0.
  + Iterate through the input array and increment the count for each number.
  + Iterate through the counting array to find indices with a count of 0, which correspond to the missing numbers.
* **Time Complexity**: **O(n)** — Two iterations through the array.
* **Space Complexity**: **O(n)** — The counting array requires additional space proportional to the size of the input array.
* **Trade-Off**: This approach is straightforward but uses more memory compared to the in-place marking approach.

**3. Mathematical Approach**

* **Algorithm**:
  + Calculate the sum of numbers from 1 to [n](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) using the formula sum = n \* (n + 1) / 2.
  + Calculate the sum of the input array.
  + The difference between the two sums gives the sum of the missing numbers.
  + This approach works only if there is exactly **one missing number** and no duplicates.
* **Time Complexity**: **O(n)** — One iteration through the array to calculate the sum.
* **Space Complexity**: **O(1)** — No additional space is used.
* **Trade-Off**: This approach is very efficient but limited to specific cases (e.g., one missing number, no duplicates).

**Conclusion:**

The **in-place marking approach** used in the current implementation is the most efficient in terms of both time and space for the general case. It avoids the additional memory overhead of a HashSet or counting array and works for arrays with duplicates. There is no need to switch to another algorithm unless the problem constraints change (e.g., if the input array cannot be modified).

## Implementation and Adjustments

Since this is the most efficient algorithm no adjustments are needed.

# Q.2

## Prompt

For function SortArrayByParity, 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.

## Response

Here is the implementation for the [SortArrayByParity](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) function. It rearranges the array in-place by using two pointers:

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

{

    try

    {

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

        while (left < right)

        {

            // If the left pointer is at an even number, move it forward

            if (nums[left] % 2 == 0)

            {

                left++;

            }

            // If the right pointer is at an odd number, move it backward

            else if (nums[right] % 2 != 0)

            {

                right--;

            }

            // Swap if left is odd and right is even

            else

            {

                int temp = nums[left];

                nums[left] = nums[right];

                nums[right] = temp;

                left++;

                right--;

            }

        }

        return nums;

    }

    catch (Exception)

    {

        throw;

    }

}

**Explanation:**

1. **Two Pointers**:
   * Use two pointers: left starts at the beginning of the array, and right starts at the end.
   * Move left forward if it points to an even number.
   * Move right backward if it points to an odd number.
   * Swap the values at left and right if left points to an odd number and right points to an even number.
2. **In-Place**:
   * The function modifies the array in-place without using extra space.
3. **Time Complexity**:
   * **O(n)**: Each element is processed at most once as the two pointers traverse the array.
4. **Space Complexity**:
   * **O(1)**: No additional space is used apart from a temporary variable for swapping.

## Implementation and Adjustments

Function added and tested.

## Prompt

What edge cases can arise in this problem and see if the current algorithm covers them all. if not update the code so it covers all edge cases.

## Response

**Edge Cases for**[SortArrayByParity](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)

1. **Empty Array**:
   * If the input array is empty ([nums.Length == 0](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)), the algorithm should return an empty array without errors.
2. **Array with All Even Numbers**:
   * If the array contains only even numbers (e.g., {2, 4, 6}), the algorithm should return the array as is.
3. **Array with All Odd Numbers**:
   * If the array contains only odd numbers (e.g., {1, 3, 5}), the algorithm should return the array as is.
4. **Array with One Element**:
   * If the array contains only one element (e.g., {2} or {1}), the algorithm should return the array as is.
5. **Array with Mixed Even and Odd Numbers**:
   * The algorithm should correctly rearrange the array so that all even numbers appear before all odd numbers.

**Current Algorithm Coverage**

The current algorithm handles most of these cases correctly:

* It works for empty arrays, arrays with all even or all odd numbers, and arrays with mixed even and odd numbers.
* However, it does not explicitly check for an empty array or handle it gracefully. While the algorithm will not throw an error, adding an explicit check for clarity and robustness is a good practice.

**Updated Code to Cover All Edge Cases**

Here is the updated [SortArrayByParity](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) function with explicit handling for edge cases:

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

{

    try

    {

        // Handle edge case: Empty array

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

        {

            return nums; // Return the empty array as is

        }

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

        while (left < right)

        {

            // If the left pointer is at an even number, move it forward

            if (nums[left] % 2 == 0)

            {

                left++;

            }

            // If the right pointer is at an odd number, move it backward

            else if (nums[right] % 2 != 0)

            {

                right--;

            }

            // Swap if left is odd and right is even

            else

            {

                int temp = nums[left];

                nums[left] = nums[right];

                nums[right] = temp;

                left++;

                right--;

            }

        }

        return nums;

    }

    catch (Exception ex)

    {

        throw new Exception("Error in SortArrayByParity: " + ex.Message);

    }

}

**Explanation of Changes**

1. **Empty Array Check**:
   * Added a check at the beginning of the function to return the array as is if it is empty ([nums == null || nums.Length == 0](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)).
2. **Robustness**:
   * The algorithm already handles arrays with all even or all odd numbers, as well as arrays with one element, without requiring additional changes.

**Testing the Edge Cases**

Here are some test cases to verify the updated function:

// Test Case 1: Empty array

int[] nums1 = { };

Console.WriteLine(string.Join(",", SortArrayByParity(nums1))); // Output: ""

// Test Case 2: All even numbers

int[] nums2 = { 2, 4, 6 };

Console.WriteLine(string.Join(",", SortArrayByParity(nums2))); // Output: "2,4,6"

// Test Case 3: All odd numbers

int[] nums3 = { 1, 3, 5 };

Console.WriteLine(string.Join(",", SortArrayByParity(nums3))); // Output: "1,3,5"

// Test Case 4: One element (even)

int[] nums4 = { 2 };

Console.WriteLine(string.Join(",", SortArrayByParity(nums4))); // Output: "2"

// Test Case 5: One element (odd)

int[] nums5 = { 1 };

Console.WriteLine(string.Join(",", SortArrayByParity(nums5))); // Output: "1"

// Test Case 6: Mixed even and odd numbers

int[] nums6 = { 3, 1, 2, 4 };

Console.WriteLine(string.Join(",", SortArrayByParity(nums6))); // Output: "4,2,1,3" (or any valid even-odd arrangement)

**Conclusion**

The updated code explicitly handles all edge cases, making it more robust and easier to understand. The time complexity remains **O(n)**, and the space complexity remains **O(1)**.

## Implementation and Adjustments

An if condition is added to handle empty arrays.

## Prompt

what if array has members other than integers? would it handle it well? It is better for it to throw an error saying that the numbers must all be integers only.

## Response

The current implementation of [SortArrayByParity](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) assumes that the input array contains only integers. If the array contains non-integer values (e.g., strings, floats, or nulls), the algorithm will throw a runtime error because operations like % (modulus) are not defined for non-integer types.

To handle this scenario gracefully, we can add a validation step to ensure that all elements in the array are integers. If any non-integer value is found, the function will throw an exception with a clear error message.

**Updated Code**

Here is the updated [SortArrayByParity](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) function with validation for integer-only arrays:

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

{

    try

    {

        // Handle edge case: Empty array

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

        {

            return nums; // Return the empty array as is

        }

        // Validate input: Ensure all elements are integers

        foreach (var num in nums)

        {

            if (num.GetType() != typeof(int))

            {

                throw new ArgumentException("Array must contain integers only.");

            }

        }

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

        while (left < right)

        {

            // If the left pointer is at an even number, move it forward

            if (nums[left] % 2 == 0)

            {

                left++;

            }

            // If the right pointer is at an odd number, move it backward

            else if (nums[right] % 2 != 0)

            {

                right--;

            }

            // Swap if left is odd and right is even

            else

            {

                int temp = nums[left];

                nums[left] = nums[right];

                nums[right] = temp;

                left++;

                right--;

            }

        }

        return nums;

    }

    catch (Exception ex)

    {

        throw new Exception("Error in SortArrayByParity: " + ex.Message);

    }

}

**Explanation of Changes**

1. **Validation for Integer-Only Arrays**:
   * Added a foreach loop to check if every element in the array is of type int.
   * If any element is not an integer, an [ArgumentException](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) is thrown with the message: "Array must contain integers only."
2. **Graceful Error Handling**:
   * If the input array contains invalid elements, the function will throw a meaningful error instead of failing silently or causing undefined behavior.

## Implementation and Adjustments

Added a for loop to validate all members of the array and make sure they are all integers before making any changes. However it will throw an exception and will terminate the program which we don’t intend to do. I removed the throw added a print command and then returned an empty array. The code:

                // Validate input: Ensure all elements are integers

                foreach (var num in nums)

                {

                    if (num.GetType() != typeof(int))

                    {

                        // print the error message

                        Console.WriteLine($"Error: Array contains non-integer value {num}.");

                        return new int[] { };

                    }

                }

# Q.3

## Prompt

for function TwoSum, 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

Here is the implementation for the [TwoSum](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) function:

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

{

    try

    {

        // Create a dictionary to store the numbers and their indices

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

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

        {

            // Calculate the complement needed to reach the target

            int complement = target - nums[i];

            // Check if the complement exists in the dictionary

            if (numIndices.ContainsKey(complement))

            {

                // Return the indices of the two numbers

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

            }

            // Add the current number and its index to the dictionary

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

            {

                numIndices[nums[i]] = i;

            }

        }

        // If no solution is found, throw an exception

        throw new ArgumentException("No two sum solution exists.");

    }

    catch (Exception ex)

    {

        throw new Exception("Error in TwoSum: " + ex.Message);

    }

}

**Explanation:**

1. **Dictionary for Lookup**:
   * A dictionary (numIndices) is used to store numbers as keys and their indices as values. This allows for O(1) lookup time.
2. **Complement Calculation**:
   * For each number in the array, calculate the complement ([target - nums[i]](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)) needed to reach the target sum.
3. **Check for Complement**:
   * If the complement exists in the dictionary, return the indices of the current number and the complement.
4. **Add to Dictionary**:
   * If the complement is not found, add the current number and its index to the dictionary for future lookups.
5. **Exception Handling**:
   * If no solution is found, an exception is thrown with a meaningful error message.

**Time and Space Complexity:**

* **Time Complexity**: **O(n)** — The array is traversed once, and dictionary operations (lookup and insertion) are O(1).
* **Space Complexity**: **O(n)** — The dictionary stores up to [n](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) elements in the worst case.

**Example Usage:**

int[] nums3 = { 2, 7, 11, 15 };

int target = 9;

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

Console.WriteLine(string.Join(",", indices)); // Output: 0,1

**Edge Cases:**

1. **No Solution**:
   * If no two numbers add up to the target, the function throws an exception.
   * Example: [nums = {1, 2, 3}, target = 10](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
2. **Multiple Solutions**:
   * The function returns the first valid pair it finds.
   * Example: [nums = {3, 3, 4}, target = 6](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) returns [0, 1].
3. **Negative Numbers**:
   * The function works correctly with negative numbers.
   * Example: [nums = {-3, 4, 3, 90}, target = 0](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) returns [0, 2].
4. **Single Element or Empty Array**:
   * If the array has fewer than two elements, no solution exists, and an exception is thrown.

## Implementation and Adjustments

Added the code and tested. The code can handle many edge cases like No solution, multiple solutions, negative numbers given, and less than two members given. The Copilot code would throw an exception if no solution was found which I changed it to return an empty array and continue running the program for other functions.

## Prompt

Can you write the code to test all these edge cases?

## Response

Here is the test code to verify all the edge cases for the [SortArrayByParity](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) and [TwoSum](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) functions:

using System;

namespace Assignment\_2

{

    class Tests

    {

        static void Main(string[] args)

        {

           // Test cases for TwoSum

            Console.WriteLine("\nTesting TwoSum:");

            // Test Case 1: Valid input with a solution

            int[] nums8 = { 2, 7, 11, 15 };

            int target1 = 9;

            Console.WriteLine(string.Join(",", Program.TwoSum(nums8, target1))); // Output: "0,1"

            // Test Case 2: No solution

            try

            {

                int[] nums9 = { 1, 2, 3 };

                int target2 = 10;

                Console.WriteLine(string.Join(",", Program.TwoSum(nums9, target2)));

            }

            catch (Exception ex)

            {

                Console.WriteLine(ex.Message); // Output: "No two sum solution exists."

            }

            // Test Case 3: Multiple solutions (should return the first valid pair)

            int[] nums10 = { 3, 3, 4 };

            int target3 = 6;

            Console.WriteLine(string.Join(",", Program.TwoSum(nums10, target3))); // Output: "0,1"

            // Test Case 4: Negative numbers

            int[] nums11 = { -3, 4, 3, 90 };

            int target4 = 0;

            Console.WriteLine(string.Join(",", Program.TwoSum(nums11, target4))); // Output: "0,2"

            // Test Case 5: Single element (should throw an exception)

            try

            {

                int[] nums12 = { 1 };

                int target5 = 1;

                Console.WriteLine(string.Join(",", Program.TwoSum(nums12, target5)));

            }

            catch (Exception ex)

            {

                Console.WriteLine(ex.Message); // Output: "No two sum solution exists."

            }

            // Test Case 6: Empty array (should throw an exception)

            try

            {

                int[] nums13 = { };

                int target6 = 0;

                Console.WriteLine(string.Join(",", Program.TwoSum(nums13, target6)));

            }

            catch (Exception ex)

            {

                Console.WriteLine(ex.Message); // Output: "No two sum solution exists."

            }

        }

    }

}

**Explanation of Test Cases:**

**For**[SortArrayByParity](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)**:**

1. **Empty Array**: Tests if the function handles an empty array gracefully.
2. **All Even Numbers**: Ensures the function returns the array as is.
3. **All Odd Numbers**: Ensures the function returns the array as is.
4. **Single Element**: Tests both even and odd single-element arrays.
5. **Mixed Numbers**: Verifies that even numbers are moved to the front.
6. **Non-Integer Values**: Ensures the function throws an exception for invalid input.

**For**[TwoSum](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)**:**

1. **Valid Input**: Tests a case where a solution exists.
2. **No Solution**: Ensures the function throws an exception when no solution exists.
3. **Multiple Solutions**: Verifies that the first valid pair is returned.
4. **Negative Numbers**: Tests the function with negative numbers.
5. **Single Element**: Ensures the function throws an exception for insufficient input.
6. **Empty Array**: Ensures the function throws an exception for an empty array.

## Implementation and Adjustments

Implemented all edge cases and tested to make sure they all work correctly.

# Q.4

## Prompt

for function MaximumProduct Given an integer array nums, find three numbers whose product is the maximum and return the product.

## Response

Here is the implementation for the [MaximumProduct](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) function:

public static int MaximumProduct(int[] nums)

{

    try

    {

        // Sort the array

        Array.Sort(nums);

        // The maximum product can be either:

        // 1. The product of the three largest numbers

        // 2. The product of the two smallest numbers (negative) and the largest number

        int n = nums.Length;

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

        return maxProduct;

    }

    catch (Exception ex)

    {

        throw new Exception("Error in MaximumProduct: " + ex.Message);

    }

}

public static int MaximumProduct(int[] nums)

{

try

{

// Sort the array

Array.Sort(nums);

// The maximum product can be either:

// 1. The product of the three largest numbers

// 2. The product of the two smallest numbers (negative) and the largest number

int n = nums.Length;

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

return maxProduct;

}

catch (Exception ex)

{

throw new Exception("Error in MaximumProduct: " + ex.Message);

}

}  
…

## Implementation and Adjustments

Copilot also considered negative numbers which changes the algorithm and it is actually not what the description of the questions says. In previous questions including negative numbers as input would have not changed the algorithm comparing to the case when no negative numbers were considered. But here it changes the algorithm. That is why I again will ask Copilot to not to consider any negative numbers given and return an empty valiw with an error message that the array contains negative numbers and not positive integers.

## Prompt

Consider only positive integers in this problem.

## Response

        // Validate input: Ensure all numbers are positive integers

        foreach (var num in nums)

        {

            if (num <= 0)

            {

                throw new ArgumentException("Array must contain only positive integers.");

            }

        }

        // Sort the array

        Array.Sort(nums);

        // The maximum product is the product of the three largest numbers

        int n = nums.Length;

        if (n < 3)

        {

            throw new ArgumentException("Array must contain at least three numbers.");

        }

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

        return maxProduct;

## Implementation and Adjustments

The only problem with this function is that it throws an exception for arrays with negative numbers and arrays with less than three numbers. I just changed them to return 0 and an error message regarding those.

# Q.5

## Prompt

for function DecimalToBinary Write a function that converts a decimal number to its binary equivalent.

## Response

Here is the implementation for the [DecimalToBinary](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) function:

public static string DecimalToBinary(int decimalNumber)

{

    try

    {

        if (decimalNumber < 0)

        {

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

        }

        // Handle the special case for 0

        if (decimalNumber == 0)

        {

            return "0";

        }

        string binary = string.Empty;

        // Convert decimal to binary

        while (decimalNumber > 0)

        {

            binary = (decimalNumber % 2) + binary;

            decimalNumber /= 2;

        }

        return binary;

    }

    catch (Exception ex)

    {

        throw new Exception("Error in DecimalToBinary: " + ex.Message);

    }

}

**Explanation:**

1. **Validation**:
   * Ensure the input is a non-negative integer. If the input is negative, throw an exception with a clear error message.
2. **Special Case for Zero**:
   * If the input is 0, return "0" directly.
3. **Conversion Logic**:
   * Use a loop to repeatedly divide the number by 2 and prepend the remainder (0 or 1) to the binary string.
   * Continue until the number becomes 0.
4. **Time Complexity**:
   * **O(log n)**: The number of iterations is proportional to the number of bits in the binary representation of the input.
5. **Space Complexity**:
   * **O(log n)**: The binary string grows with the number of bits in the binary representation.

**Example Usage:**

int decimalNumber = 42;

string binary = DecimalToBinary(decimalNumber);

Console.WriteLine(binary); // Output: "101010"

**Edge Cases:**

1. **Input is Zero**:
   * Example: [decimalNumber = 0](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
   * Output: "0".
2. **Input is Negative**:
   * Example: [decimalNumber = -5](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
   * The function should throw an exception: "Input must be a non-negative integer.".

## Implementation and Adjustments

Again I changed it so it wont throw an exception when input is negative.

                if (decimalNumber < 0)

                {

                    // print the error message and return a z=ero string

                    Console.WriteLine("Error: Input must be a non-negative integer.");

                    return "0";

                }

It is important to pay attention to the return type. Zero must be a string so it must be inside quotes.

I also added another edge case myself when the number is large to see how it reacts. For the number 1234567890 the solution was pretty fast.

# Q.6

## Prompt

for the function FindMin Given a sorted array that has been rotated, find the minimum element.

## Response

Here is the implementation for the [FindMin](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) function to find the minimum element in a rotated sorted array:

public static int FindMin(int[] nums)

{

    try

    {

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

        {

            throw new ArgumentException("Array must not be null or empty.");

        }

        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;

            }

        }

        // At the end of the loop, left == right, pointing to the minimum element

        return nums[left];

    }

    catch (Exception ex)

    {

        throw new Exception("Error in FindMin: " + ex.Message);

    }

}

**Explanation:**

1. **Binary Search**:
   * The array is rotated, so the minimum element is the "pivot" point where the rotation occurs.
   * Use binary search to efficiently find the minimum element.
2. **Logic**:
   * Compare the middle element ([nums[mid]](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)) with the rightmost element ([nums[right]](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)):
     + If [nums[mid] > nums[right]](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html), the minimum is in the right half.
     + Otherwise, the minimum is in the left half (including mid).
3. **Termination**:
   * The loop continues until [left == right](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html), at which point [nums[left]](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) is the minimum element.
4. **Time Complexity**:
   * **O(log n)**: Binary search reduces the search space by half in each iteration.
5. **Space Complexity**:
   * **O(1)**: No additional space is used.

**Example Usage:**

int[] nums = { 3, 4, 5, 1, 2 };

int minElement = FindMin(nums);

Console.WriteLine(minElement); // Output: 1

**Edge Cases:**

1. **Array is Not Rotated**:
   * Example: [nums = {1, 2, 3, 4, 5}](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
   * Output: 1.
2. **Array with One Element**:
   * Example: [nums = {10}](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
   * Output: 10.
3. **Array with Two Elements**:
   * Example: [nums = {2, 1}](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
   * Output: 1.
4. **Array is Null or Empty**:
   * Example: [nums = null](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) or [nums = {}](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
   * The function should throw an exception: "Array must not be null or empty.".

## Implementation and Adjustments

Added and tested. However the algorithm in my mind is to sort the array first and choose the first element as the minimum. However I will now ask Copilot to see if this is feasible in terms of time and space complexity.

## Prompt

Discuss how time and space complexity will be affected if we first sort the array and then find the minimum by reportign the first element.

## Response

If we first sort the array and then find the minimum by reporting the first element, the **time complexity** and **space complexity** will be affected as follows:

**Time Complexity**

1. **Sorting the Array**:
   * Sorting the array takes **O(n log n)** time using efficient sorting algorithms like QuickSort or MergeSort.
   * This is significantly slower than the **O(log n)** time complexity of the binary search approach used in the current [FindMin](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) implementation.
2. **Finding the Minimum**:
   * After sorting, accessing the first element of the array takes **O(1)** time.
3. **Overall Time Complexity**:
   * Sorting dominates the time complexity, so the overall time complexity becomes **O(n log n)**.

**Comparison**:

* + Current binary search approach: **O(log n)**.
  + Sorting-based approach: **O(n log n)**.

Sorting the array is much less efficient, especially for large arrays.

**Space Complexity**

1. **Sorting the Array**:
   * Sorting the array in-place (e.g., using QuickSort) requires **O(1)** additional space.
   * However, some sorting algorithms like MergeSort require **O(n)** additional space for temporary arrays.
2. **Finding the Minimum**:
   * Accessing the first element does not require additional space (**O(1)**).
3. **Overall Space Complexity**:
   * If sorting is done in-place, the space complexity is **O(1)**.
   * If sorting requires additional memory (e.g., MergeSort), the space complexity becomes **O(n)**.

**Comparison**:

* + Current binary search approach: **O(1)** space.
  + Sorting-based approach: **O(1)** (in-place sorting) or **O(n)** (non-in-place sorting).

Sorting may require more space depending on the algorithm used.

**Conclusion**

* **Binary Search Approach**:
  + **Time Complexity**: **O(log n)**.
  + **Space Complexity**: **O(1)**.
  + This is the most efficient approach for finding the minimum in a rotated sorted array.
* **Sorting-Based Approach**:
  + **Time Complexity**: **O(n log n)**.
  + **Space Complexity**: **O(1)** (in-place sorting) or **O(n)** (non-in-place sorting).
  + Sorting the array is unnecessary and inefficient for this problem.

**Recommendation**

The current binary search approach is optimal for finding the minimum in a rotated sorted array. Sorting the array first is not recommended, as it significantly increases the time complexity without providing any additional benefit.

## Implementation and Adjustments

Since the algorithm is using binary search it is the most efficient algorithm until today and there are no more efficient algorithms.

However to handle more edge cases I also considered checking for negative members inside the array and made it to return 0 and show an error message if this happens. This way we will see the error and the program will still continue to run for other questions.

# Q.7

## Prompt

for function IsPalindrome Given an integer x, return true if x is a palindrome, and false otherwise.

## Response

Here is the implementation for the [IsPalindrome](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) function:

public static bool IsPalindrome(int x)

{

    try

    {

        // Negative numbers are not palindromes

        if (x < 0)

        {

            return false;

        }

        // Convert the number to a string

        string str = x.ToString();

        // Use two pointers to check if the string is a palindrome

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

        while (left < right)

        {

            if (str[left] != str[right])

            {

                return false;

            }

            left++;

            right--;

        }

        return true;

    }

    catch (Exception ex)

    {

        throw new Exception("Error in IsPalindrome: " + ex.Message);

    }

}

**Explanation:**

1. **Negative Numbers**:
   * Negative numbers are not palindromes because of the - sign.
2. **String Conversion**:
   * Convert the integer to a string to easily compare characters.
3. **Two-Pointer Technique**:
   * Use two pointers ([left](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) and [right](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)) to compare characters from the beginning and end of the string.
   * If any pair of characters does not match, return false.
   * If all characters match, return true.
4. **Time Complexity**:
   * **O(n)**: The string is traversed once, where [n](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) is the number of digits in the integer.
5. **Space Complexity**:
   * **O(n)**: The string representation of the number requires additional space.

**Example Usage:**

int palindromeNumber = 121;

bool isPalindrome = IsPalindrome(palindromeNumber);

Console.WriteLine(isPalindrome); // Output: true

**Edge Cases:**

1. **Negative Numbers**:
   * Example: [x = -121](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
   * Output: false.
2. **Single Digit Numbers**:
   * Example: [x = 7](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
   * Output: true.
3. **Non-Palindromic Numbers**:
   * Example: [x = 123](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
   * Output: false.
4. **Large Palindromic Numbers**:
   * Example: [x = 123454321](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
   * Output: true.

## Implementation and Adjustments

I added the function. However when I tried to run it for a very large number:

            int largePalindromeNumber = 1234567890987654321;

            bool isPalindrome6 = IsPalindrome(largePalindromeNumber);

            Console.WriteLine(isPalindrome6);

I found out that I get an error:

Cannot implicitly convert type 'long' to 'int'. An explicit conversion exists (are you missing a cast?)[CS0266](https://msdn.microsoft.com/query/roslyn.query?appId=roslyn&k=k(CS0266))

So next I will ask Copilot to handle this case.

## Prompt

When I use very large numbers I would get this error:

Cannot implicitly convert type 'long' to 'int'. An explicit conversion exists (are you missing a cast?)CS0266

How would you handle that and say if this number is palindrome or not?

## Response

The issue arises because very large numbers exceed the range of the int type in C#. To handle this, you can use the long type, which has a larger range, or even the BigInteger type from the [System.Numerics](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) namespace for arbitrarily large numbers. Here's how you can modify the [IsPalindrome](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) function to handle very large numbers:

**Updated**[IsPalindrome](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)**Function**

using System.Numerics; // Required for BigInteger

public static bool IsPalindrome(BigInteger x)

{

    try

    {

        // Negative numbers are not palindromes

        if (x < 0)

        {

            return false;

        }

        // Convert the number to a string

        string str = x.ToString();

        // Use two pointers to check if the string is a palindrome

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

        while (left < right)

        {

            if (str[left] != str[right])

            {

                return false;

            }

            left++;

            right--;

        }

        return true;

    }

    catch (Exception ex)

    {

        throw new Exception("Error in IsPalindrome: " + ex.Message);

    }

}

**Explanation of Changes:**

1. **Use BigInteger**:
   * The BigInteger type from the [System.Numerics](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) namespace can handle numbers of arbitrary size, making it suitable for very large numbers.
2. **String Conversion**:
   * The number is converted to a string using [x.ToString()](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html), which works seamlessly with BigInteger.
3. **Two-Pointer Technique**:
   * The logic for checking if the number is a palindrome remains the same.

**Example Usage:**

using System.Numerics;

BigInteger largePalindromeNumber = BigInteger.Parse("1234567890987654321");

bool isPalindrome = IsPalindrome(largePalindromeNumber);

Console.WriteLine(isPalindrome); // Output: true

BigInteger largeNonPalindromeNumber = BigInteger.Parse("1234567890987654322");

bool isPalindrome2 = IsPalindrome(largeNonPalindromeNumber);

Console.WriteLine(isPalindrome2); // Output: false

**Edge Cases:**

1. **Negative Numbers**:
   * Example: [x = -1234567890987654321](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
   * Output: false.
2. **Single Digit Numbers**:
   * Example: [x = 7](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
   * Output: true.
3. **Very Large Palindromic Numbers**:
   * Example: [x = 1234567890987654321](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
   * Output: true.
4. **Very Large Non-Palindromic Numbers**:
   * Example: [x = 1234567890987654322](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
   * Output: false.

**Notes:**

* If you don't need to handle arbitrarily large numbers, you can use the long type instead of BigInteger. However, long has a maximum value of 9,223,372,036,854,775,807, so it won't work for numbers larger than this.
* To use BigInteger, ensure you include the [System.Numerics](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) namespace in your project.

## Implementation and Adjustments

I used BitInteger and it worked like a charm. I also had to add

using System.Numerics; // Required for BigInteger

To the first section to the program.cs file.

# Q.8

## Prompt

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) and write the function for Fibonacci()

Constraints:

• 0 <= n <= 30

## Response

Here is the implementation for the [Fibonacci](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) function:

public static int Fibonacci(int n)

{

    try

    {

        // Validate input: Ensure n is within the constraints

        if (n < 0 || n > 30)

        {

            throw new ArgumentException("Input must be between 0 and 30.");

        }

        // Base cases

        if (n == 0) return 0;

        if (n == 1) return 1;

        // Iterative approach to calculate Fibonacci numbers

        int a = 0, b = 1;

        for (int i = 2; i <= n; i++)

        {

            int temp = a + b;

            a = b;

            b = temp;

        }

        return b;

    }

    catch (Exception ex)

    {

        throw new Exception("Error in Fibonacci: " + ex.Message);

    }

}

**Explanation:**

1. **Validation**:
   * Ensure the input [n](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) is within the constraints 0 <= n <= 30. If not, throw an exception with a clear error message.
2. **Base Cases**:
   * If [n == 0](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html), return 0.
   * If [n == 1](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html), return 1.
3. **Iterative Approach**:
   * Use two variables (a and b) to store the last two Fibonacci numbers.
   * Iterate from 2 to [n](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html), updating the variables to calculate the next Fibonacci number.
4. **Time Complexity**:
   * **O(n)**: The loop runs [n - 1](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) times.
5. **Space Complexity**:
   * **O(1)**: Only a constant amount of space is used.

**Example Usage:**

int n = 10;

int fibonacciNumber = Fibonacci(n);

Console.WriteLine(fibonacciNumber); // Output: 55

**Edge Cases:**

1. **n = 0**:
   * Input: [n = 0](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
   * Output: 0.
2. **n = 1**:
   * Input: [n = 1](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
   * Output: 1.
3. **n = 30**:
   * Input: [n = 30](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
   * Output: 832040.
4. **n < 0 or n > 30**:
   * Input: [n = -1](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) or [n = 31](vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
   * The function should throw an exception: "Input must be between 0 and 30.".

## Implementation and Adjustments

Implemented and all edge cases are considered here.