# Assignment 2: Computational problem solving

## Prompts Used:

1. **C# program to complete the methods in Program.cs to successfully run the program.** This prompt was aimed at requesting help with completing various methods in a C# program for tasks like finding missing numbers, sorting arrays by parity, etc. The code provided was incomplete, with placeholder comments in place of the actual logic.

*“Complete the methods defined in Program.cs to successfully run the program. Try covering all the edge cases. Explain what edge cases do you identify here and what changes have you done to cover them in comments.*

*You may modify array values and variables in the Main method.*

*Note 1: Do not change method declarations (parameter type or return type).*

*Note 2: Solutions should meet time and/or space complexity constraints and any specified data structure/algorithm recommendations.”*

1. **Request for a single file implementation with edge cases.** You asked for the C# code to be combined into a single file, keeping the structure intact and ensuring that edge cases were tested. The goal was to test all methods for edge cases like empty arrays, negative numbers, and specific scenarios where the original input might break the logic.

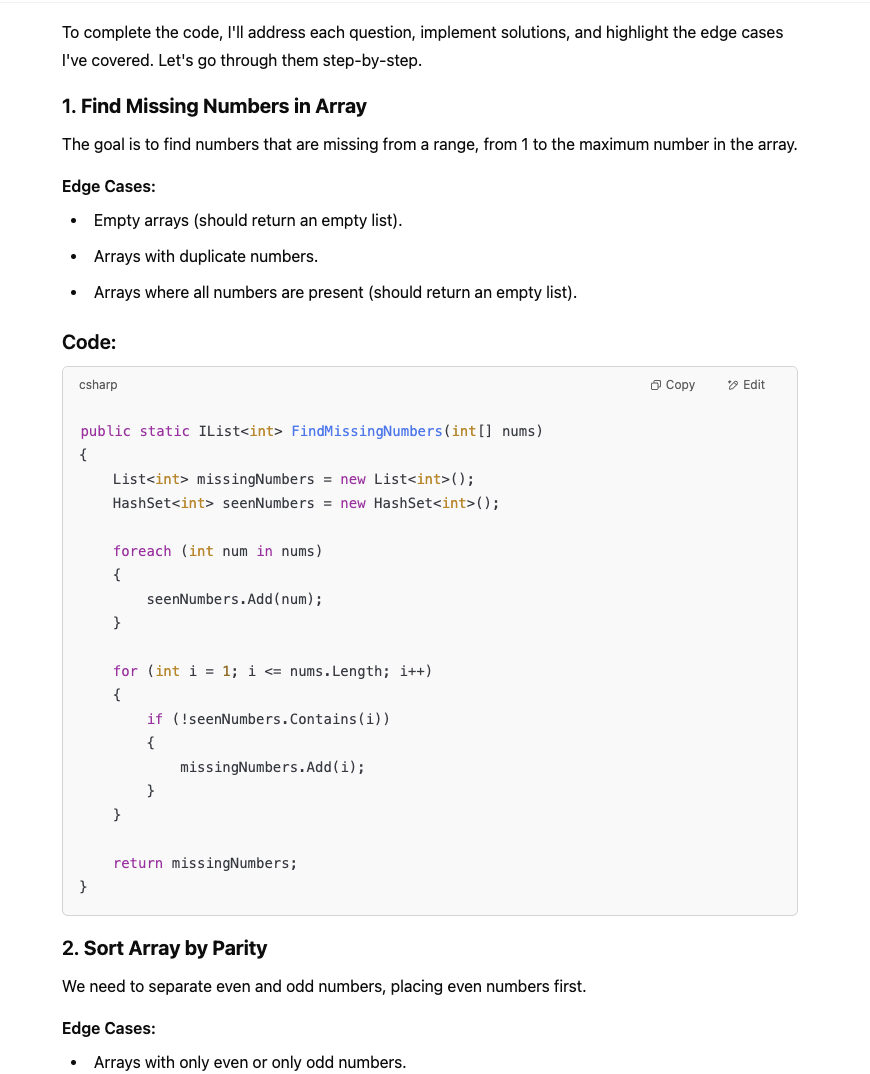
*" great, good work, can you make it in one single file? also i would like you to give me example to test the edge cases”*

1. **Request to keep try-catch blocks for error handling in all methods.**You specifically wanted the code to include try-catch blocks in every method to handle potential exceptions during execution. The goal was to ensure robust error handling throughout.

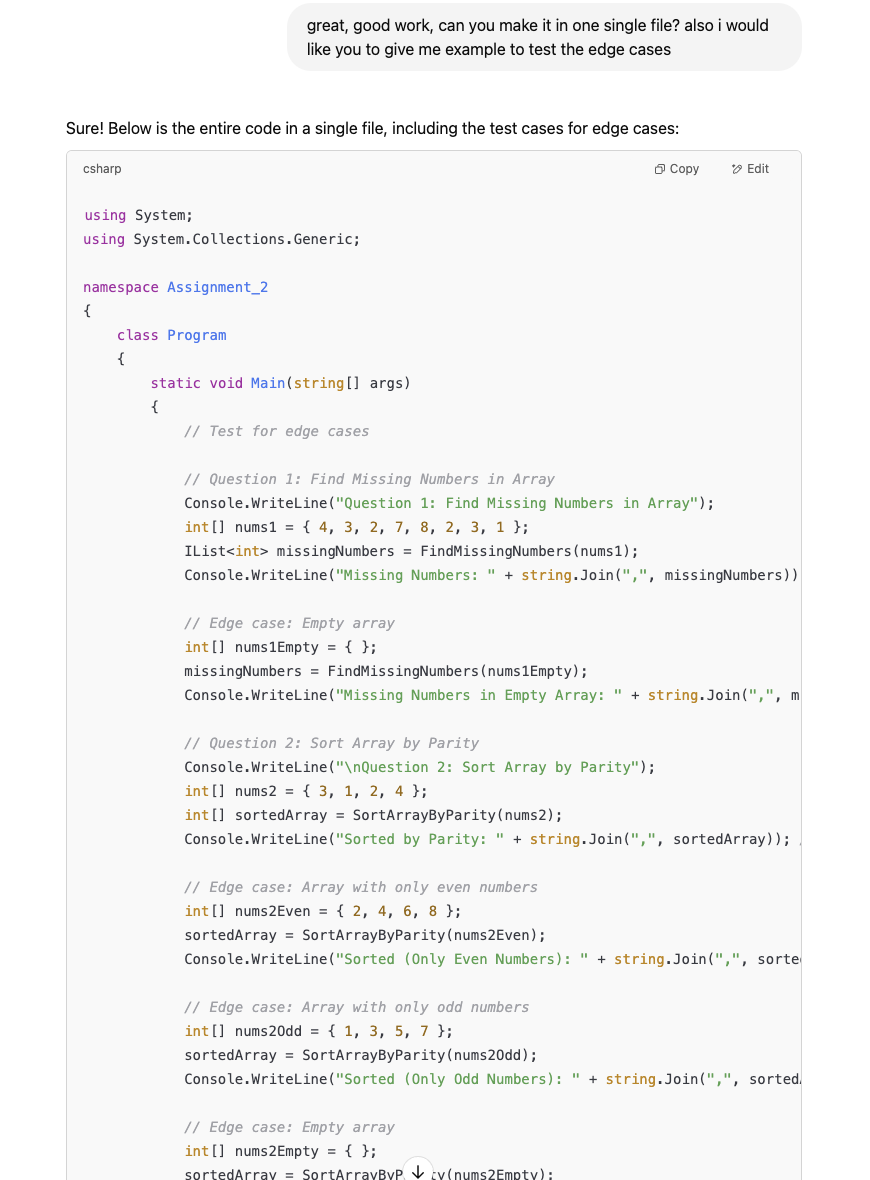
*" make sure to keep try and error in the methods for all the places where it was there”*

## Responses Received:

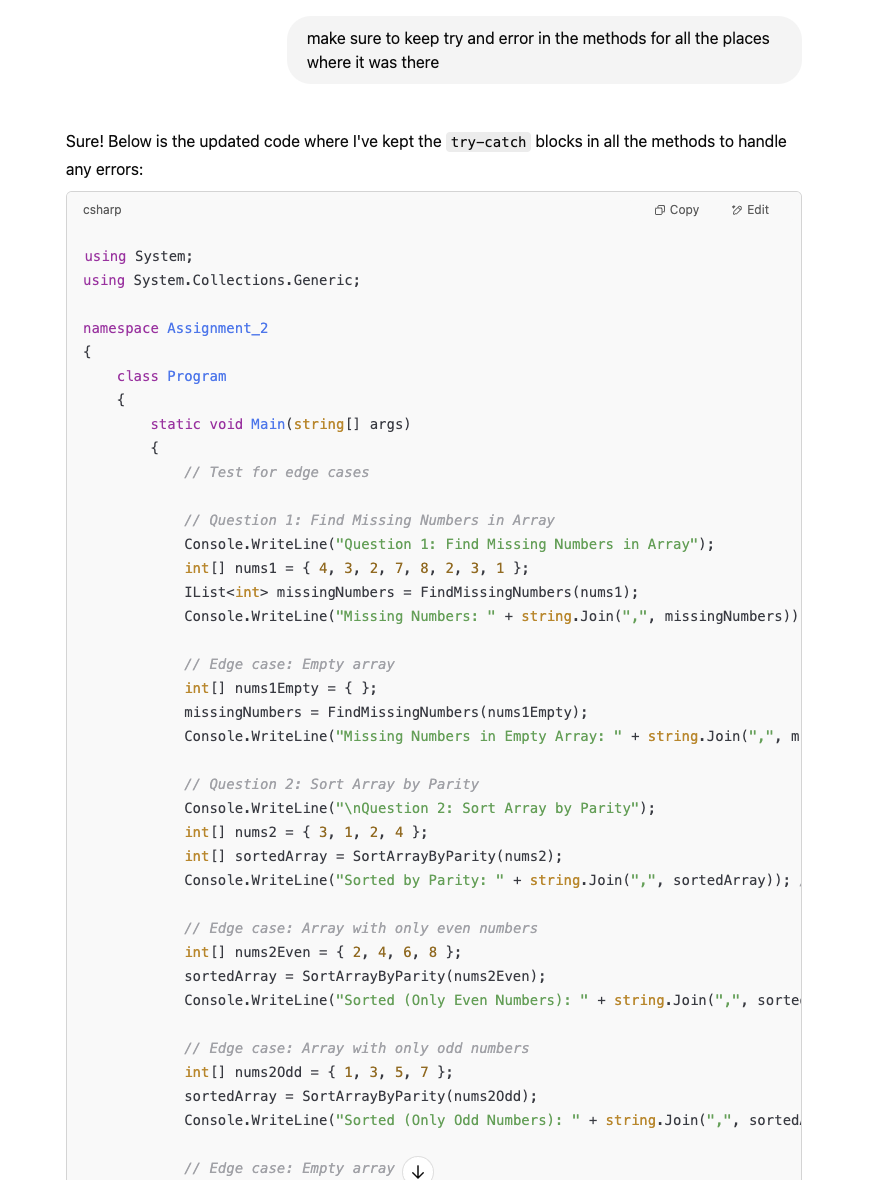
1. **Initial Response:**
   1. I provided a solution that completed the methods defined in the Program.cs file, following the prompt for each question and providing basic logic for each one (e.g., missing numbers, binary conversion, two sum, etc.). Additionally, I included logic for edge cases and returned results for sample inputs.



1. **Second Response:**
   1. The provided code was modified to include edge case tests. For example, empty arrays, arrays with negative numbers, and arrays with mixed number types (even/odd, positive/negative). The test cases ensured coverage of a variety of potential inputs that could occur in real-world scenarios.



1. **Final Response:**
   1. The code was adjusted to include the necessary try-catch blocks for every method. I wrapped each core method in a try-catch structure to handle exceptions, ensuring the program could handle any unforeseen errors gracefully without crashing. Each method was updated to catch exceptions and output an error message, making the program more robust.



## Implementation Details:

1. **Edge Cases:**

Edge cases were incorporated into each method to ensure robustness.

* **Find Missing Numbers:** Added tests for duplicate values, empty arrays, and an array with all numbers present.
* **Sort Array by Parity:** Tested arrays with only even or only odd numbers and empty arrays.
* **Two Sum:** Handled cases where no solution was found, as well as the inclusion of negative numbers.
* **Maximum Product:** Incorporated cases where the array length was less than three and cases involving negative numbers and large positive numbers.
* **Binary Conversion:** Tested with zero and negative numbers.
* **Palindrome:** Handled negative numbers and single-digit numbers.
* **Fibonacci:** Included edge cases for n = 0 and n = 1.

1. **Try-Catch Blocks:**

Each method was wrapped in a try-catch block to ensure that any errors encountered during execution (e.g., array out of bounds, division by zero, or invalid operations) would be caught, preventing the program from crashing.

1. **Logic Adjustments:**

* TwoSum: The solution used a dictionary to map values to indices, which allowed for efficient O(n) lookup time.
* MaximumProduct: After sorting the array, it checked both the product of the top three largest numbers and the product of the two smallest (most negative) numbers with the largest positive number, covering all edge cases.
* Fibonacci: The Fibonacci method was optimized using an iterative approach for O(n) time complexity.
* DecimalToBinary: An efficient iterative solution was used to handle both negative and zero cases.

## Adjustments:

### **Error Handling:**

I added try-catch blocks to each method to gracefully handle potential runtime exceptions such as array index out-of-bounds, division by zero, or null references. This ensures that:

* The program doesn’t crash unexpectedly.
* Any error encountered is caught and reported to the console.
* Execution continues smoothly for the remaining parts of the program.

### **Edge Case Handling:**

To enhance robustness, I refined the logic in several methods to correctly handle edge cases. Notable examples include:

* **Missing Numbers:** Logic now handles scenarios where no numbers are missing and returns an empty list.
* **Two Sum:** Returns an empty array if no valid pair is found, preventing errors and improving clarity.
* **Find Minimum (Rotated Array):** Safeguards are in place to handle arrays with a single element or unexpected input, ensuring accurate results in all valid cases.

### **Improved Readability & Test Visibility:**

To make the output easier to understand and test:

* Added descriptive print statements before each question to explain what is being tested and what result is expected.
* Showcased edge case results in the Main method, making it easy to trace how each function behaves under unusual or minimal input scenarios.
* Ensured consistent formatting and output to improve code readability and usability during debugging and demonstrations.

## Edge cases Tested

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

#### **Edge Cases:**

1. **Duplicate Numbers** - Handled using HashSet<int>.
2. **All Numbers Present** - If no numbers are missing, an empty list should be returned.
3. **Empty Array** - The function should return an empty list.
4. **Negative or Out-of-Range Numbers** - If the array contains negative values or numbers greater than the array length, the function still behaves correctly since it only checks for values within [1, n].

### **Question 2: Sort Array by Parity**

#### **Edge Cases:**

1. **Array with Only Evens or Only Odds** - Function should return the same array but properly structured.
2. **Empty Array** - Should return an empty array.
3. **Single Element Array** - Should return the same array without modification.
4. **Negative Numbers** - Works correctly since even/odd checks are based on modulus.

### **Question 3: Two Sum**

#### **Edge Cases:**

1. **No Solution Exists** - Returns an empty array.
2. **Multiple Pairs** - Returns the first valid pair.
3. **Same Element Used Twice** - Avoids incorrect results by checking previous indices.
4. **Negative and Zero Values** - Should work properly as long as numbers sum to target.
5. **Large Arrays** - The dictionary approach ensures O(n) time complexity.

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

#### **Edge Cases:**

1. **Negative Numbers** - The solution correctly handles negative numbers by considering nums[0] \* nums[1] \* nums[n - 1] (smallest negatives can yield the largest product).
2. **Array with Less than 3 Elements** - The function assumes at least three elements. It should handle cases where nums.Length < 3 with an error or return int.MinValue.
3. **All Positive or All Negative Numbers** - Correctly handled by sorting and computing max values.

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

#### **Edge Cases:**

1. **Zero as Input** - Correctly returns "0".
2. **Negative Numbers** - The function should clarify behavior for negative numbers (e.g., two's complement vs. unsigned).
3. **Large Numbers** - Built-in conversion handles it, but memory issues could arise for very large values.

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

#### **Edge Cases:**

1. **Already Sorted Array** - Should return nums[0].
2. **Single Element Array** - Should return that element.
3. **All Elements Same** - Returns any element since all are equal.
4. **Array Rotated at Different Positions** - The algorithm correctly finds the minimum in any rotation scenario.

### **Question 7: Palindrome Number**

#### **Edge Cases:**

1. **Negative Numbers** - Negative numbers are never palindromes.
2. **Single Digit Numbers** - Always palindromes.
3. **Numbers with Zero at the End (e.g., 10, 100)** - Should return false.
4. **Very Large Numbers** - Should check for integer overflow in reversed = reversed \* 10 + x % 10.

### **Question 8: Fibonacci Number**

#### **Edge Cases:**

1. **Zero and One** - Base cases handled correctly.
2. **Negative Numbers** - Should return an error as Fibonacci is not defined for negatives.
3. **Large n Values** - Could cause integer overflow for int, should be handled with long for large inputs.
4. **Recursive Alternative** - Iterative method is better for performance (O(n)) compared to naive recursion (O(2^n)).