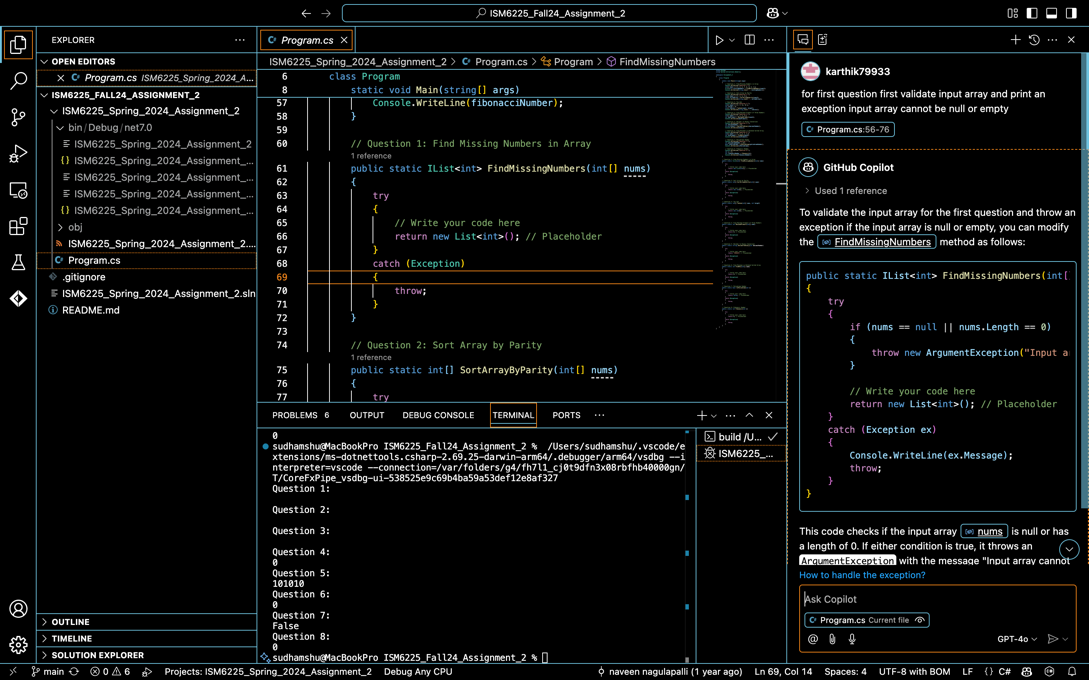
Assignment 2 - Computational problem solving

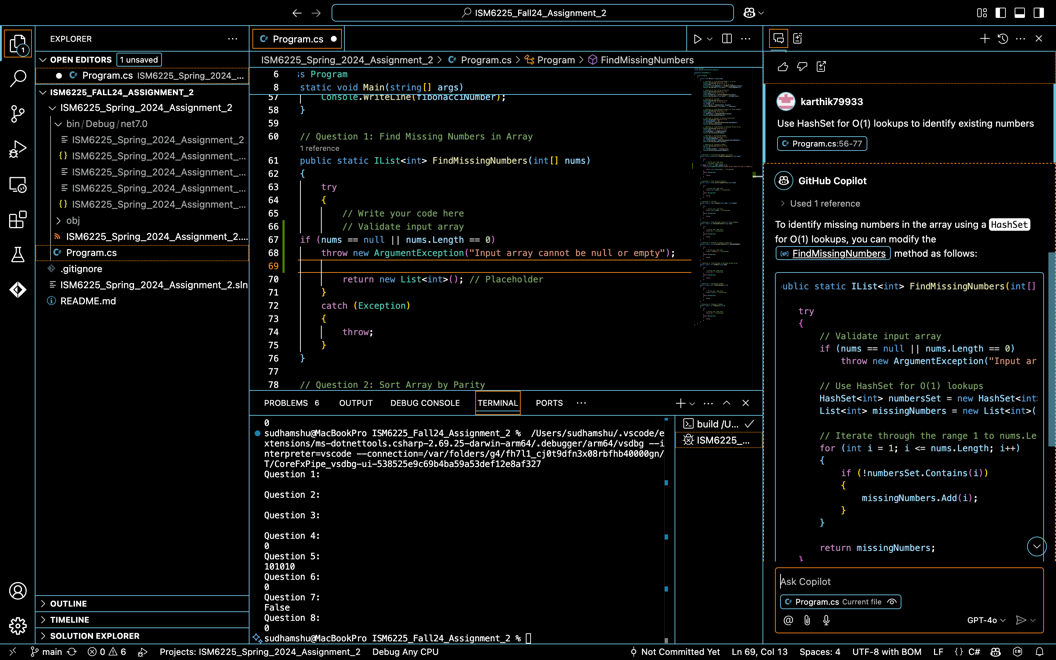
Question 1. Find Missing Numbers in Array

Copilot questions and answers:

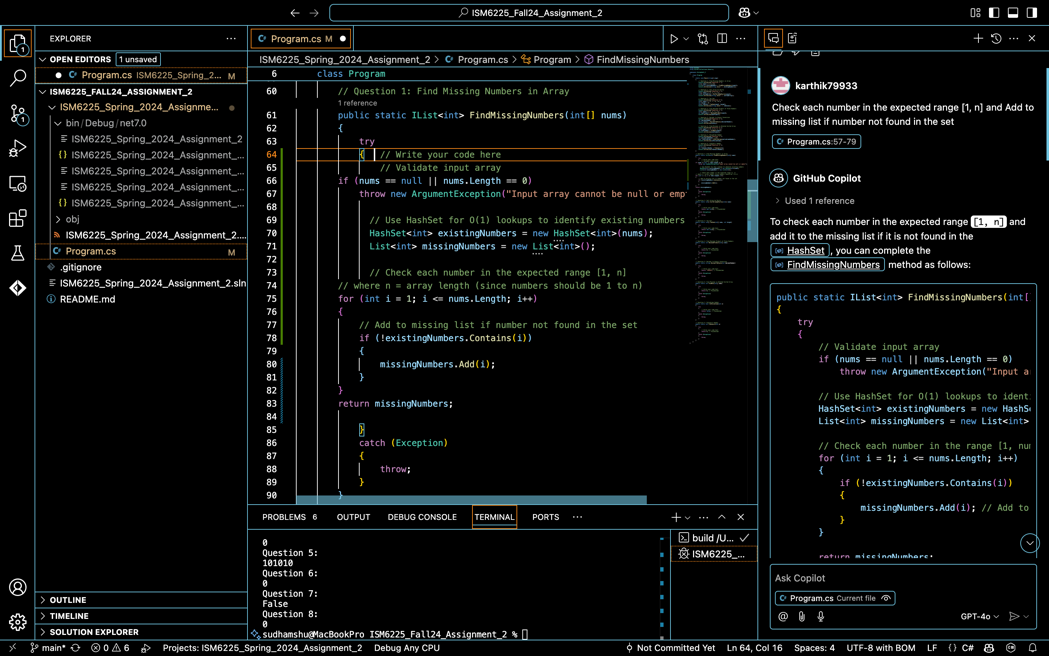
For first question first validate input array and print an exception input array cannot be null or empty



Use HashSet for O(1) lookups to identify existing numbers



Check each number in the expected range [1, n] and Add to missing list if number not found in the set



Outcome method :

// Question 1: Find Missing Numbers in Array

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

{

try

{ // Write your code here

// Validate input array

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

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

// Use HashSet for O(1) lookups to identify existing numbers

HashSet<int> existingNumbers = new HashSet<int>(nums);

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

// Check each number in the expected range [1, n]

// where n = array length (since numbers should be 1 to n)

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

{

// Add to missing list if number not found in the set

if (!existingNumbers.Contains(i))

{

missingNumbers.Add(i);

}

}

return missingNumbers;

}

catch (Exception)

{

throw;

}

}

Implementation:

You are given an array of integers nums containing numbers from 1 to n (where n is the length of the array). Some numbers may appear twice, and some may be missing.

**Goal:** Return a list of numbers in the range 1 to n that **do not appear** in the array.

This is a static method that accepts an array of integers nums and returns a list of integers (the missing numbers).

Before proceeding, we **validate** the input.

If the array is null or empty, we throw an exception to avoid errors during processing.

HashSet<int> is used here because it provides **constant time (O(1)) lookup** on average.

We initialize the HashSet with all the numbers from the array.

This helps us quickly check if a number exists in the array without iterating over it again and again.

This list will store the numbers that are **missing** from the range [1, n].

Now we iterate over every number from 1 to n.

For each number i:

If it's **not in the HashSet**, that means it’s **missing from the array**.

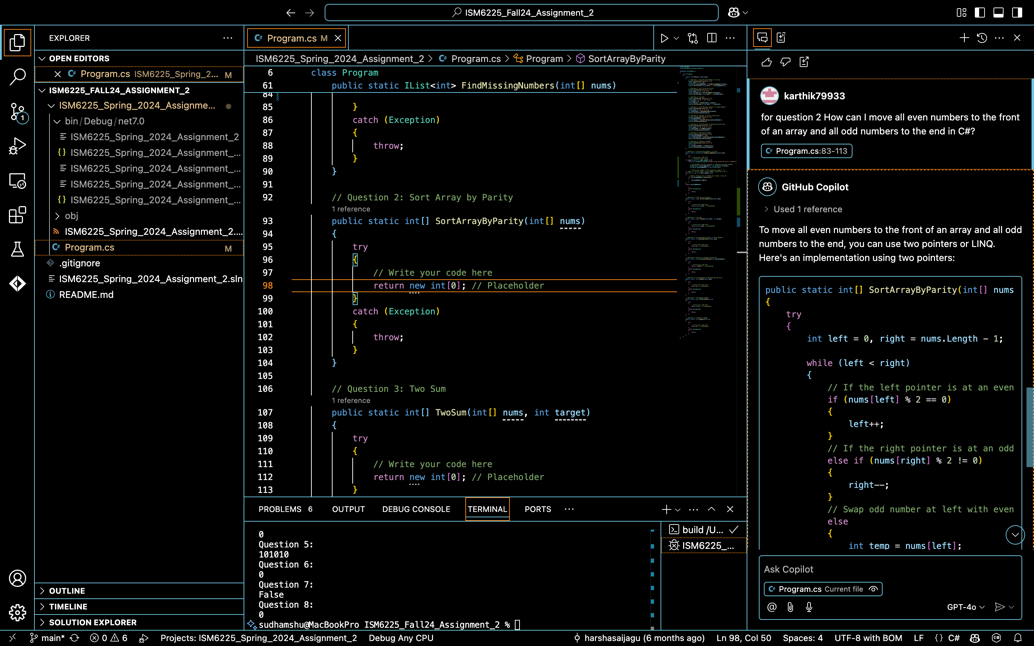
We add it to the missingNumbers list.

Finally, we return the list of missing numbers.

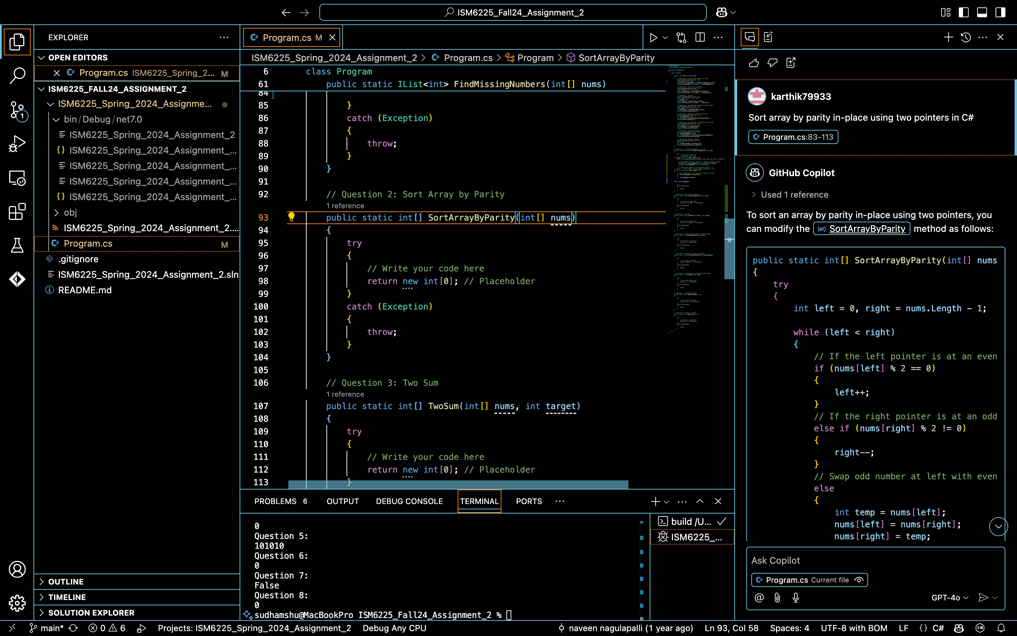
Question 2. Sort Array by Parity

Copilot questions and answers:

For question 2 How can I move all even numbers to the front of an array and all odd numbers to the end in C#?



Sort array by parity in-place using two pointers in C#



Outcome method:

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

{

try

{

// Validate input

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

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

// Separate even and odd numbers

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

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

foreach (int num in nums)

{

if (num % 2 == 0)

evens.Add(num);

else

odds.Add(num);

}

// Sort both lists

evens.Sort();

odds.Sort();

// Merge even and odd lists into a result array

int[] result = new int[nums.Length];

int index = 0;

foreach (int even in evens)

result[index++] = even;

foreach (int odd in odds)

result[index++] = odd;

return result;

}

catch (Exception)

{

throw;

}

}

Implementation:

To **rearrange the integers in an array** such that:

All **even numbers** appear before **odd numbers**.

Both **even and odd numbers are individually sorted in ascending order**.

**Input:** An array of integers (int[] nums)

**Output:** A new array of integers, sorted by parity and then by value

We check if the array is null or empty.

If it is, we throw an ArgumentException to avoid processing invalid data.

We create two lists:

* evens → to store even numbers
* odds → to store odd numbers

We iterate through each number in the array:

* If divisible by 2 (num % 2 == 0), it’s even
* Otherwise, it’s odd

We place each number in the appropriate list.

Sort the evens list in ascending order and sort the odds list in ascending order

And then create a new array result to store the final output

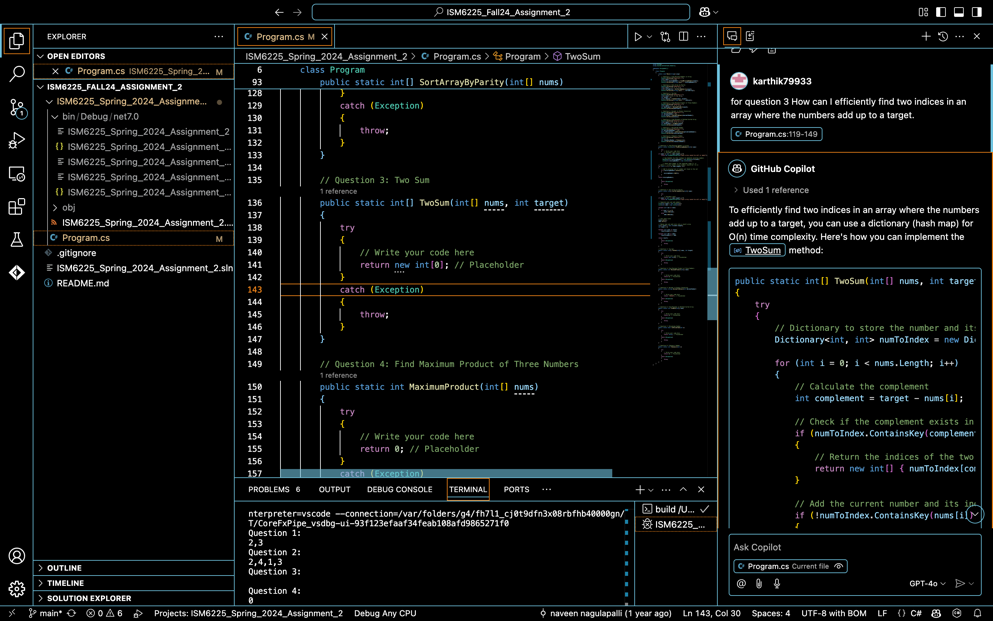
We copy all elements from the sorted evens list into result and then we copy all elements from the sorted odds list

The final array has all even numbers (sorted), followed by all odd numbers (sorted)

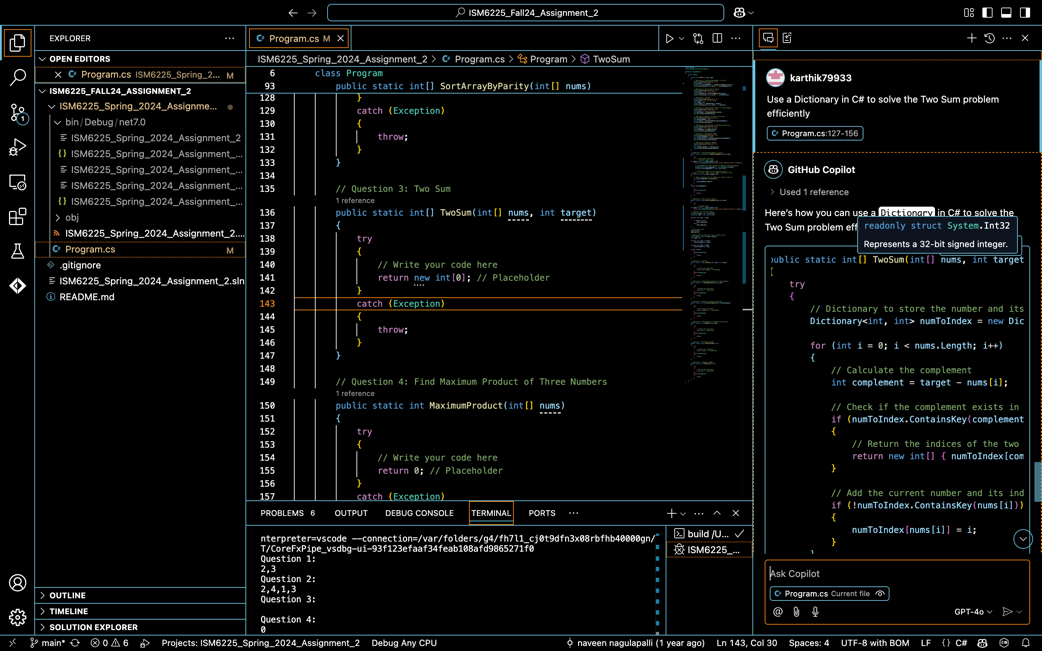
Question 3. Two Sum (Find Two Numbers that Add to Target):

Copilot questions and answers:

for question 3 How can I efficiently find two indices in an array where the numbers add up to a target.



Use a Dictionary in C# to solve the Two Sum problem efficiently



Outcome method:

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

{

try

{

// Validate input

if (nums == null || nums.Length < 2)

throw new ArgumentException("Input array must have at least two elements.");

// Dictionary to store numbers and their indices

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

// Iterate through the array

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

{

int complement = target - nums[i]; // The number we need to find

// Check if complement exists in the dictionary

if (numToIndex.ContainsKey(complement))

{

// If found, return indices of complement and current number

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

}

// If not found, store current number and its index

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

numToIndex[nums[i]] = i;

}

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

return new int[0];

}

catch (Exception)

{

throw;

}

}

Implementation:

**Input validation**: This checks whether the input array nums is valid. If the array is null or has fewer than two elements, the method throws an ArgumentException

Since it's impossible to find two numbers that sum to the target if there are fewer than two numbers.

**Initialize the dictionary**: We use a **Dictionary** (numToIndex) to store each number in the array as the **key** and its index as the **value**. This allows us to efficiently check whether we have encountered the complement of a number during iteration.

**Iterate through the array**: We loop through each element of the nums array, where i is the index of the current element.

**Calculate complement**: For the current element nums[i], we calculate the **complement** as the difference between the target and nums[i]. This complement is the number that, when added to nums[i], would equal the target.

**Check for complement**: The code checks if the complement is already in the dictionary (numToIndex.ContainsKey(complement)).

If the complement is found, it means we have previously encountered a number that adds up with nums[i] to give the target.

In this case, the method returns the indices of the complement (which is stored in numToIndex[complement]) and the current index i.

**Store current number and index**: If the current number nums[i] is not already in the dictionary, we add it along with its index. This ensures that as we continue iterating, we can check if future numbers can form a sum with the current number.

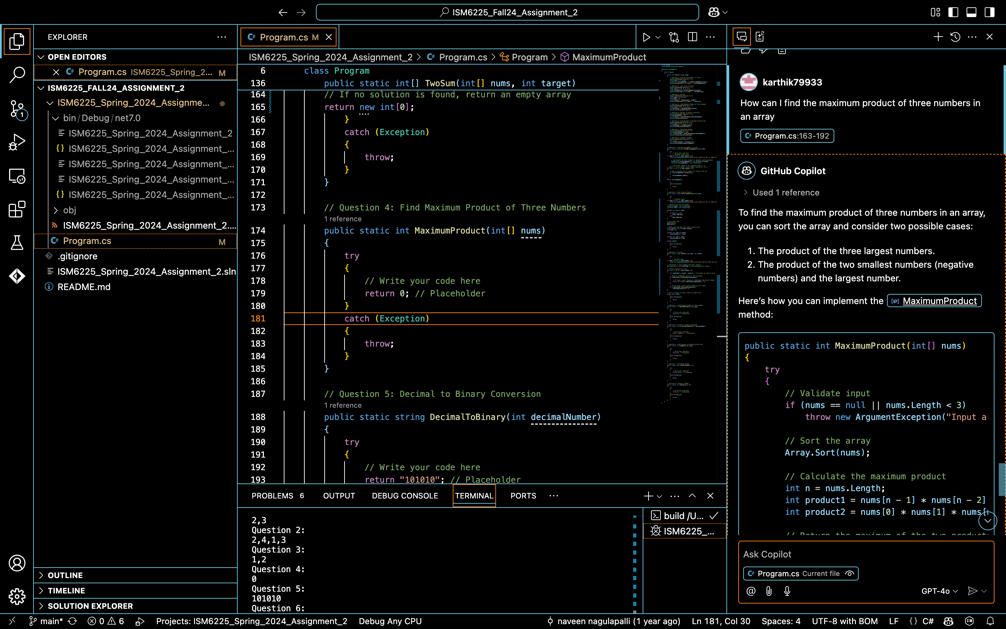
This check (!numToIndex.ContainsKey(nums[i])) prevents overwriting an index for a duplicate number, as we want to store the first occurrence of each number.

**No solution found**: If the loop finishes and no pair that sums to the target is found, the method returns an empty array, indicating that there is no solution.

Question 4. Find Maximum Product of Three Numbers:

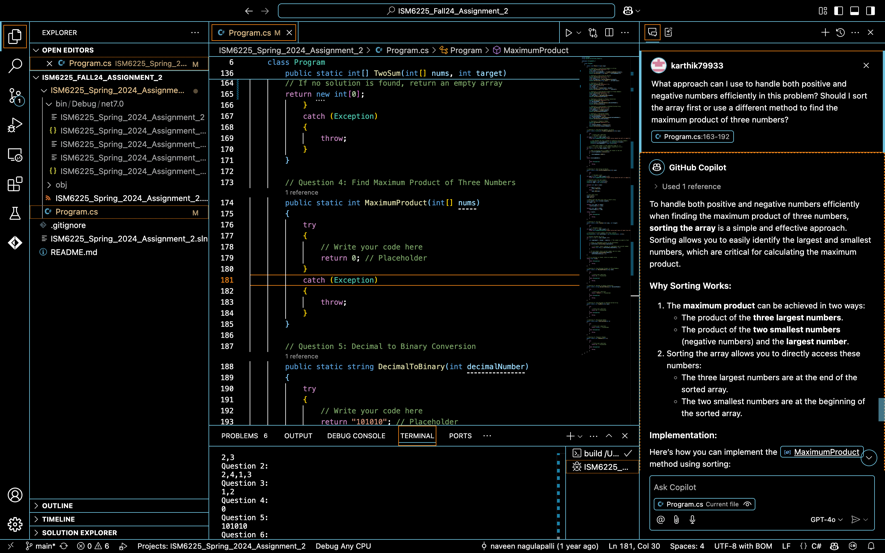
Copilot questions and answers:

How can I find the maximum product of three numbers in an array

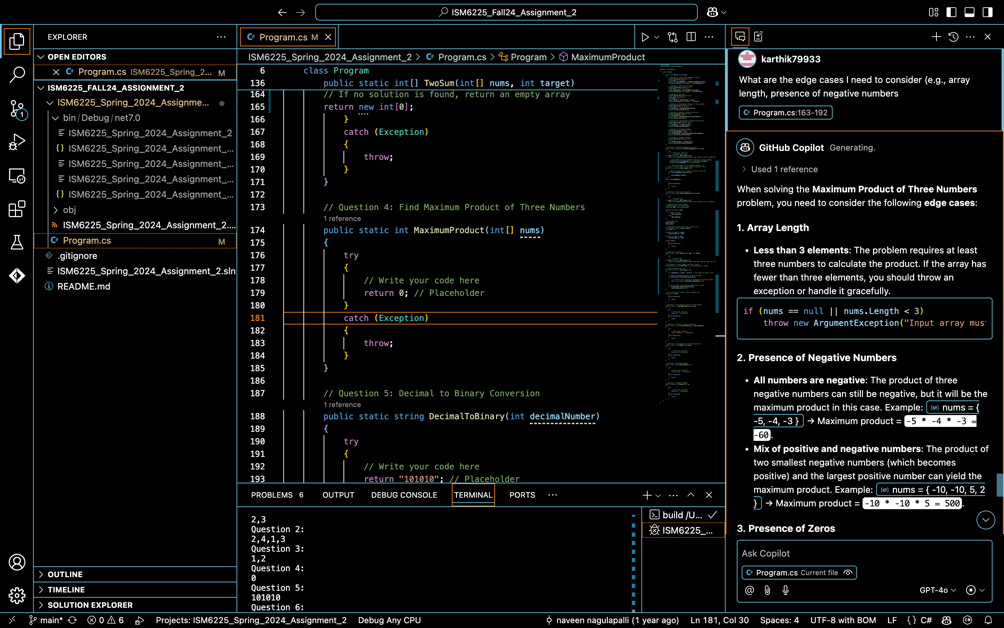


What approach can I use to handle both positive and negative numbers efficiently in this problem?

Should I sort the array first or use a different method to find the maximum product of three numbers



What are the edge cases I need to consider (e.g., array length, presence of negative numbers



Outcome method:

public static int MaximumProduct(int[] nums)

{

try

{

// Validate input

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

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

// Sort the array to easily access the largest and smallest values

Array.Sort(nums);

// After sorting, the largest product will be either:

// 1. The product of the three largest numbers (last three elements in the sorted array)

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

// Option 1: Product of the three largest numbers

int option1 = nums[nums.Length - 1] \* nums[nums.Length - 2] \* nums[nums.Length - 3];

// Option 2: Product of the two smallest numbers (most negative) and the largest number

int option2 = nums[0] \* nums[1] \* nums[nums.Length - 1];

// Return the maximum of the two options

return Math.Max(option1, option2);

}

catch (Exception)

{

throw;

}

}

Implementation:

If the array does not have at least 3 elements, it would be impossible to form a triplet, so we throw an exception.

We sort the array in ascending order.

Sorting helps in easily finding the largest and smallest numbers. Once sorted, we can quickly identify:

The largest numbers (at the end of the sorted array).

The smallest numbers (at the beginning of the sorted array).

The largest product of three numbers can come from either:

The two smallest (most negative) numbers multiplied with the largest positive number (if there are negative numbers).

In many cases, the largest product will come from multiplying the three largest numbers. This works when all numbers are positive, or when there are large positive numbers and the array is sorted in ascending order.

The product of the two smallest numbers (most negative numbers) and the largest number (to account for the possibility of large negative numbers yielding a large product when multiplied by a large positive number).

After calculating both possible products (the product of the three largest numbers and the product of the two smallest numbers with the largest number), we return the larger of the two. This ensures that we get the maximum product from all possibilities.

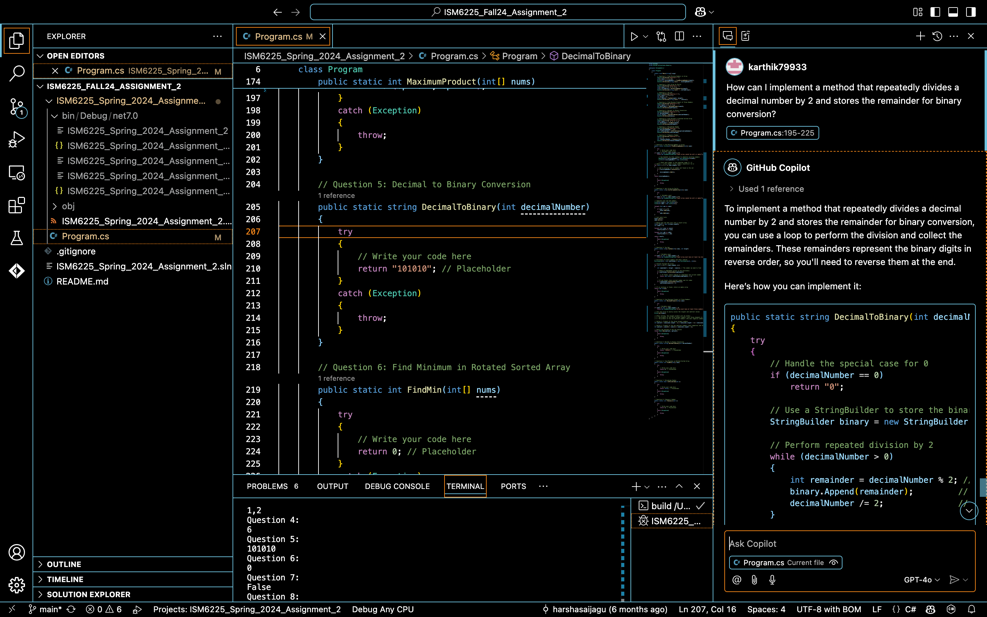
Question 5. Decimal to Binary Conversion:

Copilot questions and answers:

What is the most efficient way to convert a decimal number to binary using built-in C# functions?



How can I implement a method that repeatedly divides a decimal number by 2 and stores the remainder for binary conversion?



Outcome Method:

public static string DecimalToBinary(int decimalNumber)

{

try

{

// Edge case: if the number is 0, return "0" because the binary equivalent of 0 is simply 0

if (decimalNumber == 0)

return "0";

// Initialize a string to hold the binary result

string binaryResult = "";

// Loop to perform division and get the remainder (binary digit)

while (decimalNumber > 0)

{

// Get the remainder when divided by 2 (this gives us the binary digit)

int remainder = decimalNumber % 2;

// Add the remainder (binary digit) to the front of the result string

binaryResult = remainder.ToString() + binaryResult;

// Update decimalNumber by dividing it by 2 to process the next digit

decimalNumber = decimalNumber / 2;

}

// Return the final binary string result

return binaryResult;

}

catch (Exception)

{

throw;

}

}

Implementation:

**Edge Case Handling (Decimal Number = 0):**

This checks if the input decimal number is 0.

The binary representation of 0 is simply 0, so we handle this case explicitly by returning "0" early. Without this, the code would enter the while loop and never execute properly, because 0 % 2 would always result in 0.

We initialize an empty string binaryResult that will be used to build the binary equivalent of the number. This string will ultimately hold the binary digits (0 or 1) in the correct order.

while loop is where the actual conversion happens. It continues as long as decimalNumber is greater than zero. In each iteration of the loop:

We calculate the remainder when the decimalNumber is divided by 2 (decimalNumber % 2). This remainder gives us the binary digit for the current least significant bit (rightmost bit).

We add the binary digit to the front of the binaryResult string. This ensures that the digits are added in the correct order (most significant bit first).

We update decimalNumber by dividing it by 2 (decimalNumber = decimalNumber / 2). This step reduces the number and shifts the next binary digit into place for the next iteration.

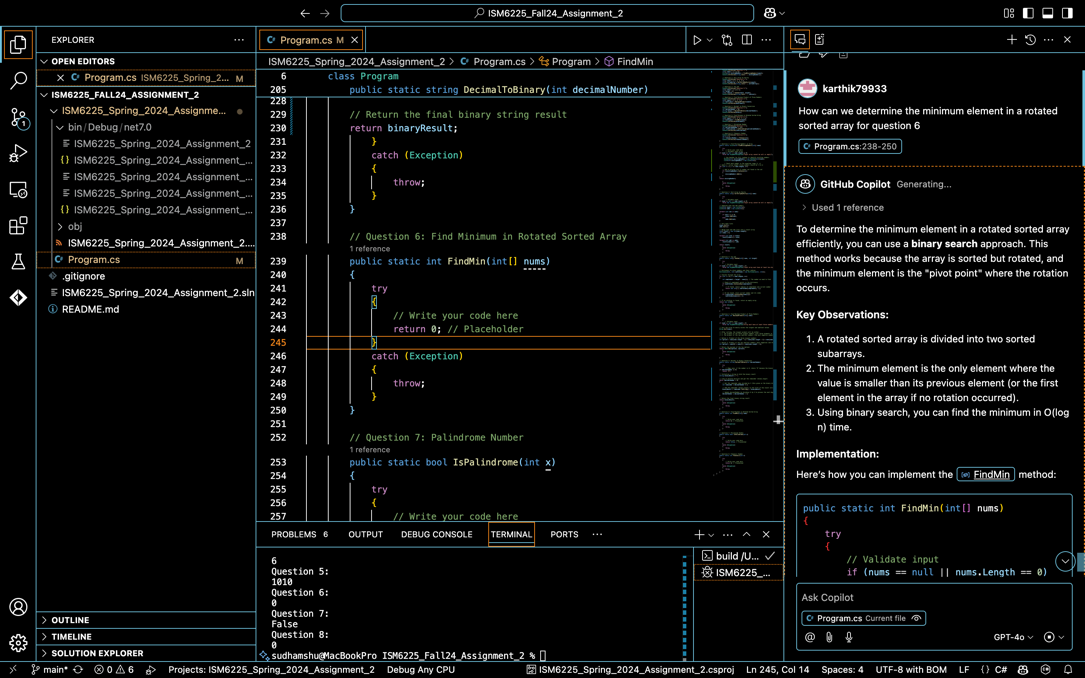
Once the while loop has completed (i.e., decimalNumber has been reduced to 0), the binaryResult string holds the binary representation of the original decimal number. We return this string as the result of the method.

If any unexpected errors occur during execution, they will be caught in the catch block. The exception is then rethrown to propagate it to the caller, ensuring that the program can handle errors properly.

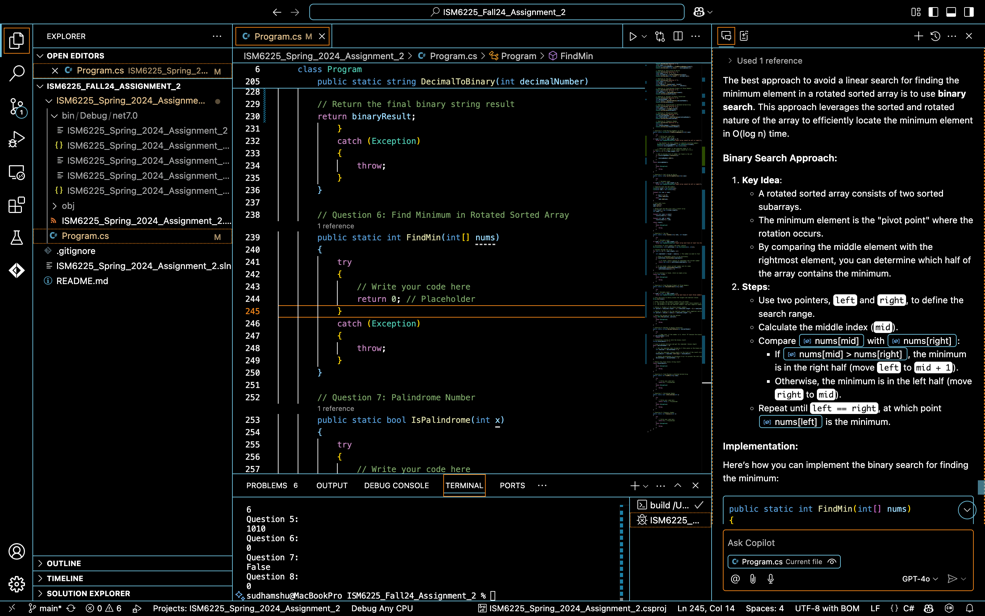
Question 6. Find Minimum in Rotated Sorted Array:

Copilot questions and answers:

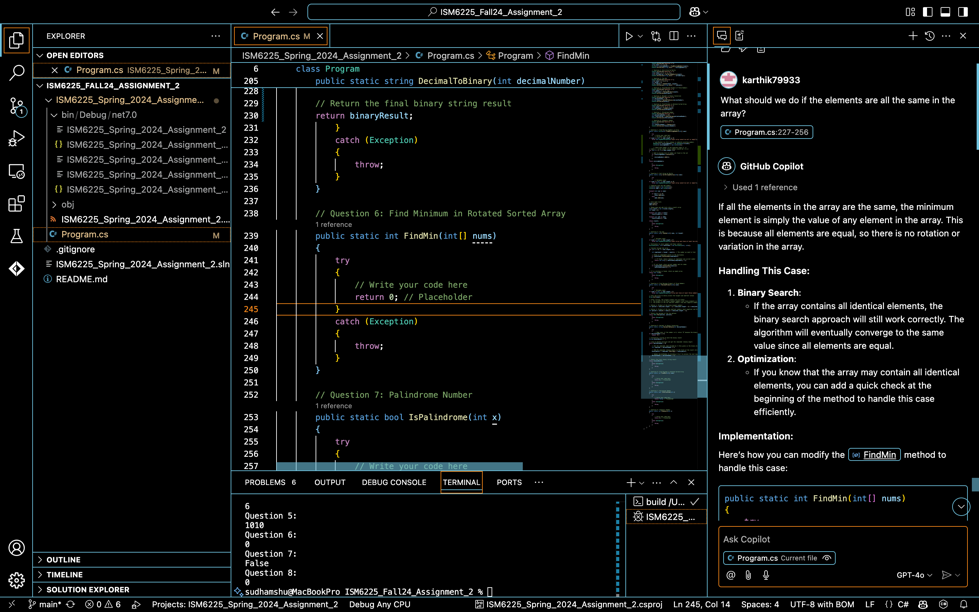
How can we determine the minimum element in a rotated sorted array



What is the best approach to avoid a linear search for finding the minimum element in a rotated sorted array?



What should we do if the elements are all the same in the array?



Outcome Method:

public static int FindMin(int[] nums)

{

try

{

// Edge case: If the array has only one element, return it as the minimum.

if (nums.Length == 1)

return nums[0];

// Initialize the pointers for binary search

int low = 0;

int high = nums.Length - 1;

// Perform binary search to find the minimum element

while (low < high)

{

int mid = low + (high - low) / 2; // Calculate mid index

// If mid element is greater than the high element, the minimum is in the right half

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

{

low = mid + 1; // Move the low pointer to the right half

}

// If mid element is less than or equal to the high element, the minimum is in the left half

else

{

high = mid; // Move the high pointer to mid (inclusive)

}

}

// After the loop ends, low will be pointing to the minimum element

return nums[low];

}

catch (Exception)

{

throw;

}

}

Implementation:

**Input**: If the array has only one element, then the array is trivially sorted, and that single element is the minimum.

**Action**: If the length of nums is 1, the method immediately returns that element, as it is the minimum.

low and high are initialized to point to the beginning (0) and end (nums.Length - 1) of the array.

These pointers will help us define the current search range.

We use a binary search to efficiently find the minimum element. This allows us to reduce the problem size by half at each iteration.

mid is calculated as the midpoint between low and high using the formula mid = low + (high - low) / 2;

**Condition**: If nums[mid] > nums[high], this means that the smallest element is to the right of mid. This happens because the array is rotated, and the portion from mid to high is not sorted anymore. The pivot point (where the rotation happens) is in the right half.

**Action**: Move the low pointer to mid + 1 because the minimum must be in the right half (we exclude the current mid).

After the loop ends, the low pointer will point to the minimum element in the array. This is because, during the binary search

We narrowed down the search range by half each time, and when low == high, we have located the minimum.

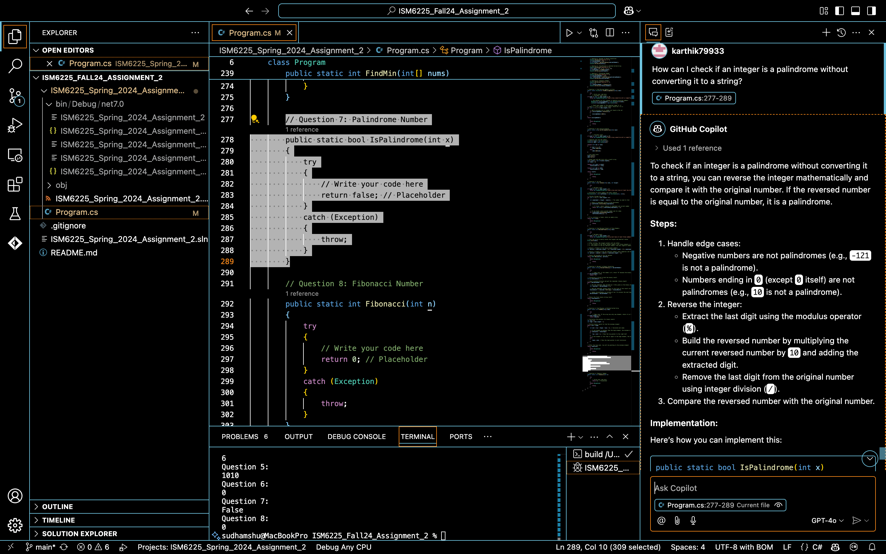
The method then returns nums[low] as the minimum value.

If any unexpected exception occurs during the execution, the method will rethrow it. This ensures that the method behaves as expected, and any issues are passed to the calling code for handling.

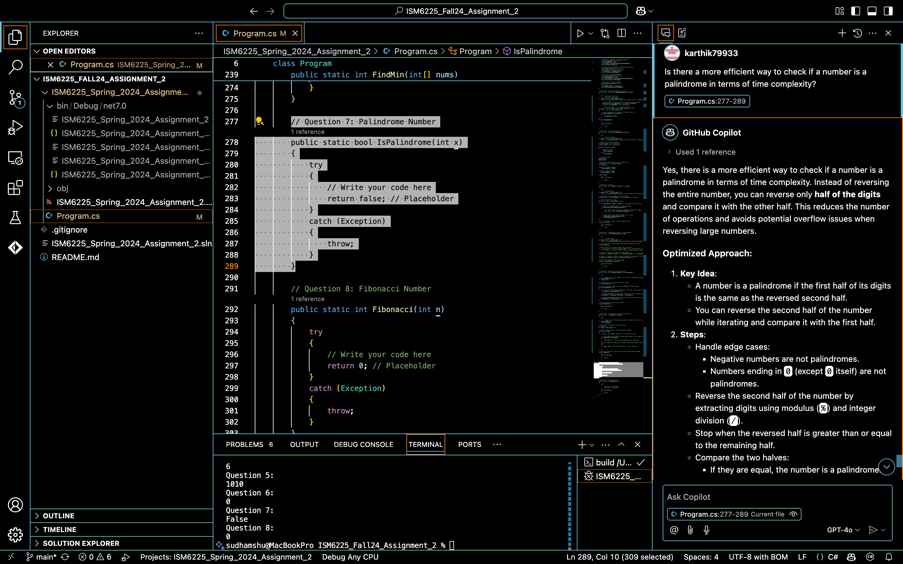
Question 7. Palindrome Number:

Copilot questions and answers:

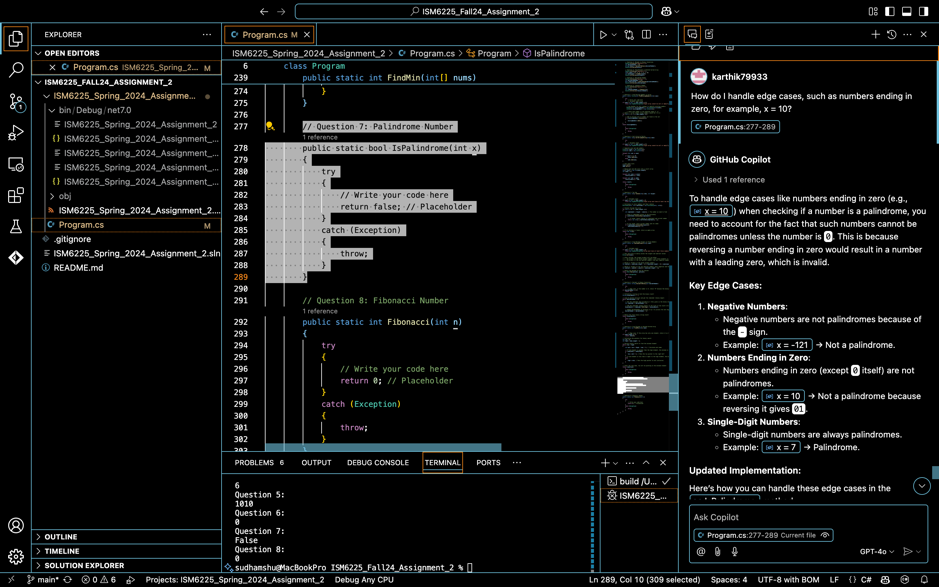
How can I check if an integer is a palindrome without converting it to a string?



Is there a more efficient way to check if a number is a palindrome in terms of time complexity?



How do I handle edge cases, such as numbers ending in zero, for example, x = 10?



Outcome Method:

**How to check for a palindrome?**

Once the loop ends, x will be reduced to a number smaller than or equal to the reversed half (reversedHalf). At this point, we need to check if x equals reversedHalf.

However, if the number of digits in x is odd, reversedHalf will have one extra digit. For example, in the number 121, after reversing reversedHalf will be 12 while x will be 1. To handle this, we can divide reversedHalf by 10 to remove the extra digit.

If x is equal to reversedHalf, it means the number is a palindrome (the reversed half matches the remaining half).

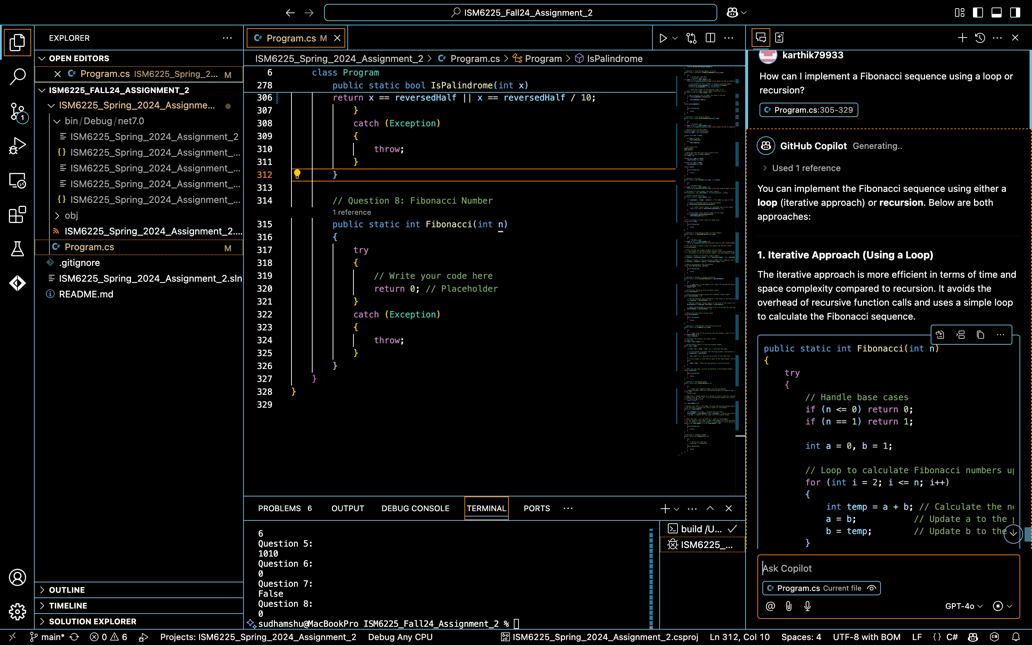
If x equals reversedHalf / 10, it means the number has an odd number of digits, and the extra digit in reversedHalf can be ignored (e.g., for 121, reversedHalf becomes 12, and x is 1).

If any exception occurs during the execution (e.g., in case of invalid input), the exception will be rethrown.

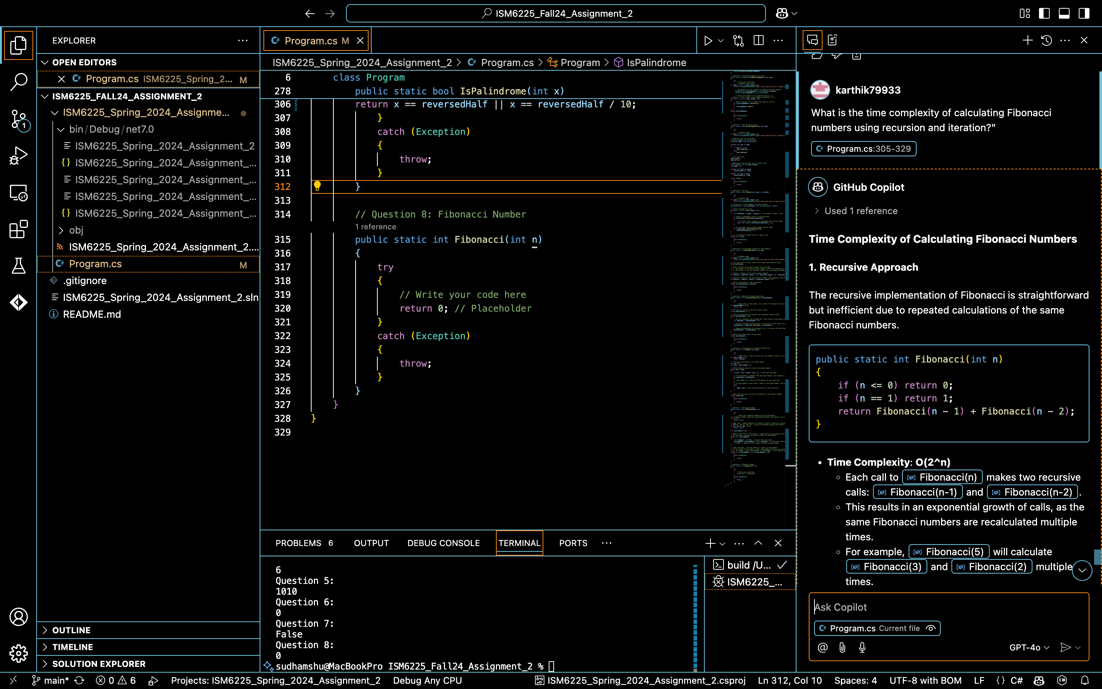
Question 7. Fibonacci Number:

Copilot questions and answers:

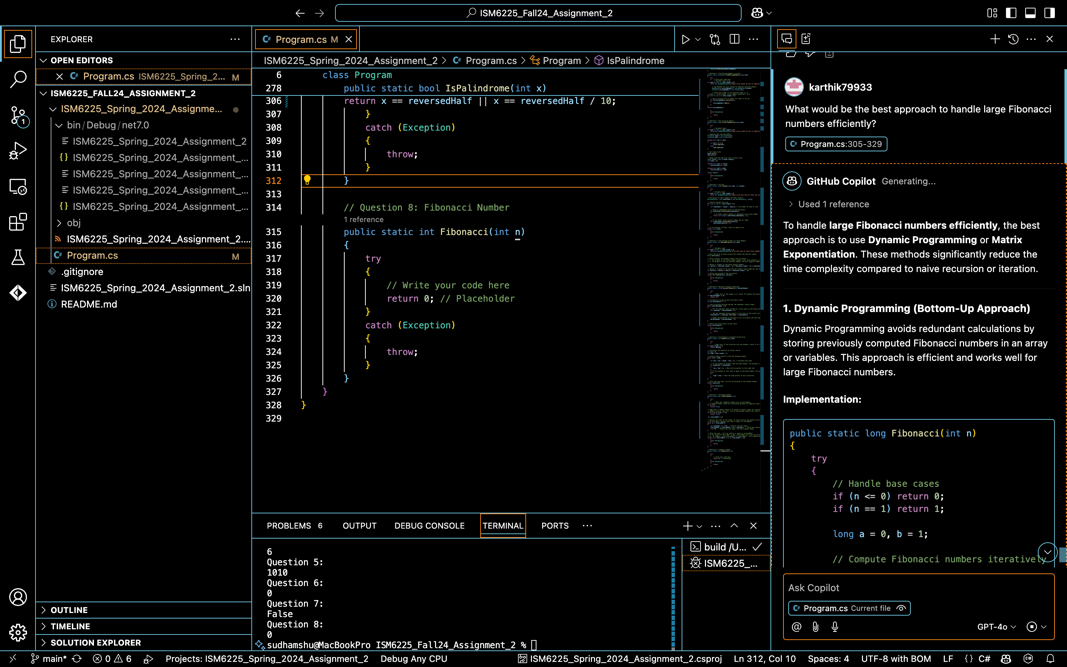
How can I implement a Fibonacci sequence using a loop or recursion?



What is the time complexity of calculating Fibonacci numbers using recursion and iteration?"



What would be the best approach to handle large Fibonacci numbers efficiently?



Outcome Method:

public static int Fibonacci(int n)

{

try

{

// Edge case: If n is 0 or 1, return n directly (base cases of Fibonacci sequence)

if (n == 0)

return 0;

if (n == 1)

return 1;

// Initialize variables to store Fibonacci numbers at previous two positions

int prev1 = 0; // F(0)

int prev2 = 1; // F(1)

// Loop to calculate Fibonacci number for n >= 2

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

{

int current = prev1 + prev2; // F(i) = F(i-1) + F(i-2)

prev1 = prev2; // Update prev1 to previous prev2 value

prev2 = current; // Update prev2 to the newly calculated Fibonacci number

}

// Return the Fibonacci number for F(n)

return prev2;

}

catch (Exception)

{

throw;

}

Implementation:

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, starting from 0 and 1:

F(0) = 0

F(1) = 1

F(n) = F(n - 1) + F(n - 2), for n > 1

We handle the base cases first:

If n = 0, the Fibonacci number is 0.

If n = 1, the Fibonacci number is 1.  
These are directly returned to avoid unnecessary computation.

prev1 and prev2 two variables are used to store the last two Fibonacci numbers:

prev1 holds F(0) = 0

prev2 holds F(1) = 1  
These will be updated during each iteration to calculate subsequent Fibonacci numbers.

**Loop to calculate Fibonacci for n >= 2**:  
The for loop runs from i = 2 to i = n. In each iteration:

We compute the Fibonacci number at position i as the sum of the previous two numbers: F(i) = F(i-1) + F(i-2).

current holds the result of F(i).

prev1 is updated to the value of prev2 (because prev2 represents F(i-1)).

prev2 is updated to current (since current is the newly computed F(i)).

This ensures that at each step, prev1 and prev2 always contain the last two Fibonacci numbers.

After the loop finishes, the variable prev2 will contain the Fibonacci number F(n), which is returned as the final result.

If the loop completes, prev2 will hold the correct Fibonacci number for n because it was updated through every iteration.