**Prompts used in GitHub Copilot (Visual Studio)**

1.Find Missing Numbers in Array

**Prompt Used:**

How to find missing numbers in a range from 1 to n using C# without extra space?

Here is my code so far:

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

{

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

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

{

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

if (nums[index] > 0)

{

nums[index] = -nums[index];

}

}

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

{

if (nums[i] > 0)

{

result.Add(i + 1);

}

}

return result;

}

**Responses Received**

The AI suggested using the input array itself for marking presence by converting values to negative at indices corresponding to the values. This avoids using additional memory and maintains O(n) time complexity.

It also confirmed that taking the absolute value during index calculations ensures correctness when elements are already marked negative.

**Edge Case Handling:** Used Math.Abs to ensure previously marked values didn’t cause incorrect indexing.

**List Return:** Used List<int> to store the result.

**Method Signature:** Wrapped the logic into a method as required.

// The final code is written in the repository

2. Sort Array by Parity:

**Prompt Used:**

"How do I sort an array by parity in C#? Place even numbers first, followed by odd numbers."

**Response received:**

Use two pointers: one for even numbers and another for odd numbers. The idea is to iterate through the array and place even numbers in the front and odd numbers at the end.

**Code Used:**

Here’s the relevant code you can use for sorting the array by parity:

// Question 2: Sort Array by Parity

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

{

try

{

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

int evenIndex = 0;

int oddIndex = nums.Length - 1;

foreach(int num in nums)

{

if(num % 2 == 0)

{

result[evenIndex++] = num;

}

else

{

result[oddIndex--] = num;

}

}

return result;

}

catch (Exception)

{

throw;

}

}

**Adjustments:**

No adjustments were necessary. The solution provided by Copilot worked as expected and handled edge cases, such as empty arrays and arrays with all even or all odd numbers.

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

**Prompt Used:**

"How do I find two numbers in an array that add up to a target value in C#? Return the indices of the two numbers."

**Response Received:**

Use a hash map (dictionary) to store the numbers as keys and their corresponding indices as values. This allows for quick look-up of the complement (target - current number) to find the two numbers that add up to the target.

**Implementation Details:**

The AI suggested using a **hash map (dictionary)** where the key is the number in the array, and the value is its index.

For each number, it calculates the complement (the number that needs to be added to the current number to reach the target).

It checks if the complement already exists in the dictionary:

* If it does, the pair is found, and it returns the indices of the two numbers.
* If it doesn't, the current number is added to the dictionary.

This approach is efficient with a time complexity of O(n), which is optimal for this problem.

**Adjustments:**

No significant adjustments were made. The solution provided by Copilot was directly applied. The only modification was ensuring that the solution returns an empty array if no pair is found, which was already covered in the response from Copilot.

// The code is written in the repository.

4. Find Maximum Product of Three Numbers:

**Prompt Used:**

"How can I find the maximum product of three numbers in an array in C#?"

**Response Received:**

Track of the **top three largest** and **two smallest** numbers in the array. This is because the maximum product can either be the product of the three largest numbers or the product of the two smallest (most negative) numbers and the largest number. The AI provided an algorithm where:

* The largest three numbers are tracked.
* The two smallest numbers are tracked.
* The result is the maximum of the product of the top three largest numbers or the product of the two smallest and the largest number.

**Implementation Details:**

The AI solution utilizes a linear scan through the array to determine the **three largest numbers** (max1, max2, max3) and the **two smallest numbers** (min1, min2).

It compares each number to update the largest and smallest numbers.

* If a number is larger than the largest tracked number (max1), we update the top three largest numbers.
* If a number is smaller than the smallest tracked number (min1), we update the smallest two numbers.

After processing the array, the function returns the **maximum product** of:

* The top three largest numbers, or
* The product of the two smallest (most negative) numbers and the largest number.

This ensures the solution handles arrays with both positive and negative values correctly.

**Adjustments:**

No adjustments were necessary. The solution provided by Copilot was directly applicable for the problem. It efficiently handles edge cases such as arrays with negative numbers or arrays that have fewer than three elements.

5. Decimal to Binary Conversion:

**Prompt Used:**

"How can I convert a decimal number to its binary representation in C#?"

**Response Received:**

Use a loop that repeatedly divides the decimal number by 2 and inserts the remainder (0 or 1) at the beginning of a string or list, which ultimately forms the binary representation. This is a standard method for decimal to binary conversion.

**Implementation Details:**

* The AI solution suggests a method of repeatedly dividing the decimal number by 2 and inserting the remainder at the beginning of a string (using StringBuilder.Insert(0, value)).
* The loop continues until the decimal number becomes 0. Each division gives the next binary digit (0 or 1), which is appended to the front of the StringBuilder.
* The resulting string is the binary representation of the decimal number.

**Adjustments:**

* The solution from Copilot worked as expected. The only modification made was to check for the special case of 0 directly, returning "0" as the binary string.
* The StringBuilder was used to efficiently handle the string concatenation, as it avoids the performance hit from repeatedly creating new strings in a loop.

// The final code is written repository.

5. Decimal to Binary Conversion:

**Prompt Used:**

"How can I find the minimum element in a rotated sorted array in C#?"

**Response Received:**

Copilot suggested implementing a binary search technique. This is a common approach for finding the minimum in a rotated sorted array in O(log n) time. The key idea is to compare the middle element with the rightmost element:

* If the middle element is greater than the rightmost element, the minimum must be to the right of the middle element.
* Otherwise, the minimum must be to the left of the middle element.

**Code Used:**

Here’s the relevant code you used to find the minimum element in a rotated sorted array:

public static int FindMin(int[] nums)

{

try

{

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

if (nums[left] <= nums[right]) return nums[left]; // Edge case: Array not rotated, returns the first element

while (left < right)

{

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

if (nums[mid] > nums[right]) left = mid + 1; // Edge case: Properly handles rotated sorted arrays

else right = mid;

}

return nums[left];

}

catch (Exception)

{

throw;

}

}

**Implementation Details:**

The first check ensures that if the array is not rotated (i.e., the first element is smaller than or equal to the last element), the minimum element is the first element.

The while loop continues adjusting the left and right pointers based on the comparison between the middle element and the rightmost element:

* If the middle elem
* ent is greater than the rightmost element, the minimum lies in the right half of the array, so we adjust left to mid + 1.
* If the middle element is less than or equal to the rightmost element, the minimum lies in the left half, so we adjust right to mid.

This results in finding the minimum element efficiently in O(log n) time.

7. Palindrome Number

**Prompt Used:**

"How can I check if a number is a palindrome in C#?"

**Response Received:**

Copilot suggested checking if the number is negative, as negative numbers cannot be palindromes. Then, it recommended reversing the digits of the number and comparing it with the original number.

It suggested considering edge cases like zero (which is a palindrome) and numbers that end in zero (which are not palindromes, unless the number itself is zero).

**Code Used:**

Here’s the relevant code you used to check if a number is a palindrome:

public static bool IsPalindrome(int x)

{

try

{

if (x < 0) return false; // Edge case: Negative numbers are not palindromes

if (x == 0) return true; // Edge case: 0 is a palindrome

int original = x, reversed = 0;

while (x > 0)

{

int digit = x % 10;

reversed = reversed \* 10 + digit;

x /= 10;

}

return original == reversed; // Edge case: Numbers ending in 0 are not palindromes, except for 0 itself

}

catch (Exception)

{

throw;

}

}

**Implementation Details:**

* The AI solution recommends reversing the digits of the number and comparing it with the original number to check if it is a palindrome.
* It first handles the edge case where the number is negative (x < 0), as negative numbers cannot be palindromes.
* The check for x == 0 returns true because zero is considered a palindrome.
* The algorithm then iteratively reverses the digits of the number by extracting the last digit (x % 10) and appending it to reverse.
* Finally, the number is compared with the reversed number, and if they are equal, the number is a palindrome.
* Additionally, the algorithm considers the case where numbers ending in zero are not palindromes unless the number itself is zero (e.g., 10 is not a palindrome, but 0 is).

8. Fibonacci Number

**Prompt Used:**

"How can I generate the nth Fibonacci number in C#?"

**Response Received:**

Use an iterative approach to calculate the Fibonacci sequence. It mentioned using two variables to store the previous two numbers and iterating to calculate the nth Fibonacci number. The response also included handling base cases for n = 0 and n = 1.

**Implementation Details:**

The AI solution recommended an iterative approach to compute the nth Fibonacci number, which is more efficient than a recursive one for larger values of n.

The algorithm first checks if n <= 1, directly returning n in such cases (handling the base cases for Fibonacci numbers where F(0) = 0 and F(1) = 1).

For n > 1, the algorithm uses two variables (a and b) to store the previous two Fibonacci numbers and iterates from 2 to n, updating a and b at each step to compute the next Fibonacci number.

The variable c stores the current Fibonacci number, which is returned after the loop completes.

**Adjustments:**

The solution provided by Copilot was implemented without changes. The iterative approach works well and avoids the performance issues associated with recursion (i.e., stack overflow or excessive computation time for large n).

The code is straightforward and efficiently handles larger Fibonacci numbers, so no adjustments were required.

**Code Given:**

public static int Fibonacci(int n)

{

try

{

if (n <= 1) return n;

if (n == 0) return 0;

if (n == 1) return 1;

int a = 0, b = 1, c = 0;

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

{

c = a + b;

a = b;

b = c;

}

return c; // Edge case: Handles larger Fibonacci numbers correctly

}

catch (Exception)

{

throw;

}

}