**Use AI for Coding Assistance**

**1)Prompts used:**

1. How to find missing numbers from an unsorted array without using two for loops.

**Responses Received:**

To find missing numbers from an unsorted array, you can use a HashSet to store the elements of the array and then iterate through the range of numbers to check which ones are missing.

**Implementation Details:**

I used the approach as suggested by Copilot. The array is first converted to a HashSet and then scanned once in a for loop from 1 to nums.Length. Missing numbers are identified using .Contains() and added to a result list. This approach eliminates the need for two for loops and keeps the solution time-efficient at O(n) while maintaining clarity.

**Adjustments:**

Added edge case handling for null and empty input arrays to avoid runtime exceptions.If the input array is null, the method returns an empty list and optionally logs a message.This prevents NullReferenceException and ensures graceful handling of unexpected inputs.

**2) Prompt Used:**

1)What are the possible edge cases for sort array by parity

2)What is the difference between null and empty array

**Response Received:**

1. When implementing the SortArrayByParity method, it's important to consider various edge cases to ensure the method handles all possible scenarios correctly. Here are some potential edge cases:

Null Array:

* + The input array is null.

Empty Array:

* + The input array has no elements.

1. Null Array: A null array is an array that has not been initialized. It does not point to any memory location.

Empty Array: An empty array is an array that has been initialized but contains no elements. It points to a valid memory location but has a length of zero.

**Implementation Details:**

Implemented the edge cases as per AI suggestion to check if input array is null or empty.

**6)Prompts used:**

How to find minimum in rotated sorted array

**Response Received:**

To find the minimum element in a rotated sorted array, you can use a modified binary search algorithm. This approach ensures an efficient O(log n) time complexity.

* Use two pointers, left and right, to represent the current search range.
* Calculate the middle index mid.
* Compare the middle element with the rightmost element to determine which half of the array contains the minimum element.
* If the middle element is greater than the rightmost element, the minimum element is in the right half.
* If the middle element is less than or equal to the rightmost element, the minimum element is in the left half.
* Adjust the pointers accordingly and repeat the process until the search range is narrowed down to a single element.

**Implementation Details:**

I followed the AI’s step-by-step logic and implemented a **binary search loop**:

* Initialized left = 0 and right = nums.Length - 1.
* Calculated mid = left + (right - left) / 2 to prevent overflow.
* Compared nums[mid] with nums[right] in each iteration:
  + If nums[mid] > nums[right], moved left to mid + 1.
  + Otherwise, moved right to mid.
* When left equals right, that index holds the minimum value.

**Adjustments:**

* Included checks for null and empty arrays to prevent runtime errors. If the input is invalid, the method returns int.MinValue and displays a clear console message.
* Reused the mid calculation pattern left + (right - left) / 2 to prevent potential integer overflow, especially when dealing with large arrays.