**Use Copilot AI for Coding Assistance**

**Question-1:**

**Prompts Used:** What's the time complexity difference between using a HashSet versus a linear search to find missing elements?

**Responses Received :** When using a HashSet, the time complexity is O(n) because adding all elements to the HashSet takes O(n) time, and checking for the presence of each number from 1 to n takes O(1) per lookup, resulting in an overall O(n) time. However, this approach requires O(n) space complexity for storing the HashSet. In contrast, using a linear search results in O(n²) time complexity because for each number from 1 to n, you perform a linear search through the entire array, though it only requires O(1) space as no additional data structures are needed. The key advantage of using a HashSet is significantly faster performance at the cost of additional memory usage.

**Implementation Details:** I implemented the HashSet approach in my solution as it offers better time complexity. The code creates a HashSet from the input array and then iterates from 1 to the length of the array, checking if each number exists in the HashSet. If a number is not found, it's added to the missing numbers list.

**Adjustments:** I added an edge case check for a null input array at the beginning of the function. This wasn't explicitly mentioned in Copilot's suggestion but is good practice to prevent null reference exceptions.

**Question-2**

**Prompts Used:** What edge cases should I handle when sorting an array by parity?

**Responses Received:** For sorting an array by parity, Copilot highlighted several important edge cases to handle. These include empty arrays (return an empty array), arrays with a single element (return as-is), arrays containing only odd or only even numbers (output should be the same as input), arrays with mixed odd and even numbers (ensure even numbers are moved to the front while odd numbers to the back), null arrays (handle gracefully by returning null or an empty array), arrays with duplicate numbers (ensure duplicates are preserved), and arrays with negative numbers (ensure parity is determined correctly, as numbers like -2 should still be treated as even).

**Implementation Details:** I implemented a solution that handles all these edge cases. The algorithm uses a single pass through the array, moving even numbers to the front by swapping them with elements at the "odd" pointer, which advances whenever an even number is found.

**Adjustments:** I relied on the try-catch block to handle exceptions. I focused on ensuring the algorithm worked correctly for arrays with all even or all odd numbers.

**Question-4**

**Prompts Used:** How should I handle negative numbers when finding maximum product of three numbers?"

**Responses Received:** The AI suggests several edge cases to consider when sorting an array by parity. These include handling empty arrays by returning an empty array, and dealing with arrays containing a single element or only odd/even numbers by returning the array as-is. For mixed odd and even arrays, the AI recommends moving even numbers to the front and odd numbers to the back while maintaining their order. It also suggests graceful handling of null arrays, ensuring that duplicates are preserved, and correctly determining the parity of negative numbers (e.g., treating -2 as even).

**Implementation Details:**  
To ensure the sorting function works effectively, the AI emphasizes incorporating checks for arrays of various lengths, including null and empty arrays. It also highlights the need to follow parity rules correctly, even with negative numbers, and to preserve duplicates to maintain data integrity.

**Adjustments:**  
No major adjustments were needed, as the AI's suggestions are comprehensive and align well with the problem's requirements. Special attention was given to handling negative numbers and duplicates, ensuring the solution is robust across different input scenarios.

**Question-6**

**Prompts Used :**What's the time complexity difference between using a HashSet versus a linear search to find missing elements?"

**Responses Received:** To find the minimum element in a rotated sorted array using binary search, the algorithm is modified by comparing the middle element (nums[mid]) with the rightmost element (nums[right]). If nums[mid] > nums[right], it indicates that the rotation point is to the right of mid, so the minimum element lies in the right half, and the search range is adjusted to left = mid + 1. Otherwise, the minimum element is in the left half, and the search range is updated to right = mid. The search continues until left == right, at which point nums[left] (or nums[right]) is the minimum element.

**Implementation Details:** The AI solution suggests modifying the binary search by updating the search range based on the comparison between nums[mid] and nums[right]. This allows the search to zero in on the rotated portion where the minimum element resides.

**Adjustments:**  
No major modifications were needed to apply the solution, The approach was directly implemented with the standard binary search adjustments based on the rotation logic.