**Prompts Used:**

Analyze all the below 8 functions and guide me whether the code I have written matches the criteria or Do you suggest any modifications so that I can improve?   
Also check the syntax thoroughly.

criteria:  
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.

**Response Received:**

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

• nums.Min() and nums.Max() each take O(n) time.

• The for loop runs from min to max, which in the worst case can be O(n) if the range is large.

• Inside the loop, nums.Contains(i) is called, which takes O(n) time.

Overall, the time complexity is O(n^2) due to the nested loop and Contains method.

• The space complexity is O(n) for storing the missingNumbers list.

The syntax of the function is correct, but the nested loop and Contains method make it inefficient. We can optimize it without changing the data structure by using a boolean array to track the presence of numbers.

**Implementation Details:**

Initially, the code had a time complexity of **O(n^2)** because of the nested loop and the Contains method, which wasn’t the most efficient. It suggested to make it more efficient I optimized the solution by using a boolean array to track which numbers are present. This avoids the need for the nested loop and speeds up the process, reducing the time complexity to **O(n)**.

**Adjustments:**

The only change I made was replacing the nested loop and Contains method with a boolean array. This small adjustment improves performance by bringing the time complexity down from **O(n^2)** to **O(n)**, while keeping the space complexity at **O(n)** for storing the **missingNumbers** list.

**Question 2: Sort Array by Parity**

**Response Received:**

* The function iterates through the array once, which is O(n).
* Adding elements to the lists and combining them are O(n) operations.
* Overall time complexity: O(n)
* The function uses two additional lists (evens and odds), which in the worst case can store all elements of the input array.
* The space complexity is therefore O(n) for the additional lists.
* The code is syntactically correct and logically sound. However, we can make a minor improvement by initializing the lists with the capacity of the input array to avoid potential resizing during additions.

**Implementation Details:**

The function works just fine with a time complexity of **O(n)** because it loops through the array once. Adding elements to the **evens** and **odds** lists and then combining them also takes **O(n)** time. In terms of space complexity, it's **O(n)** since the two extra lists can hold all the elements of the input array in the worst case.

To make the code a little more efficient, I’ve initialized the **evens** and **odds** lists with the same capacity as the input array. This prevents the lists from resizing while adding elements, which can help improve performance slightly.

**Adjustments:**  
The only change I made was initializing the **evens** and **odds** lists with the input array’s capacity. This minor tweak just helps avoid resizing during additions, but the overall time and space complexity are still the same. It's a small improvement for efficiency.

**Question 3: Two Sum**

**Response Received:**

The current implementation has a time complexity of O(n^2). This can be improved to O(n) using a dictionary.

**Implementation Details:**

Even though it was suggested to use a dictionary to improve the solution, I decided to stick with my current implementation. The reason is that the instructions specifically mention we need to follow the given data structure or algorithm recommendations. So, to stay within those guidelines, I’ve chosen not to switch to a different approach, even if it might seem more efficient. The current method still works correctly and aligns with what was asked in the criteria.

**Adjustments:**

No changes were made. Although using a dictionary was suggested for better performance, I stayed with the original approach because the instructions required using the specified data structure and algorithm. The current implementation follows those guidelines and works correctly.

**Question 4: Find Maximum Product of Three Numbers  
Response Received:**

* We maintain three variables (**max1**, **max2**, **max3**) to store the three largest numbers.
* We maintain two variables (**min1**, **min2**) to store the two smallest numbers.
* We iterate through the array once, updating these variables as needed.
* Finally, we calculate the maximum product by comparing the product of the three largest numbers and the product of the largest number with the two smallest numbers.

This approach has a time complexity of O(n), which is more efficient than the original O(n log n) approach.

**Implementation Details:**

I agreed with the suggestion and updated my code. Now, instead of sorting I keep track of the three biggest numbers and the two smallest ones while looping through the array. This helps handle both positive and negative number cases efficiently.

* max1, max2, max3 to track the three largest numbers
* min1, min2 to track the two smallest numbers  
  I iterate through the array once and update these values as needed. At the end, I just compare the product of the top three numbers with the product of the largest and two smallest numbers to get the result maximum of (max1 \* max2 \* max3) and (max1 \* min1 \* min2).

**Adjustments:**

I changed my approach from sorting the array to using a single loop with a few variables to track the required values. This made the solution faster and more efficient while still solving the problem correctly.

**Question 5: Decimal to Binary Conversion**

**Response Received:**

* **Base Case in ConvertToBinary**: The base case **if (n == 0) return "";** is correct but can be simplified for readability.

**Implementation Details:**

A suggested alternative approach was given, but it doesn’t improve efficiency or performance in any meaningful way. Regarding readability, I believe my code is already clear and well-structured, with appropriate commands that make it easy to follow. The current implementation is perfectly readable, and simplifying the base case further wouldn’t add any significant benefit.

**Adjustments:**  
No adjustments are needed. The current implementation is optimal in terms of performance, and there’s no need to change it since the suggested alternative doesn’t provide any real improvements.

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

**Response Received:**

The function uses a binary search approach, which divides the search space in half each iteration. Therefore, the time complexity is O(log n), where n is the number of elements in the array.

The function uses a constant amount of extra space (only a few integer variables), so the space complexity is O(1).

1. The function initializes low and high correctly.

2. The while loop condition low < high is appropriate for binary search.

3. The calculation of middle is correct and avoids potential overflow.

4. The comparison nums[middle] > nums[high] correctly determines which half of the array to search next.

5. The function returns the correct element nums[low] after the loop.

**Implementation Details:**

The function is correct in terms of both time and space complexity, and the syntax is correct. No changes are required.

**Adjustments:**

No adjustments were made because the function is already performing as expected and doesn’t require any improvements.

**Question 7: Palindrome Number**

**Response Received:**

The time complexity of this function is O(d), where d is the number of digits in the input number. This is because the while loop processes each digit exactly once.

The space complexity is O(1) because we are using a constant amount of extra space (for variables originalNumber, reversedNumber, and lastDigit).

• The function correctly handles negative numbers by returning false.

• It correctly reverses the digits of the number and compares the reversed number with the original number.

**Implementation Details:**

No changes are required as the function is already optimal in terms of time and space complexity, and the syntax is correct.

**Adjustments:**

No adjustments were made because the function is already performing as expected and doesn’t require any improvements.

**Question 8: Fibonacci Number**

**Response Received:**

The time complexity of the function is O(n) because it iterates from 2 to n.

The space complexity is O(1) because it uses a constant amount of space to store the variables **previous**, **current**, and **next**.

No changes are required in terms of data structure or logic. The function is already optimal.

**Implementation Details:**

There’s no need for any changes in terms of data structure or logic. The current implementation is already efficient and optimal for this problem.

**Adjustments:**

No adjustments were made because the function is already performing as expected and doesn’t require any improvements.