457. Circular Array Loop

Medium Array Hash Table Two Pointers Leetcode Link

Problem Description

In this problem, you are working with a circular array of non-zero integers. The array is called "circular" because if you move past the last element, you wrap around to the first element, and vice versa. Each value in the array tells you how many steps to move from your current position. A positive value means you move that number of steps forward, while a negative value means you move backward.

The challenge is to determine if there exists a "cycle" in the array. A cycle means that if you start at some index and follow the steps, you eventually return to the starting index after `k` moves, where `k` is greater than 1. Furthermore, all the steps taken during this process should be exclusively positive or exclusively negative, enforcing that the loop goes in a single direction.

Your task is to return `true` if there is such a cycle in the array, otherwise return `false`.

Intuition

To address this problem, we need to consider that a cycle can only exist if we're moving consistently in one direction and eventually end up where we started. This naturally brings the "fast and slow pointers" technique to mind, which is often used for cycle detection in linked lists.

The fast and slow pointers method involves two pointers moving at different speeds, and if there is a cycle, they will eventually meet. We apply the same principle here:

- The `slow` pointer moves one step at a time.
- The `fast` pointer moves two steps at a time.

If `slow` and `fast` meet at the same index, and this index is not the same as the next step (to prevent single-element loops, which aren't considered valid cycles), we have found a cycle.

At each step, we also verify that the direction does not change. If the product of `nums[slow]` and `nums[fast]` is positive, they are either both positive or both negative, thus maintaining a consistent direction. If this product is negative or if we reach an element that is already marked as visited (a value of 0), we do not have a valid cycle from that start point.

For each element, if it does not lead to a cycle, we mark the visited elements as 0 to avoid re-checking them in the future, thereby optimizing our algorithm. This marking also helps to invalidate any non-cycle paths swiftly.

Overall, the algorithm is to iterate over each element and use the fast and slow pointer method to detect a cycle. If any cycle is found, return `true`. After checking all possibilities, if no cycle is found, return `false`.

Solution Approach

The implementation of the solution for detecting a cycle in the circular array follows these main steps:

1. **Array Length Extraction**: We start by obtaining the length `n` of the input array `nums`. This is crucial since we need to calculate the next index correctly within the circular context:

```
n = len(nums)
```

2. **Helper Function for Index Calculation**: Since the array is circular, we define a function named `next(i)` that takes an index `i` and returns the next index we should move to, according to `nums[i]`, and wraps around the array if necessary:

```
def next(i):
    return (i + nums[i]) % n
```

We ensure that the result of the movement remains within the bounds of the array indices by taking the modulo with `n`.

3. **Main Loop to Check for Cycles**: We iterate through each element in the array to check for cycles starting from that index:

```
for i in range(n):
    if nums[i] == 0:  # Skip already marked elements (no cycle from this point)
        continue
```

4. **Fast and Slow Pointers Initialization**: For each starting index, we initiate `slow` and `fast` pointers, which represent the current position of each pointer:

```
slow, fast = i, next(i)
```

5. **Cycle Detection Loop**: Next, we loop to detect cycles using the following conditions:
   - The product of `nums[slow]` and `nums[fast]` must be positive, indicating they move in the same direction.
   - The product of `nums[slow]` and `nums[next(fast)]` must also be positive, ensuring that the fast pointer also continues in the same direction after two moves.

```
while nums[slow] * nums[fast] > 0 and nums[slow] * nums[next(fast)] > 0:
    if slow == fast:  # Pointers meet, indicating a potential cycle
        if slow != next(slow):  # Check to avoid single-length cycle
            return True
        break
```

6. **Marking Elements**: If not a valid cycle, we need to mark elements associated with the failed attempt to find a cycle to prevent re-processing them in future iterations. This is achieved by setting each involved element to `0`:

```
j = i
while nums[j] * nums[next(j)] > 0:
    nums[j] = 0  # Marking the element
    j = next(j)
```

7. **Final Return**: After exhaustively checking all indices, if no cycle is found, the function returns `false`.

This solution leverages the cyclical two-pointer technique to identify cycles and uses in-place marking to improve efficiency by reducing redundant checks. The use of the modulo operator ensures proper index wrapping within the circular array boundaries, and thorough condition checks maintain consistency in direction for cycle validation.

Example Walkthrough

To illustrate the solution approach using an example, let's consider the circular array `nums = [2, -1, 1, 2, 2]`.

Now let's walk through the algorithm step by step:

1. **Array Length Extraction**: The length `n` of the array `nums` is 5.
2. **Helper Function for Index Calculation**: We use the `next(i)` function to determine the subsequent index after taking a step in the array. For instance, `next(0)` would calculate `(0 + nums[0]) % 5`, which equals `2 % 5`, resulting in the next index as 2.
3. **Main Loop to Check for Cycles**: We start with index `i = 0`.
4. **Fast and Slow Pointers Initialization**: At index `i`, `slow` is initiated at 0 and `fast` is initiated at `next(0)`, which is 2.
5. **Cycle Detection Loop**:
   - On the first iteration, `slow = 0` and `fast = 2`. We calculate `nums[slow] * nums[fast]` which is `2 * 1 = 2` (positive, moving in the same direction).
   - `slow` then moves to `next(slow)`, which is 2, and `fast` moves to `next(next(fast))`, first to 4 then wrapped to 1. Both moves are in the forward direction, and `nums[slow]` is still positive. We check `nums[2] * nums[1]` which is `1 * (-1) = -1`, this indicates a change in direction, so we break out of this loop.
6. **Marking Elements**: Elements associated with index `i` are not leading to a valid cycle, so they should be marked. However, since the product of `nums[1]` and `nums[next(1)]` was not positive, we do not proceed with marking in this iteration.
7. **Continuing with the Loop**: We now increment `i` to 1 and continue the process. The array element at index 1 is `-1`, a backward step, both `slow` and `fast` pointers move backward. If at any time the pointers meet and they have traversed more than a single-element loop (by ensuring `slow != next(slow)`), it indicates there's a cycle.

   However, in this example, no cycle will be found, and each element that has been verified not to contribute to a cycle will ultimately be marked as `0` to prevent redundant future checks.
8. **Final Return**: After iterating through the whole array, if no cycle is found, the function returns `false`. For the given example, since a cycle exists when starting at index `0` where `slow` pointer would eventually catch up to the `fast` pointer while maintaining a consistent direction, we would return `true`.

Please note that this example assumes we did not encounter a valid cycle in the first iteration for illustrative purposes. In reality, once a cycle is detected, we would return `true` immediately.

Python Solution

```
class Solution:
    def circularArrayLoop(self, nums: List[int]) -> bool:
        # Get the length of the input list
        length = len(nums)

        # Define a function to find the next index in a circular manner
        def get_next_index(current_index):
            # Calculate the next index considering wrapping around
            return (current_index + nums[current_index]) % length

        # Iterate over all elements in the array
        for i in range(length):
            # Skip if the current element is already marked as 0, indicating it's not part of a loop
            if nums[i] == 0:
                continue

            # Initialize the slow and fast pointers for cycle detection
            slow_pointer = i
            fast_pointer = get_next_index(i)

            # Continue moving pointers while the signs of the elements indicate a potential loop
            # This also ensures that we are not mixing cycles of different directions
            while nums[slow_pointer] * nums[fast_pointer] > 0 and nums[slow_pointer] * nums[get_next_index(fast_pointer)] > 0:
                # If the slow and fast pointers meet, a cycle is detected
                if slow_pointer == fast_pointer:
                    # Check to ensure the loop is longer than 1 element
                    if slow_pointer != get_next_index(slow_pointer):
                        return True
                    # If the loop is just one element, break and mark it as non-looping
                    break

                # Move slow pointer by one step and fast pointer by two steps
                slow_pointer = get_next_index(slow_pointer)
                fast_pointer = get_next_index(get_next_index(fast_pointer))

            # Mark all visited elements as 0 to avoid revisiting and repeated calculations
            # This process will also ensure elimination of non-loop elements
            j = i
            while nums[j] * nums[get_next_index(j)] > 0:
                next_index = get_next_index(j)
                nums[j] = 0
                j = next_index

        # If no loop is found, return False
        return False
```

Java Solution

```
class Solution {
    // Store the length of the given array
    private int[] nums; // The given array

    // Method to check if the array contains a cycle that meets certain conditions
    public boolean circularArrayLoop(int[] nums) {
        // Initialize the array length and initialize the arrayLength with the length of nums
        this.nums = nums; // Assign the nums array to the instance variable
        int arrayLength = nums.length; // Get the length of nums
        for (int i = 0; i < arrayLength; ++i) {
            // Skip if the current element is 0 as it's already considered non-cyclic
            if (nums[i] == 0) {
                continue;
            }
            // Use a slow and fast pointers approach to find a cycle
            int slow = i;
            int fast = getNextIndex(i);
            // Continue to advance the pointers until the product of the adjacent elements is positive,
            // which indicates they move in the same direction
            while (nums[slow] * nums[fast] > 0 && nums[slow] * nums[getNextIndex(fast)] > 0) {
                if (slow == fast) {
                    // If the slow and fast pointers meet, check if the cycle length is greater than 1
                    if (slow != getNextIndex(slow)) {
                        return true; // A cycle that meets the conditions is found
                    }
                    break; // The cycle length is 1, so break out of the loop
                }
                // Move the slow pointer by one and the fast pointer by two
                slow = getNextIndex(slow);
                fast = getNextIndex(getNextIndex(fast));
            }
            // Reset all elements in the detected cycle to 0 to mark them non-cyclic
            int j = i;
            while (nums[j] * nums[getNextIndex(j)] > 0) {
                nums[j] = 0;
                j = getNextIndex(j);
            }
        }
        // No valid cycle found, return false
        return false;
    }

    // Helper method to get the next array index taking into account wrapping of the array
    // and the current item value (handles negative indices as well)
    private int getNextIndex(int i) {
        // Calculate the next index based on the current index and its value in the array.
        // The '+ arrayLength' ensures we stay within array bounds
        return (i + nums[i] % arrayLength + arrayLength) % arrayLength;
    }
}
```

C++ Solution

```
class Solution {
public:
    // Check if the array contains a cycle that meets certain criteria
    bool circularArrayLoop(vector<int>& nums) {
        int arrayLength = nums.size(); // Get the size of the array
        // Iterate over the array to find a cycle
        for (int i = 0; i < arrayLength; ++i) {
            if (nums[i] == 0) continue; // Skip elements that are already marked as 0 (visited)

            // Use two pointers: 'slow' and 'fast' to detect cycles
            int slow = i;
            int fast = getNextIndex(nums, i);

            // Keep advancing 'slow' by one step and 'fast' by two steps
            // Continue looping as long as the direction (sign) of the numbers is the same
            while (nums[slow] * nums[fast] > 0 && nums[slow] * nums[getNextIndex(nums, fast)] > 0) {
                if (slow == fast) { // Found a cycle
                    // Check if the cycle is more than one element
                    if (slow != getNextIndex(nums, slow)) {
                        return true;
                    }
                    // If not, break and move on to next element in the array
                    break;
                }
                // Move 'slow' one step forward
                slow = getNextIndex(nums, slow);
                // Move 'fast' two steps forward
                fast = getNextIndex(nums, getNextIndex(nums, fast));
            }

            // Mark all visited elements in the cycle as 0
            int j = i;
            while (nums[j] * nums[getNextIndex(nums, j)] > 0) {
                int nextIndex = getNextIndex(nums, j);
                nums[j] = 0;
                j = nextIndex;
            }
        }

        // Return false if no qualifying cycle is found
        return false;
    }

    // Helper function to get the next index in the circular array
    int getNextIndex(vector<int>& nums, int i) {
        int arrayLength = nums.size();
        // Calculate the next index accounting for wrapping around the array
        return (i + nums[i] % n + n) % n; // The double modulo ensures a positive result
    }
};
```

Typescript Solution

```
// The array of numbers
let nums: number[];

// Function to check if the array contains a cycle that meets certain criteria
function circularArrayLoop(nums: number[]): boolean {
    const n = nums.length; // Get the size of the array
    // Iterate over the array to find a cycle
    for (let i = 0; i < n; ++i) {
        if (nums[i] == 0) continue; // Skip elements that are already marked as 0 (visited)

        // Initialize two pointers: 'slow' and 'fast' to detect cycles
        let slow = i;
        let fast = getNextIndex(nums, i);

        // Keep advancing 'slow' by one step and 'fast' by two steps
        // Continue looping as long as the direction (sign) of the numbers is the same
        while (nums[slow] * nums[fast] > 0 && nums[slow] * nums[getNextIndex(nums, fast)] > 0) {
            if (slow == fast) { // Found a cycle
                // Check if the cycle is longer than one element
                if (slow != getNextIndex(nums, slow)) {
                    return true;
                }
                // If not, break and move on to next element in the array
                break;
            }
            // Move 'slow' one step forward
            slow = getNextIndex(nums, slow);
            // Move 'fast' two steps forward
            fast = getNextIndex(nums, getNextIndex(nums, fast));
        }

        // Mark all visited elements in the cycle as 0
        let j = i;
        while (nums[j] * nums[getNextIndex(nums, j)] > 0) {
            const nextIndex = getNextIndex(nums, j);
            nums[j] = 0;
            j = nextIndex;
        }
    }

    // Return false if no qualifying cycle is found
    return false;
};

// Helper function to get the next index in the circular array
function getNextIndex(nums: number[], i: number): number {
    const n = nums.length;
    // Calculate the next index accounting for wrapping around the array
    // The double modulo ensures a positive result
    return ((i + nums[i]) % n + n) % n;
};
```

Time and Space Complexity

The given Python code defines a method for detecting a cycle in a circular array. The cycle must follow certain rules – it cannot consist of a single element looping to itself, and it must maintain a consistent direction (all elements in the cycle are either all positive or all negative).

Time Complexity:

The time complexity of this method is $O(n)$, where n is the length of the array. This results from the fact that each element is visited at most twice – once by the slow pointer and once by the fast pointer. Even though there are nested loops, the inner loop executes a maximum of two times for each element: once by the slow pointer and once by the fast pointer (since we nullify elements once visited to avoid revisiting). Thus, the `while` loops do not multiply the complexity, but rather, each contributes to the linear visit of each element.

Space Complexity:

The space complexity is $O(1)$ since the algorithm only uses a fixed amount of extra space. Additional variables such as `slow`, `fast`, `i`, and `j` are used for indexing, and these do not scale with the input size. The computation is done in place, and the input list is modified directly without using any extra space proportional to the input size.