457. Circular Array Loop

`Medium` `Array` `Hash Table` `Two Pointers`

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 i moves, where i 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-cyclic 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:

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

2. **Helper Function for Index Calculation:** Since the array is circular, we define a function named next() 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:

```
1  def next(i):
2      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:

```
1  for i in range(n):
2      if nums[i] == 0:  # Skip already marked elements (no cycle from this point)
3          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:

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

5. **Cycle Detection Loop:** Next, we loop to detect cycles using the following conditions:

```
1  while nums[slow] * nums[fast] > 0 and nums[slow] * nums[next(fast)] > 0:
2      if slow == fast:  # Pointers met, indicating a potential cycle
3          if slow != next(slow):  # Check to avoid single-step cycle
4              return True
5          break
```

   - 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.

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:

```
1  j = i
2  while nums[j] * nums[next(j)] > 0:
3      nums[j] = 0  # Marking the element
4      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() 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 0, 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 0 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, so 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 3 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

```python
1  class Solution:
2      def circularArrayLoop(self, nums: List[int]) -> bool:
3          # Get the length of the input list
4          length = len(nums)
5
6          # Define a function to find the next index in a circular manner
7          def get_next_index(current_index):
8              # Calculate the next index considering wrapping around
9              return (current_index + nums[current_index]) % length
10
11         # Iterate over all elements in the array
12         for i in range(length):
13             # Skip if the current element is already marked as 0, indicating it's not part of a loop
14             if nums[i] == 0:
15                 continue
16
17             # Initialize the slow and fast pointers for cycle detection
18             slow_pointer = i
19             fast_pointer = get_next_index(i)
20
21             # Continue moving pointers while the signs of the elements indicate a potential loop
22             # This also ensures that we are not moving single cycle of different directions
23             while nums[slow_pointer] * nums[fast_pointer] > 0 and nums[get_next_index(fast_pointer)] > 0:
24                 # If the slow and fast pointers meet, a cycle is detected
25                 if slow_pointer == fast_pointer:
26                     # Check to ensure the loop is larger than 1 element
27                     if slow_pointer != get_next_index(slow_pointer):
28                         return True
29                     # If the loop is just one element, break and mark it as non-looping
30                     break
31
32                 # Move slow pointer by one step and fast pointer by two steps
33                 slow_pointer = get_next_index(slow_pointer)
34                 fast_pointer = get_next_index(get_next_index(fast_pointer))
35
36             # Mark all visited elements as 0 to avoid revisiting and repeated calculations
37             # This process will also ensure elimination of non-loop elements
38             index = i
39             while nums[index] * nums[get_next_index(index)] > 0:
40                 next_index = get_next_index(index)
41                 nums[index] = 0
42                 index = next_index
43
44         # If no loop is found, return False
45         return False
```

Java Solution

```java
1  class Solution {
2      private int arrayLength; // The length of the given array
3      private int[] nums; // The given array
4
5      // Method to check if the array contains a cycle that meets certain conditions
6      public boolean circularArrayLoop(int[] nums) {
7          arrayLength = nums.length; // Initialize the arrayLength with the length of nums
8          this.nums = nums; // Assign the nums array to the instance variable
9          // Loop through each element in the array
10         for (int i = 0; i < arrayLength; ++i) {
11             // Skip if the current element is already considered non-cyclic
12             if (nums[i] == 0) {
13                 continue;
14             }
15             // Use a slow and fast pointers approach to find a cycle
16             int slow = i;
17             int fast = getNextIndex(i);
18             // Continue to advance the pointers until the product of the adjacent elements is positive,
19             // which indicates they move in the same direction
20             while (nums[slow] * nums[fast] > 0 && nums[slow] * nums[getNextIndex(fast)] > 0) {
21                 if (slow == fast) {
22                     // If both pointers meet, check if the cycle length is greater than 1
23                     if (slow != getNextIndex(slow)) {
24                         return true; // A cycle that meets the conditions is found
25                     }
26                     break; // The cycle length is 1, so break out of the loop
27                 }
28                 // Move the slow pointer by one and the fast pointer by two
29                 slow = getNextIndex(slow);
30                 fast = getNextIndex(getNextIndex(fast));
31             }
32             // Reset all elements in the detected cycle to 0 to mark them non-cyclic
33             int j = i;
34             while (nums[j] * nums[getNextIndex(j)] > 0) {
35                 int next = getNextIndex(j);
36                 nums[j] = 0;
37                 j = next;
38             }
39         }
40         // No valid cycle found, return false
41         return false;
42     }
43
44     // Helper method to get the next array index taking into account wrapping of the array
45     // and the current item value (handles negative indices as well)
46     private int getNextIndex(int i) {
47         // Calculate the next index based on the current index and its value in the array,
48         // The result is wrapped to stay within array bounds
49         return (i + nums[i] % arrayLength + arrayLength) % arrayLength;
50     }
51 }
```

C++ Solution

```cpp
1  class Solution {
2  public:
3      // Check if the array contains a cycle that meets certain criteria
4      bool circularArrayLoop(vector<int>& nums) {
5          int n = nums.size(); // Get the size of the array
6          // Iterate over the array to find a cycle
7          for (int i = 0; i < n; ++i) {
8              if (nums[i] == 0) continue; // Skip elements that are already marked as 0 (visited)
9
10             // Use two pointers: 'slow' and 'fast' to detect cycles
11             int slow = i;
12             int fast = getNextIndex(nums, i);
13
14             // Keep advancing 'slow' by one step and 'fast' by two steps
15             // Continue looping as long as the direction (sign) of the numbers is the same
16             while (nums[slow] * nums[fast] > 0 && nums[slow] * nums[getNextIndex(nums, fast)] > 0) {
17                 if (slow == fast) {
18                     // If cycle length is 1 (larger than one element
19                     if (slow != getNextIndex(nums, slow)) {
20                         return true;
21                     }
22                     // If not, break and move on to next element in the array
23                     break;
24                 }
25                 // Move 'slow' one step forward
26                 slow = getNextIndex(nums, slow);
27                 // Move 'fast' two steps forward
28                 fast = getNextIndex(nums, getNextIndex(nums, fast));
29             }
30
31             // Mark all visited elements in the cycle as 0
32             int j = i;
33             while (nums[j] * nums[getNextIndex(nums, j)] > 0) {
34                 int nextIndex = getNextIndex(nums, j);
35                 nums[j] = 0;
36                 j = nextIndex;
37             }
38         }
39         // Return false if no qualifying cycle is found
40         return false;
41     }
42
43     // Helper function to get the next index in the circular array
44     int getNextIndex(vector<int>& nums, int i) {
45         int n = nums.size();
46         // Calculate the next index accounting for wrapping around the array
47         return ((i + nums[i]) % n + n) % n; // The double modulo ensures a positive result
48     }
49 };
```

Typescript Solution

```typescript
1  // The array of numbers
2  let nums: number[];
3
4  // Function to check if the array contains a cycle that meets certain criteria
5  function circularArrayLoop(numbers: number[]): boolean {
6      const n = nums.length; // Get the size of the array
7      // Iterate over the array to find a cycle
8      for (let i = 0; i < n; ++i) {
9          if (nums[i] === 0) continue; // Skip elements that are already marked as 0 (visited)
10
11         // Initialize two pointers 'slow' and 'fast' to detect cycles
12         let slow = i;
13         let fast = getNextIndex(nums, i);
14
15         // Keep advancing 'slow' by one step and 'fast' by two steps
16         // Continue looping as long as the direction (sign) of the numbers is the same
17         while (nums[slow] * nums[fast] > 0 && nums[slow] * nums[getNextIndex(nums, fast)] > 0) {
18             if (slow === fast) {
19                 // If cycle is found, check if it's larger than one element
20                 if (slow != getNextIndex(nums, slow)) {
21                     return true;
22                 }
23                 // If not, break and move on to next element in the array
24                 break;
25             }
26             // Move 'slow' one step forward
27             slow = getNextIndex(nums, slow);
28             // Move 'fast' two steps forward
29             fast = getNextIndex(nums, getNextIndex(nums, fast));
30         }
31
32         // Mark all visited elements in the cycle as 0
33         let j = i;
34         while (nums[j] * nums[getNextIndex(nums, j)] > 0) {
35             const nextIndex = getNextIndex(nums, j);
36             nums[j] = 0;
37             j = nextIndex;
38         }
39     }
40     // Return false if no qualifying cycle is found
41     return false;
42 }
43
44 // Helper function to get the next index in the circular array
45 function getNextIndex(numbers: number[], i: number): number {
46     const n = nums.length;
47     // Calculate the next index accounting for wrapping around the array
48     // The double modulo ensures a positive result
49     return ((i + nums[i]) % n + n) % n;
50 }
```

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.