Solution Review: Problem Challenge 3

We'll cover the following

Cycle in a Circular Array (hard)

We are given an array containing positive and negative numbers. Suppose the array contains a number 'M' at a particular index. Now, if 'M' is positive we will move forward 'M' indices and if 'M' is negative move backwards 'M' indices. You should assume that the **array is circular** which means two things:

1. If, while moving forward, we reach the end of the array, we will jump to the first element to continue the movement.
2. If, while moving backward, we reach the beginning of the array, we will jump to the last element to continue the movement.

Write a method to determine **if the array has a cycle**. The cycle should have more than one element and should follow one direction which means the cycle should not contain both forward and backward movements.

Example 1:

```
Input: [1, 2, -1, 2, 2]
Output: true
Explanation: The array has a cycle among indices: 0 -> 1 -> 3 -> 0
```

Example 2:

```
Input: [2, 2, -1, 2]
Output: true
Explanation: The array has a cycle among indices: 1 -> 3 -> 1
```

Example 3:

```
Input: [2, 1, -1, -2]
Output: false
Explanation: The array does not have any cycle.
```

Solution

This problem involves finding a cycle in the array and, as we know, the **Fast & Slow pointer** method is an efficient way to do that. We can start from each index of the array to find the cycle. If a number does not have a cycle we will move forward to the next element. There are a couple of additional things we need to take care of:

1. As mentioned in the problem, the cycle should have more than one element. This means that when we move a pointer forward, if the pointer points to the same element after the move, we have a one-element cycle. Therefore, we can finish our cycle search for the current element.

2. The other requirement mentioned in the problem is that the cycle should not contain both forward and backward movements. We will handle this by remembering the direction of each element while searching for the cycle. If the number is positive, the direction will be forward and if the number is negative, the direction will be backward. So whenever we move a pointer forward, if there is a change in the direction, we will finish our cycle search right there for the current element.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```python
def circular_array_loop_exists(arr):
  for i in range(len(arr)):
    is_forward = arr[i] >= 0  # if we are moving forward or not
    slow, fast = i, i

    # if slow or fast becomes '-1' this means we can't find cycle for this number
    while True:
      # move one step for slow pointer
      slow = find_next_index(arr, is_forward, slow)
      # move one step for fast pointer
      fast = find_next_index(arr, is_forward, fast)
      if (fast != -1):
        # move another step for fast pointer
        fast = find_next_index(arr, is_forward, fast)
      if slow == -1 or fast == -1 or slow == fast:
        break

    if slow != -1 and slow == fast:
      return True

  return False


def find_next_index(arr, is_forward, current_index):
  direction = arr[current_index] >= 0

  if is_forward != direction:
    return -1  # change in direction, return -1

  next_index = (current_index + arr[current_index]) % len(arr)

  # one element cycle, return -1
  if next_index == current_index:
    next_index = -1

  return next_index


def main():
  print(circular_array_loop_exists([1, 2, -1, 2, 2]))
  print(circular_array_loop_exists([2, 2, -1, 2]))
  print(circular_array_loop_exists([2, 1, -1, -2]))


main()

```

Run | Save | Reset

Time Complexity

The above algorithm will have a time complexity of $O(N^2)$ where 'N' is the number of elements in the array. This complexity is due to the fact that we are iterating all elements of the array and trying to find a cycle for each element.

Space Complexity

The algorithm runs in constant space $O(1)$.

An Alternate Approach

In our algorithm, we don't keep a record of all the numbers that have been evaluated for cycles. We know that all such numbers will not produce a cycle for any other instance as well. If we can remember all the numbers that have been visited, our algorithm will improve to $O(N)$ as, then, each number will be evaluated

for cycles only once. We can keep track of this by creating a separate array however the space complexity of our algorithm will increase to $O(N)$.

Back
Problem Challenge 3

Next
Introduction

Completed