Solution: Circular Array Loop

Let's solve the Circular Array Loop problem using the Fast and Slow Pointers pattern.

We'll cover the following

Statement

An input array, `nums` containing non-zero integers, is given, where the value at each index represents the number of places to skip forward (if the value is positive) or backward (if the value is negative). When skipping forward or backward, wrap around if you reach either end of the array. For this reason, we are calling it a circular array. Determine if this circular array has a cycle. A cycle is a sequence of indices in the circular array characterized by the following:

- The same set of indices is repeated when the sequence is traversed in accordance with the aforementioned rules.
- The length of the sequence is at least two.
- The loop must be in a single direction, forward or backward.

It should be noted that a cycle in the array does not have to originate at the beginning. A cycle can begin from any point in the array.

Constraints:

- $1 \leq$ `nums.length` $\leq 10^4$
- $-5000 \leq$ `nums[i]` ≤ 5000
- `nums[i]` $! = 0$

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which to follow based on considerations such as time complexity and implementation constraints.

Naive approach

The naive approach is to traverse the whole array and check for each element whether we can form a cycle starting from each element or not. We'll run a loop on every array element and keep track of the visited element using an additional array. We'll check the condition for both the forward and backward cycles. If the direction of the cycle changes at any point, we'll come out of that loop and continue verifying the loop condition for the remaining elements.

We get the required solution, but at what cost? The time complexity is $O(n^2)$ as we run a loop on every array element. The space complexity is $O(n)$ because we use extra space to keep track of the visited elements. If our array contains thousands of elements, we'll need a lot of our memory resources to solve this problem.

Optimized approach using fast and slow pointers

The circular array loop can be found efficiently using no extra memory with the fast and slow pointers technique. The algorithm uses fast and slow pointers that move through the array following the values at the array indexes. We move the fast pointer twice as fast as the slow pointer. If the pointers reach the same index, a cycle is detected. This is because the fast pointer covers twice the distance as the slow pointer does in each iteration, so the fast pointer guarantees to meet the slow pointer if the cycle exists. If the loop is found, we'll return TRUE.

If the values at the array indexes of the slow and fast pointers have different signs, i.e., one pointer is pointing to a positive value, and the other is pointing to a negative value, the loop cannot exist. Additionally, if moving a pointer takes it to the current position again, it forms a loop with one element. Since we are not considering it, we will skip it. If any of these conditions are TRUE, we'll move to the next element.

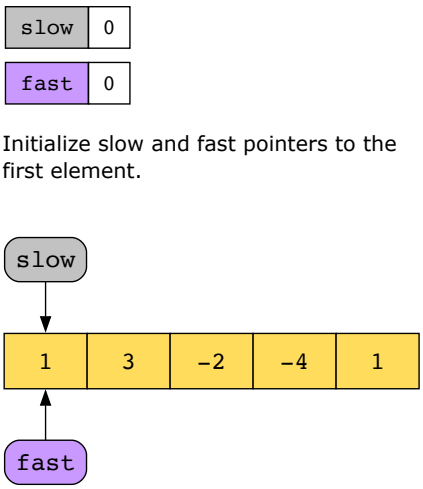
We start the algorithm by defining the following functions:

- `nextStep(pointer, value, size)`: This function will be used to find the next location of our pointer based on the value present at the current location. It will do this by adding the current index and the value present at that index and taking the modulus with the size of the array. In case of the negative value obtained, it will add size to it to make it a valid index value.
- `isNotCycle(nums, prevDirection, pointer)`: This function will return TRUE if the cycle is not possible. It will return FALSE otherwise. A cycle is not possible if either of the two following conditions is true:
    - If both pointers have different directions, i.e., one pointer's value is negative, and the other pointer's value is positive.
    - If the absolute value of `nums[i]` is equal to the length of the array, which will return to the same location. This means we have a loop with just one element, which is not the condition we're trying to detect.

We iterate through each index `i` of the array `nums`, and in each iteration, we set both `fast` and `slow` pointers to the current index. The variable `forward` determines the direction of the loop. We set it TRUE if the current element is positive and set FALSE if the current element is negative.

- Then, we iterate a loop till we find the result and perform the following actions:
    - Move the `slow` pointer `nums[i]` steps forward if its value is positive and move backward if the value of `nums[i]` is negative by calling the `nextStep` function.
    - Now, after moving the `slow` pointer, check whether or not the cycle is possible with existing pointers locations using the `isNotCycle` function.
    - If the cycle is not possible, we don't need to continue with the locations of current pointers. We break the inner loop and set both `slow` and `fast` pointers at the next index, `i`, and repeat the process.
    - Now, move the `fast` pointer twice, and after every move, check whether the cycle is possible or not. If it is not possible, break the loop and set both `slow` and `fast` pointers at the next index `i` and repeat the process.
    - After moving both `slow` and `fast` pointers, if they both meet at the same location, return TRUE, since it means we have found a cycle.
- If the cycle is not found after traversing all the elements, return FALSE.

Here's the demonstration of the steps above:



Let's look at the code for this solution below:

Java

```java
import java.util.*;

class CircularArrayLoop {

  public static boolean circularArrayLoop(int[] nums) {
  int size = nums.length;
  // Iterate through each index of the array 'nums'.
  for (int i = 0; i < size; i++) {
    // Set slow and fast pointer at current index value.
    int slow = i, fast = i;
    // Set true in 'forward' if element is positive, set false otherwise.
    boolean forward = nums[i] > 0;

    while (true) {
      // Move slow pointer to one step.
      slow = nextStep(slow, nums[slow], size);
      // If cycle is not possible, break the loop and start from next element.
      if (isNotCycle(nums, forward, slow))
        break;

      // First move of fast pointer.
      fast = nextStep(fast, nums[fast], size);
      // If cycle is not possible, break the loop and start from next element.
      if (isNotCycle(nums, forward, fast))
        break;

      // Second move of fast pointer.
      fast = nextStep(fast, nums[fast], size);
```

Solution summary

To recap, the solution to this problem can be divided into the following parts:

1. Move the slow pointer x steps forward/backward, where x is the value at the i^{th} index of the array.
2. Move the fast pointer x steps forward/backward, where x is the value at i^{th} index. Then, move fast pointer y steps forward/backward, where y is the value at x^{th} index.

3. Return TRUE when both pointers meet at the same point.
4. If the direction changes after moving the slow or fast pointer or taking a step return to the same location, then follow the steps above for the next element of the array.
5. Return FALSE if we have traversed every element of the array without finding a loop.

Time complexity

The time complexity of the solution is $O(n^2)$, where n is the number of elements in the array.

Space complexity

The algorithm has constant space complexity, that is $O(1)$.

Back

Circular Array Loop

Next

Find The Duplicate Nu...

Mark as Completed