<https://leetcode.com/problems/circular-array-loop/description/>

You are playing a game involving a circular array of non-zero integers nums. Each nums[i] denotes the number of indices forward/backward you must move if you are located at index i:

If nums[i] is positive, move nums[i] steps forward, and

If nums[i] is negative, move nums[i] steps backward.

Since the array is circular, you may assume that moving forward from the last element puts you on the first element, and moving backwards from the first element puts you on the last element.

A cycle in the array consists of a sequence of indices seq of length k where:

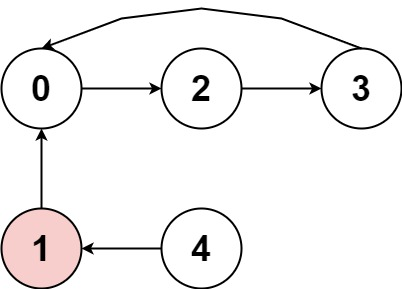
Following the movement rules above results in the repeating index sequence seq[0] -> seq[1] -> ... -> seq[k - 1] -> seq[0] -> ...

Every nums[seq[j]] is either all positive or all negative.

k > 1

Return true if there is a cycle in nums, or false otherwise.

**Example 1:**

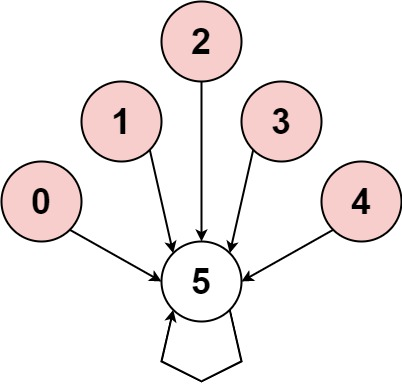


**Input:** nums = [2,-1,1,2,2]

**Output:** true

**Explanation:** The graph shows how the indices are connected. White nodes are jumping forward, while red is jumping backward.We can see the cycle 0 --> 2 --> 3 --> 0 --> ..., and all of its nodes are white (jumping in the same direction).

**Example 2:**

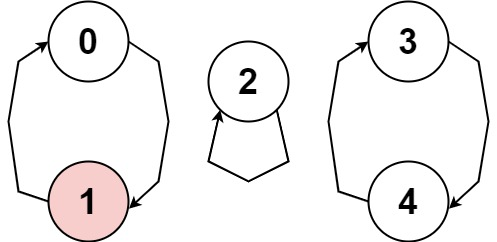


**Input:** nums = [-1,-2,-3,-4,-5,6]

**Output:** false

**Explanation:** The graph shows how the indices are connected. White nodes are jumping forward, while red is jumping backward.The only cycle is of size 1, so we return false.

**Example 3:**



**Input:** nums = [1,-1,5,1,4]

**Output:** true

**Explanation:** The graph shows how the indices are connected. White nodes are jumping forward, while red is jumping backward.We can see the cycle 0 --> 1 --> 0 --> ..., and while it is of size > 1, it has a node jumping forward and a node jumping backward, so **it is not a cycle**.We can see the cycle 3 --> 4 --> 3 --> ..., and all of its nodes are white (jumping in the same direction).

**Constraints:**

1 <= nums.length <= 5000

-1000 <= nums[i] <= 1000

nums[i] != 0

**Follow up:** Could you solve it in O(n) time complexity and O(1) extra space complexity?

**Attempt 1: 2023-03-06**

**Solution 1: Two Pointers (180 min)**

**Style 1: Start both 'slow' and 'fast' with same value as conventional "Floyd's Cycle Detection Algorithm" style**

**Wrong Solution**

**Test out by nums = {1,-1,5,1,4}, expected = true, output = false**

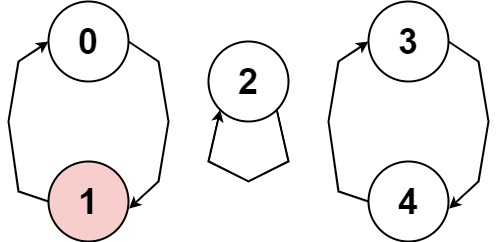
**The problem happen for nums[2] = 5 with the 'getNextIndex()' method:**

**int slow = 2, fast = 2 -> nums[slow] = nums[2] = 5, nums[getNextIndex(nums, 2)] = nums[2] = 5, nums[getNextIndex(nums, getNextIndex(nums, 2))] = nums[getNextIndex(nums, 2)] = nums[2] = 5 -> 5 \* 5 > 0 && 5 \* 5 > 0, go into while loop.**

**slow = getNextIndex(nums, slow) = getNextIndex(nums, 2) = 2**

**fast = getNextIndex(nums, getNextIndex(nums, 2)) = getNextIndex(nums, 2) = 2**

**slow == fast -> then find 'slow == getNextIndex(nums, slow)', return false, but from the graph we know we are talking about the self-circle as '2' here which should ignore as size == 1, but below code style not filter out this case which should ignore and return false, the real circle should return true is relate to 3 -> 4 -> 3**



class Solution {

    int len = 0;

    public boolean circularArrayLoop(int[] nums) {

        len = nums.length;

        for(int i = 0; i < len; i++) {

            int slow = i;

            int fast = i;

            // Condition inside ensures that all elements are positive or all negative

            // 2 negatives / 2 positives multiplied will give positive as output

            // we need to ensure all elements slow, next(fast), next(next(fast)) are same sign

            // we don't need to check next(slow) because it is some previous fast (so already checked)

            while(nums[slow] \* nums[getNextIndex(nums, fast)] > 0

                    && nums[slow] \* nums[getNextIndex(nums, getNextIndex(nums, fast))] > 0) {

                // one step for slow

                slow = getNextIndex(nums, slow);

                // two steps for fast

                fast = getNextIndex(nums, getNextIndex(nums, fast));

                if(slow == fast) {

                    // If both pointers meet, check if the cycle length is greater than 1

                    if(slow == getNextIndex(nums, slow)) {

                        // Cycle length equal to 1 not match requirement

                        return false;

                    }

                    return true;

                }

            }

        }

        // No loop found

        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[] nums, int i) {

        // Calculate the next index based on the current index and its value in the array.

        // The result is wrapped to stay within array bounds

        return (i + nums[i] % len + len) % len;

    }

}

**Now the question is why below example correct solution can handle the nums[2] = 5 with the same 'getNextIndex()' method ?**

**Refer to** <https://algo.monster/liteproblems/457>

class Solution {

    private int arrayLength; // 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) {

        arrayLength = nums.length; // Initialize the arrayLength with the length of nums

        this.nums = nums; // Assign the nums array to the instance variable

        // Loop through each element in the array

        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 both 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 result is wrapped to stay within array bounds

        return (i + nums[i] % arrayLength + arrayLength) % arrayLength;

    }

}

**The trick happens on below code section, the process of before below section is all the same as wrong solution, the only difference is it won't directly return false when slow == fast (when i = 2 we hit this logic), it just break out and touch the later tricky section.**

if (slow == fast) {

// If both 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

}

**After we break out, it touch the tricky section below which mark nums[2] = 5 to 0, then nums = {1,-1,5,1,4} change to {1,-1,0,1,4}, which exactly the same effect as how in correct solution we initially update nums[i] to nums[i] % len, which also change nums = {1,-1,5,1,4} to {1,-1,0,1,4}**

// 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);

}

**So in conclusion, we have two ways to resolve the nums[2] = 5 handling problem:**

**1.nums[i] update to nums[i] = nums[i] % len, then return false when 'slow == getNextIndex(slow)'**

**2.break when 'slow == getNextIndex(slow)', then handle by "Reset all elements in the detected cycle to 0 to mark them non-cyclic"**

**And below we will roll out two different styles:**

**Correct Solution**

**Style 1: nums[i] update to nums[i] = nums[i] % len, then return false when 'slow == getNextIndex(slow)'**

class Solution {

int len = 0;

public boolean circularArrayLoop(int[] nums) {

len = nums.length;

// Steps of size greater than n circle back, so taking remainder

// And it will help avoid nums[i] = 0 causing self-circle problem

for(int i = 0; i < len; i++) {

nums[i] = nums[i] % len;

}

for(int i = 0; i < len; i++) {

int slow = i;

int fast = i;

// Condition inside ensures that all elements are positive or all negative

// 2 negatives / 2 positives multiplied will give positive as output

// we need to ensure all elements slow, next(fast), next(next(fast)) are same sign

// we don't need to check next(slow) because it is some previous fast (so already checked)

while(nums[slow] \* nums[getNextIndex(nums, fast)] > 0

&& nums[slow] \* nums[getNextIndex(nums, getNextIndex(nums, fast))] > 0) {

// One step for slow

slow = getNextIndex(nums, slow);

// Two steps for fast

fast = getNextIndex(nums, getNextIndex(nums, fast));

if(slow == fast) {

// If both pointers meet, check if the cycle length is greater than 1

if(slow == getNextIndex(nums, slow)) {

// Cycle length equal to 1 not match requirement

return false;

}

return true;

}

}

}

// No loop found

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[] nums, int i) {

// Calculate the next index based on the current index and its value in the array.

// The result is wrapped to stay within array bounds

return (i + nums[i] % len + len) % len;

}

}

Time Complexity: O(N)

Space Complexity: O(1)

**Correct Solution with 'visited' array to accelerate the speed to 1ms**

class Solution {

    int len = 0;

    public boolean circularArrayLoop(int[] nums) {

        len = nums.length;

        boolean[] visited = new boolean[len];

        // Steps of size greater than n circle back, so taking remainder

        // And it will help avoid nums[i] = 0 causing self-circle problem

        for(int i = 0; i < len; i++) {

            nums[i] = nums[i] % len;

        }

        for(int i = 0; i < len; i++) {

            int slow = i;

            int fast = i;

            // Condition inside ensures that all elements are positive or all negative

            // 2 negatives / 2 positives multiplied will give positive as output

            // we need to ensure all elements slow, next(fast), next(next(fast)) are same sign

            // we don't need to check next(slow) because it is some previous fast (so already checked)

            while(nums[slow] \* nums[getNextIndex(nums, fast)] > 0

                && nums[slow] \* nums[getNextIndex(nums, getNextIndex(nums, fast))] > 0) {

                // One step for slow

                slow = getNextIndex(nums, slow);

                // Two steps for fast

                fast = getNextIndex(nums, getNextIndex(nums, fast));

                // Already visited, no point running slow-fast algorithm again

                if(visited[slow]) {

                    break;

                }

                visited[slow] = true;

                if(slow == fast) {

                    // If both pointers meet, check if the cycle length is greater than 1

                    if(slow == getNextIndex(nums, slow)) {

                        // Cycle length equal to 1 not match requirement

                        return false;

                    }

                    return true;

                }

            }

        }

        // No loop found

        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[] nums, int i) {

        // Calculate the next index based on the current index and its value in the array.

        // The result is wrapped to stay within array bounds

        return (i + nums[i] % len + len) % len;

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Style 2: break when 'slow == getNextIndex(slow)', then handle by "Reset all elements in the detected cycle to 0 to mark them non-cyclic"**

class Solution {

    int len = 0;

    public boolean circularArrayLoop(int[] nums) {

        len = nums.length;

        for(int i = 0; i < len; i++) {

            int slow = i;

            int fast = i;

            // Condition inside ensures that all elements are positive or all negative

            // 2 negatives / 2 positives multilplied will give positive as output

            // we need to ensure all elements slow, next(fast), next(next(fast)) are same sign

            // we don't need to check next(slow) because it is some previous fast (so already checked)

            while(nums[slow] \* nums[getNextIndex(nums, fast)] > 0

                && nums[slow] \* nums[getNextIndex(nums, getNextIndex(nums, fast))] > 0) {

                // One step for slow

                slow = getNextIndex(nums, slow);

                // Two steps for fast

                fast = getNextIndex(nums, getNextIndex(nums, fast));

                if(slow == fast) {

                    // If both pointers meet, check if the cycle length is greater than 1

                    if(slow == getNextIndex(nums, slow)) {

                        // Cycle length equal to 1 not match requirement

                        // but instead of directly return false, we break

                        // out, test out by {1,-1,5,1,4}

                        //return false;

                        break;

                    }

                    return true;

                }

            }

            // Reset all elements in the detected cycle to 0 to mark them non-cyclic

            // which will avoid duplicate revisit and by the way resolve the issue

            // for self-circle which is not real circle, test out by {1,-1,5,1,4}

            int j = i;

            while(nums[j] \* nums[getNextIndex(nums, j)] > 0) {

                nums[j] = 0;

                j = getNextIndex(nums, j);

            }

        }

        // No loop found

        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[] nums, int i) {

        // Calculate the next index based on the current index and its value in the array.

        // The result is wrapped to stay within array bounds

        return (i + nums[i] % len + len) % len;

    }

}

Time Complexity: O(N)

Space Complexity: O(1)

**Correct Solution with 'visited' array to accelerate the speed to 1ms**

class Solution {

    int len = 0;

    public boolean circularArrayLoop(int[] nums) {

        len = nums.length;

        boolean[] visited = new boolean[len];

        // Steps of size greater than n circle back, so taking remainder

        // And it will help avoid nums[i] = 0 causing self-circle problem

        for(int i = 0; i < len; i++) {

            nums[i] = nums[i] % len;

        }

        for(int i = 0; i < len; i++) {

            // We can even add one more check

            // Skip if the current element is 0 as it's already considered non-cyclic

            if(nums[i] == 0) {

                continue;

            }

            int slow = i;

            int fast = i;

            // Condition inside ensures that all elements are positive or all negative

            // 2 negatives / 2 positives multiplied will give positive as output

            // we need to ensure all elements slow, next(fast), next(next(fast)) are same sign

            // we don't need to check next(slow) because it is some previous fast (so already checked)

            while(nums[slow] \* nums[getNextIndex(nums, fast)] > 0

                && nums[slow] \* nums[getNextIndex(nums, getNextIndex(nums, fast))] > 0) {

                // One step for slow

                slow = getNextIndex(nums, slow);

                // Two steps for fast

                fast = getNextIndex(nums, getNextIndex(nums, fast));

                // Already visited, no point running slow-fast algorithm again

                if(visited[slow]) {

                    break;

                }

                visited[slow] = true;

                if(slow == fast) {

                    // If both pointers meet, check if the cycle length is greater than 1

                    if(slow == getNextIndex(nums, slow)) {

                        // Cycle length equal to 1 not match requirement

                        return false;

                    }

                    return true;

                }

            }

        }

        // No loop found

        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[] nums, int i) {

        // Calculate the next index based on the current index and its value in the array.

        // The result is wrapped to stay within array bounds

        return (i + nums[i] % len + len) % len;

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Style 1 refer to**

<https://leetcode.com/problems/circular-array-loop/solutions/556819/c-100-time-100-space-2-pointers-well-commented-code/>

class Solution {

// next function makes one move

int next(vector<int>& nums, int i){

int n = nums.size();

return (n+nums[i]+i)%n;

}

public:

bool circularArrayLoop(vector<int>& nums) {

int n = nums.size();

// visited array for making sure we visit every element just once

vector<bool> visited(n, 0);

// steps of size greater than n circle back, so taking remainder

for(int i=0; i<n; i++)

nums[i]=nums[i]%n;

for(int i=0;i<n;i++){

// initializing slow and fast

int slow = i, fast = i;

// already visited, no point running slow-fast algorithm again

if(visited[slow]) continue;

// condition inside ensures that all elements are positive or all negative

// 2 negatives / 2 positives multilplied will give +ve as output

// we need to ensure all elements slow, next(fast), next(next(fast)) are same sign

// we don't need to check next(slow) because it is some previous fast (so already checked)

while(nums[slow]\*nums[next(nums,fast)]>0 && nums[slow]\*nums[next(nums,next(nums,fast))]>0){

// one step for slow

slow = next(nums,slow);

// two steps for fast

fast = next(nums,next(nums,fast));

// if already visited break

if(visited[slow]) break;

visited[slow]=1;

if(slow==fast){

if(slow==next(nums,slow)) // single length

return false;

return true; // found a loop

}

}

}

return false; //no loop found

}

};

**Style 2 refer to**

<https://leetcode.com/problems/circular-array-loop/solutions/3277387/457-solution-with-step-by-step-explanation/>

Intuition

Approach

Check if the length of the input list is less than 2. If so, return False as there can't be any loops with less than 2 elements.

Loop through each element in the input list using the enumerate() function to keep track of the index.

If the current element is 0, skip to the next element as it can't be part of a loop.

Initialize two pointers: slow and fast. Set both pointers to the current index i.

While the product of the current element and the element at the index pointed to by fast is greater than 0, and the product of the current element and the element at the index pointed to by the next position of fast (using the \_advance() function) is also greater than 0, advance the slow pointer by one position using the \_advance() function and the fast pointer by two positions using the \_advance() function twice.

Check if the slow pointer and fast pointer have met. If so, we've found a loop.

If the slow pointer and fast pointer are equal to the next position of slow pointer, we've found a loop of length 1. In this case, break out of the loop and continue to the next element in the input list.

Set all elements in the loop to 0 by starting at the current index i and moving through the loop using the \_advance() function until we reach an element with a different sign than the current element.

Continue to the next element in the input list and repeat steps 3-8 until all elements have been processed.

If no loops have been found, return False.

class Solution:

def circularArrayLoop(self, nums: List[int]) -> bool:

if len(nums) < 2:

return False

for i, num in enumerate(nums):

if num == 0:

continue

# initialize slow and fast pointers

slow, fast = i, i

# move slow and fast pointers until they meet or go out of bounds

while num \* nums[fast] > 0 and num \* nums[self.\_advance(nums, fast)] > 0:

slow = self.\_advance(nums, slow)

fast = self.\_advance(nums, self.\_advance(nums, fast))

if slow == fast:

if slow == self.\_advance(nums, slow):

break

return True

# set all elements in the cycle to 0

slow, sign = i, num

while sign \* nums[slow] > 0:

next = self.\_advance(nums, slow)

nums[slow] = 0

slow = next

return False

def \_advance(self, nums: List[int], i: int) -> int:

return (i + nums[i]) % len(nums)

**Style 2 refer to**

<https://leetcode.com/problems/circular-array-loop/solutions/307542/c-too-much-confusion-right-see-this/>

I would like to share my approach.

**457. Circular Array Loop**

As name suggest we have to find loop that exists in array, but the **loop exist** in **two direction** i.e., either in **left-to-right** or **right-to-left** direction.

Well, we know there is one common algorithm to test whether a LinkedList / Array contains loop or not.

Slow & Fast pointer or Hare & Tortoise or Floyd's cylce detection,

for more details [Click here](https://en.wikipedia.org/wiki/Cycle_detection#Floyd's_Tortoise_and_Hare)

Our code starts with as usual like checking for loop in linked list.

/// linked list template

...

while(fast && fast->next){

/// both go in one direction, for this case left-to-right

slow = slow->next;

fast = fast->next->next;

if(slow == fast){

/// loop exists

}

}

...

Now we got the glimpse of slow & fast pointer.

**Note:** loop exists in both direction, we have to check this as well in while loop condition.

**Condition:**

nums[slow]\*nums[next(nums,fast)]>0 && nums[slow]\*nums[next(nums,next(nums,fast))]>0

See below code (Incomplete):

...

/// the condition for both direction

while(nums[slow]\*nums[next(nums,fast)]>0 && nums[slow]\*nums[next(nums,next(nums,fast))]>0){

slow = next(nums,slow);

fast = next(nums,next(nums,fast));int next(vector<int>& nums, int i){

int n = nums.size();

return (n+nums[i]+i)%n;

}

if(slow==fast){

if(slow==next(nums,slow)) // single length

return false;

return true;

}

}

...

**Note** above, next(nums, i) is defined as (n + nums[i] + i)%n. Here n is added extra to handle negative numbers (nums[i]).

Now complete code is below, with some pre-processing of input array element, to handle cases

like,

lengthOfArray = 4, and nums[i] = -9

OR

lengthOfArray = 4, and nums[i] = 12

Note: values are random.

Now **COMPLETE CODE**:

class Solution {

public:

int next(vector<int>& nums, int i){

int n = nums.size();

return (n+nums[i]+i)%n;

}

bool circularArrayLoop(vector<int>& nums) {

int n = nums.size();

// we can use slow and fast pointer to check whether there is loop or not

for(int &num: nums)

num %= n;

for(int i=0;i<n;i++){

int slow = i,

fast = i;

while(nums[slow]\*nums[next(nums,fast)]>0 && nums[slow]\*nums[next(nums,next(nums,fast))]>0){

slow = next(nums,slow);

fast = next(nums,next(nums,fast));

if(slow==fast){

if(slow==next(nums,slow)) // single length

return false;

return true;

}

}

/// DONOT TRAVERSE WHERE THERE IS NO PATH TO GET LOOP.

int j = i;

int val = nums[i];

while (nums[j] \* val > 0) {

int nexx = next(nums,j);

nums[j] = 0;

j = nexx;

}

}

return false;

}

};

**Style 2: Start both 'slow' and 'fast' with different value**

class Solution {

    private int arrayLength; // 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) {

        arrayLength = nums.length; // Initialize the arrayLength with the length of nums

        this.nums = nums; // Assign the nums array to the instance variable

        // Loop through each element in the array

        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 both 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 result is wrapped to stay within array bounds

        return (i + nums[i] % arrayLength + arrayLength) % arrayLength;

    }

}

**Refer to**

<https://algo.monster/liteproblems/457>

**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:

**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)

**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:

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.

**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

**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)

**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

**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)

**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:

Array Length Extraction: The length n of the array nums is 5.

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.

Main Loop to Check for Cycles: We start with index i = 0.

Fast and Slow Pointers Initialization: At index 0, slow is initiated at 0 and fast is initiated at next(0), which is 2.

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.

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[j] and nums[next(j)] was not positive, we do not proceed with marking in this iteration.

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.

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.

**Java Solution**

class Solution {

    private int arrayLength; // 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) {

        arrayLength = nums.length; // Initialize the arrayLength with the length of nums

        this.nums = nums; // Assign the nums array to the instance variable

        // Loop through each element in the array

        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 both 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 result is wrapped to stay within array bounds

        return (i + nums[i] % arrayLength + arrayLength) % arrayLength;

    }

}

**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.

**Refer to L141 Lined List Cycle, L142 Linked List Cycle II**

[L141.P4.1.Linked List Cycle](note://556448C6912F42628631ED8A268CC094)

[L142.Linked List Cycle II](note://82288B72A70C4136BA8570B77CB5F19F)