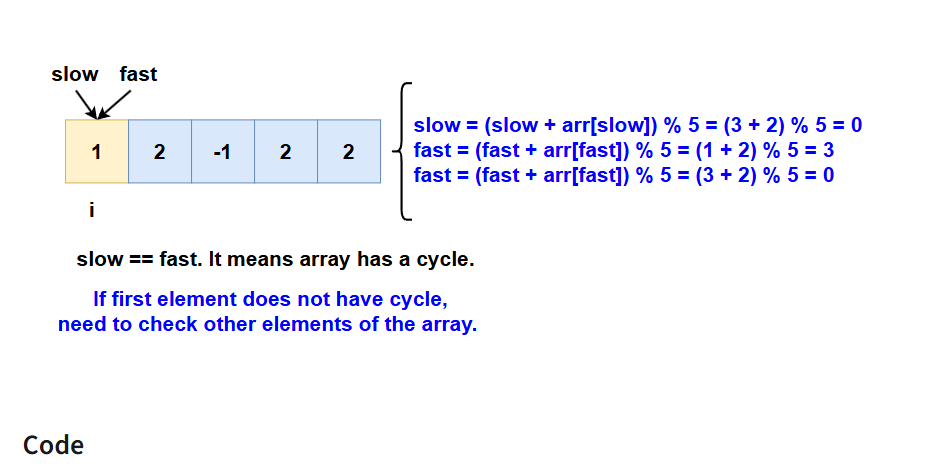
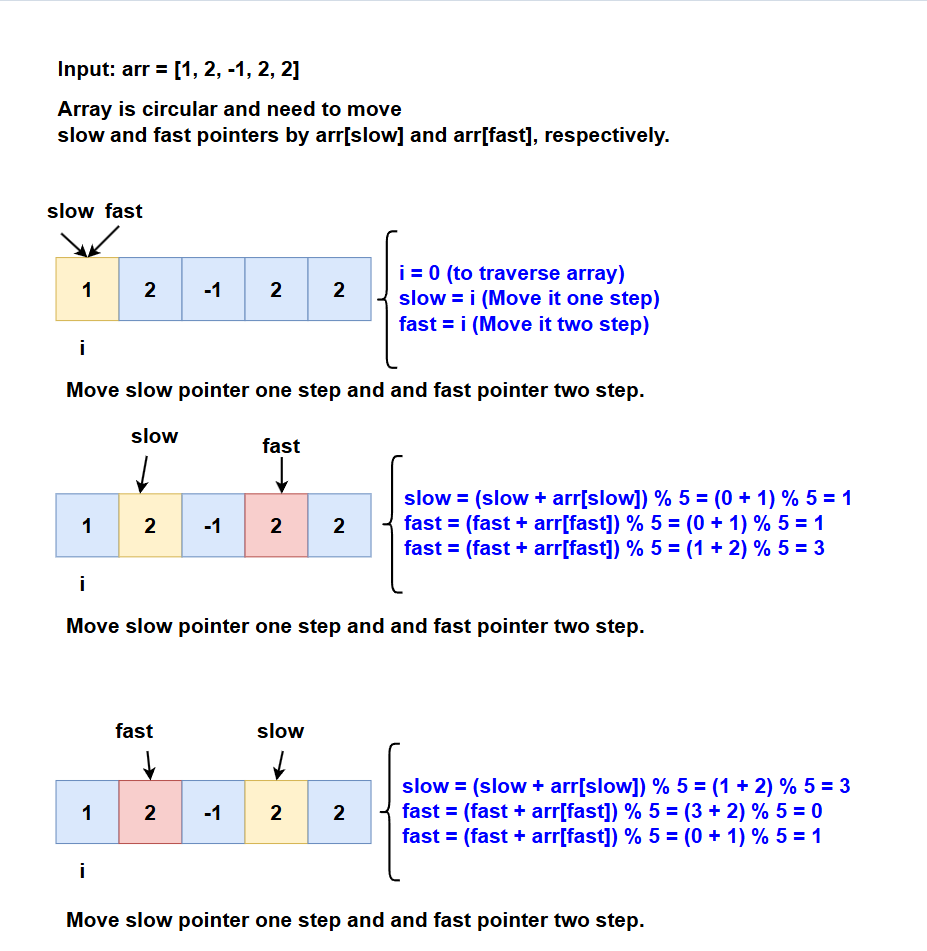
**Solution**

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.

Here is the visual representation of this algorithm for Example-1:



class Solution:

def loopExists(self, 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 = self.find\_next\_index(arr, is\_forward, slow)

# move one step for fast pointer

fast = self.find\_next\_index(arr, is\_forward, fast)

if (fast != -1):

# move another step for fast pointer

fast = self.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(self, 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():

sol = Solution()

print(sol.loopExists([1, 2, -1, 2, 2]))

print(sol.loopExists([2, 2, -1, 2]))

print(sol.loopExists([2, 1, -1, -2]))

main()

**Time Complexity**

The above algorithm will have a time complexity of  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  as, then, each number will be evaluated for cycles only once. We can keep track of this by creating a separate array, however, in this case, the space complexity of our algorithm will increase to .

**Step-by-Step Algorithm**

1. **Initialize Visited Array**:
   * Create a boolean array visited to keep track of visited indices, initially set to false.
2. **Iterate Through Each Index**:
   * Loop through each index i in the array arr.
   * If i is already visited, skip to the next index.
   * Determine the direction of movement (isForward) based on the sign of arr[i].
3. **Initialize Slow and Fast Pointers**:
   * Set both slow and fast pointers to the current index i.
4. **Cycle Detection Using Slow and Fast Pointers**:
   * Enter a loop to move the slow and fast pointers.
     + **Move Slow Pointer**:
       - Move slow one step using the findNextIndex function.
     + **Move Fast Pointer**:
       - Move fast one step using the findNextIndex function.
       - If fast is not -1, move fast another step using the findNextIndex function.
     + Continue this loop until slow and fast pointers meet or one of them becomes -1.
5. **Check for Cycle**:
   * If slow and fast pointers meet (and neither is -1), a cycle is detected.
   * Return true indicating a cycle is found.
6. **Mark Visited Indices**:
   * If no cycle is found, mark all indices visited during this cycle detection attempt:
     + Initialize index to startIndex.
     + While index is not -1 and not already visited:
       - Mark visited[index] as true.
       - Move index to the next index using the findNextIndex function.
7. **Return Result**:
   * If no cycle is detected for any index, return false.

Thank you for the correction. Here is the revised walkthrough for the array [2, 2, -1, 2] with the proper explanation of each step:

**Algorithm Walkthrough**

Let's consider the input: arr = [2, 2, -1, 2].

1. **Initial Setup**:
   * Input array: [2, 2, -1, 2]
   * Initialize visited array: [False, False, False, False]
2. **Index 0**:
   * Not visited, continue.
   * isForward is True (since arr[0] is 2).
   * slow = 0, fast = 0.
3. **First Loop**:
   * **Move Slow Pointer**:
     + slow = findNextIndex([2, 2, -1, 2], True, 0) = 2
   * **Move Fast Pointer**:
     + fast = findNextIndex([2, 2, -1, 2], True, 0) = 2
     + fast = findNextIndex([2, 2, -1, 2], True, 2) = -1 (direction change)
   * The loop terminates as fast becomes -1.
4. **Mark Visited Indices**:
   * Mark all indices visited in this cycle detection attempt starting from index 0:
     + visited = [True, False, True, False]
5. **Index 1**:
   * Not visited, continue.
   * isForward is True (since arr[1] is 2).
   * slow = 1, fast = 1.
6. **First Loop**:
   * **Move Slow Pointer**:
     + slow = findNextIndex([2, 2, -1, 2], True, 1) = 3
   * **Move Fast Pointer**:
     + fast = findNextIndex([2, 2, -1, 2], True, 1) = 3
     + fast = findNextIndex([2, 2, -1, 2], True, 3) = 1
   * slow is 3, fast is 1. Continue loop.
7. **Second Loop**:
   * **Move Slow Pointer**:
     + slow = findNextIndex([2, 2, -1, 2], True, 3) = 1
   * **Move Fast Pointer**:
     + fast = findNextIndex([2, 2, -1, 2], True, 1) = 3
     + fast = findNextIndex([2, 2, -1, 2], True, 3) = 1
   * slow is 1, fast is 1. They meet, cycle detected.
   * So, return true.

The cycle in arr is 1 -> 3 -> 1.

**Code**

**class Solution:**

**def loopExists(self, arr):**

**n = len(arr)**

**visited = [False] \* n # Initialize visited array**

**for i in range(n):**

**if visited[i]:**

**continue # Skip already visited indices**

**isForward = arr[i] >= 0 # Determine the direction**

**slow, fast = i, i**

**# Use slow and fast pointers to detect cycle**

**while True:**

**slow = self.findNextIndex(arr, isForward, slow)**

**fast = self.findNextIndex(arr, isForward, fast)**

**if fast != -1:**

**fast = self.findNextIndex(arr, isForward, fast)**

**if slow == -1 or fast == -1 or slow == fast:**

**break**

**if slow != -1 and slow == fast:**

**return True # Cycle found**

**self.markVisited(arr, i, isForward, visited)**

**return False**

**def findNextIndex(self, arr, isForward, currentIndex):**

**direction = arr[currentIndex] >= 0**

**if isForward != direction:**

**return -1 # Change in direction**

**nextIndex = (currentIndex + arr[currentIndex]) % len(arr)**

**if nextIndex < 0:**

**nextIndex += len(arr)**

**if nextIndex == currentIndex:**

**return -1 # One element cycle**

**return nextIndex**

**def markVisited(self, arr, startIndex, isForward, visited):**

**index = startIndex**

**while True:**

**visited[index] = True**

**nextIndex = self.findNextIndex(arr, isForward, index)**

**if nextIndex == -1 or visited[nextIndex]:**

**break**

**index = nextIndex**

**# Example usage**

**solution = Solution()**

**print(solution.loopExists([1, 2, -1, 2, 2])) # Expected output: True**

**print(solution.loopExists([2, 2, -1, 2])) # Expected output: True**

**print(solution.loopExists([2, 1, -1, -2])) # Expected output: False**

**Complexity Analysis**

* **Time Complexity**:
  + **Outer Loop**: The outer loop runs through each index in the array. Each index is visited at most once due to the visited array, ensuring that each element is processed only once.
  + **Inner Loop (Cycle Detection)**: The inner loop, which uses the slow and fast pointers to detect a cycle, runs in constant time for each element because we are using a do-while loop and breaking out as soon as a cycle is detected or the pointers meet a -1.

Each element is processed a constant number of times, making the overall time complexity .

* **Space Complexity**:
  + **Visited Array**: The additional visited array used to track which indices have been visited has a size equal to the input array, resulting in an  space complexity.