1306. Jump Game III

Medium Breadth-First Search Breadth-First Search Array

Leetcode Link

Problem Description

You are given an array `arr` of non-negative integers and a starting index `start`. Your goal is to determine if you can jump to any index in the array where the value is `0`. From each index `i`, you have two possible jumps: either `i + arr[i]` or `i - arr[i]`. However, you must stay within the bounds of the array; you cannot jump outside of it. The question is, can you find a path through the array, jumping from index to index, that leads you to an index whose value is `0`?

Intuition

The intuition behind solving this problem lies in considering it as a graph traversal problem, where each index is a node, and a jump represents an edge between nodes. Since we want to explore all possible jumping paths, we can use a Breadth-First Search (BFS) approach. BFS is especially useful here because it allows us to visit each node layer by layer, starting from the node we are given as `start`, and move onwards until we either find a node with the value `0` or exhaust all possible nodes that we can visit.

To implement BFS, we can use a queue to keep track of the indices we need to explore. For each index, we check whether the value is `0`; if it is, we've reached our goal, and the answer is `True`. If the value is not `0`, we "mark" this index as visited by setting its value to `-1` to avoid revisiting it, which would cause an infinite loop. We then add the new indices we can jump to the queue if we have not visited them yet and they are within the bounds of the array. We continue this process until our queue is empty or we find a value of `0`.

Solution Approach

The solution utilizes the Breadth-First Search (BFS) algorithm to explore the array. The BFS algorithm is often used in graph traversal to visit nodes level by level, and in this case, it helps to find the shortest path to an index with value `0`.

Here's how the implementation of the BFS algorithm works in the context of this problem:

1. We initiate a queue (`deque` in Python) and add the starting index to it. This queue holds the indices that we need to visit.
2. We enter a loop that continues until the queue is empty.
3. Inside the loop, we pop an index `i` from the front of the queue, which is the current index we are visiting.
4. We then check if the value at this index `i` is `0`. If it is, it means we have successfully found a path to an index with value `0`, and we return `True`.
5. If it's not `0`, we consider it visited by marking `arr[i]` as `-1`. This prevents us from revisiting it because revisiting would mean we're going in circles and not exploring new possibilities.
6. We look at the indices we can jump to from index `i`, which are `i + arr[i]` and `i - arr[i]`. For each of these two indices, we check two conditions:
   - The index must be within the bounds of the array (`0 <= j < len(arr)`).
   - The value at that index must not have been visited (indicated by `arr[j] >= 0`).
7. If an index satisfies both conditions, we append it to the queue so we can visit it in a subsequent iteration.
8. Once we have processed all possible jumps from the current index and added any new indices to visit to our queue, we continue to the next iteration of the loop.
9. This process is repeated until either the queue is empty, which means we have visited all reachable indices and did not find a value of `0` (hence we return `False`), or we find an index with value `0`.

By following the above steps, we utilize the BFS algorithm to check every index we can reach and determine if there is any path leading to an index with a value of `0`.

Example Walkthrough

Let's illustrate the solution approach with a simple example:

Consider the input array `arr = [4, 2, 3, 0, 3, 1, 2]` and the starting index `start = 5`. Our target is to find out if by jumping left or right, starting from `arr[5]`, we can reach an index with the value `0`.

Following the BFS approach:

1. We initialize a queue and add the starting index `5` to it. Our queue now looks like this: `[5]`.
2. Starting the loop, our queue is not empty.
3. We pop the first element from the queue, which is `5`, and check the value of `arr[5]`. Since `arr[5] = 1`, it's not `0`, so we continue.
4. We mark `arr[5]` as visited by setting it to `-1`. Now `arr` looks like this: `[4, 2, 3, 0, 3, -1, 2]`.
5. We check the jump positions from index `5`: `5 - arr[5]` and `5 + arr[5]` (before `arr[5]` was marked as `-1`). These positions are `5 - 1 = 4` and `5 + 1 = 6`.
6. For the first jump position:
   - Index `4` is within the bounds of the array and `arr[4] = 3` (not visited yet), so we add index `4` to our queue. Our queue now looks like this: `[4]`.
7. For the second jump position:
   - Index `6` is also within the bounds and `arr[6] = 2` (not visited), so we add index `6` to our queue. Our queue now: `[4, 6]`.
8. We continue the loop, next popping index `4` from the queue. It's value is `3` (not `0`), so we mark it as visited by setting `arr[4] = -1`. We examine the jump positions `4 + 3 = 7` (which is out of bounds) and `4 - 3 = 1`.
   - Index `1` is within bounds and `arr[1] = 2` (not visited), so we add `1` to our queue. Queue: `[6, 1]`.
9. The next index we pop from the queue is `6`, and `arr[6]` was `2` before we mark it as visited (set to `-1`). We check the jump positions `6 + 2 = 8` (out of bounds) and `6 - 2 = 4`, but since `arr[4]` is already visited (marked as `-1`), we don't add it to the queue. Queue remains the same: `[1]`.
10. Next, we pop index `1` from the queue. Here we find `arr[1] = 2`, and after marking it visited, we check positions `1 + 2 = 3` and `1 - 2 = -1` (out of bounds). Index `3` is within bounds and `arr[3] = 0`. We've found a value of `0`, so we immediately conclude our search with a `True`. We have found a path to an index with value `0`.

Thus, by using the BFS algorithm, we efficiently navigated through the possible jump indices to find a path from `start` to an index where the value is `0`.

Python Solution

```python
from collections import deque  # Make sure to import deque from collections

class Solution:
    def canReach(self, arr, start):
        # Initialize a queue and add the start index to it
        queue = deque([start])

        # Process nodes in the queue until it's empty
        while queue:
            # Pop the element from the queue
            current_index = queue.popleft()

            # Check if the value at the current index is 0, if so we've reached the target and return True
            if arr[current_index] == 0:
                return True

            # Get the jump value from the array, which indicates how far we can jump from this index
            jump_value = arr[current_index]

            # Mark this index as visited by setting its value to -1
            arr[current_index] = -1

            # Calculate the indices for the next possible jumps (forward and backward)
            for next_index in (current_index + jump_value, current_index - jump_value):
                # Ensure the new index is within bounds and not already visited
                if 0 <= next_index < len(arr) and arr[next_index] >= 0:
                    # If valid and not visited, append this index to the queue for future processing
                    queue.append(next_index)

        # If we've processed all possible indices and haven't returned True, then return False
        return False

```

Java Solution

```java
class Solution {
    public boolean canReach(int[] array, int start) {
        // Queue to hold the indices to be checked
        Deque<Integer> queue = new ArrayDeque<>();
        queue.offer(start); // Begin from the starting index

        // Continue until there are no more indices in the queue
        while (!queue.isEmpty()) {
            int currentIndex = queue.poll(); // Get the current index from the queue
            // Check if the value at the current index is 0, meaning we've reached a target
            if (array[currentIndex] == 0) {
                return true; // We can reach an index with a value of 0
            }
            int jumpDistance = array[currentIndex]; // Store the jump distance from the current position
            array[currentIndex] = -1; // Mark the current index as visited by setting it to -1

            // Prepare the next indices to jump to (forward and backward)
            int nextIndexForward = currentIndex + jumpDistance;
            int nextIndexBackward = currentIndex - jumpDistance;

            // Check if the forward jump index is within bounds and not yet visited
            if (nextIndexForward >= 0 && nextIndexForward < array.length && array[nextIndexForward] >= 0) {
                queue.offer(nextIndexForward); // If so, add it to the queue for later processing
            }

            // Check if the backward jump index is within bounds and not yet visited
            if (nextIndexBackward >= 0 && nextIndexBackward < array.length && array[nextIndexBackward] >= 0) {
                queue.offer(nextIndexBackward); // If so, add it to the queue for later processing
            }
        }
        return false; // If exited the loop, we weren't able to reach an index with a value of 0
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <queue>

class Solution {
public:
    // Determines if we can reach any index with value 0 starting from 'start' index
    bool canReach(std::vector<int>& arr, int start) {
        // Create a queue and initialize it with the start index
        std::queue<int> indicesQueue;
        indicesQueue.push(start);

        // Use Breadth-First Search to explore the array
        while (!indicesQueue.empty()) {
            // Extract the current index from the queue
            int currentIndex = indicesQueue.front();
            indicesQueue.pop();

            // If the value at the current index is 0, we've reached the target
            if (arr[currentIndex] == 0) {
                return true;
            }

            // Mark the current index as visited by setting its value to -1
            int jumpValue = arr[currentIndex];
            arr[currentIndex] = -1;

            // Calculate the index positions we can jump to from the current index
            int nextIndexForward = currentIndex + jumpValue;
            int nextIndexBackward = currentIndex - jumpValue;

            // If the next index is in bounds and hasn't been visited, add it to the queue
            if (nextIndexForward >= 0 && nextIndexForward < arr.size() && arr[nextIndexForward] != -1) {
                indicesQueue.push(nextIndexForward);
            }
            if (nextIndexBackward >= 0 && nextIndexBackward < arr.size() && arr[nextIndexBackward] != -1) {
                indicesQueue.push(nextIndexBackward);
            }
        }

        // If the queue is empty and we haven't returned true, then we cannot reach any index with value 0
        return false;
    }
};

```

Typescript Solution

```typescript
function canReach(arr: number[], start: number): boolean {
    // Create a queue for BFS and initialize with the start position
    const queue: number[] = [start];

    // Process nodes until the queue is empty
    while (queue.length > 0) {
        // Dequeue an element from the queue
        const currentIndex: number = queue.shift()!;

        // Check if the current index has a value of zero, indicating we can reach a zero value
        if (arr[currentIndex] === 0) {
            return true;
        }

        // Get the jump value from the current position
        const jumpValue: number = arr[currentIndex];

        // Mark the current position as visited by setting it to -1
        arr[currentIndex] = -1;

        // Check both forward and backward jumps
        for (const nextIndex of [currentIndex + jumpValue, currentIndex - jumpValue]) {
            // Ensure the next index is within bounds and hasn't been visited yet
            if (nextIndex >= 0 && nextIndex < arr.length && arr[nextIndex] !== -1) {
                // Add the valid next index to the queue
                queue.push(nextIndex);
            }
        }
    }

    // Return false if we can't reach a value of zero in the array
    return false;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code would be `O(N)`, where `N` is the size of the array. This is because in the worst-case scenario we might need to visit each element in the array once. The code performs a Breadth-First Search (BFS) by iterating over the array and only traversing to those indices that haven't been visited yet (marked with `-1`).

Since we are checking each element to see if it is 0 (where the jump game ends), and the elements can only be added to the deque once (due to being marked as `-1` after the first visit), the maximum number of operations is proportional to the number of elements in `arr`.

Space Complexity

The space complexity of the code is also `O(N)` due to the `deque` data structure that is used to store indices that need to be visited. In the worst case, this could store a number of indices up to the size of the `arr`.

The space complexity includes additional space for the deque and does not include the space taken up by the input itself. Since we are reusing the input array to mark visited elements, no additional space is needed for tracking visited positions outside of the input array and the deque.