1089. Duplicate Zeros

Easy Array Two Pointers Leetcode Link

Problem Description

The task is to modify a given integer array `arr` by duplicating each occurrence of zero within the array. When a zero is duplicated, all elements to the right of that zero need to be shifted to the right as well, and this process must be done without changing the length of the original array. This means that while shifting the elements, any numbers that move beyond the end of the array should not be included in the final array. The modification must be done within the array itself (in-place), without creating a copy of the array or returning a new array.

Intuition

The intuition behind the solution is to first determine the number of zeros which can be duplicated within the bounds of the original array. Not all zeros might be duplicated if there isn't enough space at the end of the array for all the shifts. We start from the beginning of the array and keep track of the position of elements if there were no bounds, counting the space needed for each zero duplication.

Then we iterate from the end of the array backward, copying over each non-zero element or duplicating zeros when needed. If we encounter a situation where duplicating a zero would exceed the actual bounds of the array, we handle that as a special case, making sure we only set the last element to zero and then move backward correctly. This reverse iteration helps in managing the elements already scanned without overwriting them while still duplicating zeros and moving elements to the right.

The solution cleverly uses two pointers and iterates over the array to accomplish the task efficiently without the need for extra space.

Solution Approach

The provided solution involves a two-pass approach. In the first pass, we calculate the position of the final element and determine how many zeros can be duplicated within the bounds of the array. The second pass is about actually duplicating the zeros and shifting the elements to the right.

First Pass:

- Initialise two pointers `i` and `k`. Pointer `i` is used to iterate over the elements of the array, and `k` is used to keep track of the 'expanded' index as if zeros are being duplicated, but without actually changing the array.
- A loop is run until `k` is less than the length of the array `n`. For non-zero elements, simply increment both `i` and `k` by one. For a zero, increment `i` by one and `k` by two because duplicating a zero would take an additional space.
- After calculating the expanded index, subtract 1 from `i` and `k` if the last incremented `k` goes beyond the array bounds (`n + 1`), AND the last inspected element was a zero. This is because such a zero cannot be duplicated within the array's bounds. So we retreat one step back to adjust for this scenario.

Second Pass:

- A reverse iteration is done using `i` from where we left off in the first pass, and a new pointer `j` which starts at the end of the array (`n - 1`).
- If the current `arr[i]` is zero, we duplicate it in the positions `arr[j]` and `arr[j - 1]` and decrement `j` twice since we added two elements. If it's a non-zero, we simply copy it to `arr[j]` (`arr[j] = arr[i]`) and decrement `j` once.
- This loop continues (moving `i` and `j` backwards) until `j` reaches -1, at which point we've filled the array from the end with duplicated zeros as required.

The key algorithmic idea here is to avoid overwriting elements we have not yet processed, which is why the loop in the second pass iterates backward.

Data structures:

- No additional data structures are used since the operation takes place in-place on the input array `arr`.

Patterns Used:

- Two-pointer technique: One pointer keeps tabs on the position as duplicated, and the other does the array manipulation in reverse to prevent overwriting unprocessed elements.

Example Walkthrough

Let's consider a simple example to illustrate the solution process using the given approach. Suppose our array, `arr`, is `[1, 0, 2, 3, 0, 4, 5, 0]`.

First Pass:

- Initialize `i` to 0 and `k` to 0. These are our two pointers.
- Start iterating over the array `arr`.
  - First element is `1` (non-zero), so we increment both `i` and `k` by one.
  - Second element is `0`. We increment `i` by one (pointing to `2`) and `k` by two (since a zero will be duplicated), `k` is now pointing to the space where the duplicated `0` would be if the array could expand.
  - Continue this process. By the time we are done, `i` is at the last element (another `0`), and `k` is pointing beyond the end of the array because this last zero cannot be doubled within the bounds of the array. Since `k` exceeds the array length after duplicating the last zero, we decrement both `i` and `k` by one to backtrack.

At the end of the first pass, `i` is at the second to last element (which is a `5`), and `k` indicates the 'expanded' length without the final impossible duplication.

Second Pass:

- Set `j` to the last index of `arr`, which is `7`.
- Iterate backwards using `i` and shift or duplicate elements.
  - The element at `i` (which is `5`) is copied to `arr[j]`. Decrement both `i` and `j`.
  - `i` now points to the second to last zero, and `j` is at the sixth position. We have two `5`s at `j` and `j-1`.
  - Next, `i` sees a `0`. Duplicate it at `arr[j]` and `arr[j-1]`, putting zero in both places and decrement `j` twice.
  - Continue this process, duplicating zeros and shifting other numbers left.
  - When we encounter another 0 (the first `0` in the array), we duplicate it again.
  - By the time `j` reaches -1, all elements are properly shifted, and zeros are duplicated without overwriting unprocessed elements.

After the second pass is complete, `arr` should look like `[1, 0, 0, 2, 3, 0, 0, 4]`. Non-zero numbers are shifted right, and every zero is duplicated, with excess elements dropped from the end to maintain the original array length.

Python Solution

```python
from typing import List

class Solution:
    def duplicateZeros(self, arr: List[int]) -> None:
        """
        Modify the input array by duplicating each occurrence of zero, shifting the remaining elements to the right.
        The elements beyond the length of the original array are discarded.
        """
        # Find the length of the modified array (including virtual duplicates of zero to the right, that will be eventually discardec
        n = len(arr)
        original_index, extended_index = -1, 0
        while extended_index < n:
            original_index += 1
            # If we encounter a zero, we count it twice for the extended array index
            extended_index += 2 if arr[original_index] == 0 else 1

        # Start filling the array from the end
        current_index = n - 1
        # If the last element in the expanded list is a virtual duplicate of zero, that we need to discard,
        # set the last position to zero and adjust indexes accordingly
        if extended_index == n + 1:
            arr[current_index] = 0
            original_index -= 1
            current_index -= 1

        # Iterate backward through the array, filling in the zeros and shifting elements
        while current_index >= 0:
            # Replace and shift if the original index points to a zero
            if arr[original_index] == 0:
                # Duplicate the zero
                arr[current_index] = 0
                current_index -= 1
                arr[current_index] = 0
            else:
                # Copy the current element if it's not a zero
                arr[current_index] = arr[original_index]
            # Move to the next elements from the end
            original_index -= 1
            current_index -= 1

# Example usage:
# sol = Solution()
# arr_example = [1, 0, 2, 3, 0, 4, 5, 0]
# sol.duplicateZeros(arr_example)
# The array will be modified to [1, 0, 0, 2, 3, 0, 0, 4] after function call

```

Java Solution

```java
class Solution {

    // Method to duplicate zeros in the array without using extra space
    public void duplicateZeros(int[] arr) {
        int n = arr.length; // n is the length of the array
        int currentIndex = -1, futureIndex = 0;

        // Find the position from which we cannot shift numbers to the right anymore
        // without going out of bounds.
        while (futureIndex < n) {
            currentIndex++;
            // If the current element is zero, it will take two positions after duplication; otherwise, it takes one.
            futureIndex += (arr[currentIndex] == 0) ? 2 : 1;
        }

        int lastIndex = n - 1; // The last index we can write to in the array

        // If futureIndex goes beyond the array length, set the last element to zero,
        // as the last zero cannot be duplicated because it does not fit within the array boundary.
        if (futureIndex == n + 1) {
            arr[lastIndex--] = 0;
            currentIndex--; // Skip this zero as it's already been placed in the array
        }

        // Move through the array from the end and duplicate zeros when necessary.
        while (lastIndex >= 0) {
            arr[lastIndex] = arr[currentIndex]; // Copy current element

            // If the current element is zero, we need to duplicate it.
            if (arr[currentIndex] == 0) {
                arr[--lastIndex] = arr[currentIndex]; // Set the previous index to zero as well
            }

            // Move one position back in both, the array and the current index tracker.
            currentIndex--;
            lastIndex--;
        }
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    void duplicateZeros(vector<int>& arr) {
        int n = arr.size(); // Total number of elements in the array.
        int i = -1; // Initialize i to point to the start of the array.
        int count = 0; // Initialize count to keep track of the number of elements including duplicates.

        // Calculate the number of elements to consider, including zeros which will be duplicated.
        while (count < n) {
            ++i;
            // If the current element is zero, increment the count by 2, else increment by 1.
            count += (arr[i] == 0) ? 2 : 1;
        }

        int j = n - 1; // Initialize j to the last index of the array.

        // If the count is exactly one more than the size, that means the last number to consider is zero
        // and it will be duplicated to go out of bounds, so we only need to set one zero at the end.
        if (count == n + 1) {
            arr[j--] = 0; // Set the last element to zero and decrement j.
            --i; // Move the i pointer back to avoid considering this zero again.
        }

        // Iterate backwards through the array and duplicate zeros where necessary.
        while (j >= 0) {
            arr[j--] = arr[i]; // Copy the current element.
            if (arr[i] == 0) {
                arr[j--] = arr[i]; // Duplicate the zero by setting the previous element to zero as well.
            }
            --i; // Move to the next element to consider.
        }
    }
};

```

Typescript Solution

```typescript
// Function to duplicate zeros in an array in-place.
function duplicateZeros(arr: number[]): void {
    let n: number = arr.length; // Total number of elements in the array.
    let i: number = -1; // Initialize 'i' to point to the start of the array.
    let count: number = 0; // Initialize 'count' to keep track of the number of elements including duplicates.

    // Calculate the number of elements to consider, including zeros which will be duplicated.
    while (count < n) {
        i++;
        count += arr[i] === 0 ? 2 : 1; // If the current element is zero, increment 'count' by 2, else by 1.
    }

    let j: number = n - 1; // Initialize 'j' to the last index of the array.

    // If 'count' is exactly one more than the size, the last number to consider is zero
    // and it will be duplicated to go out of bounds. Set one zero at the end.
    if (count === n + 1) {
        arr[j] = 0; // Set the last element to zero.
        j--; // Decrement 'j'.
        i--; // Move the 'i' pointer back to avoid considering this zero again.
    }

    // Iterate backwards through the array and duplicate zeros where necessary.
    while (j >= 0) {
        arr[j] = arr[i]; // Copy the current element.
        j--; // Decrement 'j'.
        if (arr[i] === 0 && j >= 0) {
            arr[j] = 0; // Duplicate the zero by setting the previous element to zero as well.
            j--; // Decrement 'j' after duplication.
        }
        i--; // Move to the next element to consider.
    }
}

// Example usage:
// let arrExample = [1, 0, 2, 3, 0, 4, 5, 0];
// duplicateZeros(arrExample);
// console.log(arrExample); // Output would be [1, 0, 0, 2, 3, 0, 0, 4]

```

Time and Space Complexity

The given code modifies the input array `arr` in-place to duplicate each occurrence of zero, shifting the remaining elements to the right. Here's the complexity analysis:

- **Time Complexity:**

The time complexity of the code is `O(n)`. It consists of two pass-through procedures over the array: the first pass counts the zeros (in a way that accounts for the shift that will happen once zeros are duplicated), and the second pass actually duplicates the zeros from the end to the beginning. Each pass runs in linear time relative to the number of elements in the array (`n`).

Assuming `n` is the length of `arr`, the first loop runs until `k`, which is the expanded size taking into account the duplicated zeros, is greater than or equal to `n`. Each iteration potentially advances `i` by 1 and `k` by either 1 (non-zero) or 2 (zero), meaning it will never iterate more than `n` times (because `k` starts from zero and increases to at least `n`).

The second loop runs while `j`, which starts at `n - 1` and decreases, is non-negative. On each iteration, it potentially copies a value from `i` to `j` and decreases `j` by 1 (or 2 if the value is 0 and it's not the case where `k == n + 1`). Again, this loop will never execute more than `n` times.

- **Space Complexity:**

The space complexity of the code is `O(1)`. The algorithm modifies the array in place and uses a constant amount of extra space for variables `i`, `j`, and `k`. Regardless of the size of the input array, the space used by these variables does not change, so the space complexity is constant.