503. Next Greater Element II

Medium Stack Array Monotonic Stack Leetcode Link

Problem Description

The problem gives us a circular integer array, which means that after the last element of the array, it wraps around back to the first element. We are tasked with finding the next greater number for each element in the array. The "next greater number" is defined as the first number that is bigger than the given number when you traverse the array from the current position, considering the circular nature of the array. If no such number exists, we are to return `-1` for that element.

Example: Suppose the array is `[1, 2, 1]`. For the first element (1), the next greater number is 2. For the second element (2), there is no greater number in the array, so the output is -1. For the last element (1), the next greater number is again 2 because we can traverse the array circularly.

Intuition

To solve this problem, we need to consider each element and find the next element that is greater than it. Since the array is circular, we can't just iterate linearly from the start to the end. Once we reach the end, we need to loop back to the beginning of the array and continue our search. This effectively doubles the "length" of our search space.

A naive approach would be to check each element against all others, but this would be inefficient with time complexity O(n^2), where n is the length of the array. Instead, we can use a stack to keep track of the indices of elements for which we haven't found the next greater number yet.

Here is the intuition for how the efficient solution works:

1. Create an array `ans` to store the next greater numbers, initialized with `-1`, signifying we haven't found the answer for any element yet.
2. Use a stack to store the indices of elements in decreasing order of their values.
3. Iterate through the array twice (because it's circular) using the index `i`. For each element:
   - While there is an element in the stack, and the current element is greater than the element pointed to by the index at the top of the stack (`nums[stk[-1]] < nums[i % n]`), it means we have found the next greater number for the element at the top of the stack. We then pop the index from the stack and update the `ans` array.
   - Push the current index modulo the length of the array `i % n` onto the stack. This ensures we are in bounds of the array when the loop wraps around.
4. The reason to loop twice is to simulate the circular array by giving each element a second chance to find its next greater value.

By using the stack, each element is pushed and popped at most once, yielding a linear time complexity solution, O(n), where n is the length of the array.

Solution Approach

The reference solution provided uses a stack to keep track of the indices of elements for which we need to find the next greater number. Let's break down the steps taken in the implementation:

1. **Initialize Necessary Variables**:
   - Calculate the length of the `nums` array and store it in a variable `n`.
   - Create an answer array `ans` of the same length as `nums`, initialized with `-1`. This will hold the next greater elements for the respective indices.
   - Initialize an empty list `stk` which will serve as the stack.
2. **Double Iteration Over The Array**:
   - Iterate over a range that is double the length of `nums` to simulate the circular nature of the array. The current index in the array is given by `i % n`.
3. **Finding the Next Greater Element Using Stack**:
   - As we iterate, we continuously check whether the stack is non-empty and the current number `nums[i % n]` is greater than the number at the index at the top of the stack `nums[stk[-1]]`.
   - If that's the case, we have found a next greater number for the element at the top of the stack. We update the `ans` array at the index popped from the stack with the current number.
   - This process is repeated until no such element is found or the stack becomes empty.
4. **Maintain Decreasing Stack**:
   - Each index `i % n` is added to the stack. The modulo operation ensures we cycle back to the starting indices after reaching the end of the array.
   - The stack maintains indices of elements in decreasing order. When an element with a greater value comes up, it signifies the "next greater number" for all elements on the stack it is compared against.
5. **Return the Answer**:
   - The `ans` array now contains either the next greater elements for each index, or `-1` if no greater element was found in the circular traversal. The `ans` array is returned as the final result.

The key patterns used in this implementation are stack usage for maintaining a decreasing sequence and modulus arithmetic to handle circular array traversal efficiently. This algorithm is efficient because each element is pushed onto the stack once and popped at most once, leading to a time complexity of O(n), where n is the length of the `nums` array.

Example Walkthrough

Let's use an example to illustrate the solution approach described above. Suppose we are given the circular array `nums = [4, 1, 2]`.

1. **Initialize Necessary Variables**:
   - The length of `nums` is `n = 3`.
   - We create an answer array `ans = [-1, -1, -1]`. This will hold the next greater elements.
   - We initialize an empty list `stk` as our stack.
2. **Double Iteration Over The Array**:
   - We loop over the range of `2n`, which is `6` in this case, to simulate the circular array.
3. **Iteration Details**:
   - **First Pass ([4, 1, 2])**: We start with `i = 0` (`i % n = 0`), and the stack is empty. We push the index `0` to the stack.
   - Move to `i = 1` (`i % n = 1`). `nums[1] = 1` is not greater than `nums[0] = 4`, so we push `1` onto the stack.
   - Move to `i = 2` (`i % n = 2`). `nums[2] = 2` is not greater than `nums[1] = 1` but is also not greater than `nums[0] = 4`, so we push `2` onto the stack.
   - **Second Pass ([4, 1, 2] again)**:
   - Now, `i = 3` (`i % n = 0`). `nums[3 % n] = nums[0] = 4` is not greater than `nums[2] = 2`, so nothing changes.
   - Move to `i = 4` (`i % n = 1`). `nums[4 % n] = nums[1] = 1` is not greater than `nums[2] = 2` or than `nums[0] = 4`. Nothing changes.
   - Move to `i = 5` (`i % n = 2`). `nums[5 % n] = nums[2] = 2`. Here, we see `nums[2] = 2` is greater than `nums[1] = 1` which is at the top of the stack. So, we pop `1` from the stack and update `ans[1]` to `2`. Now `stk = [0, 2]`.
4. **Finish the Stack Processing**:
   - Since we reached the end of our second pass and there is no greater element for indices `0` and `2`, we leave their corresponding values in `ans` as `-1`.
5. **Return the Answer**:
   - Our answer array `ans` is now `[-1, 2, -1]`, which correctly reflects that the next greater number for the first element (4) doesn't exist in the array, for the second element (1) the next greater number is 2, and for the third element (2) there is no next greater number considering the circular nature of the array. We return `ans` as the result.

Python Solution

```python
class Solution:
    def nextGreaterElements(self, nums):
        """
        Finds the next greater element for each element in a circular array.

        :param nums: List[int] - List of integers where we need to find the next greater element for each.
        :return: List[int] - List containing the next greater elements.
        """
        # The number of elements in the input list.
        n = len(nums)

        # Initialize the result list with -1 for each element.
        result = [-1] * n

        # Stack to keep indexes of nums for which we haven't found the next greater element.
        stack = []

        # Iterate over the list twice to simulate circular array behavior.
        for i in range(2 * n): # Shorthand for n << 1.
            # Get the index in the original nums array.
            index = i % n

            # While stack is not empty and the current element is greater than the element at
            # the index of the last element in stack, update the result for the index at the
            # top of the stack.
            while stack and nums[stack[-1]] < nums[index]:
                result[stack.pop()] = nums[index]

            # For the first pass, we need to fill the stack with index.
            # Avoid pushing index onto the stack in the second pass to prevent duplicating work.
            if i < n:
                stack.append(index)

        # Return the result list containing next greater elements.
        return result

```

Java Solution

```java
import java.util.Arrays;
import java.util.Deque;
import java.util.ArrayDeque;

class Solution {
    public int[] nextGreaterElements(int[] nums) {
        // Determine the length of the input array.
        int n = nums.length;
        // Initialize the answer array with -1 (assuming no next greater element).
        int[] answer = new int[n];
        Arrays.fill(answer, -1);
        // Use a deque as a stack to keep track of indices.
        Deque<Integer> stack = new ArrayDeque<>();

        // Iterate through the array twice to simulate a circular array.
        for (int i = 0; i < 2 * n; ++i) {
            // While the stack is not empty and the current element is greater than the
            // element at index at the top of the stack - it means we have found the
            // next greater element for the index at the top of the stack.
            while (!stack.isEmpty() && nums[stack.peek()] < nums[i % n]) {
                // Update the answer for the index at the top of the stack.
                answer[stack.pop()] = nums[i % n];
            }
            // If this index is within the original array, push its index on the stack.
            // During the second iteration, we don't push the index into the stack
            // to avoid duplicates.
            if (i < n) {
                stack.push(i);
            }
        }
        // Return the answer array with next greater elements for each index.
        return answer;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <stack>

class Solution {
public:
    // Function to find the next greater elements for each element in a circular array.
    vector<int> nextGreaterElements(vector<int>& nums) {
        int n = nums.size(); // Size of the input array
        vector<int> answers(n, -1); // Initialize the answer vector with -1, which means no greater element
        stack<int> indicesStack; // Stack to store the indices of elements

        // Loop through the array twice to simulate circular array traversal.
        for (int i = 0; i < 2 * n; ++i) {
            // While stack is not empty and the current element is greater
            // than the element corresponding to the index on the top of the stack
            while (!indicesStack.empty() && nums[indicesStack.top()] < nums[i % n]) {
                // Update the answer for the index at the top of stack as we have found
                // a next greater element
                answers[indicesStack.top()] = nums[i % n];
                // Pop the index from the stack as we've found a next greater element for it
                indicesStack.pop();
            }

            // We only push the index of the element to the stack
            // as long as we are in the first iteration over the array.
            if (i < n) {
                indicesStack.push(i);
            }
        }
        return answers;
    }
};

```

Typescript Solution

```typescript
function nextGreaterElements(nums: number[]): number[] {
    // Stack to keep indexes where we haven't found the next greater element yet
    const indexStack: number[] = [];
    const length = nums.length;
    // Result array initialized with -1, assuming we don't find a next greater element
    const result: number[] = new Array(length).fill(-1);

    // Iterate twice over the array to simulate a circular array
    for (let i = 0; i < 2 * length - 1; i++) {
        // Actual index for the original nums array (using modulo for wrap-around)
        const currentIndex = i % length;

        // Pop all elements from the stack smaller than the current element in nums
        while (indexStack.length !== 0 && nums[indexStack[indexStack.length - 1]] < nums[currentIndex]) {
            // Set the next greater element for the index on the top of the stack
            result[indexStack[indexStack.length - 1]] = nums[currentIndex];
            // Remove the index from the stack as we have found a next greater element
            indexStack.pop();
        }

        // If we're in the first pass, add the current index to the stack
        if (i < length) {
            indexStack.push(currentIndex);
        }
    }
    return result;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code can be analyzed as follows:

- The code consists of a for loop that iterates `2n` times where `n` is the length of the input array `nums`. This is evident from the loop construction `for i in range(n << 1)`, which is equivalent to `for i in range(2 * n)`.
- Inside the loop, the `while` loop can potentially run for each element of the stack. However, considering that no element is pushed onto the stack more than twice (once for its original position and once for its virtual copied position due to the circular nature of the problem), each element is also popped only once. Therefore, in amortized analysis, the `while` loop runs `O(1)` time for each iteration of the `for` loop.
- Accordingly, the overall time complexity is `O(n)` for the single pass in the doubled array, counting the amortized constant time operations inside the loop.

Thus, the time complexity is `O(n)`.

Space Complexity

The space complexity can be considered in the following points:

- A new list `ans` of size `n` is created to store the results so this takes `O(n)` space.
- Additionally, a stack `stk` is used to store indices. In the worst case, this stack can grow to store `n` indices before elements are popped.
- There are no other significant contributors to space complexity.

Taking these into account, the space complexity is also `O(n)`.