Documentation for "Next Greater Element II"

1. **Problem Statement**

Given a circular integer array nums (i.e., the next element of nums[nums.length - 1] is nums[0]), return an array where each element is the next greater element for the corresponding element in the input array. The next greater element for a number x is the first greater number that appears in the circular order after x. If no such greater element exists, return -1 for that number.

Example 1:

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

Explanation:

- The next greater element for 1 (index 0) is 2.
- The next greater element for 2 (index 1) does not exist, so it's -1.
- The next greater element for the second 1 (index 2) is 2.

Example 2:

- Input: nums = [1,2,3,4,3]
- Output: [2, 3, 4, -1, 4]

Explanation:

- The next greater element for 1 (index 0) is 2.
- The next greater element for 2 (index 1) is 3.
- The next greater element for 3 (index 2) is 4.
- The next greater element for 4 (index 3) does not exist, so it's -1.
- The next greater element for the second 3 (index 4) is 4.

2. Intuition

To find the next greater element efficiently, we can utilize the monotonic stack technique. A stack is useful because it allows us to easily traverse the array while keeping track of potential candidates for the next greater element.

By iterating over the array twice (to simulate circular behavior), we can ensure that we properly handle all cases, including when an element's next greater element might wrap around to the beginning of the array.

3. Key Observations

- Circular Nature: The array is circular, meaning after the last element, we should check the elements at the start again.
- Monotonic Stack: The stack allows us to efficiently track the next greater element by always ensuring that the stack maintains a decreasing order of values.
- Efficient Traversal: By iterating from right to left (backwards), we can ensure that each element's next greater number is found by looking at already processed elements in the stack.
- Handling the Circular Array: Since we simulate the array's circular nature by iterating twice, we ensure that every element is checked for its next greater element, even when it wraps around.

4. Approach

To solve the problem, we'll:

- Create a stack to hold the indices of potential next greater elements.
- Traverse the array in reverse order to simulate circular behavior (by iterating from the end to the beginning and wrapping around).
- For each element, pop elements from the stack that are less than or equal to the current element, as they cannot be the next greater element for any future element.
- If the stack is not empty, the top of the stack will give the next greater element for the current element. If the stack is empty, it means there is no next greater element, so set the result to -1.
- Finally, return the result array.

5. **Edge Cases**

- Single Element Array: The next greater element for a single element is always -1, since there are no other elements to compare.
- All Elements are the Same: If all elements are the same, the next greater element for each element will be -1, as there is no greater element.
- Empty Array: An empty array should return an empty result.
- Array with Descending Order: If the array is strictly decreasing, every element will have -1 as the next greater element.

6. **Complexity Analysis**

Time Complexity:

- O(n): We iterate over the array twice (once to process the elements and once for circular behavior), and each element is pushed and popped from the stack at most once, giving us a total of O(n) operations.

Space Complexity:

- O(n): We store the result array and the stack, both of which can hold up to n elements in the worst case.

7. **Alternative Approaches**

- Brute Force Approach:
    1. For each element, search through the rest of the array to find the next greater element.
    2. Time Complexity: $O(n^2)$ (inefficient for large arrays).
    3. Space Complexity: O(1).
    4. This approach is not efficient for larger arrays and should be avoided.
- Using a Deque:
    1. A deque could be used to track indices in a similar manner to the stack, but the monotonic stack approach is simpler and more intuitive for this problem.

8. Test Cases

- Test Case 1:
  - Input: [1, 2, 1]
  - Output: [2, -1, 2]
- Test Case 2:
  - Input: [1, 2, 3, 4, 3]
  - Output: [2, 3, 4, -1, 4]
- Test Case 3:
  - Input: [1]
  - Output: [-1]
- Test Case 4:
  - Input: [4, 3, 2, 1]
  - Output: [-1, 4, 4, 4]
- Test Case 5:
  - Input: [3, 3, 3, 3]
  - Output: [-1, -1, -1, -1]

9. Final Thoughts

The solution using a monotonic stack is both time-efficient and easy to implement. It allows us to find the next greater element for each element in the array while accounting for the circular nature of the array. By maintaining the stack in decreasing order and iterating twice, we can efficiently solve this problem in linear time, making it suitable for large arrays. The complexity of O(n) ensures that this solution is optimal for inputs within the given constraints.