503. Next Greater Element II

Array

found the next greater number yet.

index from the stack and update the ans array.

Monotonic Stack

Problem Description

Stack

Medium

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

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.

 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

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:

 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.

number. Let's break down the steps taken in the implementation:

Calculate the length of the nums array and store it in a variable n.

- 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.

The reference solution provided uses a stack to keep track of the indices of elements for which we need to find the next greater

Solution Approach

% n.

array.

∘ 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. **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

Finding the Next Greater Element Using **Stack**: o 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]].

The ans array is returned as the final result.

Initialize Necessary Variables:

push 2 onto the stack.

changes.

Return the Answer:

Second Pass ([4, 1, 2] again):

circular nature of the array. We return ans as the result.

Finds the next greater element for each element in a circular array.

:return: List[int] - List containing the next greater elements.

Iterate over the list twice to simulate circular array behavior.

For the first pass, we need to fill the stack with index.

// Return the answer array with next greater elements for each index.

// Function to find the next greater elements for each element in a circular array.

stack<int> indicesStack; // Stack to store the indices of elements

answers[indicesStack.top()] = nums[i % n];

// If we're in the first pass, add the current index to the stack

Finds the next greater element for each element in a circular array.

For the first pass, we need to fill the stack with index.

Return the result list containing next greater elements.

:return: List[int] - List containing the next greater elements.

// Loop through the array twice to simulate circular array traversal.

// 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]) {</pre>

// Update the answer for the index at the top of stack as we have found

// Pop the index from the stack as we've found a next greater element for it

vector<int> answers(n, -1); // Initialize the answer vector with -1, which means no greater element

vector<int> nextGreaterElements(vector<int>& nums) {

// a next greater element

for (int i = 0; i < 2 * n; ++i) {

indicesStack.pop();

if (i < length) {</pre>

def nextGreaterElements(self, nums):

return result;

class Solution:

111111

n = len(nums)

stack = []

result = [-1] * n

if i < n:

Time and Space Complexity

return result

Time Complexity

indexStack.push(currentIndex);

The number of elements in the input list.

Initialize the result list with -1 for each element.

int n = nums.size(); // Size of the input array

for i in range(2 * n): # Shorthand for n << 1.

Get the index in the original nums array.

while stack and nums[stack[-1]] < nums[index]:</pre>

result[stack.pop()] = nums[index]

Initialize Necessary Variables:

popped from the stack with the current number. This process is repeated until no such element is found or the stack becomes empty. **Maintain Decreasing <u>Stack</u>**:

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

∘ 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.

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

- 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. **Return the Answer:**
 - 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.

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

The key patterns used in this implementation are stack usage for maintaining a decreasing sequence and modulus arithmetic to

• The length of nums is n = 3. \circ 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. **Double Iteration Over The Array:**

Iteration Details:

Example Walkthrough

2].

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

Now, i = 3 (i % n = 0). nums [3 % n] = nums [0] = 4 is not greater than nums [2] = 2, so nothing changes.

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 = 5 (i % n = 2). nums[5 % n] = nums[2] = 2. Here, we see nums[2] = 2 is greater than <math>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].

• We loop over the range of 2n, which is 6 in this case, to simulate the circular array.

Finish the Stack Processing: o Since we reached the end of our second pass and there is no greater element for indices of and 2, we leave their corresponding values in ans as -1.

○ 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

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

Python

stack = []

index = i % n

top of the stack.

def nextGreaterElements(self, nums):

class Solution:

Solution Implementation

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.

:param nums: List[int] - List of integers where we need to find the next greater element for each.

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

```
# 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
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]) {</pre>
                // 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);
```

C++

public:

#include <vector>

#include <stack>

class Solution {

return answer;

```
// 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
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]) {</pre>
            // 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();
```

```
# Iterate over the list twice to simulate circular array behavior.
for i in range(2 * n): # Shorthand for n << 1.</pre>
    # 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]:</pre>
        result[stack.pop()] = nums[index]
```

Avoid pushing index onto the stack in the second pass to prevent duplicating work.

Stack to keep indexes of nums for which we haven't found the next greater element.

:param nums: List[int] - List of integers where we need to find the next greater element for each.

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).

stack.append(index)

- 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 0(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 0(n).