

503. Next Greater Element II

March 16, 2017 | 49.8K views

Average Rating: 3.82 (49 votes)

Given a circular array (the next element of the last element is the first element of the array), print the Next Greater Number for every element. The Next Greater Number of a number x is the first greater number to its traversing-order next in the array, which means you could search circularly to find its next greater number. If it doesn't exist, output -1 for this number.

Example 1:

Input: [1,2,1]
Output: [2,-1,2]
Explanation: The first 1's next greater number is 2;
The number 2 can't find next greater number;
The second 1's next greater number needs to search circularly, which is also 2.

Note: The length of given array won't exceed 10000.

Solution

Approach #1 Brute Force (using Double Length Array) [Time Limit Exceeded]

In this method, we make use of an array *doublenums* which is formed by concatenating two copies of the given *nums* array one after the other. Now, when we need to find out the next greater element for *nums[i]*, we can simply scan all the elements *doublenums[j]*, such that $i < j < \text{length}(\text{doublenums})$. The first element found satisfying the given condition is the required result for *nums[i]*. If no such element is found, we put a -1 at the appropriate position in the *res* array.

```
Java
1 public class Solution {
2
3     public int[] nextGreaterElements(int[] nums) {
4         int[] res = new int[nums.length];
5         int[] doublenums = new int[nums.length * 2];
6         System.arraycopy(nums, 0, doublenums, 0, nums.length);
7         System.arraycopy(nums, 0, doublenums, nums.length, nums.length);
8         for (int i = 0; i < nums.length; i++) {
9             res[i] = -1;
10            for (int j = i + 1; j < doublenums.length; j++) {
11                if (doublenums[j] > doublenums[i]) {
12                    res[i] = doublenums[j];
13                    break;
14                }
15            }
16        }
17        return res;
18    }
19 }
```

Complexity Analysis

- Time complexity: $O(n^2)$. The complete *doublenums* array(of size $2n$) is scanned for all the elements of *nums* in the worst case.
- Space complexity: $O(n)$. *doublenums* array of size $2n$ is used. *res* array of size n is used.

Approach #2 Better Brute Force [Accepted]

Instead of making a double length copy of *nums* array, we can traverse circularly in the *nums* array by making use of the $\%(\text{modulus})$ operator. For every element *nums[i]*, we start searching in the *nums* array(of length n) from the index $(i + 1)$ and look at the next(circularly) $n - 1$ elements. For *nums[i]* we do so by scanning over *nums[j]*, such that $(i + 1)$, and we look for the first greater element found. If no such element is found, we put a -1 at the appropriate position in the *res* array.

```
Java
1 public class Solution {
2     public int[] nextGreaterElements(int[] nums) {
3         int[] res = new int[nums.length];
4         for (int i = 0; i < nums.length; i++) {
5             res[i] = -1;
6             for (int j = i + 1; j < nums.length; j++) {
7                 if (nums[(i + j) % nums.length] > nums[i]) {
8                     res[i] = nums[(i + j) % nums.length];
9                     break;
10                }
11            }
12        }
13        return res;
14    }
15 }
```

Complexity Analysis

- Time complexity: $O(n^2)$. The complete *nums* array of size n is scanned for all the elements of *nums* in the worst case.
- Space complexity: $O(n)$. *res* array of size n is used.

Approach #3 Using Stack [Accepted]

This approach makes use of a stack. This stack stores the indices of the appropriate elements from *nums* array. The top of the stack refers to the index of the Next Greater Element found so far. We store the indices instead of the elements since there could be duplicates in the *nums* array. The description of the method will make the above statement clearer.

We start traversing the *nums* array from right towards the left. For an element *nums[i]* encountered, we pop all the elements *stack[top]* from the stack such that $\text{nums}[\text{stack}[\text{top}]] \leq \text{nums}[\text{i}]$. We continue the popping till we encounter a *stack[top]* satisfying $\text{nums}[\text{stack}[\text{top}]] > \text{nums}[\text{i}]$. Now, it is obvious that the current *stack[top]* only can act as the Next Greater Element for *nums[i]*(right now, considering only the elements lying to the right of *nums[i]*).

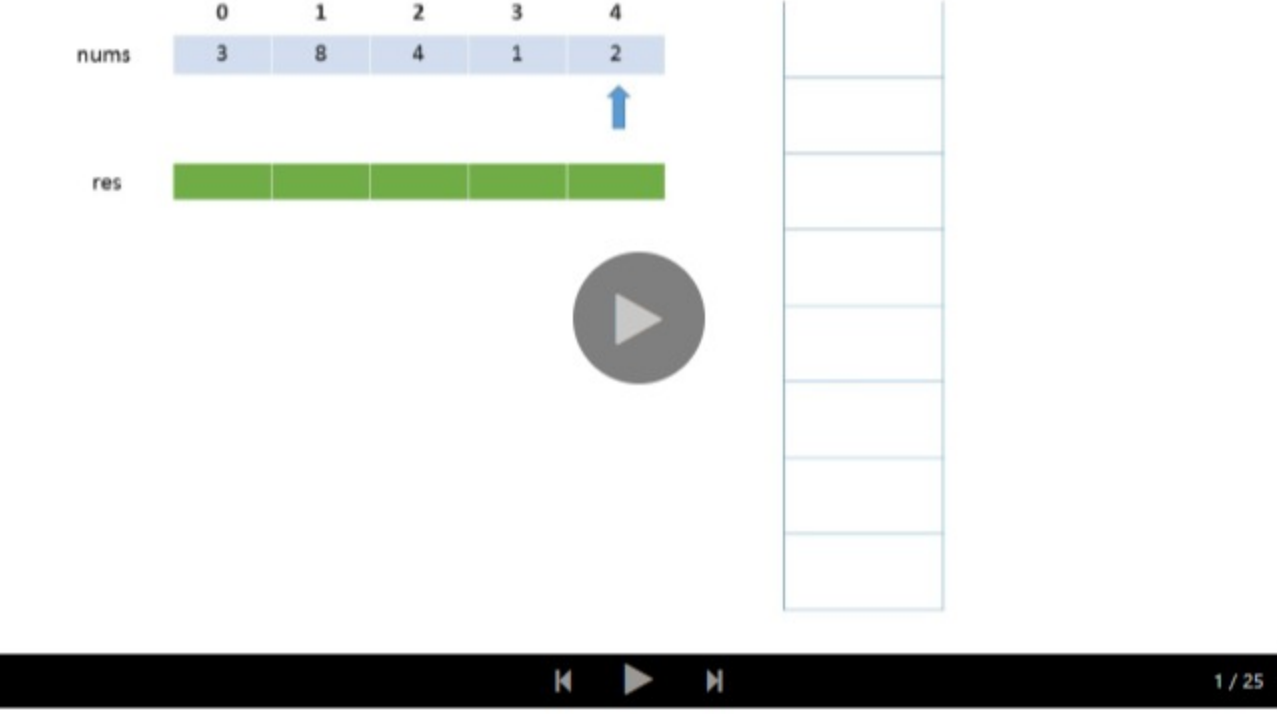
If no element remains on the top of the stack, it means no larger element than *nums[i]* exists to its right. Along with this, we also push the index of the element just encountered(*nums[i]*), i.e. *i* over the top of the stack, so that *nums[i]*(or *stack[top]* now acts as the Next Greater Element for the elements lying to its left.

We go through two such passes over the complete *nums* array. This is done so as to complete a circular traversal over the *nums* array. The first pass could make some wrong entries in the *res* array since it considers only the elements lying to the right of *nums[i]*, without a circular traversal. But, these entries are corrected in the second pass.

Further, to ensure the correctness of the method, let's look at the following cases.

Assume that *nums[j]* is the correct Next Greater Element for *nums[i]*, such that $i < j \leq \text{stack}[\text{top}]$. Now, whenever we encounter *nums[j]*, if $\text{nums}[\text{j}] > \text{nums}[\text{stack}[\text{top}]]$, it would have already popped the previous *stack[top]* and *j* would have become the topmost element. On the other hand, if $\text{nums}[\text{j}] < \text{nums}[\text{stack}[\text{top}]]$, it would have become the topmost element by being pushed above the previous *stack[top]*. In both the cases, if $\text{nums}[\text{j}] > \text{nums}[\text{i}]$, it will be correctly determined to be the Next Greater Element.

The following example makes the procedure clear:



As the animation above depicts, after the first pass, there are a number of wrong entries(marked as -1) in the *res* array, because only the elements lying to the corresponding right(non-circular) have been considered till now. But, after the second pass, the correct values are substituted.

```
Java
1 public class Solution {
2
3     public int[] nextGreaterElements(int[] nums) {
4         int[] res = new int[nums.length];
5         Stack<Integer> stack = new Stack<>();
6         for (int i = 0; i < 2 * nums.length - 1; i += 1) {
7             while (!stack.empty() && nums[stack.peek()] <= nums[i % nums.length]) {
8                 stack.pop();
9             }
10            res[i % nums.length] = stack.empty() ? -1 : nums[stack.peek()];
11            stack.push(i % nums.length);
12        }
13        return res;
14    }
15 }
```

Complexity Analysis

- Time complexity: $O(n)$. Only two traversals of the *nums* array are done. Further, atmost $2n$ elements are pushed and popped from the stack.
- Space complexity: $O(n)$. A stack of size n is used. *res* array of size n is used.


Rate this article: ★★★★★

Previous




Next

Comments: 28

Sort By

- 



Type comment here...(Markdown is supported)



 Preview  Post
- 


shanshan333333 ★ 57 · March 20, 2019 10:48 AM

I think the description of approach#3 is not straight forward for understanding as it focuses more on the implementation not the way of thinking. Actually, if the idea is described clearly, I think most of the people can easily come up with the same implementation. The key of approach#3 is that when we are trying to find the next greater number for the ith number and num[i] >= num[i+1], what do we do next? In the brute force way, we will go on to check num[i+2], num[i+3]... and there is much redundant

56





 Share  Reply



[SHOW 1 REPLY](#)
- 


zerustech ★ 237 · October 17, 2018 11:29 AM

Approach #3 is great, but the explanation is not clear enough, which didn't reveal the fact that the algorithm works because at the end of round 1, **stack[0]** is always the index of the max value in **nums[]**, and **stack[top]** is always "0", and therefore, the "next greater element" of each element from **nums[0]** through **nums[stack[0]]** has been confirmed. Besides, for any element to the right of the "max value" (**stack[0]**), if its "next greater element" is not found in round 1, it is only necessary

17



 Share  Reply



[SHOW 1 REPLY](#)
- 



vivek_23 ★ 500 · April 2, 2018 2:27 PM


In your **Stack** approach, when you iterate for the 2nd time, there is no need to again push elements from the right.

Directly check if there is any number in the stack greater than the current one and assign it.

14





 Share  Reply



[SHOW 1 REPLY](#)
- 


sylor ★ 8 · December 18, 2018 10:08 PM

Approach #2 in python still exceeds the time limit.

8





 Share  Reply



[SHOW 3 REPLIES](#)
- 


DyXrLxSTAQoDoD ★ 4655 · July 10, 2017 2:03 AM

I think first adding all element indices to stack is easier to understand for me. It means for the right most side elements, first trying to find next greater element from left beginning. The same time adding right elements to stack for elements on its left.

8



 Share  Reply

[SHOW 2 REPLIES](#)
- 



xiaoliu3 ★ 28 · November 2, 2019 6:30 AM



After one pass of mono-stack, just do another one. The maximum number should output -1.


```
Stack<Integer> stack = new Stack<>();

public int[] nextGreaterElements(int[] nums) {
```

4



 Share  Reply



[SHOW 1 REPLY](#)
- 



user0414A ★ 21 · May 1, 2020 11:42 PM


Why do we need to traverse from right to left in approach #3. Why can't we traverse from left to right? Below solution was accepted and 96.85% faster than other Python submissions

```
class Solution:
    def nextGreaterElements(self, nums: List[int]) -> List[int]:
```

2





 Share  Reply



[SHOW 1 REPLY](#)
- 


shlykovich ★ 213 · April 17, 2019 1:05 AM

Similar to stack approach is to use a min-heap and do two iterations over an array. First pass will push element to the heap with its corresponding position (value, index), and at the same time pop zero or more elements from the heap if current value is bigger than min element in To handle wrap around, repeat the same loop just skip elements if index position is smaller than current element and do not add anything to the heap.

2





 Share  Reply



[SHOW 1 REPLY](#)
- 


RogerFederer ★ 857 · January 5, 2018 3:25 AM

```
def nextGreaterElements(self, nums):
    """
    :type nums: List[int]
    :rtype: List[int]
```

2





 Share  Reply



[SHOW 1 REPLY](#)
- 

sunsys ★ 30 · November 28, 2017 8:52 PM

@vinod23 Your said "Approach #3 Using Stack's Time complexity is O(n). But I think the while is also like for loop to compare one by one, find and pick up the next Greater Element. So its Time complexity is also O(n*n) as worst.

3



 Share  Reply

[SHOW 1 REPLY](#)