<https://leetcode.com/problems/next-greater-element-ii/description/>

Given a circular integer array nums (i.e., the next element of nums[nums.length - 1] is nums[0]), return *the* ***next greater number*** *for every element in* nums.

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, return -1 for this number.

**Example 1:**

Input: nums = [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.

**Example 2:**

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

Output: [2,3,4,-1,4]

**Constraints:**

* 1 <= nums.length <= 104
* -109 <= nums[i] <= 109

**Attempt 1: 2023-03-18**

**Solution 1: Brute Force (10 min)**

**The difference between L503 and L496:**

**In L503 we can set 'j' start with 'i + 1' because L503 just need to expand original 'nums' array by concatenate itself again, every next greater number index definitely will start from 'i + 1', even when consider the expanded part.**

**But L496 we cannot set 'j' start with 'i + 1' because L496 has two arrays, nums1 is subset of nums2, the number of nums1 in nums2's index are undetermined yet, we have to use map to determine the number of nums1 in nums2's index first, then the next greater number index will only after that index**

class Solution {

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

int n = nums.length;

int[] expand = new int[n \* 2];

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

expand[i] = nums[i % n];

}

int[] result = new int[n];

Arrays.fill(result, -1);

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

// The difference between L503 and L496:

// In L503 we can set 'j' start with 'i + 1' because L503 just need

// to expand original 'nums' array by concatenate itself again, every

// next greater number index definitely will start from 'i + 1', even

// when consider the expanded part

// But L496 we cannot set 'j' start with 'i + 1' because L496 has two

// arrays, nums1 is subset of nums2, the number of nums1 in nums2's

// index are undetermined yet, we have to use map to determine the number

// of nums1 in nums2's index first, then the next greater number index

// will only after that index

for(int j = i + 1; j < n \* 2; j++) {

if(expand[i] < expand[j % n]) {

result[i] = expand[j % n];

break;

}

}

}

return result;

}

}

**Refer to**

<https://leetcode.com/problems/next-greater-element-ii/solutions/98262/typical-ways-to-solve-circular-array-problems-java-solution/>

The first typical way to solve circular array problems is to extend the original array to twice length, 2nd half has the same element as first half. Then everything become simple.

Naïve by simple solution, just look for the next greater element directly. Time complexity: O(n^2).

public class Solution {

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

int max = Integer.MIN\_VALUE;

for (int num : nums) {

max = Math.max(max, num);

}

int n = nums.length;

int[] result = new int[n];

int[] temp = new int[n \* 2];

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

temp[i] = nums[i % n];

}

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

result[i] = -1;

if (nums[i] == max) continue;

for (int j = i + 1; j < n \* 2; j++) {

if (temp[j] > nums[i]) {

result[i] = temp[j];

break;

}

}

}

return result;

}

}

**Solution 2: Decreasing Monotonic Stack (30 min)**

**Decreasing Monotonic Stack definition: All the elements in the stack are in decreasing order from bottom to top. The determine of "decreasing monotonic stack" here is based on "in process status" and the actual value based on stored 'index' on stack, NOT the "final storage status" of stack or the 'index' itself on stack.**

**Style 1: Store index not num itself**

class Solution {

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

int n = nums.length;

int[] result = new int[n];

Arrays.fill(result, -1);

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

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

while(!stack.isEmpty() && nums[stack.peek()] < nums[i % n]) {

result[stack.pop()] = nums[i % n];

}

stack.push(i % n);

}

return result;

}

}

**Refer to**

<https://leetcode.com/problems/next-greater-element-ii/solutions/98270/java-c-python-loop-twice/>

## **Explanation**

Loop once, we can get the Next Greater Number of a normal array.

Loop twice, we can get the Next Greater Number of a circular array

## **Complexity**

Time O(N) for one pass

Space O(N) in worst case

**Java**

public int[] nextGreaterElements(int[] A) {

int n = A.length, res[] = new int[n];

Arrays.fill(res, -1);

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

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

while (!stack.isEmpty() && A[stack.peek()] < A[i % n])

res[stack.pop()] = A[i % n];

stack.push(i % n);

}

return res;

}

**Style 2: Store num itself not index (Not applicable since "result[stack.pop()] = nums[i % n];" depends on index stored on stack)**

**Solution 3: Decreasing Monotonic Stack (30 min)**

**Similar as Solution 2 but traversal from right to left**

**The major difference between Solution 2 (traversal from left to right) and Solution 3 (traversal from right to left) is Solution 2 can only "Store index" but Solution 3 can "Store index" or "Store num itself", this is because in Solution 2, we have a line "result[stack.pop()] = nums[i % n];" depends on 'index' stored on stack, but in Solution 3, the result value assignment line change to "result[i % n] = stack.isEmpty() ? -1 : nums[stack.peek()];" only relate to for loop local variable 'i' and no depends on 'index', stack can either store 'index' or 'num itself'**

**Style 1: Store index not num itself**

class Solution {

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

int n = nums.length;

int[] result = new int[n];

Arrays.fill(result, -1);

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

for(int i = 2 \* n - 1; i >= 0; i--) {

while(!stack.isEmpty() && nums[stack.peek()] <= nums[i % n]) {

stack.pop();

}

result[i % n] = stack.isEmpty() ? -1 : nums[stack.peek()];

stack.push(i % n);

}

return result;

}

}

**Refer to**

<https://leetcode.com/problems/next-greater-element-ii/solutions/98273/java-10-lines-and-c-12-lines-linear-time-complexity-o-n-with-explanation/comments/102644>

class Solution {

public:

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

if (nums.empty()) return {};

int n = nums.size();

vector<int> res(n, -1);

stack<int> st;

for (int i = 2\*n-1; i >= 0; --i) {

int num = nums[i%n];

while (!st.empty() && nums[st.top()] <= num) st.pop();

res[i%n] = st.empty() ? - 1 : nums[st.top()];

st.push(i%n);

}

return res;

}

};

**Style 2: Store num itself not index**

class Solution {

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

int n = nums.length;

int[] result = new int[n];

Arrays.fill(result, -1);

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

for(int i = 2 \* n - 1; i >= 0; i--) {

while(!stack.isEmpty() && stack.peek() <= nums[i % n]) {

stack.pop();

}

result[i % n] = stack.isEmpty() ? -1 : stack.peek();

stack.push(nums[i % n]);

}

return result;

}

}

**Refer to**

<https://leetcode.com/problems/next-greater-element-ii/solutions/98273/java-10-lines-and-c-12-lines-linear-time-complexity-o-n-with-explanation/comments/102633>

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

int len = nums.length;

int[] result = new int[len];

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

for (int i = len \* 2 - 1; i >= 0; i--) {

while (!stack.isEmpty() && stack.peek() <= nums[i % len]) stack.pop();

if (i < len) result[i] = stack.isEmpty()? -1 : stack.peek();

stack.push(nums[i % len]);

}

return result;

}

**Refer to**

<https://leetcode.com/problems/next-greater-element-ii/solutions/98262/typical-ways-to-solve-circular-array-problems-java-solution/>

The second way is to use a stack to facilitate the look up. First we put all indexes into the stack, smaller index on the top. Then we start from end of the array look for the first element (index) in the stack which is greater than the current one. That one is guaranteed to be the Next Greater Element. Then put the current element (index) into the stack.

Time complexity: O(n).

public class Solution {

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

int n = nums.length;

int[] result = new int[n];

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

for (int i = n - 1; i >= 0; i--) {

stack.push(i);

}

for (int i = n - 1; i >= 0; i--) {

result[i] = -1;

while (!stack.isEmpty() && nums[stack.peek()] <= nums[i]) {

stack.pop();

}

if (!stack.isEmpty()){

result[i] = nums[stack.peek()];

}

stack.add(i);

}

return result;

}

}