

# 496. Greater Element I

March 14, 2017 | 34K views

Average Rating: 4.53 (47 votes)

You are given two arrays (without duplicates) `nums1` and `nums2` where `nums1`'s elements are subset of `nums2`. Find all the next greater numbers for `nums1`'s elements in the corresponding places of `nums2`.

The Next Greater Number of a number `x` in `nums1` is the first greater number to its right in `nums2`. If it does not exist, output -1 for this number.

## Example 1:

Input: `nums1 = [4,1,2]`, `nums2 = [1,3,4,2]`.  
Output: `[-1,3,-1]`  
Explanation:  
For number 4 in the first array, you cannot find the next greater number for it in `nums2`.  
For number 1 in the first array, the next greater number for it in the second array is 3.  
For number 2 in the first array, there is no next greater number for it in the second array.

## Example 2:

Input: `nums1 = [2,4]`, `nums2 = [1,2,3,4]`.  
Output: `[3,-1]`  
Explanation:  
For number 2 in the first array, the next greater number for it in the second array is 3.  
For number 4 in the first array, there is no next greater number for it in the second array.

## Note:

- 1. All elements in `nums1` and `nums2` are unique.
- 2. The length of both `nums1` and `nums2` would not exceed 1000.

## Summary

You are given two arrays (without duplicates) `findNums` and `nums` where `findNums`'s elements are subset of `nums`. Find all the next greater numbers for `findNums`'s elements in the corresponding places of `nums`.

The Next Greater Number of a number `x` in `findNums` is the first greater number to its right in `nums`. If it does not exist, output -1 for this number.

## Solution

### Approach #1 Brute Force [Accepted]

In this method, we pick up every element of the `findNums` array (say `findNums[i]`) and then search for its own occurrence in the `nums` array (which is indicated by setting `found` to `True`). After this, we look linearly for a number in `nums` which is greater than `findNums[i]`, which is also added to the `res` array to be returned. If no such element is found, we put a -1 at the corresponding location.

JavaCopy

```
1 public class Solution {
2     public int[] nextGreaterElement(int[] findNums, int[] nums) {
3         int[] res = new int[findNums.length];
4         int j;
5         for (int i = 0; i < findNums.length; i++) {
6             boolean found = false;
7             for (j = 0; j < nums.length; j++) {
8                 if (found && nums[j] > findNums[i]) {
9                     res[i] = nums[j];
10                    break;
11                }
12                if (nums[j] == findNums[i]) {
13                    found = true;
14                }
15            }
16            if (j == nums.length) {
17                res[i] = -1;
18            }
19        }
20        return res;
21    }
22 }
```

### Complexity Analysis

- Time complexity:  $O(m * n)$ . The complete `nums` array (of size `n`) needs to be scanned for all the `m` elements of `findNums` in the worst case.
- Space complexity:  $O(m)$ . `res` array of size `m` is used, where `m` is the length of `findNums` array.

### Approach #2 Better Brute Force [Accepted]

Instead of searching for the occurrence of `findNums[i]` linearly in the `nums` array, we can make use of a hashmap `hash` to store the elements of `nums` in the form of (`element, index`). By doing this, we can find `findNums[i]`'s index in `nums` array directly and then continue to search for the next larger element in a linear fashion.

JavaCopy

```
1 public class Solution {
2     public int[] nextGreaterElement(int[] findNums, int[] nums) {
3         HashMap< Integer, Integer > hash = new HashMap< > ();
4         int[] res = new int[findNums.length];
5         int j;
6         for (int i = 0; i < findNums.length; i++) {
7             hash.put(nums[i], i);
8         }
9         for (int i = 0; i < findNums.length; i++) {
10            for (j = hash.get(findNums[i]) + 1; j < nums.length; j++) {
11                if (findNums[i] < nums[j]) {
12                    res[i] = nums[j];
13                    break;
14                }
15            }
16            if (j == nums.length) {
17                res[i] = -1;
18            }
19        }
20        return res;
21    }
22 }
```

### Complexity Analysis

- Time complexity:  $O(m * n)$ . The whole `nums` array, of length `n` needs to be scanned for all the `m` elements of `findNums` in the worst case.
- Space complexity:  $O(m)$ . `res` array of size `m` is used. A hashmap `hash` of size `m` is used, where `m` refers to the length of the `findNums` array.

### Approach #3 Using Stack [Accepted]

In this approach, we make use of pre-processing first so as to make the results easily available later on. We make use of a stack (`stack`) and a hashmap (`map`). `map` is used to store the result for every possible number in `nums` in the form of (`element, next_greater_element`). Now, we look at how to make entries in `map`.

We iterate over the `nums` array from the left to right. We push every element `nums[i]` on the stack if it is less than the previous element on the top of the stack (`stack[top]`). No entry is made in `map` for such `nums[i]`'s right now. This happens because the `nums[i]`'s encountered so far are coming in a descending order.

If we encounter an element `nums[i]` such that `nums[i] > stack[top]`, we keep on popping all the elements from `stack[top]` until we encounter `stack[k]` such that `stack[k] ≤ nums[i]`. For every element popped out of the stack `stack[j]`, we put the popped element along with its next greater number (result) into the hashmap `map`, in the form (`stack[j], nums[i]`). Now, it is obvious that the next greater element for all elements `stack[j]`, such that `k < j ≤ top` is `nums[i]` (since this larger element caused all the `stack[j]`'s to be popped out). We stop popping the elements at `stack[k]` because this `nums[i]` can't act as the next greater element for the next elements on the stack.

Thus, an element is popped out of the stack whenever a next greater element is found for it. Thus, the elements remaining in the stack are the ones for which no next greater element exists in the `nums` array. Thus, at the end of the iteration over `nums`, we pop the remaining elements from the `stack` and put their entries in `hash` with a -1 as their corresponding results.

Then, we can simply iterate over the `findNums` array to find the corresponding results from `map` directly.

The following animation makes the method clear:

2351073

1 / 22

JavaCopy

```
1 public class Solution {
2     public int[] nextGreaterElement(int[] findNums, int[] nums) {
3         Stack< Integer > stack = new Stack< > ();
4         HashMap< Integer, Integer > map = new HashMap< > ();
5         int[] res = new int[findNums.length];
6         for (int i = 0; i < nums.length; i++) {
7             while (!stack.empty() && nums[i] > stack.peek()) {
8                 map.put(stack.pop(), nums[i]);
9                 stack.push(nums[i]);
10            }
11            while (stack.empty()) {
12                map.put(stack.pop(), -1);
13            }
14            for (int i = 0; i < findNums.length; i++) {
15                res[i] = map.get(findNums[i]);
16            }
17        }
18        return res;
19    }
20 }
```

### Complexity Analysis

- Time complexity:  $O(m + n)$ . The entire `nums` array (of size `n`) is scanned only once. The stack's `n` elements are popped only once. The `findNums` array is also scanned only once.
- Space complexity:  $O(m + n)$ . `stack` and `map` of size `n` is used. `res` array of size `m` is used, where `n` refers to the length of the `nums` array and `m` refers to the length of the `findNums` array.

Rate this article: ★★★★★

Previous Next

## Comments: 29

Sort By

Type comment here... (Markdown is supported)

PreviewPost

cg91995 ★479 · December 24, 2018 8:44 AM Report

I dont understand why this question received so many downvotes. This is a perfect example of monotonous stack... Its important.

98 · Share · Reply

SHOW 7 REPLIES

xdk3869 ★16 · June 3, 2018 5:34 AM

Not sure if we need a stack. If we scan nums2 backwards and use a map to track (element, nextGreater), we effectively get the effect of the stack:

```
public int[] nextGreaterElement(int[] nums1, int[] nums2) {
    if (nums1 == null || nums1.length == 0) r
```

16 · Share · Reply

SHOW 3 REPLIES

vishalshah3584 ★46 · April 21, 2019 1:05 PM

#approach 3 , very nice solution

2 · Share · Reply

charlem ★1 · July 12, 2019 12:37 AM

Why time complexity of Approach 3 is  $O(n+m)$  ?  
For every element in `nums` , we "loop"/pop through the stack, and a number of items there I guess could be close to `n`, thus giving us  $O(n*n)$  . Just for that first part,  $O(n*n+m)$  is in total.  
Fundamental in "Combinatoric Analysis" is "The stack's elements are popped only once." - but what once?

1 · Share · Reply

SHOW 2 REPLIES

Judew ★56 · February 25, 2019 7:32 AM Report

Concise JS Solution

```
var nextGreaterElement = function(nums1, nums2) {
    var result = [];
    var map = {};
```

0 · Share · Reply

cloudjlb ★5 · January 3, 2019 8:03 AM

I did the adjusted brute force (with hashmap) first and it was faster than 51.X % then I added the stack (came up with it finally on my own) and it's getting slower ratings. ??

0 · Share · Reply

SHOW 1 REPLY

oscardoudou ★51 · August 24, 2018 7:41 AM

Could somebody tell me why there is another 3 in gif(slide), isn't it without duplicated?  
And also what does "we keep on popping all the elements from stack[top] until we encounter stack[k] such that stack[k] ≤ nums[i]" mean? stack? Shouldn't it be nums[i] ≤ stack[k]? Consider 5 0 1 4, we push 5 then 0 then 1 first until we meet 4, we pop 1 then 0, since 5 > 4 (current nums[i]), we stop pop.

0 · Share · Reply

SHOW 2 REPLIES

prakashmanwani ★2 · February 9, 2018 1:18 AM

```
class Solution {
    public int[] nextGreaterElement(int[] nums1, int[] nums2) {
        int[] answer = new int[nums1.length];
```

0 · Share · Reply

L0u1s ★25 · January 30, 2018 11:11 PM Report

There's a way to do it in  $O(m+n)$  time and  $O(m)$  space to avoid using the stack. I posted my solution and analysis here: <https://discuss.leetcode.com/topic/118140/amortized-o-m-n-in-time-without-using-stack>

0 · Share · Reply

SHOW 1 REPLY

18fall ★0 · October 13, 2017 11:50 PM

Is the space complexity of second method is  $O(m+n)$ ? Since the size of the hashmap is `n` which is the length of `nums`.

0 · Share · Reply

1234