Previous Next

496. Greater Element I

March 14, 2017 | 34K views

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 ir
    For number 1 in the first array, the next greater number for it in the second arra
    For number 2 in the first array, there is no next greater number for it in the sec
```

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 arra
    For number 4 in the first array, there is no next greater number for it in the sec
```

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 occurence 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.

Java Copy

```java
public class Solution {
    public int[] nextGreaterElement(int[] findNums, int[] nums) {
        int[] res = new int[findNums.length];
        int j;
        for (int i = 0; i < findNums.length; i++) {
            boolean found = false;
            for (j = 0; j < nums.length; j++) {
                if (found && nums[j] > findNums[i]) {
                    res[i] = nums[j];
                    break;
                }
                if (nums[j] == findNums[i]) {
                    found = true;
                }
            }
            if (j == nums.length) {
                res[i] = -1;
            }
        }
        return res;
    }
}
```

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 occurence 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.

Java Copy

```java
public class Solution {
    public int[] nextGreaterElement(int[] findNums, int[] nums) {
        HashMap < Integer, Integer > hash = new HashMap < > ();
        int[] res = new int[findNums.length];
        int j;
        for (int i = 0; i < nums.length; i++) {
            hash.put(nums[i], i);
        }
        for (int i = 0; i < findNums.length; i++) {
            for (j = hash.get(findNums[i]) + 1; j < nums.length; j++) {

                if (findNums[i] < nums[j]) {
                    res[i] = nums[j];
                    break;
                }
            }
            if (j == nums.length) {
                res[i] = -1;
            }
        }
        return res;
    }
}
```

Complexity Analysis

- Time complexity : $O(m*n)$. The whole $nums$ array, of length n needs to be scanned for all the m elements of $finalNums$ 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 posssible number in $nums$ in the form of $(element, next\_greater\_element)$. Now, we look at how to make entries in map.

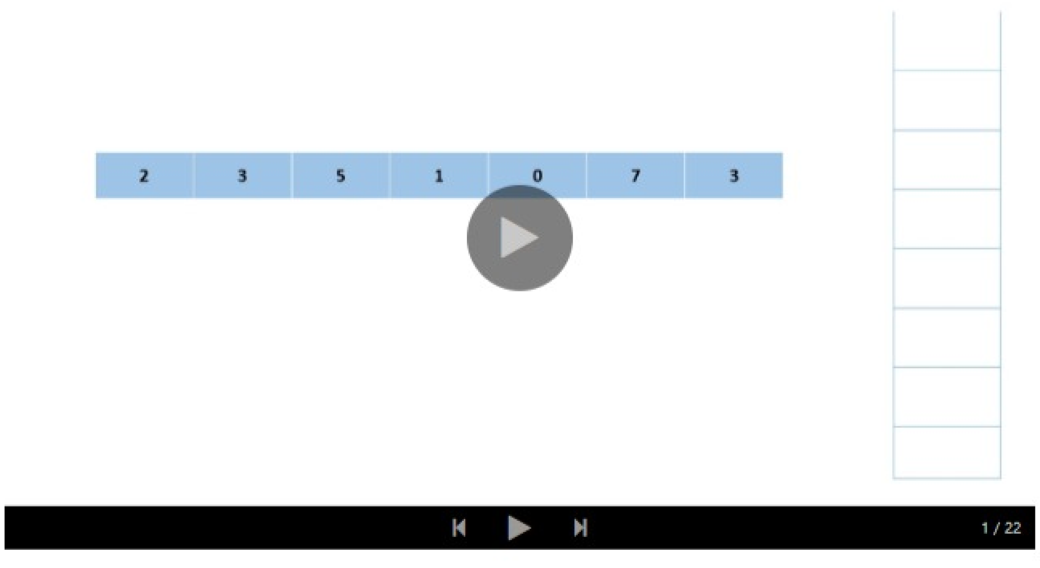
We iterate over the $nums$ array from the left to right. We push every element $nums[i]$ on the stack if it is less than the previous element on the top of the stack ($stack[top]$). No entry is made in map for such $nums[i]'s$ right now. This happens because the $nums[i]'s$ encountered so far are coming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

The following animation makes the method clear:



Java Copy

```java
public class Solution {
    public int[] nextGreaterElement(int[] findNums, int[] nums) {
        Stack < Integer > stack = new Stack < > ();
        HashMap < Integer, Integer > map = new HashMap < > ();
        int[] res = new int[findNums.length];
        for (int i = 0; i < nums.length; i++) {
            while (!stack.empty() && nums[i] > stack.peek())
                map.put(stack.pop(), nums[i]);
            stack.push(nums[i]);
        }
        while (!stack.empty())
            map.put(stack.pop(), -1);
        for (int i = 0; i < findNums.length; i++) {
            res[i] = map.get(findNums[i]);
        }
        return res;
    }
}
```

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.