496. Next Greater Element I

Easy | Stack | Array | Hash Table | Monotonic Stack — Leetcode Link

Problem Description

The problem is an application of the "next greater element" concept, where we need to find the immediate next greater element for a given element in one array, within another array. Specifically, we are given two arrays: `nums1` and `nums2`. The array `nums1` is a subset of `nums2` which means every element in `nums1` also appears in `nums2`. The task is to find out, for each element in `nums1`, what the next greater element is in `nums2`.

For each element `x` in `nums1`, we're supposed to look for the element which is the first one greater than `x` located to its right in the array `nums2`. If such an element exists, we register it as the next greater element. If there isn't any that's greater, we return `-1` for that particular element.

The output should be an array that corresponds to each element in `nums1` with its next greater element from `nums2`.

Intuition

The intuitive approach to solve this problem is by using a stack. First, we will iterate through `nums2` because this is the array where we're finding the next greater elements. As we want to find the next greater element that occurs after the given element, the stack allows us to keep track of the elements we've seen but haven't yet found a greater element for.

Here is the step-by-step intuition behind the solution:

1. We will create a dictionary called `m` to map each element in `nums2` to its next greater element. This will help in quickly looking up the result for each element in `nums1`.

2. We will also create an empty stack, `stk`, to maintain the elements for which we have to find the next greater element.

3. We go through each element `v` in `nums2`. For every element `v`, we do the following:

   - If the stack is not empty, we check the last element in the stack. If the last element in the stack is less than `v`, it means that `v` is the next greater element for that stack's top element. So we pop the top from the stack and record `v` as the next greater element in our dictionary `m`.
   - We continue to compare `v` with the new top of the stack and do the above step until the stack is empty or the top of the stack is no longer less than `v`.
   - We push `v` onto the stack because we need to find the next greater element for it.
4. Once we have completely processed `nums2` in the above way, we have our `m` dictionary with the next greater elements for all elements in `nums2` where they exist.

5. Finally, for each element `v` in `nums1`, we look up our dictionary `m`. If `v` is in the dictionary, we put `m[v]` in the result array; otherwise, we put `-1`.

This approach effectively tracks the elements that are yet to find their next greater element and finds the valid greater elements for them in a single pass through `nums2` due to the stack's LIFO property.

Solution Approach

The solution utilizes a stack and a hash map (dictionary in Python) for an efficient approach to solve the problem. Let's walk through the implementation step by step:

1. **Hash Map to Store Next Greater Elements**: A dictionary `m` is created to map each element from `nums2` to its next greater element. The key is the element from `nums2`, and the value is its next greater element in `nums2`.

2. **Stack to Keep Track of Elements**: A stack (`stk`) is used to keep track of the elements for which we need to find the next greater element. The stack maintains the indices of elements in a decreasing sequence, so whenever we encounter a greater element, we know that it is the next greater element for all elements in the stack that are less than it.

3. **Iterating Over `nums2`**: We iterate over each element `v` in `nums2`:

   - **Stack Not Empty**: While the stack is not empty and the element on top of the stack is less than `v` — `stk[-1] < v` — it means we have found the next greater element for the top of the stack. We `pop()` the element from the stack and add an entry in the dictionary `m` with the `pop`ped element as the key and `v` as the value — `m[stk.pop()] = v`.
   - **Element Has Not Found Next Greater**: If we did not find a next greater element or the stack is empty, we `append()` the current element `v` onto the stack.
4. **Mapping for `nums1` Elements**: After populating the dictionary with the correct mappings, we iterate through `nums1` and use list comprehension to build the result list. For each element `v` in `nums1`, we use the dictionary `m` to find the next greater element. If `v` is in the dictionary, we know its next greater element — `m.get(v, -1)`. If `v` is not in the dictionary, it means there is no next greater element, and we use `-1` as a default.

This pattern is an example of the Monotonic Stack, which is often used in problems where we need to find the next larger (or smaller) element in a list. The Monotonic Stack maintains elements in a sorted manner that allows for efficient retrieval of the next greater or smaller element. The dictionary (hash map) is used for direct access to the results for elements of `nums1`, making the final assembly of the output array the fast operation.

By using a combination of a stack and a hash map, we achieve a time-efficient solution that only requires a single pass through `nums2` — O(n) complexity, where n is the number of elements in `nums2`.

Example Walkthrough

Let's assume we have the following two arrays:

- `nums1 = [4, 1, 2]`
- `nums2 = [1, 3, 4, 2]`

Now, using the stacked solution approach mentioned above, we're going to find the next greater element for each item in `nums1` using `nums2`. Here's a step-by-step illustration:

1. Initialize an empty stack `stk` and an empty dictionary `m`.

2. Start iterating over `nums2`.

   - When `v = 1`, the `stk` is empty, so push `1` onto the stack.
   - When `v = 3`, `stk` top (1) is less than `3`. So according to our approach:
     - Pop `1` from the stack.
     - Add `{1: 3}` to the dictionary `m`.
     - Push `3` onto the stack since we need to find its next greater element.
   - When `v = 4`, `stk` top (3) is less than `4`. So:
     - Pop `3` from the stack.
     - Add `{3: 4}` to `m`.
     - No need to check for the next element in the stack since `stk` is now empty.
     - Push `4` onto the stack.
   - When `v = 2`, it's not greater than `stk` top (4), so just push `2` onto the stack.
3. Now we have our `m` dictionary with entries `{1: 3, 3: 4}`. Any remaining elements in the `stk` don't have a next greater element in `nums2`, so they can be assigned `-1` in `m`. After doing so, `m` will look like `{1: 3, 3: 4, 4: -1, 2: -1}`.

4. The final step is to iterate through `nums1` and compile the result using the dictionary `m`.

   - For `4`, `m[4]` gives `-1`.
   - For `1`, `m[1]` yields `3`.
   - For `2`, `m[2]` results in `-1`.

So the final output, which is the next greater element of each item in `nums1` as per their order in `nums2` is `[-1, 3, -1]`.

Python Solution

```python
from typing import List

class Solution:
    def nextGreaterElement(self, nums1: List[int], nums2: List[int]) -> List[int]:
        # Mapping to store the next greater element for each number
        next_greater_mapping = {}

        # Stack to keep track of the elements for which we have not found the next greater element yet
        stack = []

        # Iterate through each number in the second list
        for number in nums2:
            # If the current number is greater than the last number in the stack,
            # It is the next greater element for that number in the stack.
            # Pop elements from the stack and update the mapping accordingly
            while stack and stack[-1] < number:
                next_greater_mapping[stack.pop()] = number

            # Push the current number onto the stack
            stack.append(number)

        # Go through each number in the first list and get the next greater element from the mapping
        # If a number does not have a next greater element in the second list,
        # the get method returns -1
        return [next_greater_mapping.get(num, -1) for num in nums1]

```

Java Solution

```java
class Solution {
    public int[] nextGreaterElement(int[] nums1, int[] nums2) {
        // Use a stack to keep track of the elements for which we want to find the next greater element
        Deque<Integer> stack = new ArrayDeque<>();
        // Create a map to store the next greater element for each number in nums2
        Map<Integer, Integer> nextGreaterMap = new HashMap<>();

        // Loop through each element in nums2
        for (int num : nums2) {
            // While there is an element in the stack and it is smaller than the current number
            while (!stack.isEmpty() && stack.peek() < num) {
                // Pop the element from the stack and put its next greater element (num) in the map
                nextGreaterMap.put(stack.pop(), num);
            }
            // Push the current number onto the stack
            stack.push(num);
        }

        // Initialize the array to store the next greater elements for nums1
        int n = nums1.length;
        int[] result = new int[n];

        // Loop through each element in nums1
        for (int i = 0; i < n; ++i) {
            // If nums1[i] has a next greater element in nums2, use it; otherwise, use -1
            result[i] = nextGreaterMap.getOrDefault(nums1[i], -1);
        }
        // Return the result array with the next greater elements for nums1
        return result;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <stack>
#include <unordered_map>

using namespace std;

class Solution {
public:
    // Function to find the next greater element for elements in nums1 based on their
    // order in nums2.
    vector<int> nextGreaterElement(vector<int>& nums1, vector<int>& nums2) {
        stack<int> stack; // Stack to maintain the order of elements.
        unordered_map<int, int> nextGreaterMap; // Map to associate each number with its next greater element.
        vector<int> result; // Result vector to hold the next greater elements for nums1.

        // Iterate over each number in nums2 to find the next greater element.
        for (int num : nums2) {
            // While there are elements in the stack and the current element
            // is greater than the top element of the stack.
            while (!stack.empty() && stack.top() < num) {
                // Map the top element of the stack to the current element.
                nextGreaterMap[stack.top()] = num;
                stack.pop(); // Remove the top element as its next greater element is found.
            }
            // Push the current element onto the stack.
            stack.push(num);
        }

        // Iterate over each number in nums1 to build the result vector.
        for (int num : nums1) {
            // If the number has a next greater element in the map then add it to the result.
            // If not, add -1 to represent there is no next greater element.
            result.push_back(nextGreaterMap.count(num) ? nextGreaterMap[num] : -1);
        }

        return result; // Return the result vector.
    }
};

```

Typescript Solution

```typescript
// Function to find the next greater element for each element of nums1 in nums2
function nextGreaterElement(nums1: number[], nums2: number[]): number[] {
    // Create a map to hold the next greater element for numbers in nums2
    const nextGreaterMap = new Map<number, number>();

    // Initialize a stack with a sentinel value (Infinity) to handle the comparison edge case
    const monotonicallyDecreasingStack: number[] = [Infinity];

    // Iterate over each number in nums2 to compute the next greater element
    for (const num of nums2) {
        // Pop elements from the stack that are smaller than the current number
        // and record the current number as their next greater element in the map
        while (num > monotonicallyDecreasingStack[monotonicallyDecreasingStack.length - 1]) {
            const top = monotonicallyDecreasingStack.pop();
            if (top !== undefined) {
                nextGreaterMap.set(top, num);
            }
        }
        // Push the current number onto the stack
        monotonicallyDecreasingStack.push(num);
    }

    // For each number in nums1, find and return the next greater number using the precomputed map
    // If the number is not present in the map, use -1 indicating no greater number exists
    const result: number[] = nums1.map(num => nextGreaterMap.get(num) || -1);

    return result;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the `nextGreaterElement` function is primarily determined by the loop that iterates over the elements of `nums2`, and the inner while-loop that operates when the elements in the stack `stk` are smaller than the current element `v`.

Each element of `nums2` is pushed onto the stack exactly once. The `while` loop inside the `for` loop pops elements from the stack and will also execute at most once for every element, since an element will be popped only if it has found a greater element and will not be pushed back.

Therefore, every element from `nums2` is pushed and popped from the stack at most once, resulting in an overall time complexity of `O(n)`, where `n` is the number of elements in `nums2`.

The final list comprehension `[m.get(v, -1) for v in nums1]` has a time complexity of `O(m)`, where `m` is the number of elements in `nums1`. However, since the computation of the hash map `m` is the most expensive part and considering that `nums1` is a subset of `nums2`, we can conclude that the overall time complexity is `O(n)`.

Space Complexity

The space complexity of the function comes from the stack `stk` and the hash map `m`. In the worst case, the stack `stk` can store all the elements of `nums2`, which would require `O(n)` space. The hash map `m` can potentially store each element of `nums2` along with its corresponding next greater element, also resulting in `O(n)` space used.

Adding the space required by the stack and the hash map, the overall space complexity of the function is `O(n)`.

So, both the time complexity and the space complexity of the `nextGreaterElement` function are `O(n)`.