Solution: Next Greater Element

Let's solve the Next Greater Element problem using the Hash Map pattern.

We'll cover the following

- Statement
- Solution
  - Naive approach
  - Optimized solution using hash map
    - Solution summary
    - Time complexity
    - Space complexity

Statement

Given the two distinct integer arrays, `nums1` and `nums2`, where `nums1` is a subset of `nums2`, find all the next greater elements for `nums1` values in the corresponding places of `nums2`.

> **Note:** The next greater element of an element, x, in an array is the first greater element present on the right side of x in the same array.

For each element x in `nums1`, find the next greater element present on the right side of x in `nums2` and store it in the `ans` array. If there is no such element, store -1 for this number. The `ans` array should be of the same length as `nums1`, and the order of the elements in the `ans` array should correspond to the order of the elements in `nums1`.

Return the `ans` array after finding the next greater elements.

> **Note:** The input data may or may not be sorted.

Constraints:

- $1 \leq$ `nums1.length` $\leq$ `nums2.length` $\leq 10^3$
- $0 \leq$ `nums1[i]`, `nums2[i]` $\leq 10^4$
- `nums1` have distinct integers.
- `nums2` have distinct integers.
- All integers in `nums1` also appear in `nums2`.

Solution

You've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and implementation constraints.

Naive approach

The naive approach is to select each element of `nums1` and search for its occurrence in `nums2`. If the element is found, we look for the occurrence of its next greater element in `nums2` linearly. If the next greater element is obtained, we store it in the `ans` array in the corresponding place to the element in `nums1`. Otherwise, we store -1 in the `ans` array for that element.

The overall time complexity of the algorithm becomes $O(n_1 \times n_2)$, because we're searching for each element of the `nums1` array in the `nums2` array. The space complexity of this algorithm is $O(1)$.

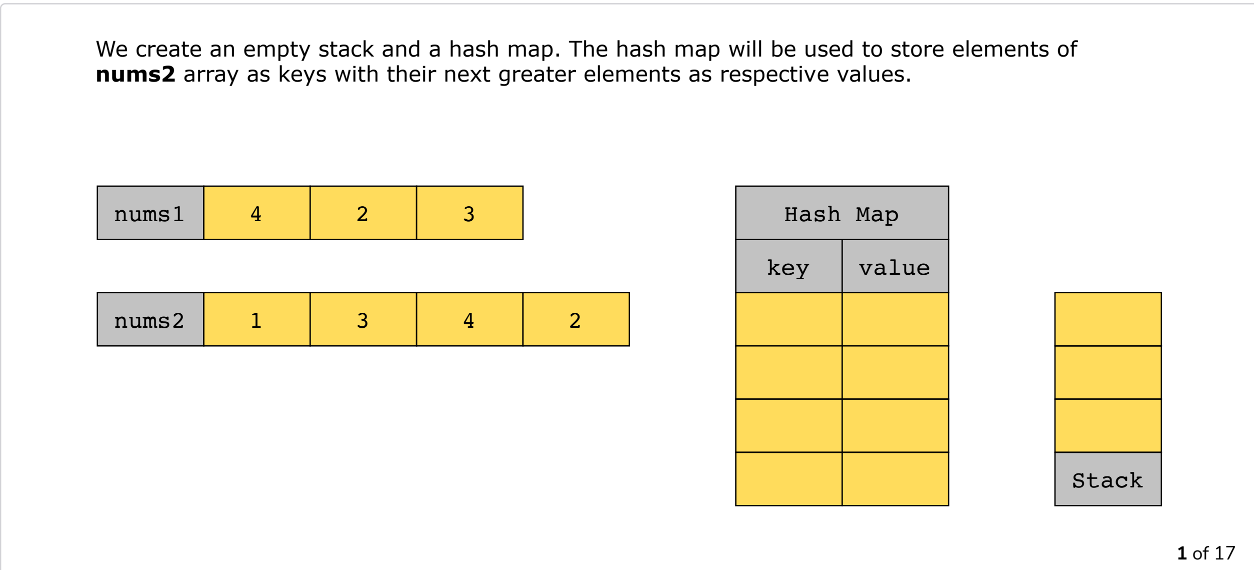
Optimized solution using hash map

An optimized approach to solve this problem is using a hash map and a stack. A hash map is used to store the elements in `nums2` as keys and their next greater elements as the respective values.

The algorithm proceeds through the following steps after creating an empty stack and a hash map:

- Iterate over each element of `nums2`, and if the stack is not empty, compare it with the top element of the stack.
    - If the current element of `nums2` is greater than the top element of the stack, pop the top element from the stack and put a key-value pair in the hash map with the popped element as the key and the current element of `nums2` as the value.
    - Repeat the step above until either the stack becomes empty or the current element of `nums2` is not greater than the top element of the stack.
- After each iteration over `nums2`, push the current element of `nums2` onto the stack.
- After processing all the elements of `nums2`, check if any elements are still remaining in the stack. If they are, pop them and put key-value pairs in the hash map with the remaining elements as the keys and -1 as their respective values.
- Finally, create an `ans` array with the same length as `nums1` and populate it with the values from the hash map that correspond to the keys in `nums1`.
- Return the `ans` array containing the next greater element for each element in `nums1`.

Let's look at the following illustration to get a better understanding of the solution:



Let's implement the algorithm as discussed above:

Java

```java
class NextGreater {

    public static int[] nextGreaterElement(int[] nums1, int[] nums2) {

        Stack<Integer> stack = new Stack<>();
        Map<Integer, Integer> map = new HashMap<>();

        // iterate over nums2
        for (int current : nums2) {
            // while stack is not empty and current element is greater than the top element of the stack
            while (!stack.empty() && current > stack.peek()) {
                // update the map with the current element as the value for the popped element
                map.put(stack.pop(), current);
            }
            // push the current element to the stack
            stack.push(current);
        }

        // iterate over remaining elements in the stack, pop them and set their values to -1 in the map
        while (!stack.empty()) {
            map.put(stack.pop(), -1);
        }

        int[] ans = new int[nums1.length];
        // iterate over nums1 and add the corresponding value from the map to ans
        for (int i = 0; i < nums1.length; i++) {
            ans[i] = map.get(nums1[i]);
        }
```

Next Greater Element

Solution summary

1. Create an empty stack and an empty hash map.
2. Iterate over `nums2`, and for each element, compare it with the top element of the stack.
3. If the current element of `nums2` is greater than the top element, pop the top element and put a key-value pair in the hash map with the popped element as the key and the current element of `nums2` as the value.
4. Push the current element onto the stack.

6. Finally, iterate over `nums1`, and for each element, append its corresponding value from the hash map to a new array, `ans`.
7. Return the `ans` array as the final result.

Time complexity

The `for` loop iterating over the elements of `nums2` takes $O(n)$ time, where n is the length of `nums2`. Each stack's n element is pushed and popped exactly once, taking $O(n)$ time. The `for` loop that populates the output array `ans` with values from the hash map takes $O(m)$ time, where m is the length of `nums1`.

So, the overall time complexity of the code is $O(n + n + m)$. Since `nums1` will always be a subset of `nums2`, m will always be less than or equal to n. Therefore, the time complexity can be simplified to $O(n)$.

Space complexity

The stack can contain a maximum of n elements, which is the length of `nums2`. The hash map can also contain a maximum of n key-value pairs, one for each element in `nums2`. Therefore, the total space used by the stack and hash map is proportional to the length of `nums2`, resulting in a $O(n)$ space complexity.