2454. Next Greater Element IV

Problem Description

The given problem presents us with an array of non-negative integers `nums`. Our task is to calculate the "second greater" integer for each element in the array. The "second greater" integer for `nums[i]` is the integer `nums[j]` that satisfies three conditions:

1. The index `j` is greater than index `i`.
2. The value at `nums[j]` is greater than the value at `nums[i]`.
3. There exists exactly one index `k`, such that `i < k < j`, and the value `nums[k]` is greater than `nums[i]` but less than `nums[j]`.

If no such `nums[j]` exists, then we should mark the second greater integer as `-1`.

For example, in the array `[1, 2, 4, 3]`, the second greater integer for the value `1` at index `0` is `4` at index `2`, since it's the only value that satisfies all the conditions (greater than `1`, comes after `1`, and there is exactly one value between `1` and `4` which is also greater than `1`). Following the same rules, the second greater integer for `3` is `-1`, and for `4` and `2` it is `-1` since no suitable integers follow them.

The goal is to generate and return an array `answer` where `answer[i]` is the second greater integer of `nums[i]`.

Intuition

The intuition for solving this problem is to make use of a stack (`stk`) and a heap queue (`q`) to efficiently track the greater elements. The approach can be broken down into the following steps:

1. Initialize two support structures: a stack (`stk`) to keep indices of elements from `nums` for potential candidates of the "first greater" integer, and a min-heap (`q`) to keep track of potential candidates for the "second greater" integer.
2. Traverse through the elements of `nums` and use the two data structures to check and assign the "second greater" integer for the current element if applicable.
3. When a new element `x` is found to be greater than the element of the top index in `stk`, we need to check if it can be the "second greater" for previous elements. Thus, we move such indices from `stk` to `q` because we've found their "first greater" and are now looking for their "second greater".
4. While encountering a new element `x` that is greater than the items in `q`, we update the answer for those indices because `x` now acts as their "second greater" integer. We can do this safely because the heap ensures the smallest element is always at the top, so we will encounter the "second greater" in the correct order.
5. Append the current index `i` to `stk`, because we've yet to find even the "first greater" for this element.
6. After traversing all elements, we return the `ans` array that contains the "second greater" integers or `-1` for each corresponding index.

This approach allows us to effectively find the necessary "second greater" integer for each element in a single pass throughout the array.

Solution Approach

The solution makes clever use of a stack and a heap queue (min-heap) to keep track of the potential "first greater" and "second greater" elements for each item in the `nums` array respectively. Here's a step-by-step explanation of the implementation details:

1. **Initialize Helper Structures**: We utilize a stack `stk` and a min-heap `q`. `stk` holds indices of elements for which we have not yet encountered any greater elements, and `q` maintains a pair `(value, index)` of potential "second greater" elements in ascending value order due to its heap structure.
2. **Iterate Through Elements**:
   - We iterate over each element `x` in `nums` array using its index `i`. We compare `x` with the elements represented by indices in `stk` and `q` to find and set their "second greater" elements.
3. **Update Answer for Elements in the Heap Queue**:
   - While the min-heap is not empty and the smallest element (at the heap's top) is less than `x`, we set the answer for that index (the second element of the heap's top pair) to `x`, as `x` is their "second greater" element. We then remove this top element from the heap with `heappop(q)`.
4. **Move Indices from Stack to Heap**:
   - For every element in `stk` where the corresponding value in `nums` is less than `x`, we push a pair `(value, index)` onto the min-heap and pop it from the stack. We do this because we've found their "first greater" element, which is `x`, and now we're interested in finding their "second greater" element.
5. **Maintain Stack**:
   - After dealing with the heap queue, we add the current index `i` to the `stk`. This is because for the current element at index `i`, either no greater element has been found yet, or it's potentially the "first greater" for upcoming elements.
6. **Final Answer**:
   - The array `ans` is initialized with `-1` for all indices to cover scenarios where no "second greater" element is found. Upon completion of the iteration, `ans` is updated with the "second greater" elements wherever applicable and is ready to be returned.

Using a stack and a min-heap like this allows us to maintain the invariant that `stk` contains indices strictly in increasing order of their corresponding values in `nums`. Simultaneously, the min-heap maintains the potential "second greater" candidates in the order in which they should be considered, ensuring that we can efficiently update `ans` as we traverse `nums`.

This algorithm effectively collapses nested loops that would otherwise be needed to find the "second greater" element for each item, thus improving the overall running time, especially for large arrays.

Example Walkthrough

Let's consider a small example to illustrate the solution approach: Assume our input array is `nums = [2, 7, 3, 5, 4]`.

1. **Initialize Helper Structures**: We start with an empty stack (`stk`) and min-heap (`q`), and an answer array `ans` initialized to `-1` for each index.
2. **Iterate Through Elements**:
   - For `i = 0, v = 2`: Since both `stk` and `q` are empty, we just push index `0` onto `stk`.
   - For `i = 1, v = 7`: `7` is greater than `2` (`nums[stk[-1]]`), so we move index `0` from `stk` to `q` with value `2`, and push index `1` onto `stk`.
   - For `i = 2, v = 3`: It's greater than `2` (value at top of the `q`) but less than `7` (`nums[stk[-1]]`) so we pop `(2, 0)` from `q` and update `ans[0]` to `3`, then push index `2` onto `stk`.
3. **Update Answer for Elements in the Heap Queue**:
   - The heap queue `q` currently has no elements less than the current `v`, so we skip this for index `2`.
4. **Move Indices from Stack to Heap**:
   - Index `2` stays in `stk` as `7` is greater than `3`, so we do not modify `q` for index `2`.
5. **Maintain Stack**:
   - We simply add index `3` to `stk`.
6. **Repeat Iteration**:
   - For `i = 3, v = 5`: It's greater than `3` (`nums[stk[-1]]`), so we move index `2` from `stk` to `q` with value `3`. Since `q` is now non-empty and the value at the top `(3, 2)` is less than `v = 5`, we pop `(3, 2)` from `q` and update `ans[2]` to `5`. Index `3` is pushed onto `stk`.
   - For `i = 4, v = 4`. It's less than `5` (`nums[stk[-1]]`), so index `4` is pushed onto `stk`.
7. **Final Answer**:
   - After iterating through the array, we have `ans = [-1, -1, 5, -1, -1]`. We have not updated `ans[1]` because `7` was never popped from `q`, indicating no second greater element was found for `7`. The same reasoning applies to `ans[3]` and `ans[4]`.

In summary, we performed an iterative process where we maintained a stack for indices of potential first greater elements and a heap queue for indices of potential second greater elements, popping from and pushing to these structures as needed while traversing the array. This allowed us to efficiently find and update the second greater element for each item, giving us `ans = [-1, -1, 5, -1, -1]` as our result.

Python Solution

```python
from typing import List
from heapq import heappush, heappop

class Solution:
    def secondGreaterElement(self, nums: List[int]) -> List[int]:
        # A stack to keep track of indices of elements in decreasing order
        stack = []
        # A min-heap to store the pair (element, index) for the potential second greater elements
        min_heap = []
        # Initialize the answer list with -1 for each element
        answer = [-1] * len(nums)

        # Iterate over the indices and values of nums.
        for i, num in enumerate(nums):
            # If the current value is greater than the element pointed by the current index
            # of the queue, pop elements from the heap and update their answer to the current value.
            while min_heap and min_heap[0][0] < num:
                _, index = heappop(min_heap)
                answer[index] = num

            # While the stack is not empty and the current value is greater than the top
            # element of the stack, push the pair (element of the stack, index) into the heap.
            while stack and nums[stack[-1]] < num:
                heappush(min_heap, (nums[stack[-1]], stack.pop()))

            # Append the current index to the stack.
            stack.append(i)

        # The stack now contains indices of elements for which no second greater element exists
        # and therefore their corresponding values in the answer list remain -1

        return answer

# The rewritten code now uses Python 3 syntax, has standardized variable naming,
# and includes comments explaining the functionality of the code.
```

Java Solution

```java
class Solution {
    public int[] secondGreaterElement(int[] nums) {
        // Stack to keep indexes of elements in a descending order
        Deque<Integer> stack = new ArrayDeque<>();

        // Priority Queue to store the elements and their corresponding indexes which are waiting for the second greater element
        PriorityQueue<int[]> queue = new PriorityQueue<>(a, b) -> a[0] - b[0]);

        // Length of the input array
        int length = nums.length;

        // Array to hold the final answer, initialized to -1 as default value
        int[] answer = new int[length];
        Arrays.fill(answer, -1);

        // Iterating over input array elements
        for (int i = 0; i < length; ++i) {
            int value = nums[i];

            // Process elements in the priority queue with value smaller than the current element
            while (!queue.isEmpty() && queue.peek()[0] < value) {
                // Assign the current value as the second greater element to the popped element
                answer[queue.poll()[1]] = value;
            }

            // Process elements in the stack with value smaller than the current element
            while (!stack.isEmpty() && nums[stack.peek()] < value) {
                // Add elements to the priority queue with their value and index
                queue.offer(new int[] {nums[stack.peek()], stack.pop()});
            }

            // Push current index onto the stack
            stack.push(i);
        }

        // Return the answer array containing the second greater element for each position
        return answer;
    }
}
```

C++ Solution

```cpp
// Including the necessary headers
#include <vector>
#include <stack>
#include <queue>
#include <utility>
using namespace std;

// Aliasing the 'pair<int, int>' as 'Pair',
typedef pair<int, int> Pair;

class Solution {
public:
    // This function finds the 'next greater element' for each element.
    // The 'next greater element' is defined as the first element to the right which is larger.
    vector<int> nextGreaterElement(vector<int>& nums) {
    vector<int> secondGreaterElement(vector<int>& nums) {
        // Stack used to keep track of elements indices for which we haven't found a greater element.
        stack<int> indexStack;
        // Priority queue used to store values with their indices, where the smallest is at the top.
        priority_queue<Pair, vector<Pair>, greater<Pair>> minHeap;

        int n = nums.size(); // Number of elements in the input vector.

        // Vector to store the answers, initialized with -1 for each element.
        vector<int> results(n, -1);

        // Iterate over each element in 'nums'.
        for (int i = 0; i < n; ++i) {
            int currentValue = nums[i];

            // Check if the top element of the min-heap is smaller than 'currentValue'.
            // If it is, the 'currentValue' is the next greater element for the index at the top of the heap.
            while (!minHeap.empty() && minHeap.top().first < currentValue) {
                // Assign 'currentValue' as the result for the corresponding index.
                results[minHeap.top().second] = currentValue;
                // Remove the element from the heap since it has been processed.
                minHeap.pop();
            }

            // Check if the elements remaining in the stack have a smaller value than 'currentValue'.
            while (!indexStack.empty() && nums[indexStack.top()] < currentValue) {
                // Push the pair of (value, index) onto the min-heap with their index.
                minHeap.push({nums[indexStack.top()], indexStack.top()});
                // Remove the element from the stack.
                indexStack.pop();
            }

            // Add the current index to the stack for future comparisons.
            indexStack.push(i);
        }

        // Return the final result vector.
        return results;
    }
};
```

Typescript Solution

```typescript
// The type alias for a pair of numbers where the first is the element and the second is the index.
type Pair = [number, number];

// The stack used to keep track of elements' indices for which we haven't found a greater element.
let indexStack: number[] = [];

// Simulating the priority queue (min-heap) using an array and a sort function.
let minHeap: Pair[] = [];

// The function to find the 'next greater element' for each element in the array.
// The 'next greater element' is the first larger element to the right of it or -1 if none exists.
function secondGreaterElement(nums: number[]): number[] {
    const n: number = nums.length; // Number of elements in the input array.
    let results: number[] = new Array(n).fill(-1); // Initialize the result array with -1s.

    // Function to simulate the 'pop' functionality on the minHeap.
    function popMinHeap() {
        minHeap.sort((a[0], b[0]) => a[0] - b[0]); // Ensure the smallest value is on top.
        return minHeap.shift(); // Remove and return the first element (top of the heap).
    }

    // Iterate over each element in 'nums'.
    for (let i = 0; i < n; ++i) {
        const currentValue = nums[i];

        // While there's a smaller element at the 'top' of the minHeap, update 'results'.
        while (minHeap.length > 0 && minHeap[0][0] < currentValue) {
            const [, index] = popMinHeap()!; // 'pop' the minHeap and get the index.
            results[index] = currentValue; // Assign 'currentValue' as the result for the index.
        }

        // While elements remaining in the stack have smaller values, add them to the minHeap.
        while (indexStack.length > 0 && nums[indexStack.top()] < currentValue) {
            const index = indexStack.pop()!; // Pop the index from the stack.
            minHeap.push([nums[index], index]); // Add the element to the minHeap.
        }

        // Add the current index to the stack.
        indexStack.push(i);
    }

    // Return the final result vector.
    return results;
}
```

Time and Space Complexity

The given algorithm finds the second greater element for each element in the list `nums`. It utilizes a stack (`stk`) and a priority queue (`q`, implemented as a min-heap using the heapq module in python).

Time Complexity:

- The algorithm traverses through each element in the list exactly once, which gives us a linear component of $O(n)$.
- Inside the loop, there are a couple of operations involving the stack and heap:
  - The `while stk and nums[stk[-1]] < x` loop can push a number of elements into the heap `q`, however, each element is pushed at most once, because once an element is popped from `stk` and pushed into `q`, it is not handled by this loop again.
  - The `while q and q[0][0] < x` loop pops elements from the heap until a number greater than the current number `x` is not found. Each element in the list could be popped at most once during the entire traversal. Heap operations (push and pop) are $O(\log k)$ where k is the number of elements in the heap. However, since we can perform at most n such operations in total (as the heap can never contain more elements than the number in the list), the complexity for heap operations over the traversal is $O(n \log n)$.
- Therefore, the total time complexity for the algorithm combines the linear traversal of the list and the $O(n \log n)$ complexity of heap operations, which gives us $O(n \log n)$.

Space Complexity:

- The stack `stk` and the heap `q` both store indices of elements. In the worst case, they could store all indices. Hence, the space complexity of these data structures is $O(n)$.
- An additional list `ans` is created to store the result, which also takes $O(n)$ space.
- Thus, combining the space required for the stack, heap, and the results list, the overall space complexity of the algorithm is $O(n)$, as the constants are dropped in Big O notation.