1673. Find the Most Competitive Subsequence

Medium Stack Greedy Array Monotonic Stack

Problem Description

In this problem, we are given an array `nums` of integers and another integer `k`. Our task is to find the most competitive subsequence of the array `nums` that has a size of `k`. A subsequence here means a sequence that can be derived by deleting any number of elements (including zero) from the array without changing the order of the remaining elements.

A subsequence is considered more competitive if at the first point of difference between two subsequences, the number in the more competitive one is smaller. For example, the subsequence `[1,3,4]` is more competitive than `[1,3,5]` because, at their first differing position, `4` is smaller than `5`.

To put it another way, we are looking for the smallest lexicographically subsequence of length `k` from the given array, where "smallest" refers to the subsequence that would come first if you listed all possible subsequences of the given length in numerical order.

Intuition

The intuition behind approaching this problem lies in maintaining a stack that can help us keep track of potential candidates for the most competitive subsequence. Here's the thought process:

1. We move through the array and for each element, we decide if it should be included in the stack (which represents the current most competitive subsequence we have built up).
2. We apply two checks while considering to push an element onto the stack: a. If the current element is smaller than the top of the stack (last element in our current subsequence), it could lead to a more competitive subsequence. So we might want to remove the larger elements from the top of the stack. b. However, we can only remove elements from the stack if we are sure that we can still complete our subsequence of size `k`. In other words, there must be enough elements left in the array to fill the stack after popping. This is guaranteed when `len(stk) + n - i > k` where `stk` is the stack, `n` is the number of elements in `nums`, and `i` is the current index.
3. After considering the above two points, if our stack size is less than `k`, we push the current element onto the stack as a potential candidate for the most competitive sequence. We keep doing this until we have scanned through all elements in the array.
4. After the loop finishes, the stack contains the most competitive subsequence of size `k`.

The stack effectively stores the smallest numbers seen so far and pops the larger ones that could be replaced by the following smaller numbers to maintain the competitiveness of the sequence. By the time the scanning of the array is complete, we have the most competitive subsequence.

Solution Approach

The solution uses a stack as a data structure to implement a greedy approach. Here's a step-by-step breakdown of the algorithm:

1. Initialize an empty list called `stk` which will be treated like a stack. This will be used to store the elements of the most competitive subsequence.
2. Determine the length of the input array `nums` and store it in a variable `n`.
3. Loop through each element `v` and its index `i` in the array using `enumerate(nums)`.
4. Inside the loop, there is a `while` loop with a compound condition that is used to decide whether the top element should be popped from the stack:
   - The stack should not be empty (there's an element to potentially pop).
   - The current element `v` is less than the last element in the stack (`stk[-1]`). This check is important because a smaller element makes a subsequence more competitive.
   - There are still enough elements left in `nums` (after `i`th index) to fill the stack to a length of `k` (the condition `len(stk) + n - i > k` ensures this).
5. If the above conditions are met, the top element of the stack is popped because keeping it would make our subsequence less competitive. We continue popping until the conditions fail, meaning we either exhaust the stack or the current element `v` is not smaller than the last in `stk` or there aren't enough elements remaining to reach length `k`.
6. After the `while` loop, a check ensures that we only add elements to `stk` if its length is less than `k`. Otherwise, our subsequence would exceed the desired length.
7. We append the current element `v` to the stack if the length of the stack is less than `k`.
8. After iterating over all elements in `nums`, the list `stk` will contain exactly `k` elements, which forms the most competitive sequence possible, and thus is returned.

During each iteration, the algorithm greedily tries to make the subsequence as small as possible by comparing the current number with the ones already in `stk`. This use of a stack allows the algorithm to keep the subsequence always in a competitive state by ensuring that each new added number would not make it any worse compared to other potential subsequences.

The main algorithmic patterns used here are:

- Greedy algorithm: Making local optimal choices at each step with the hope of finding a global optimum, i.e., the most competitive subsequence.
- Stack: Providing a way to keep track of elements and efficiently facilitate the necessary add and remove operations as per competitive conditions.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the following array `nums` and integer `k`:

```
nums = [3, 5, 2, 6]
k = 2
```

We want to find the most competitive subsequence of size `k` from `nums`.

Following the solution approach:

1. Initialize an empty stack `stk`.
2. Determine the length of `nums`, which is `n = 4`.
3. Begin iterating through `nums`:
   - At index `i = 0`, `v = 3`, there is nothing to pop because `stk` is empty. We add `3` to `stk`.
   - At index `i = 1`, `v = 5`, there is nothing to pop because `5` is not less than the last element in `stk`, which is `3`. Since `len(stk) < k`, we add `5` to `stk`.
   - At index `i = 2`, `v = 2`, `v` is less than the last element in `stk` (which is `5`) and `len(stk) + (n - i) > k` (`2` elements in `stk` + `2` elements left in `nums` > `2`). So we pop `5` from `stk` and add `2`.
   - At index `i = 3`, `v = 6`, there is nothing to pop since `6` is greater than the last element in `stk` (which is `2`). However, `len(stk)` is already `k`, so we don't add `6` to `stk`.
4. After the loop finishes, `stk = [2, 3]` which forms the most competitive subsequence because it is lexicographically the smallest subsequence of length `k` that we can form from `nums`.

The resulting most competitive subsequence is `[2, 3]`. The stack has helped us maintain the potential candidates for the subsequence efficiently, ensuring at each step that any larger numbers that could be replaced by smaller ones and do not contribute to the competitiveness are removed.

Python Solution

```python
class Solution:
    def mostCompetitive(self, nums: List[int], k: int) -> List[int]:
        # Initialize an empty list to use as a stack
        stack = []
        # Get the total number of elements in nums
        num_len = len(nums)

        # Iterate over the indices and values of nums
        for i, value in enumerate(nums):
            # While stack is not empty and the last element in stack is greater than the current value
            # and there are enough remaining elements to replace and still build a sequence of length k
            while stack and stack[-1] > value and len(stack) + num_len - i > k:
                # Pop the last element from the stack
                stack.pop()
            # If the stack contains fewer than k elements, append the current value
            if len(stack) < k:
                stack.append(value)

        # Return the stack which now contains the most competitive subsequence of length k
        return stack

```

Java Solution

```java
class Solution {

    public int[] mostCompetitive(int[] nums, int k) {
        // Use a deque as a stack to maintain the most competitive sequence.
        Deque<Integer> stack = new ArrayDeque<>();
        int n = nums.length;

        // Iterate over the array elements.
        for (int i = 0; i < nums.length; ++i) {
            // While the stack is not empty, and the current element is smaller than the top element of the stack,
            // and there are enough elements remaining to form a sequence of length k
            // we pop the stack since the current element gives a more competitive sequence.
            while (!stack.isEmpty() && stack.peek() > nums[i] && stack.size() + n - i > k) {
                stack.pop();
            }

            // If the stack size is less than k, we can push the current element onto the stack.
            if (stack.size() < k) {
                stack.push(nums[i]);
            }
        }

        // Create an array to store the most competitive sequence.
        int[] answer = new int[stack.size()];
        // Pop elements from the stack to fill the answer array in reverse order
        // since the elements in deque are in reverse order of the required sequence.
        for (int i = answer.length - 1; i >= 0; --i) {
            answer[i] = stack.pop();
        }
        // Return the array containing the most competitive sequence
        return answer;
    }
}
```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Function to find the most competitive subsequence of `nums` with size `k`
    std::vector<int> mostCompetitive(std::vector<int>& nums, int k) {
        std::vector<int> stack; // Stack to store the most competitive subsequence
        int numsSize = nums.size(); // Store the size of the input vector

        // Iterate over all the elements in the input vector
        for (int i = 0; i < numsSize; ++i) {
            // While the stack is not empty, the last element in the stack is greater than
            // the current number, and the potential size of the competitive subsequence
            // is still greater than `k`, pop the last element from the stack
            while (!stack.empty() && stack.back() > nums[i] && stack.size() + numsSize - i > k) {
                stack.pop_back();
            }
            // If the size of the stack is less than `k`, we can add the current number
            // to the subsequence
            if (stack.size() < k) {
                stack.push_back(nums[i]);
            }
        }
        // Return the most competitive subsequence
        return stack;
    }
};

```

Typescript Solution

```typescript
function mostCompetitive(nums: Array<number>, k: number): Array<number> {
    let stack: Array<number> = []; // Stack to store the most competitive subsequence
    let numsSize = nums.length; // Store the length of the input array

    // Iterate over all the elements in the input array
    for (let i = 0; i < numsSize; ++i) {
        // While the stack is not empty and the last element in the stack is greater than
        // the current element, and there are enough remaining elements to reach a
        // competitive subsequence of size `k`, pop the last element from the stack
        while (
            stack.length > 0 &&
            stack[stack.length - 1] > nums[i] &&
            stack.length + numsSize - i > k
        ) {
            stack.pop();
        }

        // If the current size of the stack is less than `k`, we can add the current element
        // to the subsequence
        if (stack.length < k) {
            stack.push(nums[i]);
        }
    }

    // Return the most competitive subsequence
    return stack;
}

// Example usage:
// let result = mostCompetitive([3, 5, 2, 6], 2);
// result would be [2, 6]

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the input list `nums`. Even though there is a nested loop caused by the `while` loop within the `for` loop, each element is pushed to and popped from the stack at most once. This guarantees that each operation is bounded by `2n` (each element pushed and popped at most once), which simplifies to `O(n)`.

Space Complexity

The space complexity is `O(k)` since we are using a stack to keep track of the most competitive subsequence, which, according to the problem, can have a maximum length of `k`. No other data structures are used that grow with the size of the input.