2869. Minimum Operations to Collect Elements

Easy Array Hash Table Leetcode Link

Problem Description

You are given an array of positive integers, named `nums`, and a target number represented by the integer `k`. Your goal is to collect a list of elements labeled from `1` to `k` through a specific operation. The operation is defined as removing the last element from the array `nums` and placing that element in your collection. You must determine the fewest number of operations needed to collect all elements from `1` to `k`. If an element is not present in `nums` or cannot be obtained by performing the allowed operations, it implies that it is impossible to collect all elements from `1` to `k`.

For example, if your array `nums` is `[1,3,2,4,3]` and `k` is `3`, you can perform the following operations:

- Remove `3` from `nums` and add it to the collection. Now your collection has `[3]` and `nums` is `[1,3,2,4]`.
- Remove `4` from `nums` but do not add to the collection since `4` is not needed (we want elements from `1` to `k=3`).
- Remove `2` from `nums` and it goes to the collection. Your collection is now `[3,2]` and `nums` is `[1,3]`.
- Finally, remove `3` from `nums` and since it's already in the collection, you can ignore it.
- Remove `1` from `nums` and add it to your collection.

After these 4 operations, your collection has the elements `[1, 2, 3]`, and thus, the minimum number of operations is `4`. The task is to figure out this minimum number of operations for any given array `nums` and integer `k`.

Intuition

The intuition behind the solution is to address the problem efficiently by working backwards, starting from the end of the array - since it's the only place where we can remove elements - and moving towards the front. By doing this, we keep a check on what elements are getting added to the collection and ensure the following:

- We only care about elements that are less than or equal to `k`, because we want to collect elements `1` to `k`.
- We avoid adding duplicates to our collection because each number from `1` to `k` should only be collected once.

We use a list called `is_added` that keeps track of whether an element has been added to the collection. This array is of size `k`, where each index represents an element from `1` to `k`, and the value at each index represents whether the corresponding element has been added to the collection.

If we encounter an element, while traversing from the end, that is less than or equal to `k` and has not been added (`is_added[element - 1]` is `False`), we mark it as added and increase our count of unique elements. We continue this process until our count reaches `k`, which means we have all the elements from `1` to `k`.

The number of operations is then simply the total number of elements in `nums` minus the index of the last added element because all elements from the end of the array to this index have to be removed in order to collect all elements from `1` to `k`. Essentially, we're tracking how many removals we made from the end of the array to get all elements from `1` to `k`.

Solution Approach

The solution involves a single pass through the given array in reverse order, beginning from the last element and moving towards the first. This strategy is chosen because elements can only be removed from the end of the array. The language of choice for the implementation is Python.

Here's a step-by-step explanation of the solution with reference to the provided code snippet:

- An array `is_added` of size `k` is created to keep track of elements from `1` to `k` that have been added to our collection. Initially, all values in `is_added` are set to `False`, indicating that no elements have been collected yet.
- We define a count variable `count`, which keeps the count of unique elements that we have collected so far, starting the count from `0`.
- We iterate through the array `nums` from the last element to the first, using a reverse loop. This is implemented by the loop `for i in range(n - 1, -1, -1):`, where `n` is the length of the `nums` array.
- For each element `nums[i]` encountered during the traversal:
  - We check if `nums[i]` is greater than `k` or if `is_added[nums[i] - 1]` is `True` (the element has already been added to the collection). If either condition is true, we continue to the next iteration without performing any operations since we either don't need the element or have already collected it.
  - If `nums[i]` is needed and has not been added to the collection yet, we set `is_added[nums[i] - 1]` to `True` and increment our `count`.
  - As soon as our `count` equals `k`, we know we have collected all elements from `1` up to `k`. The minimum number of operations required is then the total length of the array minus the current index `i`, which gives us `n - i`.

This approach efficiently ensures that we do not collect unnecessary elements and simultaneously avoid duplication in our collection. Once we've collected all required elements, we immediately return the result.

Using this algorithm, we leverage simple data structures such as an array (`is_added`) and a counter variable to reach an optimal solution with a time complexity of O(n), where n is the number of elements in `nums`.

Example Walkthrough

Let's consider a small example to illustrate the solution approach with the array `nums = [5,4,2,3,1,2]` and target `k = 3`.

- The `is_added` array will be initialized with values `[False, False, False]` which signifies that none of the numbers `1`, `2`, or `3` have been added to our collection yet.
- We iterate over `nums` starting from the last element, so our loop begins at `nums[5]` which is `2`.
  - Since `2` is less than or equal to `k` and hasn't been added to the collection yet (`is_added[2 - 1]` is `False`), we add it by setting `is_added[1]` to `True`. Our collection becomes `[False, True, False]`, indicating that `2` has been added.
- Moving to `nums[4]` which is `1`:
  - Following our rules, we add `1` to the collection and `is_added` becomes `[True, True, False]`.
- Next, we look at `nums[3]` which is `3`:
- `3` is less than or equal to `k` and is not present in the collection (`is_added[3 - 1]` is `False`), so we add it. Our `is_added` array now becomes `[True, True, True]`. At this point, we have all elements from `1` to `k` in our collection, and we stop the iteration.

Since we have collected all the numbers from `1` to `k` after reaching the 3rd index from the right (inclusive), we have done a total of `6 (length of nums) - 3 (current index) = 3` operations to collect all necessary elements. Therefore, the fewest number of operations required in this example is `3`.

Python Solution

```python
from typing import List

class Solution:
    def min_operations(self, nums: List[int], k: int) -> int:
        # Create a list to track if the required numbers have been added
        is_added = [False] * k
        count = 0  # Counter for unique numbers added
        n = len(nums)  # Calculate the length of the nums list

        # Start iterating over the list from the end to the beginning
        for i in range(n - 1, -1, -1):
            if nums[i] > k or is_added[nums[i] - 1]:
                # Skip if the number is greater than k or already added
                continue

            # Mark the number as added
            is_added[nums[i] - 1] = True
            count += 1  # Increment the counter by 1

            if count == k:
                # If we have added k unique numbers, return the number of operations
                return n - i

        # If it is not possible to perform the operation, return -1
        return -1

```

Java Solution

```java
import java.util.List;

class Solution {
    // Method to find the minimum number of operations to add the first k positive integers into the list
    public int minOperations(List<Integer> nums, int k) {
        // Array to keep track of which numbers between 1 to k have been added
        boolean[] isNumberAdded = new boolean[k];
        // Get the size of the input list
        int n = nums.size();
        // Counter for unique numbers added to the list
        int count = 0;

        // Iterate in reverse through the list
        for (int i = n - 1; i >= 0; i--) {
            int currentValue = nums.get(i);
            // If the current value is greater than k or already marked as added, skip it
            if (currentValue > k || isNumberAdded[currentValue - 1]) {
                continue;
            }
            // Mark this number as added
            isNumberAdded[currentValue - 1] = true;
            // Increment the count of unique numbers
            count++;
            // If we have added k unique numbers, return the number of operations
            if (count == k) {
                return n - i;
            }
        }
        // If we exit the loop without returning, there's an error, so return -1 as it shouldn't happen
        // Each number between 1 and k should exist in a properly-sized list
        return -1;
    }
}

```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // This function calculates the minimum number of operations required
    // to reduce array 'nums' such that there are 'k' distinct integers
    int minOperations(vector<int>& nums, int k) {
        int n = nums.size(); // Obtain the size of nums
        vector<bool> isAdded(n, false); // Create a boolean vector to track added numbers

        int countDistinct = 0; // Variable to count distinct integers
        // Start from the end of the vector and look for distinct integers until 'k' are found
        for (int i = n - 1;; --i) {
            // If current number is greater than k or already counted as distinct, skip it
            if (nums[i] > k || isAdded[nums[i] - 1]) {
                continue;
            }

            // Mark the number as added because it is distinct
            isAdded[nums[i] - 1] = true;

            // Increase the count of distinct numbers
            countDistinct++;

            // If we have found 'k' distinct numbers, return the number of operations,
            // which is the difference between array length and starting index
            if (countDistinct == k) {
                return n - i;
            }
        }

        // Note that the loop is missing an exit condition and might lead
        // to an out of bounds access in the nums vector or an infinite loop
        // if 'k' distinct numbers are not found.
        // The problem's constraints should ensure that this situation doesn't happen.
    }
};

```

Typescript Solution

```typescript
function minOperations(nums: number[], k: number): number {
    // Get the length of the input array.
    const arrayLength: number = nums.length;

    // Initialize an array to keep track of which numbers have been added to the sequence.
    const isAdded: boolean[] = Array(k).fill(false);

    // Initialize count to keep track of unique numbers encountered that are not more than k.
    let uniqueCount: number = 0;

    // Iterate backwards through the array.
    for (let i: number = arrayLength - 1;; --i) {
        // If the current number is greater than k or it has already been counted, skip it.
        if (nums[i] > k || isAdded[nums[i] - 1]) {
            continue;
        }

        // Mark the current number as added.
        isAdded[nums[i] - 1] = true;

        // Increment the count of unique numbers.
        ++uniqueCount;

        // If we have encountered k unique numbers, return the size of the sequence.
        if (uniqueCount === k) {
            return arrayLength - i;
        }
    }
    // The loop was intentionally constructed to run indefinitely, control exits from within the loop.
    // If the function has not returned within the loop, it's unexpected as per the problem statement,
    // and may indicate an issue with the inputs. The following return statement is technically unreached.
    return -1; // Return an impossible count as indication of an error.
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the input array `nums`. This is because there is a single for loop that iterates backwards over the array `nums`, and in each iteration, it performs a constant time check and assignment operation. Since these operations do not depend on the size of `k` and there are no nested loops, the iteration will occur `n` times, leading to a linear time complexity with respect to the size of the array.

Space Complexity

The space complexity of the given code is `O(k)`. The `is_added` list is the only additional data structure whose size scales with the input parameter `k`. Since it is initialized to have `k` boolean values, the amount of memory used by this list is directly proportional to the value of `k`. The rest of the variables used within the function (like `count`, `n`, and `i`) use a constant amount of space, and therefore do not contribute to the space complexity beyond a constant factor.