3065. Minimum Operations to Exceed Threshold Value I

Easy Array

Problem Description

In this problem, we're presented with an array of integers called `nums`, which is indexed starting at 0, and a separate integer `k`. Our objective is to remove occurrences of the smallest element in the array, one at a time, until all the elements left in the array are greater than or equal to `k`.

The operation we're allowed to perform repeatedly is the removal of one instance of the smallest number in the array. We want to determine the smallest number of such operations needed to ensure no number in the array is less than `k`.

For instance, if our array is `[1, 4, 3, 2, 2, 7]` and `k` is 3, we should remove the `1` and both `2`s. We performed this operation a minimum of 3 times to ensure all remaining elements are at least `k`.

Intuition

To solve this problem, we don't actually need to perform the operations of removing elements. We can take a more direct approach by just counting how many elements in the array are less than `k`. Those are exactly the elements that would need to be removed, one by one, in the operations described.

The solution involves a straightforward approach which is to traverse the entire array once and count the occurrences of numbers that are strictly smaller than `k`. Since each of these occurrences will require one operation to remove, the count directly gives the minimum number of operations needed.

Solution Approach

The solution's implementation is quite straightforward and is due to a pattern that is often used in problems related to counting specific elements in a list: traversal and counting.

The Python code uses a generator expression inside the built-in `sum` function. Let's dissect it:

1. `for x in nums`: This is a simple for-loop over the list `nums`, where each element is represented as `x`.
2. `x < k`: For each element, we check whether it's less than `k`.
3. `(x < k for x in nums)`: This is a generator expression that will go through each element `x` in `nums`, yielding `True` if `x` is less than `k`, and `False` otherwise. In Python, `True` is equivalent to `1`, and `False` is equivalent to `0` when they are used in arithmetic operations.
4. `sum(x < k for x in nums)`: The `sum` function then adds up these `1`s and `0`s, effectively counting the number of times `x < k` is `True`, that is, the number of elements less than `k`.

There aren't any complex algorithms, data structures, or design patterns at play here. The solution merely employs a few fundamental programming constructs: a generator expression for the condition check and a summing operation to tally the count. This works efficiently because we iterate over the list exactly once, leading to an O(n) solution, where n is the number of elements in `nums`. Since we are not using any additional data structures, the space complexity is O(1), meaning that it uses a constant amount of extra space.

In pseudocode, the process could be described as:

```
initialize counter to 0
for each element x in array nums
    if x < k
        increment counter
return counter
```

The provided Python code matches this simple pseudocode logic, except it uses Python's ability to treat booleans as integers and encapsulates the for-loop and if-check in a concise one-liner.

Example Walkthrough

Let's take an example to illustrate the solution approach. Suppose we have the following array of integers `nums` and integer `k`:

```
nums = [1, 5, 3, 2, 8, 5, 2]
k = 4
```

According to the problem, we want to remove the smallest element from the array repeatedly until all elements are greater than or equal to `k`. Here is the step-by-step walkthrough:

1. We start by examining each element in the array to see if it is smaller than `k`. Here `k` is `4`, so we look for elements smaller than `4`.
2. As we traverse `nums`, we find `1`, `3`, `2`, and `2` to be the elements less than `k`.
3. Counting them up, there's a total of `4` items that meet the condition (`x < k`).
4. Since the operation is to remove one instance of the smallest number one by one, and we have identified `4` such instances, we infer that it will take exactly `4` removal operations to meet our condition that no number is less than `k`.
5. We finish with the understanding that the array does not need to be modified, but just the count of smaller elements is enough to answer the problem. Thus, we avoid the unnecessary operations of actually removing them.

Applying the provided solution approach:

```
for x in [1, 5, 3, 2, 8, 5, 2]:
    if x < 4: # only when x is 1, 3, 2, 2 this condition is true
        ... # the sum function will count this as 1

# By using the generator expression, we get the sum as 4.
```

Therefore, to follow the problem's constraint of removing occurrences of the smallest numbers until all elements are `>= k`, we will need to perform the removal operation 4 times. The rest of the elements will be `5, 5, 8` which are all `>= k`. This matches our manual count and confirms the correctness of our solution approach.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def minOperations(self, nums: List[int], target: int) -> int:
        # Initialize the count variable to keep track of numbers less than the target.
        count = 0

        # Iterate over each number in the nums list.
        for num in nums:
            # If the current number is less than the target,
            # increment the count.
            if num < target:
                count += 1

        # Return the total count of numbers less than the target.
        return count
```

Java

```java
class Solution {

    // Method to count the number of elements less than the given threshold 'k'
    public int minOperations(int[] nums, int k) {
        int count = 0; // Initialize count to record the number of operations

        // Loop through all the elements of the array 'nums'
        for (int num : nums) {
            // If the current element is less than 'k'
            if (num < k) {
                count++; // Increment the count
            }
        }

        return count; // Return the total number of elements less than 'k'
    }
}
```

C++

```cpp
#include <vector>

class Solution {
public:
    // Function to find the minimum number of operations needed
    // such that each element in the nums vector is at least k.
    // In each operation, you can increment any element by 1.
    int minOperations(std::vector<int>& nums, int k) {
        int numOperations = 0; // Initialize the count of operations

        // Iterate over each element in the nums vector
        for (int num : nums) {
            // If the current element is less than k
            if (num < k) {
                // Increment the number of operations
                // by the difference between k and the current element.
                numOperations += (k - num);
            }
        }

        // Return the total number of operations needed
        return numOperations;
    }
};
```

TypeScript

```typescript
/**
 * Calculates the minimum number of operations to increase all elements less than k
 *
 * @param nums - The array of numbers to be processed
 * @param k - The target value that elements in nums should reach or exceed
 * @returns The number of elements in nums that are less than k
 */
function minOperations(nums: number[], k: number): number {
    // Filter the array to get all elements which are less than 'k'
    const elementsLessThanK = nums.filter((element) => element < k);

    // Return the count of elements that are less than 'k'
    return elementsLessThanK.length;
}
```

```python
from typing import List

class Solution:
    def minOperations(self, nums: List[int], target: int) -> int:
        # Initialize the count variable to keep track of numbers less than the target.
        count = 0

        # Iterate over each number in the nums list.
        for num in nums:
            # If the current number is less than the target,
            # increment the count.
            if num < target:
                count += 1

        # Return the total count of numbers less than the target.
        return count
```

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`. This is because the code iterates through each element of `nums` once to check if it is less than `k`.

Space Complexity: The space complexity of the code is `O(1)`, as it uses a fixed amount of extra space regardless of the input size. The summation operation does not require additional space that scales with the input size.