2772. Apply Operations to Make All Array Elements Equal to Zero

Medium Array Prefix Sum

Leetcode Link

Problem Description

You are given an array of integers, `nums`, which is indexed starting from zero. In addition, you're provided with a positive integer, `k`, representing the size of a subarray that you can use in an operation. The operation consists of selecting any `k`-sized subarray from `nums` and decreasing each element within that subarray by `1`. The goal of the problem is to determine whether it's possible to apply this operation any number of times in such a way that all elements of the array can be made equal to `0`. If it's possible, the function should return `true`; otherwise, it should return `false`. A subarray mentioned in the problem is defined as a contiguous, non-empty segment of the array.

Intuition

The intuition behind the solution to this problem relies on the idea that if you need to reduce an element to zero, you would need to perform the operation a number of times equal to the value of that element. We can start this process from the leftmost element and move right.

At each step, you check the element's value, taking into account any alterations made by previous operations. If the value at the current index can be reduced to zero, you proceed. If you encounter a negative value or an index where the `k`-sized subarray goes out of bounds of the array, it implies that it is not possible to reduce all elements to zero; therefore, you would return `false`.

Interestingly, instead of directly manipulating the array after each operation (which would be time-consuming), you utilize a technique involving a difference array. With the help of this difference array, you can represent increments and decrements in the array values at specific intervals effectively. By keeping track of the cumulative changes using a prefix sum approach, you can determine the actual value of each element as you iterate through the original array.

The clever insight here is the use of the difference array to make the operations of decrementing a subarray's elements efficient. As you iterate through the original array, you adjust the difference array accordingly to represent the needed operations. Once the iteration is complete without encountering any issues mentioned earlier, you can confidently return `true`, indicating that all elements can be made equal to `0`.

Solution Approach

The solution approach involves the use of a difference array to efficiently track the cumulative operations that must be performed on the original array, `nums`. This difference array, `d`, is initially all zeros and has a length one greater than `nums` to accommodate changes that affect elements beyond the end of `nums`.

Here's the detailed process:

1. We initialize the difference array `d` and a variable `s` which will hold the sum of difference values as we progress through `nums`.
2. We iterate through the elements of the `nums` array. For each index `i`:
   - We accumulate the sum `s` by adding `d[i]` to it. This variable `s` represents the total change made to `nums[i]` due to operations on earlier elements because a change to an earlier element may affect all subsequent elements within its `k`-sized subarray.
   - We apply the accumulated sum `s` to `nums[i]` to calculate its modified value after previous operations.
   - If the modified value of `nums[i]` is already `0`, we move to the next element, as no action is required.
   - If the modified value is less than `0`, or if the `k`-sized subarray starting at `i` would go out of bounds (i.e., `i + k > n`), we return `false`. This check ensures that it's impossible to perform an operation without reducing some element below zero or dealing with a subarray that extends beyond the array's end.
   - If the value is positive and within bounds, this means we need to perform the operation `nums[i]` times on the subarray starting at index `i` and ending at index `i + k - 1`. Thus, we decrease `s` by `nums[i]` to indicate that we've made a plan to carry out these operations. We also increment `d[i + k]` by `nums[i]`, effectively canceling out the operation's effect beginning at index `i + k`. This is where the difference array `d` shows its utility by allowing us to make adjustments at specific points that automatically alter the sum in between.
3. Once the iteration is complete without returning `false`, it means that we have a valid sequence of operations that can reduce all elements to `0`, and we return `true`.

Through the use of a difference array and prefix sums, the solution avoids repeated iterations over subarrays making it efficient and effective for achieving the task in linear time complexity, O(n), where n is the number of elements in `nums`.

Example Walkthrough

Let's walk through an example to illustrate the solution approach described above. Consider an array `nums = [3, 4, 2, 3]` and `k = 3`.

1. We initialize the difference array `d` which will have one extra element than `nums`, so `d = [0, 0, 0, 0, 0]`. We also set a variable `s = 0`.
2. Begin iterating through `nums`:
   - For `i = 0`, `s = d[0] + s = 0`. The value at `nums[0]` after adjustments is `3`. Since we have a `k`-size subarray (which fits within bounds, as `i + k = 3` is not greater than `n`), and the value is above `0`, we need to perform the operation `3` times. So, we decrease `s` by `3` to `-3`, and increment `d[3]` by `3`, yielding `d = [0, 0, 0, 3, 0]`. This represents our commitment to decrease the values in the subarray starting from index `0` to index `2`.
   - For `i = 1`, `s = d[1] + s = -3`. The value at `nums[1]` is modified to `4 - 3 = 1`. Again, we perform the operation `1` time, so we decrement `s` by `1` to `-4`, and increment `d[4]` by `1`, giving `d = [0, 0, 0, 3, 1]`.
   - For `i = 2`, `s = d[2] + s = -4`. The value at `nums[2]` is modified to `2 - 4 = -2`. Here we encounter a value less than `0` after applying the effect of previous operations, which means it's not possible to reduce the original array to all zeros. Hence, we would return `false`.
3. In this scenario, we don't get to iterate through the entire `nums` because our solution requires us to return `false` once we encounter a modified value below `0`. Therefore, it is not possible to make all elements equal to `0` with the given `k`-sized subarray operations.

This example clearly shows that if at any point in iterating over the array we come across a value below zero, we conclude that it is not possible to use the given operations to reduce the entire array to zeros and thus return `false`. The difference array technique allows us to simulate the operations without repetitively decrementing values in `k`-sized chunks, resulting in an efficient approach to solving the problem.

Python Solution

```python
from typing import List

class Solution:
    def checkArray(self, nums: List[int], k: int) -> bool:
        # Calculate the length of the array nums
        array_length = len(nums)

        # Initialize an array to track dynamic modifications
        dynamic_changes = [0] * (array_length + 1)

        # Initialize a variable to carry the sum of modifications
        sum_modifications = 0

        # Iterate through nums array
        for index, value in enumerate(nums):
            # Update the value with modifications seen so far
            sum_modifications += dynamic_changes[index]
            value += sum_modifications

            # If the modified value is zero, continue to next iteration
            if value == 0:
                continue

            # Return False if value is negative or index + k exceeds the array bounds
            if value < 0 or index + k > array_length:
                return False

            # Update the sum of modifications to negate the excess value
            sum_modifications -= value

            # Record the necessary change at an index k steps ahead
            dynamic_changes[index + k] += value

        # If all checks pass, return True
        return True
```

Java Solution

```java
class Solution {

    public boolean checkArray(int[] nums, int distance) {
        int length = nums.length;
        // Array 'differences' will be used to track the changes to be applied.
        int[] differences = new int[length + 1];

        // 'sum' is used to keep track of the accumulated value to apply.
        int sum = 0;

        // Iterate through the array 'nums'.
        for (int index = 0; index < length; ++index) {
            // Apply the accumulated changes to the current element.
            sum += differences[index];
            nums[index] += sum;

            // If the updated value of nums[index] is zero, continue to the next iteration.
            if (nums[index] == 0) {
                continue;
            }

            // If the updated value of nums[index] is negative,
            // or index + distance is out of bounds, return false.
            if (nums[index] < 0 || index + distance > length) {
                return false;
            }

            // Decrease 'sum' by the value of nums[index] since it needs to be "cancelled out"
            // in the future iterations.
            sum -= nums[index];

            // Schedule a change to be applied 'distance' indices ahead of current index.
            differences[index + distance] += nums[index];
        }

        // If the method has not returned false, the conditions were satisfied, therefore return true.
        return true;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to check if the array can be transformed to a zero array, with increments of 'k' indices
    bool checkArray(vector<int>& nums, int k) {
        int size = nums.size(); // Store the size of nums
        vector<int> diff(size + 1); // Initialize a difference array with an extra element

        int sum = 0; // Variable to handle the running sum

        // Loop through each element of the nums array
        for (int i = 0; i < size; ++i) {
            sum += diff[i]; // Update sum according to the diff array
            nums[i] += sum; // Modify the current element of nums using sum

            // If the value at nums[i] after modification is 0, no changes are needed
            if (nums[i] == 0) {
                continue;
            }

            // If the current value is negative or we're unable to increment subsequent elements, return false
            if (nums[i] < 0 || i + k > size) {
                return false;
            }

            // Modify the running sum and the diff array at the position i+k
            sum -= nums[i];
            diff[i + k] += nums[i];
        }
        // If all operations are possible without violating any constraints, return true
        return true;
    }
};
```

Typescript Solution

```typescript
// This function checks if it's possible to convert an array to an array of zeroes,
// by subtracting a number at a position and also from all elements.
// 'k' positions ahead. It returns true if possible, false otherwise.
function checkArray(nums: number[], k: number): boolean {
    // Get the length of the array.
    const length = nums.length;

    // Initialize a difference array with an extra space for ease of processing.
    // This helps us to keep track of increments and decrements which will be
    // propagated along the array.
    const differenceArray: number[] = Array(length + 1).fill(0);

    // This variable `sum` will keep the running total of elements added
    // or subtracted because of the difference array.
    let sum = 0;

    // Iterate over the array elements.
    for (let index = 0; index < length; ++index) {
        // Add the current difference to sum.
        sum += differenceArray[index];

        // Apply the current sum to the current element, simulating propagation.
        nums[index] += sum;

        // If after adding the sum, the number is zero,
        // we move on without making any changes.
        if (nums[index] === 0) {
            continue;
        }

        // If the number is negative, or the index surpasses bounds when adding `k`,
        // we cannot fulfill the condition and return false.
        if (nums[index] < 0 || index + k > length) {
            return false;
        }

        // Since we want to zero out the current number, we would need to
        // subtract it from the current sum and from each of the next `k`
        // elements in the array.
        sum -= nums[index];

        // Increment the difference array `k` positions from the current index
        // by the value of the current number.
        differenceArray[index + k] += nums[index];
    }

    // If the loop finishes, it's possible to convert the array
    // into an array of zeroes by using the method described, so return true.
    return true;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)` where `n` is the length of the array `nums`. This is because the code consists of a single loop that runs `n` times. Inside the loop, there are constant time operations being performed, which do not affect the linear nature of the time complexity.

Space Complexity

The space complexity of the code is also `O(n)` because an extra array `d` of size `n + 1` is created to store the cumulative difference. No other data structures are used that grow with the size of the input, hence the space complexity does not exceed linear bounds.