2960. Count Tested Devices After Test Operations

Easy Array Simulation

Problem Description

In this problem, we have an array `batteryPercentages` representing the battery levels of `n` devices, each indexed from `0` to `n - 1`. Our goal is to test each device sequentially, starting from the first device. For each device `i` that has a battery level greater than 0, we perform some operations. We increment a counter that tracks the number of tested devices. Then we reduce the battery percentage of all subsequent devices (`j` where `j > i`) by 1, making sure their battery levels don't drop below zero. If the current device's battery level is not greater than 0, we simply move on to the next device without increasing the counter or changing other devices' battery levels.

The challenge is to determine the total number of devices that we can test given that testing one device will consume a bit of battery from the subsequent devices.

The process continues until all devices have been addressed, and we return the count of devices that have been tested successfully.

Intuition

The intuition behind the solution is realizing that the operation's sequence affects each device exactly once. As we move through the array, each device's battery level is decremented by the number of devices tested before it, representing the cumulative battery consumption by previous testing. By keeping track of how many devices we have already tested (`ans`), we can determine whether the current device can be tested by checking if the adjusted battery level (`batteryPercentages[i] - ans`) is greater than 0.

The key insight is that we can simulate the process of testing devices in a single pass through the `batteryPercentages` array. For each device, we decrement the current battery percentage by the number of devices tested so far and check if it still has a positive battery percentage. If it does, we increment our tested device count (`ans`). Otherwise, we leave the count as is and move to the next device.

Solution Approach

The implementation of the solution uses a straightforward algorithm with a single pass through the input array, representing a greedy approach. The data structure used is very simple; we just need the input list `batteryPercentages`, and an integer counter `ans` to keep track of the number of devices we've tested successfully.

We iterate through each element `x` in `batteryPercentages`, and for each device, we simulate the battery usage up to that point by subtracting `ans` from `x`. Here is the step-by-step breakdown of the algorithm:

1. Initialize `ans` to 0, which will hold the count of tested devices.
2. Iterate through each battery level `x` in the array `batteryPercentages`.
    - Calculate the effective battery level after accounting for previous tests by subtracting `ans` from `x`.
    - Check if the effective battery level is greater than 0:
        - If yes, this means the device can be tested, so we increment `ans` by 1.
        - If no, the device cannot be tested, and we continue to the next device without incrementing `ans`.
3. After walking through all devices, `ans` will now contain the total number of devices that were able to be tested.

This solution employs a simple greedy strategy, as it always takes the opportunity to test a device whenever possible by checking the current battery level at each step. The algorithm operates in O(n) time complexity where n is the number of devices, as it only requires going through the array once.

The mathematical formula used in this implementation is:

```
batteryPercentages[i] = max(0, batteryPercentages[i] - ans)
```

It is used to simulate the reduction in battery level of each device, and it ensures that the battery level doesn't go below 0.

By applying this simple yet effective approach, we are able to determine the total number of devices that can be tested.

Example Walkthrough

Let's consider a small example with the following battery levels for the devices: `batteryPercentages = [3, 2, 5, 1]`. We now go through the approach step by step:

1. Initialize `ans = 0`. This will hold the count of devices we have tested successfully. Now our counts are `batteryPercentages = [3, 2, 5, 1]` and `ans = 0`.
2. Start with the first device (`batteryPercentages[0]`):
    - Effective battery level = `3 - 0`, which is 3.
    - Since 3 is greater than 0, the device can be tested. We increment `ans` by 1.
    - Now `ans = 1` and `batteryPercentages` remains unchanged for now.
3. Move to the second device (`batteryPercentages[1]`):
    - Effective battery level = `2 - 1`, which is 1.
    - Once again the number is greater than 0, so we increment `ans` by 1.
    - Now `ans = 2` and we still don't change `batteryPercentages`.
4. Next, the third device (`batteryPercentages[2]`):
    - Effective battery level = `5 - 2`, which is 3.
    - The number is greater than 0, we increment `ans` by 1.
    - Now `ans = 3`.
5. Finally, the fourth device (`batteryPercentages[3]`):
    - Effective battery level = `1 - 3`, which is -2. But we can't have negative battery, so we consider it 0.
    - This number is not greater than 0, so we cannot test the device and hence do not increment `ans`.
    - The value of `ans` remains at 3.

After walking through all devices, we have `ans = 3`, which means we were able to successfully test 3 devices out of 4 with the given batteryPercentages array without reducing any individual device's battery below zero.

In summary, the result for the input array `[3, 2, 5, 1]` is 3, which is the total number of devices that can be tested following the solution approach outlined.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def countTestedDevices(self, battery_percentages: List[int]) -> int:
        # Variable to hold the count of tested devices
        tested_devices_count = 0

        # Iterate through the battery percentages
        for battery_percentage in battery_percentages:
            # Subtract the currently tested devices' count from the battery percentage
            battery_percentage -= tested_devices_count

            # If the adjusted battery percentage is greater than zero, increment the count
            if battery_percentage > 0:
                tested_devices_count += 1

        # Return the total count of tested devices
        return tested_devices_count
```

Java

```java
class Solution {
    // Method to count the number of devices tested based on their battery percentages
    public int countTestedDevices(int[] batteryPercentages) {
        // Initialize the count of tested devices to 0
        int testedDeviceCount = 0;

        // Iterate through each device's battery percentage
        for (int batteryPercentage : batteryPercentages) {
            // Deduct the current tested device count from the battery percentage
            batteryPercentage -= testedDeviceCount;
            // If the adjusted battery percentage is still positive,
            // it means the device is tested, thus increment the count
            if (batteryPercentage > 0) {
                ++testedDeviceCount;
            }
        }

        // Return the total count of tested devices
        return testedDeviceCount;
    }
}
```

C++

```cpp
#include <vector> // Include the vector header to use std::vector

class Solution {
public:
    // Function to count the number of tested devices based on their battery percentages
    int countTestedDevices(std::vector<int>& batteryPercentages) {
        int count = 0; // Initialize a variable to store the count of devices tested

        // Iterate through each device's battery percentage in the vector
        for (int &percentage : batteryPercentages) {
            // Decrease the current battery percentage by the count of devices tested so far
            percentage -= count;

            // If the adjusted battery percentage is greater than 0, increase the count
            if (percentage > 0) {
                ++count;
            }
        }

        // Return the final count of devices tested
        return count;
    }
};
```

TypeScript

```typescript
/**
 * Counts the number of devices that can be tested based on their remaining battery percentages.
 * Devices are tested one by one, and each test is assumed to consume 1% battery from the device being tested.
 *
 * @param {number[]} batteryPercentages - An array of integers representing battery percentages for each device.
 * @returns {number} - The number of devices that can be tested.
 */
function countTestedDevices(batteryPercentages: number[]): number {
    // Initialize the count of devices that can be tested.
    let testedDevicesCount = 0;

    // Iterate over the array of battery percentages.
    for (let batteryPercentage of batteryPercentages) {
        // Decrease the battery percentage by the number of devices already tested.
        batteryPercentage -= testedDevicesCount;

        // If the adjusted battery percentage is still positive,
        // it means the current device can be tested.
        if (batteryPercentage > 0) {
            // Increment the count of devices tested.
            ++testedDevicesCount;
        }
    }

    // Return the total count of tested devices.
    return testedDevicesCount;
}
```

```python
from typing import List

class Solution:
    def countTestedDevices(self, battery_percentages: List[int]) -> int:
        # Variable to hold the count of tested devices
        tested_devices_count = 0

        # Iterate through the battery percentages
        for battery_percentage in battery_percentages:
            # Subtract the currently tested devices' count from the battery percentage
            battery_percentage -= tested_devices_count

            # If the adjusted battery percentage is greater than zero, increment the count
            if battery_percentage > 0:
                tested_devices_count += 1

        # Return the total count of tested devices
        return tested_devices_count
```

Time and Space Complexity

The time complexity of the provided code is `O(n)` where `n` is the length of the `batteryPercentages` list. This is because the code contains a single for-loop that iterates through each element of the list exactly once.

The space complexity is `O(1)` as there are no additional data structures that grow with the input size. The only extra variables used are `ans` and `x` which do not depend on the size of the input list, thus the space used is constant.