1124. Longest Well-Performing Interval

Medium | Stack | Array | Hash Table | Prefix Sum | Monotonic Stack | Leetcode Link

Problem Description

The problem presents us with an array called `hours` which represents the number of hours worked by an employee each day. Our goal is to find the length of the longest interval of days where there are more "tiring days" than "non-tiring days". A "tiring day" is defined as any day where the employee works more than 8 hours. We need to understand the interval dynamics where the "well-performing intervals" are those having a greater count of tiring days compared to non-tiring days. The challenge lies in finding the maximum length of such an interval.

Intuition

The solution to this problem uses an interesting approach akin to finding the longest subarray with a positive sum, which can be solved efficiently using a prefix sum and a hash map. The intuition here is to designate tiring days as a positive contribution (+1) and non-tiring days as a negative contribution (-1) to the sum. As we iterate over the array:

- We keep a running sum (`s`), which is increased by 1 for tiring days and decreased by 1 for non-tiring days.
- If at any point, the running sum is positive, it means there are more tiring days than non-tiring days so far, so we update the answer to the current length of days (`i + 1`).
- If the running sum is not positive, we look to see if there is a previous running sum (`s - 1`). If it exists, then the subarray between the day when `s - 1` was the running sum and the current day is a "well-performing interval". We then update our answer if this interval is longer than our current longest interval.
- To efficiently find these previous running sums, we use a hash map (`pos`) that records the earliest day that each running sum occurred. This way, we only store the first occurrence of each sum since we want the longest possible interval.

This approach relies on the idea that if we find a running sum that is greater than a past running sum, then there must have been more tiring days than non-tiring days in between those two points. The efficiency of this solution comes from the fact that we traverse the list only once and access/update the hash map in constant time.

Solution Approach

The solution uses a hash map and a running sum to efficiently track well-performing intervals. Here's the detailed breakdown of how the approach is implemented:

1. Initialize variables:
   - `ans` tracks the length of the longest well-performing interval found so far and initializes to 0.
   - `s` is our running sum, which helps determine if an interval is well-performing.
   - `pos` is a hash map recording the first occurrence of each running sum.
2. Iterate over the `hours` array using `enumerate` to have both index `i` and value `x` for each day.
3. Update the running sum `s`:
   - Add 1 to `s` if `x > 8` (tiring day).
   - Subtract 1 from `s` if `x` is not greater than 8 (non-tiring day).
4. After updating the running sum:
   - If `s > 0`, it means we've encountered more tiring days than non-tiring days up to day `i`, so update `ans` to `i + 1`.
   - If `s <= 0`, we look for `s - 1` in `pos`. If it's found, it indicates there's an interval starting right after the first occurrence of `s - 1` up to the current day `i`, which is a well-performing interval. Hence, we calculate the length of this interval (`i - pos[s - 1]`) and update `ans` if it's longer than the current `ans`.
5. Update the hash map `pos`:
   - If the current running sum `s` has not been seen before, record its first occurrence (`pos[s] = i`). We only update the first occurrence because we're interested in the longest interval.

The code uses `pos` to remember the earliest day an intermediate sum occurs. By checking if `s - 1` is in `pos`, we can infer if a corresponding earlier sum would allow for a well-performing interval to exist between it and the current day.

Using this pattern allows us to efficiently process each day in constant time, resulting in an overall time complexity of `O(n)`, where `n` is the number of days. The space complexity is also `O(n)` due to storing the sum indices in the hash map, possibly equal to the number of days if all running sums are unique.

Example Walkthrough

Let's consider a small example using the solution approach described above. Suppose we have the following `hours` array where we need to find the length of the longest well-performing interval:

```
hours = [9, 9, 6, 0, 6, 6, 9]
```

Here's a step-by-step walkthrough using the solution approach:

1. We initialize our variables: `ans` to 0, `s` to 0, and `pos` as an empty hash map.
2. We start iterating through `hours` with the values and their indices:
   - Day 0 (`i=0`, `x=9`): It's a tiring day because `x > 8`. We add 1 to `s`, making it 1. Since `s > 0`, we update `ans = i + 1 = 1`. The hash map `pos` is updated with `pos[1] = 0` because we haven't seen this sum before.
   - Day 1 (`i=1`, `x=9`): Another tiring day. We increment `s` to 2. `ans` is updated to `i + 1 = 2` and `pos[2] = 1`.
   - Day 2 (`i=2`, `x=6`): A non-tiring day, so we subtract 1 from `s`, making it 1 again. Since `s` is still positive, we don't update `ans`, but we don't update `pos[1]` either as it already exists.
   - Day 3 (`i=3`, `x=0`): Non-tiring, subtract 1 from `s` to 0. `ans` remains unchanged, and we add `pos[0] = 3` to the hash map.
   - Day 4 (`i=4`, `x=6`): Non-tiring, subtract 1 from `s` to -1. Since `s <= 0`, we check for `s - 1` which is -2 in `pos`, but it's not there. No update to `ans` and we set `pos[-1] = 4`.
   - Day 5 (`i=5`, `x=6`): Non-tiring, `s` goes to -2. We check for `s - 1` which is -3 in `pos`, but it's not found. `ans` stays the same and `pos[-2] = 5`.
   - Day 6 (`i=6`, `x=9`): Tiring, we add 1 to `s`, bringing it up to -1. We check for `s - 1` which is -2 in `pos` and find it at position 5. We calculate the interval length `i - pos[-2] = 6 - 5 = 1`. Since this does not exceed our current maximum of `ans = 2`, we do not update `ans`.
3. The iteration is now complete. The longest well-performing interval we found has a length of `2`, which occurred between days 0 and 1 (inclusive).

Therefore, the answer for this given `hours` array is `2`.

Python Solution

```python
class Solution:
    def longest_wpi(self, hours) -> int:
        # Initialize the maximum length of well-performing interval and sum so far
        max_length = cumulative_sum = 0
        # Initialize a dictionary to store the earliest index of a particular cumulative sum
        sum_indices = {}

        # Iterate through each hour in the list
        for index, hour in enumerate(hours):
            # Increment or decrement the cumulative sum based on the hour's value
            cumulative_sum += 1 if hour > 8 else -1

            # If the cumulative sum is positive, we found a well-performing interval
            # from the beginning up to the current index
            if cumulative_sum > 0:
                max_length = index + 1
            else:
                # If cumulative_sum - 1 is in the sum_indices, it means we previously had a smaller sum
                # By finding the length from that index to the current index, we ensure a positive hour count
                if cumulative_sum - 1 in sum_indices:
                    max_length = max(max_length, index - sum_indices[cumulative_sum - 1])

                # If this sum has not been seen before, map it to the current index
                # We only want to record the first occurrence of a cumulative sum to achieve the longest interval
                if cumulative_sum not in sum_indices:
                    sum_indices[cumulative_sum] = index

        # The resulting max_length is the length of the longest well-performing interval
        return max_length

```

Java Solution

```java
class Solution {
    public int longestWPI(int[] hours) {
        int longestSequence = 0;  // This will hold the final result, length of the longest well-performing interval.
        int score = 0;  // This tracks the current score indicating the balance of hours (tiring vs. non-tiring).
        Map<Integer, Integer> scoreToIndexMap = new HashMap<>();  // Mapping from scores to their first occurrence index.

        // Iterate over the input array.
        for (int i = 0; i < hours.length; ++i) {
            // If the number of hours worked is more than 8 in a day, increment score, otherwise decrement.
            score += hours[i] > 8 ? 1 : -1;

            // If the current score is positive, it means there is a well-performing interval from 0 to i-th day.
            if (score > 0) {
                longestSequence = i + 1;  // Update the length of the longest sequence.
            } else {
                // If there's a previous score that is one less than the current score...
                if (scoreToIndexMap.containsKey(score - 1)) {
                    // ... then there's a well-performing interval from that previous score's index to the current index i.
                    longestSequence = Math.max(longestSequence, i - scoreToIndexMap.get(score - 1));
                }
                // Store the current score's first occurrence index if it's not already stored.
                // This means for any score, we save the earliest index at which the score occurred.
                scoreToIndexMap.putIfAbsent(score, i);
            }
        }
        return longestSequence;  // Return the length of the longest well-performing sequence found.
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to find the length of the longest well-performing interval
    int longestWPI(vector<int>& hours) {
        int longestInterval = 0;    // Variable to store the length of the longest interval found
        int score = 0;              // A score to determine well-performing days vs. non-well-performing days
        unordered_map<int, int> firstOccurrence;  // Map to store the first occurrence of a score

        // Iterate over the hours array
        for (int i = 0; i < hours.size(); ++i) {
            // Increase the score for well-performing days (hours > 8), decrease for non-well-performing days
            score += hours[i] > 8 ? 1 : -1;

            // If the score is positive, we've found a well-performing interval from the start
            if (score > 0) {
                longestInterval = i + 1;  // Update the longest interval length
            } else {
                // If the score becomes non-positive, try to find a well-performing interval in the middle
                // Check if there's a previous score that is one less than the current score
                if (firstOccurrence.count(score - 1)) {
                    // Update the longest interval found if necessary
                    longestInterval = max(longestInterval, i - firstOccurrence[score - 1]);
                }
            }
            // Record the first occurrence of a score if it hasn't already been recorded
            if (!firstOccurrence.count(score)) {
                firstOccurrence[score] = i;
            }
        }
        return longestInterval;  // Return the length of the longest well-performing interval found
    }
};

```

Typescript Solution

```typescript
// Import necessary libraries from JavaScript/TypeScript (no include statement needed in TypeScript)
// The typing for unordered_map in TypeScript would use a Map or Record

// Define the function to find the longest well-performing interval
function longestWPI(hours: number[]): number {
    let longestInterval: number = 0; // Variable to store the length of the longest interval found
    let score: number = 0; // A score to determine well-performing days vs non-well-performing days
    let firstOccurrence: Map<number, number> = new Map(); // Map to store the first occurrence of a score

    // Iterate over the hours array
    for (let i = 0; i < hours.length; ++i) {
        // Increase score for well-performing days (hours > 8), decrease for non-well-performing days
        score += hours[i] > 8 ? 1 : -1;

        // If the score is positive, a well-performing interval from the start has been found
        if (score > 0) {
            longestInterval = i + 1; // Update the longest interval length
        } else {
            // If the score is non-positive, try to find a well-performing interval in the middle
            // Check if there's a previous score that is one less than the current score
            if (firstOccurrence.has(score - 1)) {
                // Update the longest interval found if necessary
                longestInterval = Math.max(longestInterval, i - (firstOccurrence.get(score - 1) as number));
            }
        }

        // Record the first occurrence of a score if it hasn't already been recorded
        if (!firstOccurrence.has(score)) {
            firstOccurrence.set(score, i);
        }
    }

    return longestInterval; // Return the length of the longest well-performing interval found
}

```

Time and Space Complexity

Time Complexity

The given Python function `longestWPI` exhibits a time complexity of `O(N)`, where `N` represents the length of the input list `hours`. This is due to the fact that the function iterates through the list exactly once. During each iteration, it performs a constant amount of work: updating the sum `s`, checking conditions, and updating the `pos` dictionary or `ans` as needed.

Space Complexity

The space complexity of the function is also `O(N)`. The `pos` dictionary is the primary consumer of space in this case, which in the worst-case scenario might need to store an entry for every distinct sum `s` encountered during the iteration through `hours`. In a worst-case scenario where every value of `s` is unique, the dictionary's size could grow linearly with respect to `N`, the number of elements in `hours`.