2432. The Employee That Worked on the Longest Task

Easy Array Leetcode Link

Problem Description

In this problem, we are working with a record of tasks completed by different employees, each of whom has a unique identifier ranging from 0 to `n - 1`. There is a given 2D integer array named `logs`, where each element is a pair consisting of an employee id and a leave time. This array essentially tracks which employee finished which task and the time they finished it. The leave times are unique for each task.

Each task begins immediately after the previous one ends, and the first task (0th task) starts at time 0. The goal is to determine the employee id that worked on the task that took the longest time to complete. In case there are tasks that took the same maximum amount of time and were worked on by different employees, the employee with the smaller id is the one that should be returned.

Intuition

To solve this problem, we need to determine the duration that each employee spent on their task. Since the tasks are completed in a serial manner, the duration an employee spent on a task can be calculated by subtracting the finish time of the previous task from the finish time of the current task. The `0th` task starts at time 0, so for this first task, the duration is simply the finishing time of the task since it started at time 0.

Now, we keep track of the maximum duration observed and the associated employee id. As we iterate through the `logs` array, we compare the current task's duration with what we've recorded as the maximum duration so far. If the current duration is greater, we update our records to now consider this as the max duration, and we update the "hardest worker" to be the current employee's id. If the current duration matches the maximum duration seen so far (a tie), we check the employee ids involved and update our "hardest worker" to be the employee with the smaller id.

The managing of the "last" task finish time is critical because it marks the start of the next task (other than the first task, which starts at time 0). After determining the duration for the current task, we update the "last" finish time to match the current task's finish time, effectively setting it up for the next comparison.

The given solution employs a for-loop to scan the `logs` list once, maintaining the current maximum duration and the employee with the smallest id that worked for this duration. Once the loop is complete, it returns the identifier of the "hardest worker" as the result.

Solution Approach

The solution approach uses a simple linear scan algorithm to determine which employee worked the longest on a single task. It makes use of basic comparison and arithmetic operations to track and update the task with the longest duration and the corresponding employee id.

Here is the step-by-step explanation of the solution algorithm:

1. Initialize three variables: `last`, `mx`, and `ans`. Set both `last` and `mx` (maximum task duration so far) to `0`, and `ans` (the id of the employee who worked the longest) to the first employee's id.
2. Iterate over each entry in the `logs` list. Each entry is a pair `[uid, t]` which stands for an employee id (`uid`) and the time the task was completed (`t`).
3. For each entry, update the duration of the current task by calculating the difference between the current task's completion time `t` and the last task's completion time (stored in `last`). The expression for this is:

   ```
   t -= last
   ```

4. Now check if the current duration `t` is greater than the maximum duration `mx` or if there is a tie and the current employee's id is smaller than the one previously stored. If either condition is true:
   - Update `mx` to match the current duration `t`.
   - Update `ans` to the current employee's id (`uid`).
5. Update the `last` task completion time to account for the subsequent task's duration calculation:

   ```
   last += t
   ```

6. After processing all entries, return `ans` as the solution. This is the id of the employee who worked the most extended task.

The solution uses no additional data structures; it relies only on a few variables to track the information needed and operates with an O(n) time complexity where n is the number of entries in the `logs` list. The algorithm is efficient as it scans the logs only once, and the space complexity is constant O(1), as no extra space is needed regardless of the size of the input list `logs`.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach:

Suppose we have these `logs` for employees with durations in which they completed the tasks:

```
logs = [[0,3],[2,5],[0,10],[1,15]]
```

In this `logs` array, the first element `[0,3]` indicates that employee `0` finished a task at time `3`. The second element `[2,5]` indicates that employee `2` finished the following task at time `5`; and so on.

Now, let's apply the solution approach:

1. We initialize `last`, `mx`, and `ans` to `0` since no task has been completed yet, and we need a starting point.
2. We begin iteration with the first entry `[0,3]`. This task took `3` time units because it started at time `0`. We set `mx` to `3` and `ans` to `0`, since this is the longest duration we have seen so far, and it belongs to employee `0`.

```
last = 0
mx = 3
ans = 0
```

3. We continue to the next entry `[2,5]`. The duration for this task is `5 - last` which is `2` (`5 - 3`). This duration is not longer than `mx` which is `3`, so we don't change `mx` or `ans`. We then update `last` to `5`.

```
last = 5
mx = 3
ans = 0
```

4. The next entry is `[0,10]`. The duration for this task is `10 - last` which is `5` (`10 - 5`). This duration is greater than the current `mx` which is `3`, so we update `mx` to `5` and `ans` to `0`. Then we update the `last` to `10`.

```
last = 10
mx = 5
ans = 0
```

5. The last entry is `[1,15]`. The duration for this task is `15 - last` which is `5`. This is equal to our current `mx`. Now, we compare the employee ids; since `1` is larger than the current `ans` which is `0`, we keep `ans` as `0`. We update `last` to `15`.

```
last = 15
mx = 5
ans = 0
```

6. We have exhausted all entries, so we finish the iteration, and the `ans` is the final result. Employee `0` is the one who worked the longest on a single task.

In this example, the final output would be `0`, which indicates that employee `0` worked the longest time on a task, which took `5` time units to complete.

Python Solution

```python
class Solution:
    def hardestWorker(self, n: int, logs: List[List[int]]) -> int:
        # Initialize variables:
        # 'last_event_time' to capture the end time of the last event.
        # 'max_duration' to store the duration of the longest task.
        # 'hardest_worker_id' to store the ID of the hardest worker.
        last_event_time = 0
        max_duration = 0
        hardest_worker_id = 0

        # Iterate through the logs
        for user_id, current_time in logs:
            # Calculate the duration of the current task
            current_duration = current_time - last_event_time

            # Check if the current task duration is greater than the max duration found so far,
            # or if it's equal and the current user ID is lower than the one stored.
            if max_duration < current_duration or (max_duration == current_duration and hardest_worker_id > user_id):
                # Update the hardest worker ID and the max task duration
                hardest_worker_id, max_duration = user_id, current_duration

            # Update 'last_event_time' for the next iteration
            last_event_time = current_time

        # Return the ID of the hardest worker
        return hardest_worker_id

```

Java Solution

```java
class Solution {
    public int hardestWorker(int n, int[][] logs) {
        int employeeIdWithMaxWork = 0; // To hold the ID of the employee who worked the most in a single log
        int lastEndTime = 0; // To keep track of the end time of the last task
        int maxDuration = 0; // To hold the maximum duration of work done continuously

        // Iterate through the work logs
        for (int[] log : logs) {
            int currentEmployeeId = log[0]; // ID of the current employee
            int currentTime = log[1]; // Time when the current task was finished

            // Calculate the duration the current employee worked
            int currentDuration = currentTime - lastEndTime;

            // Determine if the current employee worked more than the previous max,
            // or if it's equal to the max duration and the current employee's ID is smaller.
            if (maxDuration < currentDuration || (maxDuration == currentDuration && employeeIdWithMaxWork > currentEmployeeId)) {
                employeeIdWithMaxWork = currentEmployeeId; // Update the employee ID with the max work
                maxDuration = currentDuration; // Update the max duration
            }

            // Update the end time of the last task to the current task's end time
            lastEndTime = currentTime;
        }
        return employeeIdWithMaxWork; // Return the ID of the employee who worked the hardest
    }
}

```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Function to find the hardest worker where 'n' is the number of workers,
    // and 'logs' is the vector of vectors containing the worker ID and the time they logged out.
    int hardestWorker(int n, vector<vector<int>>& logs) {
        int hardestWorkerId = 0; // Store the ID of the hardest worker.
        int maxWorkDuration = 0; // Store the maximum work duration.
        int lastLogTime = 0; // Store the last log time to calculate the duration.

        // Loop through the logs to identify the hardest worker.
        for (auto& log : logs) {
            int workerId = log[0]; // Get the worker's ID from the log.
            int logTime = log[1]; // Get the log time from the log.

            // Calculate the work duration for the current worker.
            int workDuration = logTime - lastLogTime;

            // Check if this worker has worked more or equal than the current maximum work duration
            // and if this worker has a smaller ID than the current hardest worker's ID in case of a tie.
            if (maxWorkDuration < workDuration || (maxWorkDuration == workDuration && hardestWorkerId > workerId)) {
                maxWorkDuration = workDuration; // Update the maximum work duration.
                hardestWorkerId = workerId; // Update the hardest worker ID.
            }

            // Update the last log time for the next iteration.
            lastLogTime = logTime;
        }

        // Return the ID of the hardest worker after all logs have been processed.
        return hardestWorkerId;
    }
};

```

Typescript Solution

```typescript
// Define the function hardestWorker which takes a total number of workers (n)
// and an array of logs with worker IDs and timestamps, and
// returns the ID of the worker who worked the hardest.
function hardestWorker(totalWorkers: number, logs: number[][]): number {
    // Initialize variables:
    // 'longestWork' to track the longest amount of work done.
    // 'maxWorkDuration' to keep the maximum continuous work duration.
    // 'lastTimestamp' to keep the last timestamp we calculated work duration from.
    let [longestWorkId, maxWorkDuration, lastTimestamp] = [0, 0, 0];

    // Iterate through each log entry.
    for (let [workerId, timestamp] of logs) {
        // Calculate the work duration by subtracting lastTimestamp from current timestamp.
        let workDuration = timestamp - lastTimestamp;

        // Check if the current work duration exceeds our maxWorkDuration or if it is equal and the workerId is lower.
        if (maxWorkDuration < workDuration || (maxWorkDuration === workDuration && longestWorkId > workerId)) {
            // Update the longestWorkId and maxWorkDuration with the current worker's ID and work duration.
            longestWorkId = workerId;
            maxWorkDuration = workDuration;
        }

        // Update lastTimestamp to calculate next work duration correctly.
        lastTimestamp = timestamp;
    }

    // Return the ID of the worker who worked the hardest.
    return longestWorkId;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the number of entries in the `logs` list. The reason is that the code iterates through each log entry exactly once, performing a constant amount of work for each entry (updating variables and simple comparisons).

Space Complexity

The space complexity of the code is `O(1)` because the code uses a fixed amount of space that does not depend on the input size. Variables `last`, `mx`, `ans`, and any other temporary variables used during iteration do not scale with the size of the input, as they are simply storage for current and maximum time differences as well as the corresponding worker ID.