2225. Find Players With Zero or One Losses

Medium Array Hash Table Counting Sorting

Leetcode Link

Problem Description

The problem provides you with an integer array `matches`, where each element represents the outcome of a match as `[winner_i, loser_i]`. The task is to find all players who have not lost any matches and all players who have lost exactly one match, ensuring that all listed players have participated in at least one match.

Here's how you need to organize the output:

- The first list (`answer[0]`) should contain all players who have never lost a match.
- The second list (`answer[1]`) should include all players who have lost exactly one match.

It's important to list the players in ascending order. If a player does not appear in the `matches` array, they are not considered since no match records for them exist. Additionally, players will not lose more than one match because match outcomes are unique.

Intuition

The intuition behind the solution is to track the number of losses for each player. This is achievable by counting the occurrences of each player in the loser position of every match.

To do this systematically, we can use the following steps:

1. Utilize a `Counter` to keep a tally of the losses for each player.
2. Iterate over the `matches` list, incrementing the count for losers. If a player wins a match, ensure they're in the `Counter` with a loss count of 0 (since winning means they have not lost that match).
3. After processing all matches, go through the Counter and categorize players based on their loss count.
   - Players with 0 losses go into `answer[0]`.
   - Players with exactly 1 loss go into `answer[1]`.
4. Sort both lists to satisfy the ascending order requirement.
5. Return the sorted lists as the final result.

Using the `Counter`, we abstract the complexity of tracking individual losses and make it simple to determine which list a player belongs to based on their number of losses after processing all match outcomes.

Solution Approach

The solution's implementation follows an efficient approach to categorize players based on their match outcomes:

1. A `Counter` object is utilized to maintain the tally of losses for each player. This data structure is optimal for this purpose because it allows us to keep track of how many times each player appears as the loser and does not count their wins.
2. Iterate through the `matches` list, and for each match `[winner, loser]`:
   - Check if the `winner` is already in the `Counter`. If not, initialize their loss count to 0 because winning a match implies they haven't lost it.
   - Increment the `loser`'s count by one to signify their loss in this match.
3. After tallying the losses for each player, create a two-dimensional list `ans`, where `ans[0]` will eventually contain players with 0 losses, and `ans[1]` will contain players with exactly one loss.
4. Iterate through the items in the `Counter` (for `u, v` in `cnt.items()`). For each player (`u`) and their loss count (`v`):
   - If `v` is less than 2 (meaning the player has either won all their matches or lost just one), add the player to `ans[v]`. This works because `v` can only be 0 or 1, as we are only interested in players with no losses or exactly one loss.
5. The lists need to be sorted in increasing order, as per the problem specifications. Thus, both `ans[0]` and `ans[1]` are sorted using the `sort()` method.
6. Finally, return the list `ans` as the result, where `ans[0]` contains all players that have not lost any matches and `ans[1]` contains all players that have lost exactly one match.

This algorithm efficiently uses a hash map (provided by `Counter` in Python) to count occurrences, which is ideal for frequency counting tasks. The overall time complexity of the algorithm is determined by the number of matches and the sorting step, and the space complexity is largely influenced by the `Counter` used to store losses per player.

Example Walkthrough

Let's go through a small example to illustrate the solution approach. Assume we have the following `matches` array:

```
matches = [[1, 3], [2, 1], [4, 2], [5, 2]]
```

In this array, the subarrays represent match outcomes with winners and losers respectively.

1. Initialize a `Counter` to count losses:
   - After initializing: `Counter({})`
2. Start iterating through `matches`:
   - Match `[1, 3]`: `Counter({3: 1})`
   - Match `[2, 1]`: `Counter({3: 1, '1': 1})`
   - Before proceeding with the loser of the next match, ensure that winner `4` has a loss count of 0: `Counter({'3': 1, '1': 1, '4': 0})`
   - Match `[4, 2]`: `Counter({'3': 1, '1': 1, '2': 1, '4': 0})`
   - Again, ensure that winner `5` has a loss count of 0 before the next loser is processed: `Counter({'3': 1, '1': 1, '2': 1, '4': 0, '5': 0})`
   - Match `[5, 2]`: `Counter({'3': 1, '1': 1, '2': 2, '4': 0, '5': 0})`
3. Now we have all loss counts, create list `ans` with two sublists for 0 losses and 1 loss:
   - `ans = [[], []]`
4. Iterate through the counts and populate `ans` accordingly:
   - Player `3` has 1 loss: `ans = [[], [3]]`
   - Player `1` has 1 loss: `ans = [[], [3, 1]]`
   - Player `4` has 0 losses: `ans = [[4], [3, 1]]`
   - Player `5` has 0 losses: `ans = [[4, 5], [3, 1]]`
   - Player `2` is not added to either sublist since they've lost more than one match.
5. Sort both `ans[0]` and `ans[1]` as required:
   - `ans[0].sort()`: `ans = [[4, 5], [3, 1]]`
   - `ans[1].sort()`: `ans = [[4, 5], [1, 3]]`
6. The final sorted `ans` list is returned from the function:

```
ans = [[4, 5], [1, 3]]
```

This holds our result, where `ans[0]` contains players `4` and `5` who have never lost any matches, and `ans[1]` contains players `1` and `3` who have lost exactly one match, listed in ascending order.

Python Solution

```python
from collections import Counter

class Solution:
    def findWinners(self, matches):
        loss_counts = Counter()

        # Count the number of losses for each player
        for winner, loser in matches:
            # Ensure the winner is in the Counter, implicitly check if 'winner' is in 'count_losses'
            # Since getting count_losses[winner] will default to 0 if not present
            loss_counts[loser] += 1

        winners_with_zero_losses = []
        winners_with_one_loss = []

        # Iterate over the players and their loss counts
        for player, loss_count in loss_counts.items():
            # If a player has less than 2 losses, add them to the respective list
            if loss_count < 2:
                if loss_count == 0:
                    winners_with_zero_losses.append(player)
                else:
                    winners_with_one_loss.append(player)

        # Players with no losses are served first & with exactly one loss served after
        # Sort the lists to meet the output criteria
        winners_with_zero_losses.sort()
        winners_with_one_loss.sort()

        # Combine the two lists into a single list of lists for the result
        result = [winners_with_zero_losses, winners_with_one_loss]
        return result
```

Java Solution

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Collections;

class Solution {
    public List<List<Integer>> findWinners(int[][] matches) {
        // HashMap to store the number of losses for each player
        Map<Integer, Integer> lossCountMap = new HashMap<>();
        // Process all match results
        for (int[] match : matches) {
            int winner = match[0];
            int loser = match[1];
            // Initialize the winner's loss count if not already present in map
            lossCountMap.putIfAbsent(winner, 0);
            // Increment the loss count for the loser
            lossCountMap.put(loser, lossCountMap.getOrDefault(loser, 0) + 1);
        }

        // Create the result list containing two lists: one for all the players who have never lost, and one for the players who have
        List<List<Integer>> winnersList = new ArrayList<>();
        winnersList.add(new ArrayList<>());
        winnersList.add(new ArrayList<>());

        // Iterate through each entry in the loss count map
        for (Map.Entry<Integer, Integer> entry : lossCountMap.entrySet()) {
            int loser = entry.getKey();
            int losses = entry.getValue();
            // If the player has lost fewer than 2 matches, include them in the appropriate sublist (0 losses or 1 loss)
            if (losses < 2) {
                winnersList.get(losses).add(loser);
            }
        }

        // Sort the sublists of players with 0 losses and exactly 1 loss
        Collections.sort(winnersList.get(0));
        Collections.sort(winnersList.get(1));

        // Return the result list
        return winnersList;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>
#include <algorithm>

class Solution {
public:
    // Function to find the players who have never lost a match (winners)
    // and those who have lost exactly one match (one-match-losers)
    vector<vector<int>> findWinners(vector<vector<int>>& matches) {
        unordered_map<int, int> lossCount; // Map storing the loss count for each player

        // Process each match result and update the loss count for the players
        for (auto& match : matches) {
            int winner = match[0];
            int loser = match[1];

            // Make sure every player is included in the map
            if (!lossCount.count(winner)) {
                lossCount[winner] = 0;
            }

            // Increment the loss count for the loser of the match
            ++lossCount[loser];
        }

        vector<vector<int>> answer(2); // To hold the final result

        // Iterate through the map to classify players based on their loss counts
        for (auto& playerLossPair : lossCount) {
            int player = playerLossPair.first;
            int losses = playerLossPair.second;

            // If the player has less than 2 losses, add them to the respective list
            if (losses < 2) {
                answer[losses].push_back(player);
            }
        }

        // Sort the list of winners and one-match-losers
        sort(answer[0].begin(), answer[0].end());
        sort(answer[1].begin(), answer[1].end());

        return answer; // Return the sorted lists
    }
};
```

Typescript Solution

```typescript
// Import statements are not required in TypeScript for data structures like arrays and maps.

// Type Alias for readability, representing a match result with winner and loser.
type MatchResult = [number, number];

// Function to update the loss count map with match results.
function processMatch(lossCount: Map<number, number>, winner: number, loser: number): void {
    // Ensure every player is present in the map; if not add them with zero losses
    if (!lossCount.has(winner)) {
        lossCount.set(winner, 0);
    }
    // Increment the loss count for the loser of the match
    const currentLossCount = lossCount.get(loser) || 0;
    lossCount.set(loser, currentLossCount + 1);
}

// Function to sort and categorize players based on their loss counts into winners and one-match-losers
function findWinners(matches: MatchResult[]): number[][] {
    const lossCount: Map<number, number> = new Map(); // Map to store the loss count for each player

    // Process each match result and update the loss count for the players
    for (const match of matches) {
        const [winner, loser] = match;
        processMatch(lossCount, winner, loser);
    }

    // Arrays to hold the list of players who never lost and those who lost exactly one match
    const oneMatchLosers: number[] = [];
    const neverLostWinners: number[] = [];

    // Iterate through the map to classify players based on their loss counts
    for (const [player, losses] of lossCount) {
        // If the player has no losses, they are a winner. If only one loss, they are a one-match-loser.
        if (losses === 0) {
            neverLostWinners.push(player);
        } else if (losses === 1) {
            oneMatchLosers.push(player);
        }
    }

    // Sort the list of winners and one-match-losers
    neverLostWinners.sort((a, b) => a - b);
    oneMatchLosers.sort((a, b) => a - b);

    // Return the sorted lists as a 2D array
    return [neverLostWinners, oneMatchLosers];
}

// Usage of the findWinners function can be demonstrated with an example:
const exampleResults: MatchResult[] = [
    [1, 3],
    [2, 3],
    [3, 6],
    [5, 6],
    [5, 7],
    [4, 7],
];
const results = findWinners(exampleResults);
console.log('Winners: ${results[0]}, One-Match Losers: ${results[1]}');
```

Time and Space Complexity

The given Python code seeks to find all players who never lost a game (who are undefeated), and all players who lost exactly one game. Let's break down its time and space complexity.

Time Complexity

1. Creating the counter: `O(N)` - Creating the counter object requires iterating over the list of matches where `N` is the number of matches.
2. Initializing player counts: `O(N)` - We iterate through all matches, and for each match we perform a check and counter increment which operates in constant time, resulting in `O(N)`.
3. Traversing the counter for players with losses less than 2: `O(P)` - We go through the counter which contains `P` unique players.
4. Sorting winners and players with one loss: `O(PlogP)` - Sorting is performed on the players' list for those who have not lost and those who have lost one match. In the worst case, all players could either be winners or have one loss, hence the sorting can be `O(PlogP)` for `P` unique players.

The overall time complexity is the sum of these operations, dominated by the sorting steps: `O(N) + O(P) + O(PlogP)`. Since the sorting term is usually the most significant for large lists, we can simplify this expression to: `O(PlogP)`.

Space Complexity

1. Counter object: `O(P)` - The counter object holds at most `P` unique players, which is the space required.
2. Answer List: `O(P)` - The answer list is a list of two lists, which in the worst case would hold all `P` players, resulting in `O(P)` space.

The overall space complexity combines both aspects, remaining `O(P)` since it is not multiplied by any factor.

Therefore, the final complexity of the provided code is:

- Time Complexity: `O(PlogP)`
- Space Complexity: `O(P)`