2923. Find Champion I

Easy Array Matrix

Problem Description

In this problem, you're given data representing the results of matches in a tournament. This data is in the form of a 2D boolean matrix `grid`, where each dimension represents the `n` teams in the tournament. If `grid[i][j]` is `1`, it means team `i` is stronger than team `j`. Conversely, if it's `0`, then team `i` is weaker than team `j`. The objective is to find out which team is the champion, with the definition of a champion being a team that is stronger than all other teams.

To reiterate, we're looking for a team that has won against every other team. If such a team exists, it will be declared the champion. Importantly, we know that every team will only have a row where they are the winner if they've won every match against others, and we need to identify such a row to find the champion.

Intuition

When thinking about how to solve this problem, one straightforward method that comes to mind is to enumerate each team and check their match outcomes. If we find a team 'i' that has beaten every other team, which in the matrix translates to a row filled with '1's except the diagonal (as a team does not play against itself), then `i` is the champion.

To implement this solution in an efficient way, we iterate over each team `i` by going through each row in the matrix. Then, using a comprehension, we check if every element `x` in the row is `1` for all other teams `j`, implying that team `i` has won every match with others.

Solution Approach

The Python solution provided takes advantage of a few Python-specific features. Firstly, it uses list comprehensions - a compact way to process all entries in a list. In the context of the given solution, the comprehension checks if a team has all victories (`1`) against other teams, which can be checked by looking through the row corresponding to that team in the `grid` matrix. The condition `x == 1 for j, x in enumerate(row) if i != j` makes sure we skip the team's own position since a team cannot compete against itself – that's why the check `i != j` is there.

Here's a breakdown of key elements used in the implementation:

- **Enumeration**: The `enumerate` function is used to get both the index (`i`) and the value (`row`) from the `grid`. The index represents the team number, and the row contains the match outcomes for that team.
- **List Comprehensions**: Comprehensions are used to go through each match result in the row while skipping the match result against themselves, as indicated by `i != j`.
- **All Function**: `all()` is a built-in function in Python that returns `True` if all elements of the given iterable are true (or if the iterable is empty). In the given context, it is used to check if a team has a row of all wins (`1`), which would satisfy the condition to be the champion.
- **Early Return**: Instead of iterating through the entire matrix and then deciding the result, the solution returns the index (the team number) as soon as it finds the row that satisfies the champion criteria. This is efficient because it stops further unnecessary computation once the champion is identified.

To summarize the solution approach as per the problem:

```
def findChampion(self, grid: List[List[int]]) -> int:
    for i, row in enumerate(grid):
        if all(x == 1 for j, x in enumerate(row) if i != j):
            return i
```

This method iterates through each row with index `i`:

1. For each team `j`, check through all match outcomes `x` in the corresponding `row`.
2. Skip the outcome where a team played against themselves with the condition `i != j`.
3. If all outcomes `x` against other teams `j` are victories (`1`), then team `i` is the champion, and we return `i`.

In this problem, we make some assumptions:

- There can only be one champion.
- A draw is not possible in a match between two teams.
- When a team wins, their corresponding cell is set as `1` and `0` otherwise.

The solution uses a basic iteration of rows in combination with conditional checks and short-circuits as soon as the condition for being a champion is satisfied, making it an efficient solution to find the champion of the tournament.

Example Walkthrough

Let's walk through a small example using a hypothetical 3-team tournament to illustrate the solution approach. Suppose we have the following tournament results in a boolean matrix:

```
grid = [
  [0, 1, 1],  # Team 0's results
  [0, 0, 1],  # Team 1's results
  [0, 0, 0]   # Team 2's results
]
```

In this matrix:

- `grid[0]` represents that Team 0 lost to Team 1 (`grid[0][1] = 1` indicates Team 1 is stronger than Team 0) and won against Team 2 (`grid[0][2] = 1` indicates Team 0 is stronger than Team 2).
- `grid[1]` represents that Team 1 lost to Team 0 (`grid[1][0] = 0` indicates Team 0 is stronger than Team 1) but won against Team 2 (`grid[1][2] = 1`).
- `grid[2]` represents that Team 2 lost to both Team 0 (`grid[2][0] = 0`) and Team 1 (`grid[2][1] = 0`).

We want to find out which team is the champion, which is the team that won against every other team. This means we are looking for a row in `grid` which, except for the diagonal entry, contains all `1`s.

Let's apply our solution approach:

1. Start by iterating through the `grid` matrix, getting both the index `i` and the row.
   - `i = 0`, `row = [0, 1, 1]`: We pass over the first entry (since team doesn't compete against itself) and check the rest. Team 0 has one loss (`grid[0][0] = 0`), so it cannot be the champion.
   - `i = 1`, `row = [0, 0, 1]`: Again, we pass over the diagonal entry. Team 1 has a loss, so it cannot be the champion either.
   - `i = 2`, `row = [0, 0, 0]`: We ignore the diagonal, but all other entries are `0`, so Team 2 has not won any matches.
2. We used the `all()` Python function to check if all entries in a row, excluding the diagonal, are `1`.

Since none of the teams have a row with all `1`s, we conclude that there is no champion in this tournament. If there was a team that won against every other team, the `all()` call would return `True` for that team's row, and the function would return that team's index as the champion.

Following is what the Python code might look like for this example:

```
def findChampion(grid):
    for i, row in enumerate(grid):
        if all(x == 1 for j, x in enumerate(row) if i != j):
            return i
    return -1  # Using -1 to indicate no champion found

# Testing the example grid
grid = [
  [0, 1, 1],
  [0, 0, 1],
  [0, 0, 0]
]

print(findChampion(grid))  # Output will be -1 since no team meets the criteria
```

As we can see, the function checks each team's winning record against others and concludes correctly based on the problem description. No team satisfies the condition to be the champion in this particular case.

Solution Implementation

Python

```
class Solution:
    def findChampion(self, grid: List[List[int]]) -> int:
        # Iterate over each row in the grid
        for row_index, row in enumerate(grid):
            # Check if the current player (row) has defeated all others.
            # This is done by verifying that all elements in the row equal to 1,
            # except the element where the opponent would be the same as the player,
            # which we skip by checking if the column index is not equal to the row index.
            if all(value == 1 for col_index, value in enumerate(row) if row_index != col_index):
                # A champion (player who defeated all others) is found - return their index
                return row_index
        # If we get to this point, there's no champion found in the grid
        return -1
        # Add -1 to indicate no champion is found if that's the expected behavior.
        # It wasn't in the original code, but it might be an oversight to handle the condition when no champion exists.
```

Java

```
class Solution {
    // Method to find the champion in a given grid
    // A champion is defined as the person who has defeated all other persons
    public int findChampion(int[][] grid) {
        int numberOfPersons = grid.length; // Get the number of persons from the grid's length

        // Iterate through each person (indexed from 0 to numberOfPersons-1)
        for (int personIndex = 0;; ++personIndex) {
            int winCount = 0; // Initialize a count of victories against other persons

            // Iterate through all potential opponents
            for (int opponentIndex = 0; opponentIndex < numberOfPersons; ++opponentIndex) {
                // Check if the current personIndex has a victory against the opponentIndex
                // and they are not the same person
                if (personIndex != opponentIndex && grid[personIndex][opponentIndex] == 1) {
                    winCount++; // Increment the win count for each victory
                }
            }

            // If the win count equals the number of potential opponents,
            // then this person is the champion
            if (winCount == numberOfPersons - 1) {
                return personIndex; // Return the index (ID) of the champion
            }
        }
    }
}
```

C++

```
#include <vector>

class Solution {
public:
    // Method to find the champion index in a grid, where a champion is a person who has defeated all others.
    int findChampion(std::vector<std::vector<int>>& grid) {
        int n = grid.size(); // Size of the grid (number of players)

        // Loop through each person in the grid
        for (int i = 0; i < n; ++i) {
            int victoryCount = 0; // Counter to keep track of victories for the current person

            // Count the number of people the current person has defeated.
            for (int j = 0; j < n; ++j) {
                // If it's not the same person and the current person has defeated person j, increment the count.
                if (i != j && grid[i][j] == 1) {
                    victoryCount++;
                }
            }

            // If the current person has defeated n-1 other people, return this person's index.
            if (victoryCount == n - 1) {
                return i;
            }
        }

        // If no champion exists, return -1 to indicate failure (this part was missing in the original code).
        return -1;
    }
};
```

TypeScript

```
/**
 * Finds a "champion" in the given grid, where a champion is defined as a row that defeats all others.
 * Each element grid[i][j] = 1 indicates i defeats j, a row is a champion if it defeats every other row.
 * @param {number[][]} grid - A square matrix representing wins (1) and losses (0).
 * @returns {number} The index of the champion row, if it exists.
 */
function findChampion(grid: number[][]): number {
    // Obtain the total number of competitors from the grid's length
    const totalCompetitors = grid.length;

    // Iterate over each competitor to find a champion
    for (let competitorIndex = 0; competitorIndex < totalCompetitors; ++competitorIndex) {
        let victoriesCount = 0;

        // Count the number of victories for the current competitor against all others
        for (let opponentIndex = 0; opponentIndex < totalCompetitors; ++opponentIndex) {
            // Increment the victory count if the competitor defeated the opponent,
            // making sure not to count self-defeats
            if (competitorIndex !== opponentIndex && grid[competitorIndex][opponentIndex] === 1) {
                ++victoriesCount;
            }
        }

        // If the competitor defeated all other competitors, return the champion's index
        if (victoriesCount === totalCompetitors - 1) {
            return competitorIndex;
        }
    }

    // If no champion is found (which this code won't hit because the loop is supposed to run indefinitely
    // or until it finds a champion), this statement would normally throw an error or handle it according to the requirements.
    throw new Error('No champion found.');
}
```

```
class Solution:
    def findChampion(self, grid: List[List[int]]) -> int:
        # Iterate over each row in the grid
        for row_index, row in enumerate(grid):
            # Check if the current player (row) has defeated all others.
            # This is done by verifying that all elements in the row equal to 1,
            # except the element where the opponent would be the same as the player,
            # which we skip by checking if the column index is not equal to the row index.
            if all(value == 1 for col_index, value in enumerate(row) if row_index != col_index):
                # A champion (player who defeated all others) is found - return their index
                return row_index
        # If we get to this point, there's no champion found in the grid
        return -1
        # Add -1 to indicate no champion is found if that's the expected behavior.
        # It wasn't in the original code, but it might be an oversight to handle the condition when no champion exists.
```

Time and Space Complexity

The time complexity of the provided code is `O(n^2)` where `n` is the number of teams. This is because for each team `i`, the code checks whether all other teams `j` have been defeated by team `i`. This involves a nested iteration where the inner loop runs `n-1` times (because it skips the case where `i == j`) for each of the `n` teams, hence the quadratic time complexity.

The space complexity is `O(1)` as the code only uses a fixed amount of additional memory for variables like `i`, `row`, and `x`. There are no data structures used that grow with the size of the input.