2410. Maximum Matching of Players With Trainers

Medium Greedy Array Two Pointers Sorting

Leetcode Link

Problem Description

In this LeetCode problem, you are given two 0-indexed integer arrays: `players` and `trainers`. Each entry in the `players` array represents the ability of a particular player, and each entry in the `trainers` array represents the training capacity of a particular trainer. A player can be matched with a trainer if the player's ability is less than or equal to the trainer's capacity. It's important to note that each player can only be matched with one trainer, and each trainer can match with only one player. The task is to find the maximum number of such player-trainer pairings where the conditions of matching ability to capacity are met.

Intuition

The intuition behind the solution is straightforward: we want to pair as many players with trainers as we can, prioritizing the pairing of less able players with trainers who have just enough capacity to train them. This ensures that we do not 'waste' a trainer with high capacity on a player with low ability when that trainer could be matched with a more able player instead.

To achieve this, we sort both the `players` and `trainers` arrays. Sorting helps us easily compare the least able player with the least capable trainer and move up their respective arrays. If a match is found, both the player and the trainer are effectively removed from the potential pool by moving to the next elements in the arrays (incrementing the index). If no match is found, we move up the `trainers` array to find the next trainer with a higher capacity that might match the player's ability.

The result is incrementally built by adding a match whenever we find a capable trainer for a player. We stop when we've either paired all players or run out of trainers. The sum of matches made gives us the maximum number of possible pairings.

Solution Approach

The implementation provided uses a simple but effective greedy approach, heavily relying on sorting. The steps involve:

1. Sorting the `players` array in ascending order, which ensures we start with the player with the lowest ability.
2. Sorting the `trainers` array in ascending order, which allows us to start with the trainer having the lowest capacity.

The algorithm works with two pointers, one (`i`) iterating through the `players` array, and the other (`j`) through the `trainers` array.

Here is the implementation broken down:

- Initialize a variable `ans` to `0` to keep track of the number of matches made.
- Initialize the trainer index pointer `j` to `0`.
- Iterate over each player `p` in the sorted `players` list using a for-loop.
  - Within the loop, proceed with an inner while-loop which continues as long as `j` is less than the length of `trainers` and the current trainer's capacity `trainers[j]` is less than the ability of the player `p`. The purpose of this loop is to find the first trainer in the sorted list whose capacity is sufficient to train the player.
  - After the while-loop, if `j` is still within the bounds of the `trainers` array, it means a suitable trainer has been found for the player `p`. We increment `ans` to count the match and also increment `j` to move to the next trainer.
- Once all players have been considered, return the value of `ans`.

Algorithm and Data Structures:

- Lists: The main data structure used here are lists (`players` and `trainers`), which are sorted.
- Greedy approach: The algorithm uses a greedy method by matching each player with the "smallest" available trainer that can train them.
- Two pointers: It uses two pointers to pair players with trainers without revisiting or re-comparing them, thus optimizing the process.
- Sorting: By sorting the arrays, the solution leverages the property that once a player cannot be paired with a current trainer, they cannot be paired with any trainers before that one in the sorted list either.

Patterns used:

- Sorting and Two-pointers: This is a common pattern for efficiently pairing elements from two different sorted lists.

Example Walkthrough

Let's say we have the following `players` and `trainers` arrays:

- `players`: `[2, 3, 4]`
- `trainers`: `[1, 2, 5]`

Following the solution approach:

1. We first sort both the `players` and `trainers` arrays:

   After sorting,

   - `players`: `[2, 3, 4]` (already sorted)
   - `trainers`: `[1, 2, 5]` (already sorted)
2. We initialize `ans` to `0`. This variable will keep track of the successful player-trainer matches.
3. We also initialize the trainer index pointer `j` to `0`.

Now, let's go through each player and try to find a trainer match:

- Player with ability `2` is the first in the `players` array.
  - We check the trainers in order: trainer `1` cannot train the player because the trainer's capacity is too low.
  - Move to the next trainer (`2`), and we find a match. Trainer `2` can train the player with ability `2`.
  - We increment `ans` to `1` and move to the next trainer (`j` becomes `2`).
- Player with ability `3` is the next.
  - The trainer in position `2` in the sorted `trainers` list has a capacity of `5`, which is sufficient.
  - We match this player with the trainer.
  - We increment `ans` to `2`, and since there are no more trainers, we move on to the final player.
- Player with ability `4`:
  - There are no more trainers to compare since we exceeded the length of the `trainers` array.

At the end of this process, the value of `ans` is `2`, which signifies that we were able to match two pairs of players and trainers successfully.

Python Solution

```python
from typing import List  # Import List from typing module for type hints

class Solution:
    def matchPlayersAndTrainers(self, players: List[int], trainers: List[int]) -> int:
        # Sort the list of players in ascending order
        players.sort()
        # Sort the list of trainers in ascending order
        trainers.sort()

        # Initialize the count of matched players and trainers
        match_count = 0
        # Initialize the pointer for trainers list
        trainer_index = 0

        # Iterate over each player in the sorted list
        for player in players:
            # Skip trainers which have less capacity than the current player's strength
            while trainer_index < len(trainers) and trainers[trainer_index] < player:
                trainer_index += 1

            # If there is a trainer that can match the current player
            if trainer_index < len(trainers):
                # Increment the match count
                match_count += 1
                # Move to the next trainer for the next player
                trainer_index += 1

        # Return the total number of matches
        return match_count
```

Java Solution

```java
class Solution {
    public int matchPlayersAndTrainers(int[] players, int[] trainers) {
        // Sort the players array in ascending order
        Arrays.sort(players);
        // Sort the trainers array in ascending order
        Arrays.sort(trainers);

        // Initialize the count of matches to 0
        int matches = 0;
        // Initialize the index for trainers to 0
        int trainerIndex = 0;

        // Iterate over each player
        for (int player : players) {
            // Increment the trainerIndex until we find a trainer that can match the player
            while (trainerIndex < trainers.length && trainers[trainerIndex] < player) {
                trainerIndex++;
            }
            // If there is a trainer that can match the player, increment the match count
            if (trainerIndex < trainers.length) {
                matches++; // A match is found
                trainerIndex++; // Move to the next trainer.
            }
        }
        // Return the total count of matches
        return matches;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include the necessary header for std::sort

class Solution {
public:
    // Function to match players with trainers based on their strength.
    // Each player can only be matched with a trainer that is equal or greater in strength.
    int matchPlayersAndTrainers(vector<int>& players, vector<int>& trainers) {
        std::sort(players.begin(), players.end()); // Sort the players in ascending order
        std::sort(trainers.begin(), trainers.end()); // Sort the trainers in ascending order

        int matches = 0; // Initialize the number of matches to zero
        int trainersIndex = 0; // Initialize the trainers index to the start of the array

        // Iterate through each player
        for (int playerStrength : players) {
            // Find a trainer that can match the current player's strength
            while (trainersIndex < trainers.size() && trainers[trainersIndex] < playerStrength) {
                trainersIndex++; // Increment trainers index if the current trainer is weaker than the player
            }

            // If a suitable trainer is found, make the match
            if (trainersIndex < trainers.size()) {
                matches++; // Increment match count
                trainersIndex++; // Move to the next trainer for the following players
            }
        }

        return matches; // Return the total number of matches made
    }
};
```

Typescript Solution

```typescript
// Import the necessary module for sorting
import { sort } from 'some-sorting-module'; // Please replace 'some-sorting-module' with the actual module you would use for sorting

// Function to sort an array in ascending order
function sortArrayAscending(array: number[]): number[] {
    return sort(array, (a, b) => a - b);
}

// Function to match players with trainers based on their strength.
// Each player can only be matched with a trainer that is equal or greater in strength.
function matchPlayersAndTrainers(players: number[], trainers: number[]): number {
    // Sort the players and trainers in ascending order
    const sortedPlayers: number[] = sortArrayAscending(players);
    const sortedTrainers: number[] = sortArrayAscending(trainers);

    let matches: number = 0; // Initialize the number of matches to zero
    let trainersIndex: number = 0; // Initialize the trainers index to the start of the array

    // Iterate through each player and match with trainers based on strength
    for (const playerStrength of sortedPlayers) {
        // Find a trainer that can match the current player's strength
        while (trainersIndex < sortedTrainers.length && sortedTrainers[trainersIndex] < playerStrength) {
            trainersIndex++; // Increment trainers index if the current trainer is weaker than the player
        }

        // If a suitable trainer is found, make the match
        if (trainersIndex < sortedTrainers.length) {
            matches++; // Increment match count
            trainersIndex++; // Move to the next trainer for the following players
        }
    }

    return matches; // Return the total number of matches made
}

// Export the function if this is part of a module
export { matchPlayersAndTrainers };
```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by the sorting operations and the subsequent for-loop with the inner while loop.

1. Sorting both the `players` and `trainers` list takes $O(n\log n)$ and $O(m\log m)$, respectively, where n is the number of players and m is the number of trainers.
2. The for-loop iterates over each player, which is $O(n)$. Within this loop, the while loop progresses without resetting, which makes it linear over m elements of trainers in total. In the worst case, all players are checked against all trainers, hence it contributes a maximum of $O(m)$ time complexity.

The combined time complexity of these operations would be $O(n\log n) + O(m\log m) + O(n + m)$. However, since $O(n\log n)$ and $O(m\log m)$ are the dominant terms, the overall time complexity simplifies to $O(n\log n + m\log m)$.

Space Complexity

The space complexity of the code is $O(1)$, which is the additional space required. The sorting happens in place, and no additional data structures are used aside from a few variables for keeping track of indexes and the answer.