1996. The Number of Weak Characters in the Game

Medium Stack Greedy Array Sorting Monotonic Stack Leetcode Link

Problem Description

In the given LeetCode problem, you are playing a game with multiple characters. Each character has two properties: attack and defense. These properties are provided in a 2D integer array `properties`, where `properties[i] = [attack_i, defense_i]` contains the properties of the `i-th` character. A character is considered weak if there is another character with both an attack and defense level strictly greater than that of the said character. The goal is to determine how many characters are weak according to these conditions.

Intuition

The key to solving this problem is to optimize the way we look for characters that can be dominated (or are weak compared to others). A brute-force approach of comparing every character with every other character would result in a time-consuming solution. Thus, the main challenge is to reduce the number of comparisons needed.

One effective tactic for reducing comparisons is to sort the characters in a way that allows us to easily identify the weak ones. We can do this by sorting the characters based on their attack values in descending order. If their attack values are the same, we sort based on their defense values in ascending order. This specific ordering makes sure that when we traverse the sorted array, any character we see with a higher defense value will definitely have a lower attack value (because we have sorted them in descending order), thus indicating the presence of a weak character.

The intuition behind sorting by descending attack is to ensure that when iterating over the characters, any subsequently encountered character with a lower defense is guaranteed to be weak since there cannot be a character with a higher attack preceding it.

Once the array is sorted, we initialize two variables, `ans` and `mx`. The variable `ans` keeps track of the number of weak characters, and `mx` stores the maximum defense value encountered so far. As we iterate through the properties array, we compare each character's defense value with `mx`. If the defense value is strictly less than `mx`, this character is weak, and we increment `ans` by one. Otherwise, we update `mx` with the current character's defense value if it's higher than the current `mx`. The final result is the value of `ans`, which represents the total number of weak characters found.

Solution Approach

The solution uses sorting and a one-pass scan through the sorted list of character properties to determine the weak characters.

Here's a step-by-step explanation of the algorithm:

1. **Sorting the Properties**: We start by sorting the `properties` array based on two criteria. First, we sort by the attack value in descending order, and in case of a tie, we sort by defense value in ascending order. This can be achieved using a custom sort function:

   ```python
   properties.sort(key=lambda x: (-x[0], x[1]))
   ```

   This ensures that for any character `i` and `j` where `i < j` in the sorted array, `attack_i >= attack_j`. If `attack_i == attack_j`, then `defense_i <= defense_j`.

2. **Initialising Counters**: Two variables are initialised: `ans` to count the number of weak characters, and `mx` to keep track of the maximum defense value seen so far in the iteration.

3. **Iterating Through the Sorted Properties**: The array is iterated through after being sorted:

   ```python
   for _, defense in properties:
       ans += defense < mx
       mx = max(mx, defense)
   ```

   Each step of the iteration does the following:

   - If the defense of the current character is less than the maximum defense seen so far (`mx`), then this character is weak, and `ans` is incremented by 1.
   - The `mx` is then updated with the maximum of its current value and the defense value of the current character.

4. **Return Weak Characters Count**: After the iteration is complete, the value of `ans` represents the total count of weak characters. It is returned as the final result of the function.

This approach uses a sorting algorithm that typically has a time complexity of `O(n log n)` and a scan of the sorted array with a time complexity of `O(n)`, resulting in an overall time complexity of `O(n log n)` where `n` is the number of characters in the game.

Using a sorted list as the data structure enables an efficient single pass to determine weak characters, leveraging the sorted order to minimize the required comparisons.

Example Walkthrough

Let's illustrate the solution approach with a simple example. Suppose we have the following `properties` array for the characters:

```
properties = [[5,5],[6,3],[3,6]]
```

This array tells us that we have three characters:

- Character 1 has an attack of 5 and a defense of 5.
- Character 2 has an attack of 6 and a defense of 3.
- Character 3 has an attack of 3 and a defense of 6.

Following through the steps of the solution:

1. **Sorting the Properties**: Sort the characters so that those with higher attack values come first. In case of a tie in the attack values, the one with the lower defense value comes first. After sorting:

```
properties after sorting = [[6,3], [5,5], [3,6]]
```

Now the list is ordered in such a way that it's easier to determine whether a character is weak without having to compare it with all other characters.

2. **Initialising Counters**: Initialize `ans` to 0 and `mx` to -1 (assuming all defense values are positive).

3. **Iterating Through the Sorted Properties**:

   - Character 1: Attack = 6, Defense = 3. Since `mx` is -1, we don't increase `ans`. We update `mx` to 3.
   - Character 2: Attack = 5, Defense = 5. Because the defense (5) is greater than `mx` (3), Character 2 is not weak. We update `mx` to 5.
   - Character 3: Attack = 3, Defense = 6. Here, the defense (6) is greater than the current `mx` (5), so Character 3 is also not weak. `mx` updates to 6.

No character's defense was strictly less than the maximum defense seen so far after them, so:

4. **Return Weak Characters Count**: `ans` remains 0. Therefore, according to our algorithm, there are no weak characters in this example.

Python Solution

```python
from typing import List

class Solution:
    def numberOfWeakCharacters(self, properties: List[List[int]]) -> int:
        # Sort the characters in descending order of their attack value.
        # If two characters have the same attack value, sort them in
        # ascending order of their defense value.
        properties.sort(key=lambda character: (-character[0], character[1]))

        # Initialize the count of weak characters to zero.
        weak_characters_count = 0

        # Initialize the maximum defense found so far to zero.
        max_defense = 0

        # Iterate over the characters, which are now sorted by the rules above.
        for _, defense in properties:
            # If the current character's defense is less than the maximum
            # defense found so far, it is a weak character.
            if defense < max_defense:
                weak_characters_count += 1
            else:
                # Update the maximum defense for future comparisons.
                max_defense = defense

        # Return the total count of weak characters.
        return weak_characters_count

```

Java Solution

```java
class Solution {
    public int numberOfWeakCharacters(int[][] properties) {
        // Sorting the properties array with a custom comparator
        // Characters are sorted by attack in descending order
        // If attacks are equal, they're sorted by defense in ascending order
        Arrays.sort(properties, (a, b) -> {
            if (b[0] - a[0] == 0) {
                return a[1] - b[1]; // Sort by defense ascending if attacks are equal
            } else {
                return b[0] - a[0]; // Otherwise, sort by attack descending
            }
        });

        int countWeakCharacters = 0; // Initialize counter for weak characters
        int maxDefense = 0; // To store the maximum defense seen so far

        // Iterate over the sorted array to count weak characters
        for (int[] character : properties) {
            // If the current character's defense is less than the maximum
            // defense seen so far, it is a weak character
            if (character[1] < maxDefense) {
                countWeakCharacters++; // Increment weak character count
            }
            // Update the maximum defense seen so far
            maxDefense = Math.max(maxDefense, character[1]);
        }

        return countWeakCharacters;  // Return the total count of weak characters
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This function calculates the number of weak characters in the game.
    // A weak character is defined as one whose attack and defense are both
    // strictly less than another character's attack and defense.
    // @param properties: A 2D vector where each inner vector contains the attack
    //                    and defense values of a character.
    // @return: The number of weak characters.
    int numberOfWeakCharacters(vector<vector<int>>& properties) {
        // Sort the properties in descending order by attack and if two characters have
        // the same attack, then sort in ascending order by defense.
        sort(properties.begin(), properties.end(), [](const auto& a, const auto& b) {
            return a[0] == b[0] ? a[1] < b[1] : a[0] > b[0];
        });

        int countWeakCharacters = 0; // This variable stores the count of weak characters.
        int maxDefense = 0; // This variable stores the maximum defense seen so far.

        // Iterate through each of the characters in the sorted properties.
        for (const auto& character : properties) {
            // If the current character's defense is less than the maximum defense seen
            // so far, it means that there is another character with both higher attack
            // and defense, so the current character is weak.
            if (character[1] < maxDefense) {
                ++countWeakCharacters;
            }
            // Update the maximum defense seen so far.
            maxDefense = max(maxDefense, character[1]);
        }

        // Return the final count of weak characters.
        return countWeakCharacters;
    }
};

```

Typescript Solution

```typescript
/**
 * Determines the number of weak characters.
 * @param properties - A 2D array where each subarray contains the attack and defense values of a character.
 * @returns The count of weak characters.
 */
function numberOfWeakCharacters(properties: number[][]): number {
    // Sort the characters primarily by their attack in descending order,
    // and secondarily by their defense in ascending order.
    properties.sort((firstCharacter, secondCharacter) => (
        firstCharacter[0] === secondCharacter[0]
            ? firstCharacter[1] - secondCharacter[1] // Ascending defense sort if attacks are equal
            : secondCharacter[0] - firstCharacter[0] // Descending attack sort
    ));

    // Initialize the count of weak characters and the maximum defense found so far.
    let weakCharacterCount = 0;
    let maxDefense = 0;

    // Iterate over each character to determine if it's weak.
    for (const [, defense] of properties) {
        if (defense < maxDefense) {
            // If the character's defense is less than the max defense found, it is weak.
            weakCharacterCount++;
        } else {
            // Update the maximum defense seen so far if the current defense is higher.
            maxDefense = defense;
        }
    }

    // Return the total count of weak characters.
    return weakCharacterCount;
}

```

Time and Space Complexity

The time complexity of the given code is primarily governed by the sorting operation and then the single pass through the `properties` array. The sorting operation has a time complexity of `O(n log n)` where `n` is the number of elements in `properties`. The subsequent for loop is linear, providing a time complexity of `O(n)`. Therefore, the overall time complexity is `O(n log n)` due to the sort being the most significant operation.

The space complexity of the code is `O(1)` or constant space, not counting the space used for the input and output of the function. While the `sort()` method itself is implemented in a way that can provide `O(n)` space complexity in Python's Timsort algorithm worst-case scenario, it's generally considered in-place for most practical purposes as it's a part of the language's standard functionality. Hence, beyond what is needed to store the `properties` list, only a fixed amount of additional space is used for variables such as `ans` and `mx`.