1626. Best Team With No Conflicts

Medium Array Dynamic Programming Sorting

Leetcode Link

Problem Description

In this problem, you are playing the role of a manager who is forming a basketball team for a tournament. The primary goal is to assemble a team with the highest possible aggregate score. The overall score of a team is calculated by adding up the individual scores of all the players on the team. However, assembling the team comes with a constraint to avoid conflicts. A conflict is defined as a situation where a younger player has a higher score than an older player. If two players have the same age, then there is no conflict regardless of their scores.

You are provided with two lists: `scores` and `ages`. The `scores` list contains the score for each player, while the `ages` list contains ages of the players. The indices of the two lists correspond, meaning `scores[i]` and `ages[i]` represent the score and age of the same player.

The task is to find the highest team score possible without any conflicts arising from the age and score constraints when picking team members.

Intuition

To solve the problem of assembling a conflict-free team with the maximum possible score, we can leverage a variation of the classic Dynamic Programming (DP) approach known as the Longest Increasing Subsequence (LIS). The LIS typically aims to find the longest increasing subsequence within a sequence of numbers.

However, in this scenario, because we need to respect both the age and the scores of players, and since we cannot have a younger player with a higher score than an older player, the players must be sorted in a way that respects both conditions. We sort the players by age and then by score when the ages are identical.

Once the players are sorted, we can then use a data structure called a Binary Indexed Tree (BIT) or Fenwick Tree to efficiently calculate the final solution. BIT is usually used for range queries and updates in log(n) time, but in this scenario, it is used to find the maximum cumulative score while updating age-score pairs.

The core idea of the solution code is to iterate through each player, as sorted by age, and at each step, update their respective position in the BIT with their score plus the maximum score obtained from all players of a lesser or equal age. The update operation in the BIT involves setting the current index with the maximum of its current value and the new cumulative score. The query operation retrieves the maximum cumulative score up until the given index.

This will result in the BIT reflecting the maximum cumulative scores obtainable up until each age, ensuring no age-based conflicts. The final answer is obtained by querying the maximum cumulative score from the BIT which reflects the maximum overall score for the entire team.

Solution Approach

The solution uses a Binary Indexed Tree (BIT), which is a data structure that helps with range sum queries and updates in logarithmic time. It allows us to efficiently keep track of the maximum score we can achieve up to a certain age without having to compare each player with every other player.

The approach requires sorting the players first. This sorting is not just by age or score, but it's a composite sort: primarily by age, and secondarily by score within the same age group. This ensures that whenever we are processing a player, all potential team members that won't cause conflicts due to age have already been considered.

The `BinaryIndexedTree` class provides two main methods: `update` and `query`. The `update` method is used to update the BIT with the maximum score for a given age while the `query` method is used to retrieve the maximum score up to a certain age.

Here's a step-by-step breakdown of the implementation steps:

1. Create an instance of the `BinaryIndexedTree` class, called `tree`, with a size that is equal to the maximum age of the players.
2. Sort the players by their ages and scores as explained before.
3. Iterate through each player in the sorted order and perform the following actions:
   - Query the current maximum score we can get with a team of the current player's age or younger using `tree.query(age)`.
   - Calculate the new score by adding the current player's score to this queried score.
   - Update the BIT at the index corresponding to the player's age with the new score if it's greater than what's currently stored there.

The final maximum team score is found by querying the BIT for the maximum cumulative score after we have iterated through all players.

Each update and query operation in the BIT runs in O(log(m)), where `m` is the maximum age. Since each player is processed exactly once, and assuming that sorting the players takes O(n log n), where `n` is the number of players, the total time complexity of the solution is O(n log n + n log(m)).

The usage of BIT in this problem is akin to the dynamic programming approach for calculating LIS, except that it's optimized with a tree structure for faster updates and queries. The overall method finds an optimized subset of players that adheres to the non-conflict criteria and sums up to the largest possible team score.

Example Walkthrough

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

Suppose we are given two lists as input:

- `scores = [4, 3, 5, 7, 2]`
- `ages = [23, 24, 22, 22, 25]`

The objective is to create a team with maximum total score without any conflicts regarding the ages and scores.

1. **Sort Players**: The first step is to sort the players by their ages, and scores if the ages are equal. After sorting, our lists will look like:
   - Ages Sorted: `ages = [22, 22, 23, 24, 25]`
   - Scores Sorted with Ages: `scores = [5, 7, 4, 3, 2]`
2. **Binary Indexed Tree (BIT) Initialization**: We initialize a BIT based on the maximum age which is 25. This means that the BIT will have indices ranging from 1 to 25 (the value at index 0 is not used).
3. **Iterate and Update BIT**: The third step is to iterate over the sorted players and use their scores to update the BIT.
   - For the player with age 22 and score 5, the tree does not have any score yet, so we update index 22 with 5.
   - For the next player with age 22 and score 7, we query the BIT at index 22, which is 5, add the player's score to get 12, and update the BIT at index 22 with 12 because it's higher than the existing score.
   - For the player with age 23, we query BIT at index 23. As there are no players with age 23 or less with scores in the BIT, the maximum score is zero, so we add this player's score to 0, getting 4, and update the tree at index 23 with 4.
   - For the player with age 24, we query the BIT for the maximum score at index 24, which would be the maximum we updated for age 22 or 23, which is 12. We add the current player's score to 12, getting 15, and update the BIT at index 24 if 15 is higher than what's already there.
   - For the last player with age 25, the process is the same, fetching the maximum score up to age 25 (which is still 15), adding the player's score of 2 to get 17, and updating the BIT with 17 at index 25.
4. **Retrieve Maximum Score**: The final maximum score is the maximum value in the BIT at the end of processing, which is 17 at index 25.

Throughout the process, we have ensured no conflicts arose due to age, as all updates to a certain age index only consider scores from the same age or younger. The running time for this process is efficient because each update and query on the BIT happens in logarithmic time and we update each player exactly once.

Python Solution

```python
class BinaryIndexedTree:
    # Initialize the BinaryIndexedTree with a given size.
    def __init__(self, size):
        self.size = size
        self.tree = [0] * (size + 1)

    # Update the BinaryIndexedTree by setting the value at index x to the maximum of the current value and val.
    # Progress upward through the tree by jumping from one index to another by utilizing the least significant bit (LSB).
    def update(self, index, val):
        while index <= self.size:
            self.tree[index] = max(self.tree[index], val)
            index += index & -index

    # Query the maximum value in the BinaryIndexedTree from index 1 to x.
    # Start at index x and move downward through the tree by subtracting the least significant bit (LSB).
    def query(self, index):
        max_val = 0
        while index > 0:
            max_val = max(max_val, self.tree[index])
            index -= index & -index
        return max_val


class Solution:
    # Calculate the best team score given the scores and ages of the players.
    # The function sorts the players by their scores and ages, then utilizes a BinaryIndexedTree to find the optimal score.
    def bestTeamScore(self, scores: List[int], ages: List[int]) -> int:
        max_age = max(ages)  # Find the maximum age.
        tree = BinaryIndexedTree(max_age)

        # Pair each player's score with their age, and sort the list of pairs.
        player_info = sorted(zip(scores, ages))

        # Iterate over the sorted player_info.
        for score, age in player_info:
            # Update the BinaryIndexedTree:
            # for each player, find the best previous score including players with equal or lesser age
            # and update the tree with the new score if it's higher.
            tree.update(age, score + tree.query(age))

        # Query the BinaryIndexedTree for the maximum score possible with the given conditions.
        return tree.query(max_age)
```

Please note, the type hint `List[int]` should be imported from the `typing` module for the code to work correctly. You can add the following import statement at the beginning of the code:

```python
from typing import List

```

Java Solution

```java
class BinaryIndexedTree {
    private int size; // The number of elements
    private int[] tree; // Binary Indexed Tree array

    // Constructor to initialize the tree array based on the given size
    public BinaryIndexedTree(int size) {
        this.size = size;
        this.tree = new int[size + 1];
    }

    // Update the tree with a given value at a specified index
    public void update(int index, int value) {
        while (index <= size) {
            // Store the maximum value for the current index
            tree[index] = Math.max(tree[index], value);
            // Move to the next index to update the tree
            index += index & -index;
        }
    }

    // Query the tree for the maximum value up to a given index
    public int query(int index) {
        int max = 0; // Initialize the maximum value
        while (index > 0) {
            // Compare and get the maximum value encountered
            max = Math.max(max, tree[index]);
            // Move to the previous index to continue the query
            index -= index & -index;
        }
        return max; // Return the maximum value found
    }
}

class Solution {
    public int bestTeamScore(int[] scores, int[] ages) {
        int playerCount = ages.length; // Number of players
        // Array to store the pairs of score and age
        int[][] playerInfo = new int[playerCount][2];
        for (int i = 0; i < playerCount; ++i) {
            playerInfo[i] = new int[] {scores[i], ages[i]};
        }
        // Sort the player information based on scores, and then by age if scores are the same
        Arrays.sort(playerInfo, (a, b) -> a[0] == b[0] ? a[1] - b[1] : a[0] - b[0]);

        int maxAge = 0; // Variable to store the maximum age
        // Find the maximum age among all players
        for (int age : ages) {
            maxAge = Math.max(maxAge, age);
        }
        // Initialize the Binary Indexed Tree with the maximum age
        BinaryIndexedTree tree = new BinaryIndexedTree(maxAge);
        // Fill the tree with the scores using the ages as indices
        for (int[] player : playerInfo) {
            int age = player[1];
            int score = player[0];
            // Update the tree: maximum score for the age considering the current score and previous scores
            tree.update(age, score + tree.query(age));
        }
        // Query the tree to find the best team score
        return tree.query(maxAge);
    }
}

```

C++ Solution

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

using namespace std;

// BinaryIndexedTree (Fenwick Tree) class for efficient updates and queries on prefix maximums.
class BinaryIndexedTree {
public:
    // Constructor initializes the tree with given size.
    explicit BinaryIndexedTree(int size)
        : size(size),
        tree(size + 1) {}

    // Update method to set the maximum value at a specific position.
    void update(int index, int value) {
        while (index <= size) {
            // Assign the maximum value at the current index.
            tree[index] = max(tree[index], value);
            // Move to the next index to update.
            index += index & -index;
        }
    }

    // Query method to find the maximum value up to a specific position.
    int query(int index) {
        int maximumValue = 0;
        while (index > 0) {
            // Get the maximum value encountered so far.
            maximumValue = max(maximumValue, tree[index]);
            // Move to the previous index to continue the query.
            index -= index & -index;
        }
        return maximumValue;
    }

private:
    int size; // The size of the tree.
    vector<int> tree; // The underlying container for the tree.
};

// Solution class to solve the problem.
class Solution {
public:
    // Method to calculate the best team score given players' scores and ages.
    int bestTeamScore(vector<int>& scores, vector<int>& ages) {
        int n = ages.size(); // Number of players.
        // Create a vector to store players' scores and ages together.
        vector<pair<int, int>> players(n);
        for (int i = 0; i < n; ++i) {
            players[i] = {scores[i], ages[i]};
        }
        // Sort the players primarily by score and secondarily by age.
        sort(players.begin(), players.end());
        // Find the maximum age among all players.
        int maxAge = *max_element(ages.begin(), ages.end());
        // Create a Binary Indexed Tree with the maximum age as size.
        BinaryIndexedTree tree(maxAge);
        // Iterating over each player to populate the tree.
        for (auto& [score, age] : players) {
            // Update the tree by adding the player's score to the maximum score of the previous age.
            tree.update(age, score + tree.query(age));
        }
        // Query the tree for the maximum score up to the maximum age.
        return tree.query(maxAge);
    }
};

```

Typescript Solution

```typescript
function bestTeamScore(scores: number[], ages: number[]): number {
    // Combine ages and scores into a single array of tuples
    const ageScorePairs = ages.map((age, index) => [age, scores[index]]);

    // Sort the array of tuples by age, and then by score if ages are equal
    ageScorePairs.sort((a, b) => (a[0] === b[0] ? a[1] - b[1] : a[0] - b[0]));

    const teamSize = ageScorePairs.length;
    // Initialize an array to store the maximum score at each index
    const dp = new Array(teamSize).fill(0);

    // Iterate over the sorted age-scores pairs
    for (let i = 0; i < teamSize; ++i) {
        // Compare with each previous player
        for (let j = 0; j < i; ++j) {
            // If the current player's score is greater or equal,
            // update the dp array with the maximum score found so far
            if (ageScorePairs[i][1] >= ageScorePairs[j][1]) {
                dp[i] = Math.max(dp[i], dp[j]);
            }
        }
        // Add the current player's score to the maximum score at the current index
        dp[i] += ageScorePairs[i][1];
    }

    // Return the maximum score from the dp array
    return Math.max(...dp);
}

```

Time and Space Complexity

Time Complexity:

The time complexity of the `bestTeamScore` function is determined by the iteration over the sorted list of player scores and ages, and operations using the Binary Indexed Tree (BIT).

1. Sorting the `zip(scores, ages)`, which has a time complexity of `O(N log N)` where `N` is the number of players.
2. Iterating over the sorted list and performing `update` and `query` operations on the BIT for each player. Both `update` and `query` methods consist of a while loop that performs at most `log M` operations, where `M` is the maximum age. Because each player invokes one `update` and one `query` operation, the total number of operations is `N * log M`.

Combining these two steps, the overall time complexity is `O(N log N) + O(N log M)`. Since the age can be considered constant for this question, it simplifies to `O(N log N)`.

Space Complexity:

The space complexity is determined by the space needed to store the BIT and the space required for sorting.

1. The Binary Indexed Tree occupies a space of `O(M)` where `M` is the maximum age.
2. Sorting requires a space of `O(N)` to store the sorted pairs.

While sorting contributes `O(N)`, the BIT contributes `O(M)`, so the overall space complexity is `O(N + M)` where `N` is the number of players and `M` is the maximum possible age.