1900. The Earliest and Latest Rounds Where Players Compete

Hard · Memoization · Dynamic Programming · Leetcode Link

Problem Description

In this tournament, `n` players are lined up in a single row, and they are numbered from `1` to `n`. The players compete in rounds where the `i`th player from the front of the row competes with the `i`th player from the end of the row. The winner goes on to the next round. If there's an odd number of players in a round, the middle player automatically advances without competing.

The objective of the problem is to determine, given two specific players (`firstPlayer` and `secondPlayer`), the earliest and latest round in which these two players could potentially compete against each other. It is important to note that `firstPlayer` and `secondPlayer` are the strongest and will win against any other player. When other players compete, the outcome of who wins can be decided by us in order to find out the earliest and latest round these two specific players could compete.

Intuition

The intuition behind finding the earliest and latest rounds `firstPlayer` and `secondPlayer` can compete is to simulate the tournament while making decisions that would either hasten or delay their match.

Since we can control the outcome of matches between other players, we use this to our advantage to find extremes. For the earliest, we try to eliminate players in such a way that our `firstPlayer` and `secondPlayer` meet as soon as possible. For the latest, we do the opposite, we pick winners such that `firstPlayer` and `secondPlayer` can avoid meeting until it's inevitable.

The simulation uses dynamic programming (DP) to keep track of the earliest and latest rounds given different conditions. The DP state is defined by the position of `firstPlayer` from the front, the position of `secondPlayer` from the end, and the total number of players `k` in the current round. Therefore, `dp(l, r, k)` keeps track of the earliest and latest rounds they can meet given those conditions.

The solution uses recursion with memoization (`functools.lru_cache`) to ensure that overlapping subproblems are calculated only once, hence optimizing the overall run time of our DP approach.

The base cases are clear: if `l == r`, it means the `firstPlayer` and `secondPlayer` are facing each other, and the round numbers for both earliest and latest would be `1`. Alternatively, if `l > r`, we swap the values to keep positions consistent with our definition.

The recursive step involves iterating through all possible positions the players could end up in after the round (`i` and `j`) and keeping track of the minimum and maximum rounds for all these scenarios. We also constrain our iterations based on the number of players left (`k`).

In the end, `dp(firstPlayer, n - secondPlayer + 1, n)` gives us the earliest and latest rounds `firstPlayer` and `secondPlayer` can meet.

Solution Approach

The solution is implemented in Python using a recursive function with memoization. The algorithm utilizes dynamic programming (DP) to store the outcome of previously computed scenarios, thus avoiding redundant calculations. Here is how each part of the solution contributes to the overall approach:

- **Dynamic Programming (DP)**: The function `dp(l: int, r: int, k: int) -> List[int]` represents the DP state, where `l` is the position of `firstPlayer` from the front, `r` is the position of `secondPlayer` from the back, and `k` is the total number of players left. The function returns the earliest and latest rounds in which `firstPlayer` and `secondPlayer` can compete.
- **Memoization**: Implemented using the `functools.lru_cache` decorator, which caches the results of each unique call to `dp` based on the parameter values to eliminate redundant calculations.
- **Recursive Cases**: These are handled through the nested `for` loops where `i` and `j` loop through possible positions of `firstPlayer` and `secondPlayer` after each round. The if-statement ensures that only feasible matchups are considered. The recursive call `dp(i, j, (k + 1) // 2)` computes the result for the next round considering the current scenario.
- **Constraints on Positions**: The check `not l + r - k // 2 <= i + j <= (k + 1) // 2` enforces that players cannot jump over the middle. It ensures that the possible positions of the players adhere to the rules of the tournament.
- **Updating Earliest and Latest Rounds**: Temporary variables `a` and `b` hold the minimum and maximum round numbers respectively, during iteration. With each valid scenario, we update them to reflect the earliest and latest possible competition rounds for `firstPlayer` and `secondPlayer`.
- **Base Cases**: When `l == r`, it indicates a direct match between `firstPlayer` and `secondPlayer`, and both earliest and latest rounds are `1`. When `l > r`, the positions are swapped for consistency.
- **Returning the Result**: The initial call to the DP function `dp(firstPlayer, n - secondPlayer + 1, n)` provides the final answer, which takes into account the total number of players `n` and the positions of `firstPlayer` and `secondPlayer`.

The combination of these approaches yields an optimized algorithm that computes the desired range of rounds efficiently through recursive exploration of the tournament bracket while maintaining the logical constraints of the problem.

Example Walkthrough

Let's walk through a small example. Suppose there are `n = 4` players: 1, 2, 3, 4. We want to find out the earliest and latest rounds in which player 1 (`firstPlayer`) and player 4 (`secondPlayer`) could meet, assuming the strongest players always win against any other player.

Initially, the players are lined up as 1, 2, 3, 4.

Round 1

- Matches: (1 vs. 4) and (2 vs. 3).
- Possible Outcomes:
  - First match: Player 1 wins (since player 1 is `firstPlayer`), player 4 wins (since player 4 is `secondPlayer`).
  - Second match: We can decide the outcome since these are not the players we are tracking. Let's assume player 2 wins for the earliest possible match and player 3 wins for the latest possible match.

By selecting player 2 in the first scenario, we are setting up direct competition between 1 and 2 in the next round, whereas selecting player 3 will delay the match between player 1 and 4.

Earliest Scenario

- Remaining players for the next round: 1, 2.
- Competition: (1 vs. 2).
- Since player 1 (`firstPlayer`) is the strongest, they win and this would be the earliest round (Round 2) that player 1 could potentially face player 4.

Latest Scenario

- Remaining players for next round: 1, 3.
- Competition: Since there's an odd number of players, player 1 advances without competing.
- In the following round, the remaining players would be 1 and 4, and they would compete then.
- This would be the latest round (Round 3) that player 1 could potentially face player 4.

Application of the Solution Approach

In this example, we applied the solution approach as follows:

- We invoked the `dp` function to calculate the earliest and latest rounds for players `1` (firstPlayer) and `3` (secondPlayer), adjusted for the number of players counting from the back, which results in `dp(1, 4 - 4 + 1, 4)`.
- The DP and memoization help us avoid recalculating the same scenario twice.
- The base cases ensure that if `firstPlayer` (`l`) and `secondPlayer` (`r`) meet (`l == r`), we return `1` for both earliest and latest rounds.
- The recursive calls and updates to `a` and `b` are used to determine the rounds while adhering to the rules of the tournament.
- Eventually, we found that the earliest round they could meet was Round 2 and the latest was Round 3.

This walkthrough illustrates how dynamic programming with memoization, coupled with logical rules of the tournament, can be used to find the earliest and latest possible rounds for any two given players to compete.

Python Solution

```python
from typing import List
import functools
import math

class Solution:
    def earliestAndLatest(self, n: int, firstPlayer: int, secondPlayer: int) -> List[int]:
        # Define the memoization (dynamic programming) function with caching
        @functools.lru_cache(None)
        def dp(left_index: int, right_index: int, num_players: int) -> List[int]:
            # left_index is the position of the first player from the start (front),
            # right_index is the position of the second player from the end,
            # num_players is the total number of players in the current round.

            # Base case: if both players are about to meet
            if left_index == right_index:
                return [1, 1]

            # Ensure that left_index is always before right_index
            if left_index > right_index:
                return dp(right_index, left_index, num_players)

            # Initialize earliest and latest rounds as infinity
            # and negative infinity, respectively
            earliest_round = math.inf
            latest_round = -math.inf

            # Enumerate all possible new positions for players after matching
            for i in range(1, left_index + 1):
                for j in range(left_index - i + 1, right_index - i + 1):
                    # Ignore invalid matchings
                    if not left_index + right_index - num_players // 2 <= i + j <= (num_players + 1) // 2:
                        continue
                    # Recurse to find the new earliest and latest rounds
                    new_earliest, new_latest = dp(i, j, (num_players + 1) // 2)
                    earliest_round = min(earliest_round, new_earliest + 1)
                    latest_round = max(latest_round, new_latest + 1)

            return [earliest_round, latest_round]

        # Adjust indices based on player positions and invoke dp
        return dp(firstPlayer, n - secondPlayer + 1, n)

```

Java Solution

```java
import java.util.*;

public class Solution {
    private Map<String, int[]> memo = new HashMap<String, int[]>();

    public int[] earliestAndLatest(int n, int firstPlayer, int secondPlayer) {
        return dp(firstPlayer, n - secondPlayer + 1, n);
    }

    private int[] dp(int leftIndex, int rightIndex, int numPlayers) {
        // Ensure that leftIndex is always before rightIndex for consistency
        if (leftIndex > rightIndex) {
            return dp(rightIndex, leftIndex, numPlayers);
        }

        // Convert the state to a string to use as a key for memorization
        String key = leftIndex + "," + rightIndex + "," + numPlayers;
        if (memo.containsKey(key)) {
            return memo.get(key);
        }

        // If both players are about to meet, this is the end case
        if (leftIndex == rightIndex) {
            return new int[]{1, 1};
        }

        // Initialize earliest and latest rounds
        int earliestRound = Integer.MAX_VALUE;
        int latestRound = -1;

        // Iterate over all possible new positions for players after matching
        for (int i = 1; i <= leftIndex; ++i) {
            for (int j = leftIndex - i + 1; j <= rightIndex - i + 1; ++j) {
                // Skip invalid matchings
                if (!(leftIndex + rightIndex - numPlayers / 2 <= i + j && i + j <= (numPlayers + 1) / 2)) {
                    continue;
                }
                // Recursive call to find the new earliest and latest rounds
                int[] rounds = dp(i, j, (numPlayers + 1) / 2);
                earliestRound = Math.min(earliestRound, rounds[0] + 1);
                latestRound = Math.max(latestRound, rounds[1] + 1);
            }
        }

        // Memorize the calculated rounds
        memo.put(key, new int[]{earliestRound, latestRound});

        return memo.get(key);
    }
}

```

C++ Solution

```cpp
#include<vector>
#include<functional>
#include<climits>

class Solution {
public:
    std::vector<int> earliestAndLatest(int n, int firstPlayer, int secondPlayer) {
        // Use a lambda function to replace the functools.lru_cache in Python.
        // The lambda function will serve as the memoization function with caching.
        std::function<std::vector<int>(int, int, int)> dp;
        dp = [&](int leftIndex, int rightIndex, int numPlayers) -> std::vector<int> {
            // If both players are about to meet, return [1, 1] because both earliest and latest are at this round.
            if (leftIndex == rightIndex) {
                return {1, 1};
            }

            // Ensure that leftIndex is always before rightIndex.
            if (leftIndex > rightIndex) {
                return dp(rightIndex, leftIndex, numPlayers);
            }

            // Initialize earliest and latest rounds.
            int earliestRound = INT_MAX;
            int latestRound = INT_MIN;

            // Enumerate all possible new positions for players after matching.
            for (int i = 1; i <= leftIndex; ++i) {
                for (int j = leftIndex - i + 1; j <= rightIndex - i + 1; ++j) {
                    // Ignore invalid matchings.
                    if (!(leftIndex + rightIndex - numPlayers / 2 <= i + j && i + j <= (numPlayers + 1) / 2)) {
                        continue;
                    }
                    // Recurse to find the new earliest and latest rounds.
                    std::vector<int> result = dp(i, j, (numPlayers + 1) / 2);
                    int newEarliest = result[0];
                    int newLatest = result[1];
                    earliestRound = std::min(earliestRound, newEarliest + 1);
                    latestRound = std::max(latestRound, newLatest + 1);
                }
            }

            return {earliestRound, latestRound};
        };

        // Adjust indices based on player positions (firstPlayer stays the same, secondPlayer is adjusted from the end)
        // and invoke the dp (dynamic programming) function with caching.
        return dp(firstPlayer, n - secondPlayer + 1, n);
    }
};

```

Typescript Solution

```typescript
type RoundResults = [number, number]; // Tuple type for holding the earliest and latest rounds.

// Decorator-like function to simulate memoization in TypeScript (Python's functools.lru_cache equivalent).
function memoize(fn: (...args: any[]) => RoundResults): (...args: any[]) => RoundResults {
  const cache: Map<string, RoundResults> = new Map();
  return function(...args: any[]): RoundResults {
    const key = JSON.stringify(args); // Serialize arguments array to use as a cache key.
    if (cache.has(key)) return cache.get(key)!;
    const result = fn(...args);
    cache.set(key, result);
    return result;
  };
}

// Global recursive function for dynamic programming with memoization.
const dp: (...args: any[]) => RoundResults = memoize(function (leftIndex: number, rightIndex: number, numPlayers: number): RoundRes
  // Base case: if both players are about to meet or have already met.
  if (leftIndex >= rightIndex) return [1, 1];

  let earliestRound: number = Infinity;
  let latestRound: number = -Infinity;

  // Iterate over all possible new positions for players after matching.
  for (let i = 1; i <= leftIndex; i++) {
    for (let j = leftIndex - i + 1; j <= rightIndex - i; j++) {
      // Calculate the sum of the new positions.
      let sum = i + j;

      // Ignore invalid matchings based on the constraints.
      if (!(leftIndex + rightIndex - Math.floor(numPlayers / 2) <= sum && sum <= Math.floor((numPlayers + 1) / 2))) continue;

      let [newEarliest, newLatest] = dp(i, j, Math.floor((numPlayers + 1) / 2));
      earliestRound = Math.min(earliestRound, newEarliest + 1);
      latestRound = Math.max(latestRound, newLatest + 1);
    }
  }

  return [earliestRound, latestRound];
});

// Global function to find the earliest and latest rounds where two players could meet.
function earliestAndLatest(n: number, firstPlayer: number, secondPlayer: number): RoundResults {
  // Adjust indices based on player positions and invoke the memoized dynamic programming function.
  return dp(firstPlayer, n - secondPlayer + 1, n);
}

```

Time and Space Complexity

The given code is a dynamic programming solution that calculates the possible rounds in which two players would meet in a tournament bracket. The state of the dynamic programming `dp[i][j][k]` maintains the earliest and latest rounds where the `i-th` player from the front can meet the `j-th` player from the back out of `k` total players.

Time Complexity

To analyze the time complexity, we should consider three parameters `l`, `r`, and `k` corresponding to the first player's position, the second player's position from the end, and the total number of players, respectively.

The maximum range of `l` and `r` is `n` since `n` is the total number of players. The range of `k` is also `O(n)` because, in the worst case, `k` takes on half the previous value, leading to a logarithmic number of states in terms of `k`.

The nested loops in the code iterate and process each combination of `l` and `r` within their bounds, so in the worst case the innermost statements will run `O(l * r)` times for a particular value of `k`.

Considering the recursive nature, `dp(i, j, (k + 1) // 2)` will be invoked multiple times, but memoization using `functools.lru_cache(None)` ensures that each state is computed only once.

Therefore, the overall time complexity is `O(n^3 log n)`, since the range of `k` contributes a logarithmic factor due to the halving in each recursive step, and `l` and `r` contribute a factor of `n^2` due to the nested loops.

Space Complexity

The space complexity is determined by the number of items that can be stored in the cache (memoization) and the recursion stack depth.

The cache size corresponds to the number of distinct states that the dynamic programming algorithm must keep track of, which involves the ranges of `l`, `r`, and `k`. As concluded before, `l` and `r` have a maximum range of `n`, and `k` can take on `O(log n)` different values.

Hence, the space complexity of the cache is `O(n^2 log n)`.

The recursion stack depth will be proportional to the number of recursive calls we can have in the worst case, which is `O(log n)` since in each recursive call `k` is approximately halved until it reaches the base case.

In conclusion, the overall space complexity of the solution is `O(n^2 log n)` when considering the cache size as the dominant factor.