1406. Stone Game III

Hard · Array · Math · Dynamic Programming · Game Theory   Leetcode Link

Problem Description

Alice and Bob are playing a game with an arrangement of stones in a row. Each stone has a value, and these values are listed in an array called `stoneValue`. The players take turns, with Alice going first, and on each player's turn, they may take 1, 2, or 3 stones from the beginning of the remaining row of stones.

The score for a player increases by the value of the stones they take, and everyone starts with a score of 0. The goal is to finish the game with the highest score, and it is possible that the game ends in a tie if both players have the same score when all the stones have been taken.

The key part of the problem is that both players will play optimally, meaning they will make the best possible moves to maximize their own score.

The objective is to determine the winner of the game or to find out if it will end in a tie, based on the values of the stones.

Intuition

To solve this problem, we need to use dynamic programming, as we're looking for the best strategy over several turns, considering the impact of each possible move.

The intuition behind the solution is to look at each position in the array and decide the best move for the player whose turn it is. Since each player is trying to maximize their own score, they should be looking to maximize their current score minus whatever score the opponent would get after that move. This is because while a player aims to increase their score, they should also try to minimize the future opportunities for the opponent.

The dynamic programming function, here defined as `dfs(i)`, returns the maximum score difference (player score - opponent score) from the `i`th stone to the end of the array. When it's the current player's turn, they look ahead at up to 3 moves and calculate the score difference they would get if the next player played optimally from that point. We recursively calculate `dfs(i)` for the possible moves and choose the one with the maximum score difference.

Since Python's `@cache` decorator is being used, computations for each position are remembered and not recalculated multiple times, which greatly improves the efficiency.

After performing the DFS, the final answer is compared against 0. If the final score difference resulting from `dfs(0)` (starting at the first stone) is zero, it is a 'Tie.' If greater than zero, the current player (Alice, since she starts) wins, and if it is less than zero, it means the second player (Bob) wins.

Solution Approach

The solution is implemented using dynamic programming, one of the most powerful algorithms in solving optimization problems, especially those involving the best strategy to play a game. In this case, memoization via Python's `@cache` decorator is used to store the results of subproblems.

The `dfs(i)` function defined within the `stoneGameIII` method is the core of the solution, which represents the best score difference that can be obtained starting with the `i`th stone.

Here's a breakdown of how the `dfs(i)` function operates:

- It takes a parameter `i` which represents the index of the starting stone in the `stoneValue` list.
- The function is initially called with `i = 0` as we start evaluating from the first stone.
- A check is made to see if `i` is beyond the last stone in the array, in which case the function returns `0`, indicating no more scores can be obtained.
- The function maintains two variables: `ans`, which will store the maximum score difference possible from this position, and `s`, which is the cumulative sum of stone values that have been picked.
- A loop explores taking 1, 2, or 3 stones from the current position by incrementing `j`. The constraint is to break out of the loop if the end of the stones array is reached.
- For each possible move (taking `j` stones), we calculate the sum `s` of values of stones taken, and we calculate a potential answer as `s - dfs(i + j + 1)`. This calculation ensures we consider the opponent's optimal play after our move.
- We update `ans` to be the maximum of itself or the newly calculated potential answer.
- Once we loop through all possible moves, we return the maximum answer.

The memoization ensures that we don't compute the same state multiple times, reducing the problem to linear complexity.

Finally, when the `dfs(0)` is called, we check if the result is `0`, indicating a tie. If it's positive, Alice wins because it means she can have a greater score difference by playing optimally. If it's negative, Bob wins for the opposite reason.

The data structure used here is essentially the list called `stoneValue`. The `@cache` decorator creates an implicit hashtable-like structure to remember previously calculated results of the `dfs` function.

This approach uses the concept of minimax, which is widely used in decision-making and turn-based games, to always make the best move considering the opponent will also play optimally.

Example Walkthrough

Let's illustrate the solution approach with an example. Suppose we have the `stoneValue` array as follows: `[1,2,3,7]`.

Alice and Bob will play optimally, taking turns starting with Alice. Let's see how the `dfs` function would be used to determine the game's outcome.

1. Initially, `dfs(0)` is called since Alice starts with the first stone.
2. At `i = 0`, there are three choices for Alice:
   - Take `stoneValue[0]` which is `1`, and then `dfs(1)` will be called for Bob.
   - Take `stoneValue[0]` and `stoneValue[1]` which sums to `3`, and then `dfs(2)` will be called for Bob.
   - Take `stoneValue[0]`, `stoneValue[1]`, and `stoneValue[2]` which sums to `6`, and then `dfs(3)` will be called for Bob.
3. `dfs(1)` will be Bob's turn with remaining stones `[2,3,7]`. Bob has the same choice to take 1, 2, or 3 stones, and after each choice, a new `dfs` with the appropriate index would be called for Alice.
4. This process continues until `i` reaches the length of the `stoneValue` array, at which point the function returns `0` since no stones are left to take.
5. During each call, `dfs` calculates the maximum score difference Alice or Bob can have at that point. For instance, if Alice takes one stone, then Bob can optimally choose the best outcome from the remaining stones, and the score difference is updated accordingly.
6. Eventually, all possible options and their score differences are computed, and `dfs(i)` will return the optimal score difference the current player can achieve from `i` to the end of the array.

The game would unfold as follows, showing the choices and corresponding `dfs` calls:

- `dfs(0)`:
  - Choice 1: Take `1`. `dfs(1)` (Bob's turn) is now evaluated.
  - Choice 2: Take `1 + 2 = 3`. `dfs(2)` (Bob's turn) is now evaluated.
  - Choice 3: Take `1 + 2 + 3 = 6`. `dfs(3)` (Bob's turn) is now evaluated.
- `dfs(1)` (Bob's turn with `[2,3,7]`):
  - Bob will look ahead and follow similar steps, aiming to minimize Alice's score following his moves.

After evaluating all the possibilities, `dfs(0)` will yield the best score difference Alice can achieve by taking stones optimally from the beginning of the array. If the score difference is:

- Greater than 0: Alice wins.
- Less than 0: Bob wins.
- Exactly 0: The game results in a tie.

For this example, Alice can secure a win by taking the first stone (with a value of 1), because now the score difference (`dfs(0)`) can be maximized as follows:

- Alice takes `1`, and `dfs(1)` is called.
- If Bob takes `2`, Alice can take `3` and `7` and wins. If Bob takes `2` and `3`, Alice takes `7` and still wins. Hence Bob will choose the option that minimizes loss, which might be taking `2`, making the sequence of plays `[1], [2], [3,7]`, and Alice wins by 7 (Alice's total: 11 - Bob's total: 4 = 7).

Using this strategy, we understand that `dfs(0)` would return a positive score indicating Alice as the winner. Each call to `dfs` was efficiently processed due to memoization, which saved the state to prevent re-evaluation.

Python Solution

```python
from functools import lru_cache  # Import the lru_cache decorator from functools
from math import inf  # Import 'inf' from the math library to represent infinity

class Solution:
    def stoneGameIII(self, stone_values: List[int]) -> str:
        """
        Determine the result of the stone game III.

        :param stone_values: List[int] - A list of integers representing the values of stones.
        :return: str - The result of the game, either 'Alice', 'Bob', or 'Tie'.
        """
        # Define the depth-first search function with memoization
        @lru_cache(maxsize=None)
        def dfs(current_index: int) -> int:
            """
            Calculate the maximum score difference the player can obtain starting from 'current_index'.

            :param current_index: int - The current index a player is at.
            :return: int - The maximum score difference.
            """
            if current_index >= total_stones:
                return 0

            # Initialize the answer to negative infinity and a sum accumulator
            max_score_diff, accumulated_sum = -inf, 0

            # The player can pick 1, 2, or 3 stones in a move
            for j in range(3):
                # If the range exceeds the length of stone_values, stop the loop
                if current_index + j >= total_stones:
                    break
                # Accumulate the value of stones picked
                accumulated_sum += stone_values[current_index + j]
                # Calculate the max score difference recursively by subtracting the opponent's best score after the current player's
                max_score_diff = max(max_score_diff, accumulated_sum - dfs(current_index + j + 1))

            return max_score_diff

        # Get the total number of stones
        total_stones = len(stone_values)
        # Start the game from index 0 to get the overall answer
        final_score_diff = dfs(0)

        # Compare the final score difference to determine the winner
        if final_score_diff == 0:
            return 'Tie'
        elif final_score_diff > 0:
            return 'Alice'
        else:
            return 'Bob'

```

Java Solution

```java
class Solution {
    private int[] stoneValues; // An array to hold the values of the stones.
    private Integer[] memoization; // A memoization array to store results of subproblems.
    private int totalStones; // The total number of stones.

    // Determines the outcome of the stone game III.
    public String stoneGameIII(int[] stoneValues) {
        totalStones = stoneValues.length; // Initialize the total number of stones.
        this.stoneValues = stoneValues; // Set the class's stoneValues array.
        memoization = new Integer[totalStones]; // Initialize the memoization array.

        // Result of the DFS to compare against.
        int result = dfs(0);

        if (result == 0) {
            return "Tie"; // If result is zero, then the game is tied.
        }

        // If the result is positive, Alice wins; otherwise, Bob wins.
        return result > 0 ? "Alice" : "Bob";
    }

    // Depth-First Search with memoization to calculate the optimal result.
    private int dfs(int index) {
        // Base case: if the index is out of the right boundary of array.
        if (index >= totalStones) {
            return 0;
        }

        // Return the already computed result if present, avoiding redundant computation.
        if (memoization[index] != null) {
            return memoization[index];
        }

        int maxDifference = Integer.MIN_VALUE; // Initialize to the smallest possible value.
        int sum = 0; // Sum to store the total values picked up until now.

        // Try taking 1 to 3 stones starting from the current index 'i' and calculate
        // the maximum score difference taking the subproblem (i+j+1) into account.
        for (int j = 0; j < 3 && index + j < totalStones; ++j) {
            sum += stoneValues[index + j]; // Increment sum with stone value.
            // Update maxDifference with the maximum of its current value and the score
            // difference. The score difference is current sum - result of dfs(i + j + 1).
            maxDifference = Math.max(maxDifference, sum - dfs(index + j + 1));
        }

        // Store the result in memoization and return it.
        return memoization[index] = maxDifference;
    }
}

```

C++ Solution

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

using namespace std;

class Solution {
public:
    // Function to decide the winner of the stone game III
    string stoneGameIII(vector<int>& stoneValue) {
        int n = stoneValue.size(); // Get the number of stones
        vector<int> dp(n, INT_MIN); // Initialize the dynamic programming table with minimum int values

        // Recursive lambda function to perform depth-first search
        function<int(int)> dfs = [&](int index) -> int {
            if (index >= n) {
                return 0; // Base case: if we've reached or exceeded the number of stones, the score is 0
            }
            if (dp[index] != INT_MIN) {
                return dp[index]; // If the value is already computed, return it
            }
            int maxScore = INT_MIN; // Initialize the max score for the current player
            int sum = 0; // Variable to store the cumulative value of stones picked up

            // Explore upto 3 moves ahead, because a player can pick 1, 2, or 3 stones
            for (int j = 0; j < 3 && index + j < n; ++j) {
                sum += stoneValue[index + j]; // Accumulate value of stones picked
                maxScore = max(maxScore, sum - dfs(index + j + 1)); // Choose the move which gives the max score
            }
            dp[index] = maxScore; // Memoize the result for the current index
            return maxScore;
        };

        int finalScore = dfs(0); // Start the game from the first stone

        // Using the calculated final score, determine the winner or if it's a tie
        if (finalScore == 0) {
            return "Tie";
        }
        return finalScore > 0 ? "Alice" : "Bob";
    }
};

```

Typescript Solution

```typescript
// Function that determines the winner of the stone game
function stoneGameIII(stoneValues: number[]): string {
    const stoneCount = stoneValues.length;
    const INFINITY = 1 << 30; // A representation of infinity.
    const dp: number[] = new Array(stoneCount).fill(INFINITY); // DP array initialized with 'infinity' for memoization.

    // Helper function that uses Depth First Search and dynamic programming to calculate the score
    const dfs = (currentIndex: number): number => {
        if (currentIndex >= stoneCount) { // Base case: no stones left.
            return 0;
        }
        if (dp[currentIndex] !== INFINITY) { // If value already computed, return it from memoization storage.
            return dp[currentIndex];
        }

        let maxDiff = -INFINITY; // Initialize max difference as negative infinity.
        let currentSum = 0; // Holds the running total sum of stone values.
        for (let count = 0; count < 3 && currentIndex + count < stoneCount; ++count) {
            currentSum += stoneValues[currentIndex + count];
            // Recursive call to dfs for the opponent's turn.
            // Update maxDiff to be the maximum value Alice can get by considering the current pick.
            maxDiff = Math.max(maxDiff, currentSum - dfs(currentIndex + count + 1));
        }
        // Store the computed maxDiff value in the dp array (memoization)
        dp[currentIndex] = maxDiff;
        return maxDiff;
    };

    const finalScore = dfs(0); // Start the game with the first stone.

    if (finalScore === 0) {
        return 'Tie'; // If final score is 0, then the game is a tie.
    }
    // If final score is positive, Alice wins; otherwise, Bob wins.
    return finalScore > 0 ? 'Alice' : 'Bob';
}

```

Time and Space Complexity

Time Complexity

The time complexity of the function `stoneGameIII` is determined by the recursion process performed by the helper function `dfs`. In this function, the algorithm attempts to maximize the score for the current player by choosing up to three consecutive stones (which is implemented in the for loop that iterates at most three times).

The memoization (through the `@cache` decorator) stores the results of the subproblems, which are the optimal scores starting from each index `i`. There are `n` possible indices to start from (where `n` is the length of `stoneValue`), and since each function call of `dfs` examines at most 3 different scenarios before recursion, the total number of operations needed is proportional to `n`.

Hence, the time complexity is `O(n)` because each state is computed only once due to memoization, and the recursive calls made from each state are at most three.

Space Complexity

The space complexity of `stoneGameIII` is determined by two factors: the space used for recursion (the recursion depth) and the space used to store the results of subproblems (due to memoization).

1. **Recursion Depth**: Since the function `dfs` is called recursively and can potentially make up to three recursive calls for each call made, the depth of the recursion tree could theoretically go up to `n` in a worst-case scenario. However, due to memoization, many recursive calls are pruned, and thus, the true recursion depth is limited. Generally, the recursion does not go beyond `n`.

2. **Memoization Storage**: The memoization of `dfs` subproblem results is implemented through the `@cache` decorator, which could potentially store results for each of the `n` starting indices.

Combining these two factors, the space complexity is also `O(n)`, primarily due to the space needed to store the computed values for memoization for each possible starting index and the call stack for the recursive calls.