1690. Stone Game VII

Medium Array Math Dynamic Programming Game Theory

Leetcode Link

Problem Description

In this game, Alice and Bob are playing a turn-based game with an array (row) of stones. Each stone has a certain value associated with it. Alice starts first and at each turn, a player can remove either the leftmost or the rightmost stone. The score for each turn is calculated as the sum of values of the remaining stones in the array. The objective for Alice is to maximize the score difference by the end of the game, while Bob wants to minimize this difference. The final output should be the difference in scores between Alice and Bob if both play optimally. In other words, we are looking for the best strategy for both players such that Alice maximizes and Bob minimizes the end score difference, and we want to know what that difference is.

Intuition

The intuition behind solving this problem lies in recognizing it as a dynamic programming (DP) problem and understanding game theory. Given that both players are playing optimally, we need to decide at each step which choice (taking the leftmost or rightmost stone) will lead to the best possible outcome for the player whose turn it is while considering that the other player will also play optimally.

Since the score that a player gets in their turn depends not just on their immediate decision but also on the remaining array of stones, this gives us a hint that we need a way to remember outcomes for particular situations to avoid redundant calculations. This kind of optimization is where dynamic programming shines.

The solution uses depth-first search (DFS) with memoization (or DP), which essentially remembers ('caches') the results of certain calculations (here, score differences for given i and j where i and j denote the current indices of the stones in the row that players can pick from).

The caching is achieved using the `@cache` decorator, which means that when the `dfs` function is called repeatedly with the same `i, j` arguments, it will not recompute the function but instead retrieve the result from an internal cache.

The 'dfs' function is designed to select the optimal move at each step. It does this by calculating the score difference when removing a stone from the left (denoted by 'a') and from the right (denoted by 'b'), and then choosing the maximum of these two for the player whose turn it is. This ensures that each player's move – whether it's Alice maximizing the score difference or Bob minimizing it – contributes to the overall optimal strategy.

Finally, the precomputed prefix sums (`accumulate(stones, initial=0)`) help to quickly calculate the sum of the remaining stones with constant time lookups, which significantly improves the performance of the algorithm. The intuition here is to optimize the sum calculation within the row of stones, as this operation is required repeatedly for determining scores.

Solution Approach

The solution to the problem leverages dynamic programming (DP) and depth-first search (DFS) to find the optimal moves for Alice and Bob.

Here's a walk-through of the implementation steps of the solution:

1. **Memoization with `@cache`:** We begin by defining a recursive function `dfs(i, j)` which accepts two parameters indicating the indexes of the row from which the players can take stones. The `@cache` decorator is used to store the results of the function calls, ensuring that each unique state is only calculated once, thus reducing the number of repeat computations.
2. **Base Case of Recursion:** When the indices `i` and `j` cross each other (`i > j`), it means there are no stones left to be removed, and the function returns 0 as there are no more points to be scored.
3. **Recursive Case – Computing Scores:**
   - `a = s[j + 1] - s[i + 1] - dfs(i + 1, j)` calculates the score difference if the leftmost stone is taken. It computes the sum of remaining stones using prefix sums and then subtracts the optimal result of the sub-problem where the leftmost has been removed (we advance `i` by 1).
   - `b = s[j] - s[i] - dfs(i, j - 1)` does the similar calculation for the rightmost stone.
4. **Choosing the Optimal Move:** The function then returns the maximum of these two options, `max(a, b)`. For Alice, this means maximizing the score difference, while for Bob, due to the recursive nature of the algorithm, this means choosing the move that minimally increases Alice's advantage.
5. **Prefix Sums with `accumulate`:** `s = list(accumulate(stones, initial=0))` is used to create an array of prefix sums to speed up the calculation of the sum of the remaining stones.
6. **Computing the Answer:** The main function then calls `dfs(0, len(stones) - 1)` to initiate the recursive calls starting from the whole array of stones. The final answer represents the best possible outcome of the difference in scores, assuming both Alice and Bob play optimally.
7. **Clearing the Cache (Optional):** `dfs.cache_clear()` can be called to clear the cache if necessary, but it's optional and doesn't affect the output for this single-case computation.

By employing DP with memoization, the solution ensures that the computation for each sub-array of stones defined by indices `i` and `j` is only done once. Coupled with an optimally designed recursive function and the use of prefix sums for rapid summation, this approach considerably reduces the time complexity that would otherwise be exponential due to the overlapping subproblems present in naive recursion.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose we have an array `stones` of `[3, 9, 1, 2]`, where each number represents the value of a stone.

1. **Initialize Prefix Sums:** Using the `accumulate` function, we create the prefix sums array `s`, which becomes `[0, 3, 12, 13, 15]`. The `0` is an initial value for easier computation.
2. **Invoke dfs(0, 3):** We start the depth-first search with the entire array, with `i=0` and `j=3` (index of the first and last stones).
3. **Determine Score Differences:**
   - When taking the leftmost (3), `dfs(1, 3)` is called, resulting in `a = s[4] - s[2] - dfs(1, 3)` ⇒ `a = 15 - 12 - dfs(1, 3)` ⇒ `a = 3 - dfs(1, 3)`.
   - When taking the rightmost (2), `dfs(0, 2)` is called, resulting in `b = s[3] - s[1] - dfs(0, 2)` ⇒ `b = 13 - 3 - dfs(0, 2)` ⇒ `b = 10 - dfs(0, 2)`.

At this point, we need the results of `dfs(1, 3)` and `dfs(0, 2)` to continue.

4. **Recursive Calls:**
   - For `dfs(1, 3)`: Again, we decide between `dfs(2, 3)` (taking 9) and `dfs(1, 2)` (taking 2).
     - If we take 9, `dfs(2, 3)` results in `a = s[4] - s[3] - dfs(2, 3)` ⇒ `a = 15 - 13 - dfs(2, 3)` ⇒ `a = 2 - dfs(2, 3)`.
     - If we take 2, `dfs(1, 2)` results in `b = s[3] - s[2] - dfs(1, 2)` ⇒ `b = 13 - 12 - dfs(1, 2)` ⇒ `b = 1 - dfs(1, 2)`.
   - For `dfs(0, 2)`: This will follow similar steps, evaluating the removal of stones at the positions (0, 2).
5. **Base Case:** Eventually, the calls will reach a base case where `i > j` in which case `dfs` returns `0`.
6. **Memoization:** As these calls are made, the results are stored thanks to the `@cache` decorator, meaning that any repeated calculations for the same state are fetched from cache rather than recomputed.
7. **Determine Best Move at Each Step:** `dfs` function will return the maximum value of the `a` or `b` calculated at each step, providing the score difference up to that point.
8. **Trace Back to First Call:** The results of the sub-problems build upon each other to provide the result of the initial call `dfs(0, 3)`, which gives the final optimal score difference.

For our case:

- We start with the full array: `dfs(0, 3)`
- Then explore taking a stone from the left and right, recursively: `dfs(1, 3)` and `dfs(0, 2)`
- This process continues, exploring all possibilities, but efficiently with memoization.
- We'll get a sequence of decisions that will maximize the difference for Alice if she starts, and minimize it for Bob.

9. **Final Answer:** After dfs(0, 3) is fully executed with all its recursive dependencies solved, the result is then the optimal score difference with Alice starting, given that both players play optimally.

Python Solution

```python
from typing import List
from functools import lru_cache
from itertools import accumulate

class Solution:
    def stoneGameVII(self, stones: List[int]) -> int:
        # Helper function that uses dynamic programming with memoization
        @lru_cache(maxsize=None)  # Use lru_cache for memoization
        def dp(low, high):
            # Base condition: if no stones are left
            if low > high:
                return 0

            # If Alice removes the stone at the low end, the new score is computed
            # from the remaining pile (low+1 to high) and current total score
            score_when_remove_low = prefix_sums[high + 1] - prefix_sums[low + 1] - dp(low + 1, high)

            # If Alice removes the stone at the high end, the new score is computed
            # from the remaining pile (low to high-1) and current total score
            score_when_remove_high = prefix_sums[high] - prefix_sums[low] - dp(low, high - 1)

            # Return the max score Alice can achieve from the current situation
            return max(score_when_remove_low, score_when_remove_high)

        # Construct the prefix sums list where prefix_sums[i] is the sum of stones[0] to stones[i-1]
        prefix_sums = [0] + list(accumulate(stones))

        # Calculate the maximum score difference Alice can achieve by starting from the full size of the stones pile
        answer = dp(0, len(stones) - 1)

        # Clear the cache since it is no longer needed
        dp.cache_clear()

        return answer  # Return the answer

```

Java Solution

```java
class Solution {
    // Prefix sum array to efficiently calculate the score.
    private int[] prefixSum;
    // Memoization table to store results of subproblems.
    private Integer[][] memo;

    public int stoneGameVII(int[] stones) {
        int n = stones.length;
        prefixSum = new int[n + 1];
        memo = new Integer[n][n];

        // Compute prefix sums for stones to help calculate the score quickly.
        for (int i = 0; i < n; ++i) {
            prefixSum[i + 1] = prefixSum[i] + stones[i];
        }

        // Begin the game from the first stone to the last stone.
        return dfs(0, n - 1);
    }

    // Recursive function with memoization to compute the maximum score difference.
    private int dfs(int left, int right) {
        // Base case: when there are no stones, the score difference is 0.
        if (left > right) {
            return 0;
        }

        // Check if the result for this subproblem is already computed.
        if (memo[left][right] != null) {
            return memo[left][right];
        }

        // The score difference if we remove the left-most stone.
        int scoreRemoveLeft = prefixSum[right + 1] - prefixSum[left + 1] - dfs(left + 1, right);
        // The score difference if we remove the right-most stone.
        int scoreRemoveRight = prefixSum[right] - prefixSum[left] - dfs(left, right - 1);

        // The player chooses the option that maximizes the score difference.
        // The result of the subproblem is the maximum score that can be achieved.
        memo[left][right] = Math.max(scoreRemoveLeft, scoreRemoveRight);

        return memo[left][right];
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <cstring>
#include <functional>
using namespace std;

class Solution {
public:
    int stoneGameVII(vector<int>& stones) {
        int n = stones.size();
        // Define a 2D array f to memorize the results
        vector<vector<int>> dp(n, vector<int>(n, 0));

        // Define an array to store the prefix sums of the stones
        vector<int> prefixSum(n + 1, 0);
        for (int i = 0; i < n; ++i) {
            prefixSum[i + 1] = prefixSum[i] + stones[i];
        }

        // Define a recursive lambda function for depth-first search
        function<int(int, int)> dfs = [&](int left, int right) {
            // Base case: if the game is over (no stones left), the score is 0.
            if (left > right) {
                return 0;
            }
            // If we have already computed the result for this interval, return it
            if (dp[left][right] != 0) {
                return dp[left][right];
            }
            // Calculate score if removing the leftmost stone
            int scoreIfRemoveLeft = prefixSum[right + 1] - prefixSum[left + 1] - dfs(left + 1, right);
            // Calculate score if removing the rightmost stone
            int scoreIfRemoveRight = prefixSum[right] - prefixSum[left] - dfs(left, right - 1);

            // The result for the current interval is the maximum score the current player can achieve
            dp[left][right] = max(scoreIfRemoveLeft, scoreIfRemoveRight);
            return dp[left][right];
        };

        // Start the game from the full range of stones and return the maximum possible score
        return dfs(0, n - 1);
    }
};

```

Typescript Solution

```typescript
function stoneGameVII(stones: number[]): number {
    const n: number = stones.length;
    // Define a 2D array dp to memorize the results
    const dp: number[][] = Array.from({length: n}, () => Array(n).fill(0));

    // Define an array to store the prefix sums of the stones
    const prefixSum: number[] = Array(n + 1).fill(0);
    for (let i = 0; i < n; ++i) {
        prefixSum[i + 1] = prefixSum[i] + stones[i];
    }

    // Define a recursive lambda function for depth-first search
    const dfs: (left: number, right: number) => number = (left, right) => {
        // Base case: if the game is over (no stones left), the score is 0.
        if (left > right) {
            return 0;
        }
        // If we have already computed the result for this interval, return it
        if (dp[left][right] !== 0) {
            return dp[left][right];
        }
        // Calculate score if removing the leftmost stone
        const scoreIfRemoveLeft: number = prefixSum[right + 1] - prefixSum[left + 1] - dfs(left + 1, right);
        // Calculate score if removing the rightmost stone
        const scoreIfRemoveRight: number = prefixSum[right] - prefixSum[left] - dfs(left, right - 1);

        // The result for the current interval is the maximum score the current player can achieve
        dp[left][right] = Math.max(scoreIfRemoveLeft, scoreIfRemoveRight);
        return dp[left][right];
    };

    // Start the game from the full range of stones and return the maximum possible score
    return dfs(0, n - 1);
}

```

Time and Space Complexity

The time complexity of the provided code can be analyzed as follows:

- There are `n = j - i + 1` states to compute, where `n` is the total number of stones.
- For each state (i, j), we have two choices: to take the stone from the left or the right.
- Caching results of subproblems with memoization reduces repeated calculations such that each pair `(i, j)` is computed once.
- Therefore, there are `O(n^2)` states due to the combination of starting and ending positions.

Hence, the overall time complexity is `O(n^2)`.

The space complexity analysis is as follows:

- Space is used to store the results of subproblems; this uses `O(n^2)` space due to memoization.
- The array `s` has a space complexity of `O(n)`.

The memoization's space complexity is dominant. Therefore, the overall space complexity is `O(n^2)`.