1140. Stone Game II

Medium Array Math Dynamic Programming Game Theory Prefix Sum

Leetcode Link

Problem Description

Alice and Bob are playing a game with a row of piles of stones, where each pile has a positive number of stones, given by `piles[i]`. The goal is to end with the most stones. Alice starts the game and the players alternate turns. Initially, the variable $M = 1$.

On each turn, the current player can take all the stones from the first X piles, where $1 <= X <= 2M$. After choosing the number of piles to take the stones from, the player updates M to be the maximum of M and the number X chosen.

The game ends when all stones have been taken. The objective is to maximize the number of stones Alice can obtain if both players play optimally.

Intuition

To find the solution, we need to consider the game from a dynamic programming perspective, specifically using a minimax approach with memorization to handle the overlapping subproblems.

Since Alice aims to maximize the number of stones she can get, while Bob aims to minimize that same value, we can define a recursive function that returns the maximum number of stones a player can achieve from the current state, defined by the index `i` of the first available pile and the value of M.

The stopping condition for our recursive function `dfs(i, m)` is when there are $2M$ or more piles remaining starting from index `i`. In this case, the current player can take all remaining stones. Otherwise, the player iterates over all valid choices of X (from 1 to $2M$), recursively calling `dfs(i + x, max(m, x))` which represents the new state after the opponent has played their turn.

The function then chooses the maximum value from these recursive calls, which will represent the optimal decision for the current player. To optimize and avoid recalculating the same states, we use the `@cache` decorator from the `functools` module, which memorizes the results of the function calls with the given arguments.

We also utilize the `accumulate` function from the `itertools` module to create a prefix sum array, `s`, which allows us to quickly calculate the sum of stones from any range of piles during the recursive calls. The final answer for the maximum number of stones Alice can get will be the return value of `dfs(0, 1)`, which is the starting state of the game.

Solution Approach

The given Python solution uses a depth-first search (DFS) recursive strategy with memoization. The implementation comprises several key parts:

1. **Memoization**: The `@cache` decorator from `functools` is used to automatically store the results of expensive function calls and return the cached result when the same inputs occur again. This optimization reduces the computational complexity by preventing redundant calculations of subproblems.
2. **Depth-First Search (DFS) Recursive Function**: The `dfs(i, m)` function is the heart of the solution. Here, `i` represents the current index starting from which the stones can be taken, and `m` is the maximum number of piles that can be chosen based on the previous move.
3. **Prefix Sum Array**: The `accumulate` function from the `itertools` module creates a prefix sum array, `s`. It allows for constant-time computation of the sum of stones in any subarray of `piles`. The initial `0` is added to the sums to make sure that the sum from the start is easily accessible.
4. **Minimax Algorithm**: Given that Alice wants to maximize her stones and Bob wants to minimize Alice's stones, the algorithm looks at each possible choice of X (from 1 to $2M$) that the players can make. For each choice, it recursively calculates what would be the best score Alice can get if she picks X piles and then the opponent plays optimally after her. The result of this recursive call is subtracted from the total number of stones from `i` to the end (`s[n] - s[i]`) to get the sequence of optimal choices for Alice.
5. **Base Case**: If we can take all remaining piles ($x + 2 >= n - i$), we simply take all the remaining stones (`s[n] - s[i]`) without further recursion because this is the maximum we can get from this point.
6. **Calculation and Return**: In the `dfs` function, we use a generator expression inside the `max` function to iterate over all possible X within the `1` to $2M$ range and calculate the corresponding score Alice would get if the optimal choice for X is made.

By calling `dfs(0,1)` we start the DFS recursion from the beginning of the piles and with an initial M of 1, as per the game's rules. The return value of this call provides the answer to the problem, which is the maximum number of stones Alice can get if both she and Bob play optimally.

Example Walkthrough

Let's illustrate the solution approach with an example. Suppose the `piles` array is `[2, 7, 9, 4]`, representing four piles with 2, 7, 9, and 4 stones each.

We apply the depth-first search (DFS) recursive function `dfs` with memoization to compute the maximum number of stones Alice can obtain.

1. **Initial Call**: Start with `dfs(0, 1)` with an index of `0` and M of `1`.
2. **Prefix Sum Array**: First, we construct a prefix sum array `s`: `[0, 2, 9, 18, 22]`. This represents the total stones one can get by taking piles up to that index. E.g., taking the first three piles yields `s[3] - s[0]` which equals `18`.
3. **Possible Choices:** Now, Alice can take from 1 to `2 * M` piles. Since M is 1, Alice can take 1 or 2 piles.
4. **Alice's Turn**:
   - If Alice takes 1 pile, she gets 2 stones. Now Bob will use `dfs(1, max(1, 1)) = dfs(1, 1)` on the remaining `[7, 9, 4]` piles.
   - If Alice takes 2 piles (1 to $2M$), she gets 9 stones ($2 + 7$). Now Bob will use `dfs(2, max(1, 2)) = dfs(2, 2)` on the remaining `[9, 4]` piles.
5. **Bob's Turn**: Bob plays optimally to give Alice the minimum number of remaining stones.
   - From `dfs(1, 1)`: Bob follows the same logic and has the option to take 1 or 2 piles. If he takes 1 pile, Alice is left with `dfs(2, 1)` and if he takes 2 piles, Alice is left with `dfs(3, 2)`.
   - From `dfs(2, 2)`: Bob has the option to take 1, 2, 3, or 4 piles. However, only the options taking 1 or 2 piles are valid as $2M = 4$ and there are only two piles left. Both lead to Alice not being able to pick any more piles, as Bob would clear the remaining stones.
6. **Subproblem Results**: The return values of these recursive calls are cached, preventing redundant calculations. Alice's choice will be the one that maximizes her stones considering optimal plays by Bob.
7. **Calculation**:
   - `dfs(0, 1)` is the maximum of:
     - `2 + (Total stones - 2 - dfs(1, 1))`
     - `9 + (Total stones - 9 - dfs(2, 2))`
   - The recursive calls continue breaking down subsequent turns in the same manner until the game ends.
8. **Base Case**:
   - When `dfs(2, 2)` is called, M is 2 and there are only 2 piles left, so Alice can take all the remaining stones, returning `s[4] - s[2] = 22 - 18 = 4`.
   - When `dfs(3, 2)` is called, M is 2 and there is only 1 pile left, so Alice can take all the remaining stones, returning `s[4] - s[3] = 22 - 18 = 4`.

By the end of these calculations, we deduce that Alice's optimal strategy yields the most stones based on the initial call `dfs(0, 1)`. This result is the maximum number of stones Alice can collect if both players play optimally. In this case, Alice's best initial move is to take the first two piles, and she will end up with 9 stones.

Python Solution

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

class Solution:
    def stoneGameII(self, piles: List[int]) -> int:

        @lru_cache(maxsize=None)
        def dfs(current_index, max_take):
            # If we can take all remaining piles, return the sum of those piles.
            if max_take * 2 >= total_piles - current_index:
                return prefix_sums[total_piles] - prefix_sums[current_index]

            # Try every possible number of stones we can take, and choose the option
            # that maximizes our stones.
            return max(
                prefix_sums[total_piles] - prefix_sums[current_index] -
                dfs(current_index + x, max(max_take, x))
                for x in range(1, 2 * max_take + 1)
            )

        total_piles = len(piles)  # The total number of piles.
        prefix_sums = list(accumulate(piles, initial=0))  # Prefix sum array to calculate the sum efficiently.

        # Start the game from the first pile, with the initial maximum of 1 stone to take.
        return dfs(0, 1)
```

Java Solution

```java
class Solution {
    // A 2-D array to hold the prefix sum of the piles array
    private int[] prefixSums;
    // 2-D array to memoize the results of dfs(i, m)
    private Integer[][] memo;
    // Number of piles
    private int n;

    public int stoneGameII(int[] piles) {
        n = piles.length;
        prefixSums = new int[n + 1];
        memo = new Integer[n][n + 1];
        // Calculate prefix sums for the piles array for easy range sum queries
        for (int i = 0; i < n; ++i) {
            prefixSums[i + 1] = prefixSums[i] + piles[i];
        }
        // Start the game with the first pile (index 0) and initial 'M' value of 1
        return dfs(0, 1);
    }

    private int dfs(int i, int m) {
        // If the next player can take all remaining piles, return the sum of those piles
        if (i + 2 * m >= n) {
            return prefixSums[n] - prefixSums[i];
        }
        // If we have already computed this state, return the stored value
        if (memo[i][m] != null) {
            return memo[i][m];
        }
        int result = 0;
        // Try all possible x moves from the current position
        for (int x = 1; x <= 2 * m; ++x) {
            // Choose the move that maximizes the current player's score
            result = Math.max(result, prefixSums[n] - prefixSums[i] - dfs(i + x, Math.max(m, x)));
        }
        // Store the result in the memoization table before returning
        memo[i][m] = result;
        return result;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to play the stone game and determine the maximum number of stones the player can get.
    int stoneGameII(vector<int>& piles) {
        int n = piles.size();

        // Prefix sum array 's' to efficiently calculate the sum of stones in 'piles' from any index.
        vector<int> prefixSum(n + 1, 0);
        for (int i = 0; i < n; ++i) {
            prefixSum[i + 1] = prefixSum[i] + piles[i];
        }

        // Memoization array 'dp', initialized to 0. Stores the results of subproblems.
        vector<vector<int>> dp(n, vector<int>(n + 1, 0));

        // Recursive lambda function to perform a depth-first search (DFS).
        // It uses memoization to store results of subproblems.
        // 'i' denotes the current index and 'm' the current 'M' value.
        function<int(int, int)> dfs = [&](int i, int m) -> int {
            // If we're able to take all remaining stones, we'll do so.
            if (i + 2 * m >= n - i) {
                return prefixSum[n] - prefixSum[i];
            }
            // If we've already computed this subproblem, return the stored result.
            if (dp[i][m] != 0) {
                return dp[i][m];
            }

            int best = 0; // Keeping track of the best score we can achieve from this state.

            // Try taking x stones where x is from 1 to the maximum number of stones we can take.
            for (int x = 1; x <= 2 * m; ++x) {
                best = max(best, prefixSum[n] - prefixSum[i] - dfs(i + x, max(x, m)));
            }

            // Store the computed result in 'dp' before returning.
            dp[i][m] = best;
            return best;
        };

        // Initial call to the DFS function starting from index 0 and M value 1.
        return dfs(0, 1);
    }
};
```

Typescript Solution

```typescript
function stoneGameII(piles: number[]): number {
    const pileCount = piles.length;

    // F is a memoization table where F[i][m] represents the maximum number of stones
    // a player can get when starting from the i-th pile with M = m.
    const memoTable = Array.from({ length: pileCount }, () => new Array(pileCount + 1).fill(0));

    // S is a prefix sum array where s[i] represents the total number of stones
    // in the first i piles.
    const prefixSum = new Array(pileCount + 1).fill(0);
    for (let i = 0; i < pileCount; ++i) {
        prefixSum[i + 1] = prefixSum[i] + piles[i];
    }

    // Define depth-first search function to determine the maximum stones
    // that can be taken starting from the i-th pile with m as the current M value
    const dfs = (currentIndex: number, currentM: number): number => {
        // If the current player can take all remaining piles, return the sum directly
        if (currentM * 2 >= pileCount - currentIndex) {
            return prefixSum[pileCount] - prefixSum[currentIndex];
        }

        // Use memoization to avoid re-calculating states
        if (memoTable[currentIndex][currentM]) {
            return memoTable[currentIndex][currentM];
        }

        let maxStones = 0;
        // Try taking x piles where x is from 1 to M*2 and maximize the stones
        for (let x = 1; x <= currentM * 2; ++x) {
            maxStones = Math.max(maxStones, prefixSum[pileCount] - prefixSum[currentIndex] - dfs(currentIndex + x, Math.max(currentM, x)));
        }
        // Store and return the result in memo table
        memoTable[currentIndex][currentM] = maxStones;
        return maxStones;
    };

    return dfs(0, 1);
}
```

Time and Space Complexity

The time complexity of the code is $O(n^3)$. This is because in the function `dfs(i, m)`, we iterate up to $2m - 1$ times where m can go up to $n/2$ in the worst case. For each iteration, we have a recursive call, which can lead to a maximum of n levels of recursion (since i can range from 0 to $n - 1$), and the update of m in the recursive calls could happen n times as well. Since we are memoizing the results of the recursive calls, each state (i, m) is only computed once. The number of unique states for i is n and for m is also up to n, this results in n^2 unique states. Thus the overall time complexity is $O(n * n * n)$ which simplifies to $O(n^3)$.

The space complexity of the function is $O(n^2)$. The memoization cache `dfs(i, m)` will store at most $n * n$ states, since both i and m range in $[0, n]$. The list s that stores the accumulated sums introduces another $O(n)$ space, but since it grows linearly with the input, it doesn't affect the overall $O(n^2)$ space complexity dominated by the memoization cache.