1563. Stone Game V

Hard Array Math Dynamic Programming Game Theory Leetcode Link

Problem Description

In the "Stone Game V" problem, you are given a row of stones with each stone having an associated value. The objective is for the player, Alice, to score as high as possible through a series of rounds. During each round, Alice divides the row of stones into two non-empty parts. Then, Bob, another player, calculates the sum of values for each part. Bob discards the part with the higher sum, and Alice's score is increased by the sum of the remaining part. If both parts have an equal sum, Alice gets to decide which part to discard. This process repeats until only one stone remains. Alice's score starts at zero, and the goal is to find the maximum score Alice can achieve by the end of the game.

Intuition

The intuition behind the solution is to recognize that we are dealing with an optimization problem with overlapping subproblems and optimal substructure, meaning the problem can be effectively solved using dynamic programming. More specifically, since Alice can make different choices leading to different outcomes, we explore different ways of splitting the stones, always considering the best score Alice can get from any given point.

Memoization (caching the results of expensive function calls) is applied to ensure we do not compute the same state more than once. The problem is recursive in nature - for every split Alice makes, there are subsequently different potential splits that can be made thereafter. Therefore, we define a recursive function `dfs(i, j)`, which represents the maximum score Alice can achieve from stones in the range `stoneValue[i]` to `stoneValue[j]`.

This function compares the sum of values in the left and right partitions of the current range, always trying to maximize the score obtained by Alice. If the left partition has a lesser sum than the right, we only consider the left partition for further recursive calls (since Bob would discard the right). Conversely, if the right partition has a lesser sum, we consider the right partition for recursion. If both partitions have equal sums, we consider both partitions to find the maximum score. We use an accumulated sum array `s` to efficiently calculate the sum of stones for each partition.

The solution uses top-down dynamic programming strategy to break down the game into manageable states (represented by the recursive function `dfs`) and relies on a memoization technique to cache answers to subproblems. By exploring all possible splits and choosing the optimal ones at each step, the function calculates the maximum score that Alice can gain.

Solution Approach

The implementation uses dynamic programming with memoization to optimize the recursive search. The key elements of the solution are:

- **Memoization**: The `dfs` function is decorated with Python's `@cache` feature, which memoizes the results. It stores the output of the function for each unique set of arguments `(i, j)` to ensure we do not repeat computations. This is essential for reducing the problem's complexity from exponential to polynomial time.

- **Recursive Function (`dfs`)**: The `dfs(i, j)` function calculates the maximum score from the subarray of `stoneValue` starting at index `i` and ending at index `j`. It uses recursion to try each possible division of the array into left and right parts and finds the optimal score that can be achieved from that point onwards.

- **Accumulated Sum Array (`s`)**: An accumulated sum array `s` is created using Python's `accumulate` function with `initial=0`. This array helps in efficiently calculating the sum of values in any subarray `stoneValue[i:j+1]` by simply calculating `s[j+1] - s[i]` without needing to sum the elements each time.

- **Optimization Logic**: As we divide the array and calculate the sum `a` of the left part, we also calculate the sum `b` of the right part as `s[j + 1] - s[i] - a`. Depending on whether `a` is less than, greater than, or equal to `b`, we make recursive calls for the next round accordingly:

  - If `a < b`, we only need to consider the maximum score obtained from the left part (`dfs(i, k)`) because Bob would throw away the right part.
  - If `a > b`, we only need to consider the maximum score obtained from the right part (`dfs(k + 1, j)`) for similar reasons.
  - If `a == b`, Alice has the chance to choose either part, so we calculate and compare the scores for both possible decisions, `dfs(i, k)` and `dfs(k + 1, j)`, and take the one which gives the maximum score.
- **Pruning**: To further optimize, the implementation includes pruning steps where it avoids unnecessary recursive calls. If `ans` (current max score in our range) is already greater than or equal to twice the sum of a part, then we know exploring that part further won't yield a better outcome and we can skip making those recursive calls.

By applying these techniques, the solution efficiently explores all possibilities while avoiding redundant computations, leading to the maximum score Alice can achieve in the Stone Game V.

Example Walkthrough

To illustrate the solution approach for the Stone Game V problem, let's consider a small example where the `stoneValue` array is `[6, 2, 3, 4]`. Alice is trying to maximize her score by choosing the optimal places to split the array of stones.

1. We initialize the accumulated sum array `s` as `[0, 6, 8, 11, 15]`, which includes an initial zero for simplifying the sum calculations.

2. We start by calling the recursive function `dfs(0, 3)` to calculate the maximum score Alice can obtain with the full array of stones from index 0 to 3.

3. Inside `dfs(0, 3)`, we try splitting the array into two parts at different positions:

   - Split between 6 and 2: `a = s[1] - s[0] = 6`, `b = s[4] - s[1] = 9`
     - Since `a < b`, the maximum score for this division would be the result of `dfs(0, 0)` which is 6 (because Bob would throw away the part that includes [2, 3, 4]).
   - Split between 2 and 3: `a = s[2] - s[0] = 8`, `b = s[4] - s[2] = 7`
     - Since `a > b`, the maximum score for this division would be the result of `dfs(2, 3)` which is 7 (because Bob would throw away the part that includes [6, 2]).
   - Split between 3 and 4: `a = s[3] - s[0] = 11`, `b = s[4] - s[3] = 4`
     - Since `a > b`, the maximum score for this division would be the result of `dfs(3, 3)` which is 4 (because Bob would throw away the part that includes [6, 2, 3]).
4. Since `dfs(0, 0)`, `dfs(2, 3)`, and `dfs(3, 3)` do not involve any further splits (single element arrays), they simply return their element's value as the maximum score.

5. We compare the possible outcomes of each split for `dfs(0, 3)`:

   - The result of `dfs(0, 0)` is 6.
   - The result of `dfs(2, 3)` is 7.
   - The result of `dfs(3, 3)` is 4.
6. The maximum score that Alice can achieve is the maximum of the results above, which is 7, achieved by splitting the array between indexes [6, 2] and [3, 4].

7. By caching these results in our memoization structure, we avoid recalculating `dfs(0, 0)`, `dfs(2, 3)`, and `dfs(3, 3)` in future iterations.

8. The memoization ensures that subsequent calls to `dfs` with the same `(i, j)` arguments will not repeat the calculation but will retrieve the result from the cache.

By following this approach for all possible splits and recursing as needed, we can find that the maximum score Alice can achieve with the `stoneValue` array `[6, 2, 3, 4]` is 7.

Python Solution

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

class Solution:
    def stoneGameV(self, stone_values: List[int]) -> int:
        # Using memoization to optimize recursive calls
        @lru_cache(None)
        def dfs(left_index, right_index):
            # Base case: when there's only one stone, no score can be obtained
            if left_index == right_index:
                return 0

            max_score = 0
            left_sum = 0

            # Try splitting the array at different points to maximize the score
            for split_index in range(left_index, right_index):
                # Sum of values on the left of the split point
                left_sum += stone_values[split_index]
                # Sum of values on the right of the split point
                right_sum = prefix_sums[right_index + 1] - prefix_sums[left_index] - left_sum

                # Compare sums to decide which side to choose
                if left_sum < right_sum:
                    # If the left sum is less, consider the left portion;
                    # skip if not better than double the current max_score
                    if max_score >= left_sum * 2:
                        continue
                    max_score = max(max_score, left_sum + dfs(left_index, split_index))
                elif left_sum > right_sum:
                    # If the right sum is less, consider the right portion;
                    # stop splitting further right if not better than double the current max_score
                    if max_score >= right_sum * 2:
                        break
                    max_score = max(max_score, right_sum + dfs(split_index + 1, right_index))
                else:
                    # If both are equal, consider both portions and take the maximum
                    max_score = max(max_score, left_sum + dfs(left_index, split_index), right_sum + dfs(split_index + 1, right_inde

            return max_score

        # Compute prefix sums for stone_values to use later in splitting
        prefix_sums = list(accumulate(stone_values, initial=0))

        # Compute the maximum score by starting from the entire range
        return dfs(0, len(stone_values) - 1)

# Example usage:
# stone_values = [6, 2, 3, 4, 5]
# solution = Solution()
# result = solution.stoneGameV(stone_values)
# print(result)  # This will print the result of the stone game V for the given stone values.

```

Java Solution

```java
class Solution {
    private int numStones; // Total number of stones
    private int[] prefixSums; // Array to store prefix sums
    private int[] stoneValues; // Array to store values of stones
    private Integer[][] memo; // Memoization array to avoid repeated calculation

    public int stoneGameV(int[] stoneValue) {
        numStones = stoneValue.length;
        this.stoneValues = stoneValue;
        prefixSums = new int[numStones + 1]; // One additional space to include sum up to 0

        // Pre-calculate prefix sums to enable fast range sum queries
        for (int i = 1; i <= numStones; ++i) {
            prefixSums[i] = prefixSums[i - 1] + stoneValue[i - 1];
        }

        memo = new Integer[numStones][numStones]; // Initialize memoization matrix
        return dfs(0, numStones - 1);
    }

    // Helper method to recursively find the maximum score using DFS and memoization
    private int dfs(int left, int right) {
        if (left == right) {
            // Base case: when there is only one stone, score is 0
            return 0;
        }
        if (memo[left][right] != null) {
            // Return previously computed value if available to avoid repeat calculations
            return memo[left][right];
        }

        int maxScore = 0; // Store the maximum score found so far
        int leftSum = 0; // Sum of the left partition

        // Iterate over all possible partitions between left and right
        for (int k = left; k < right; ++k) {
            // Accumulate value of stones for the current partition
            leftSum += stoneValues[k];
            // Compute right partition sum
            int rightSum = prefixSums[right + 1] - prefixSums[left] - leftSum;

            if (leftSum < rightSum) {
                // If left partition has less sum, consider the left partition
                if (maxScore >= leftSum * 2) {
                    continue;
                }
                maxScore = Math.max(maxScore, leftSum + dfs(left, k));
            } else if (leftSum > rightSum) {
                // If right partition has less sum, consider the right partition
                if (maxScore >= rightSum * 2) {
                    break;
                }
                maxScore = Math.max(maxScore, rightSum + dfs(k + 1, right));
            } else {
                // If sums are equal, consider the maximum value from both partitions
                maxScore = Math.max(maxScore, Math.max(leftSum + dfs(left, k), rightSum + dfs(k + 1, right)));
            }
        }
        // Store the computed value in memo and return it
        return memo[left][right] = maxScore;
    }
}

```

C++ Solution

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

class Solution {
public:
    int stoneGameV(std::vector<int>& stoneValue) {
        int n = stoneValue.size();
        std::vector<int> prefixSum(n + 1, 0);
        // Calculate the prefix sums for quick range sum queries.
        for (int i = 1; i <= n; ++i) {
            prefixSum[i] = prefixSum[i - 1] + stoneValue[i - 1];
        }

        // Initialize memoization table
        std::vector<std::vector<int>> memo(n, std::vector<int>(n, 0));

        // Define the DFS function for dynamic programming with memoization.
        std::function<int(int, int)> dfs = [&](int left, int right) -> int {
            // If the range consists of a single stone, no more score can be accrued.
            if (left == right) {
                return 0;
            }
            // If we have already computed the solution for this subrange, return the result.
            if (memo[left][right]) {
                return memo[left][right];
            }
            int maxScore = 0;
            int leftSum = 0;
            // Iterate through all possible partitions of the range
            for (int k = left; k < right; ++k) {
                leftSum += stoneValue[k];
                int rightSum = prefixSum[right + 1] - prefixSum[left] - leftSum;
                // Split into [left, k] and [k+1, right] and calculate recursively.
                if (leftSum < rightSum) {
                    maxScore = std::max(maxScore, leftSum + dfs(left, k));
                } else if (leftSum > rightSum) {
                    maxScore = std::max(maxScore, rightSum + dfs(k + 1, right));
                } else {
                    // If leftSum equals rightSum, calculate both and take the maximum.
                    maxScore = std::max({maxScore, leftSum + dfs(left, k), rightSum + dfs(k + 1, right)});
                }
            }
            // Store the result in the memoization table before returning.
            return memo[left][right] = maxScore;
        };

        // Call the DFS function for the full range.
        return dfs(0, n - 1);
    }
};

// Example usage:
// Solution sol;
// std::vector<int> stoneValue = {6, 2, 3, 4, 5};
// int result = sol.stoneGameV(stoneValue);

```

Typescript Solution

```typescript
function stoneGameV(stoneValue: number[]): number {
    const n: number = stoneValue.length;
    const prefixSum: number[] = new Array(n + 1).fill(0);
    // Calculate the prefix sums for quick range sum queries
    for (let i = 1; i <= n; ++i) {
        prefixSum[i] = prefixSum[i - 1] + stoneValue[i - 1];
    }

    // Initialize memoization table
    const memo: number[][] = new Array(n).fill(0).map(() => new Array(n).fill(0));

    // Define the recursive function for dynamic programming with memoization
    const dfs: (left: number, right: number) => number = (left, right) => {
        // If the range consists of a single stone, no more score can be accrued
        if (left === right) {
            return 0;
        }
        // If we have already computed the solution for this subrange, return the result
        if (memo[left][right]) {
            return memo[left][right];
        }
        let maxScore = 0;
        let leftSum = 0;
        // Iterate through all possible partitions of the range
        for (let k = left; k < right; ++k) {
            leftSum += stoneValue[k];
            const rightSum = prefixSum[right + 1] - prefixSum[k + 1];
            // Split into [left, k] and [k+1, right] and calculate recursively
            if (leftSum < rightSum) {
                maxScore = Math.max(maxScore, leftSum + dfs(left, k));
            } else if (leftSum > rightSum) {
                maxScore = Math.max(maxScore, rightSum + dfs(k + 1, right));
            } else {
                // If leftSum equals rightSum, calculate both and take the maximum
                maxScore = Math.max(maxScore, leftSum + dfs(left, k), rightSum + dfs(k + 1, right));
            }
        }
        // Store the result in the memoization table before returning
        memo[left][right] = maxScore;
        return maxScore;
    };

    // Call the recursive function for the full range
    return dfs(0, n - 1);
}

// Example usage:
// const stoneValue: number[] = [6, 2, 3, 4, 5];
// const result: number = stoneGameV(stoneValue);

```

Time and Space Complexity

Time Complexity

The time complexity of the `dfs` function is defined by the number of subproblems it needs to solve and the time it takes to solve each problem. Since the function uses memoization, each subproblem (defined by different `(i, j)` pairs where `i` and `j` are the indices of `stoneValue`) is only solved once. The number of subproblems that can be formed in a list of `N` stones is `O(N^2)` because there are `N` choices for where to start (`i`) and `N` choices for where to end (`j`), though in practice, it's less because `i` has to be less than `j`.

The function `dfs` itself does a single pass from `i` to `j` in each call. This means that for each subproblem, you have to do `O(j-i)` work to find the partition point `k` that gives the optimal score. In the worst case, this is `O(N)` work per subproblem. So if every `(i, j)` pair were considered and we did linear work on each, the time complexity of solving all subproblems would be `O(N^3)`.

However, by using pruning (if-statements that use a running total to skip unnecessary work), the code ensures that it doesn't continue to look for a better partition once it's clear that no better partition is possible. This optimization makes it difficult to define an exact worst-case time complexity, as it's heavily input-dependent, but the use of memoization and pruning brings the expected time complexity down considerably in typical cases. Nonetheless, the upper bound in the absence of any pruning would still be `O(N^3)`.

Space Complexity

The space complexity is determined by the size of the memoization cache and the call stack due to recursion. As noted above, there can be `O(N^2)` subproblems, so `dfs` can be called that many times in different contexts, and that many results can be stored. Thus, the memoization cache has a space complexity of `O(N^2)`.

Recursion also consumes stack space. In the worst case, the `dfs` function might recurse `N` times before reaching a base case (when `i` equals `j`). Thus, the stack space used in the worst case is `O(N)`.

Adding the space for the prefix sum array, which is `O(N)`, the total space complexity becomes `O(N^2) + O(N) + O(N)`, which simplifies to `O(N^2)` because in big-O notation, we focus on the term that grows fastest as `N` increases.

Therefore, the overall space complexity is `O(N^2)`.