877. Stone Game

Medium | Array | Math | Dynamic Programming | Game Theory | Leetcode Link

Problem Description

The problem presents a game between two players, Alice and Bob, involving a row of piles of stones. There is an even number of these piles, and a strictly positive number of stones is in each pile. The aim for each player is to collect as many stones as possible. Alice takes the first turn, and then players alternate turns. On each turn, a player can only take all the stones from either the first pile or the last pile in the row. The game ends when no piles remain, with the player having the most stones declared the winner. Given that the total number of stones is odd, there cannot be a tie. The goal is to determine if Alice can win the game, provided both players act optimally. The outcome should be a boolean value, `true` if Alice can win, or `false` otherwise.

Intuition

Given that there is an optimal strategy for both players, we can approach this problem with dynamic programming. Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. It's particularly well-suited for optimization problems such as this game, where the outcome depends on decisions made at each step.

The intuition behind this solution involves understanding that each player wants to maximize the difference between their own score and their opponent's score at the end of the game. To represent this, we can use a 2-dimensional array `f` where `f[i][j]` represents the maximum score difference a player can achieve when considering the subarray of piles from index `i` to index `j`.

Here's the thought process for building the solution:

1. Initialize a 2D array `f` of size `n` by `n`, where `n` is the number of piles.
2. Start by filling the diagonal `f[i][i]` with the number of stones in pile `i`, because if there's only one pile, the player who is taking the turn will just take it.
3. Fill the 2D array in a bottom-up manner. For every pair of piles `(i, j)`, where `i` is less than `j`, calculate the optimal score difference by considering two scenarios:
   - The player takes the first pile `i`, leaving the piles `i+1` to `j` for the opponent. The score difference would be `piles[i] - f[i + 1][j]`.
   - The player takes the last pile `j`, leaving the piles `i` to `j-1`. The score difference would be `piles[j] - f[i][j - 1]`.
4. We should maximize the score difference, so we take the maximum from both scenarios to fill in `f[i][j]`.
5. To achieve the maximum score difference, choose the optimal move at each step based on the values calculated in the 2D array `f`.

If `f[0][n - 1]` is greater than 0, it means Alice can end the game with more stones than Bob, hence she wins. The relaxation of constraints due to the piles' count being even and total stones being odd allows us to conclude that Alice always wins by taking the optimal first move. Therefore, the function always returns true for the given constraints.

Solution Approach

The solution to this problem uses a dynamic programming approach, which involves creating a table to store the results of subproblems. In this case, the subproblems are the maximum score difference that can be achieved from any subset of piles from `i` to `j`.

Here is how the solution is implemented step by step:

1. **Initialization of the DP Table**: A 2D list `f` is initialized with `n` rows and `n` columns filled with zeros, where `n` is the number of piles. This list is used to store the maximum score difference between the player's score and the opponent's score.

2. **Base Case**: For `i` ranging from 0 to `n-1`, we fill the diagonal `f[i][i]` with the number of stones in pile `i`. Since this scenario involves a single pile, the score difference is equal to the number of stones in that pile.

3. **Bottom-up Calculation**: The table `f` is filled in a bottom-up manner. This means we calculate the values for subsets of piles starting from the smallest range `(i, i)` and building up to the entire range `(0, n-1)`. The outer loop iterates `i` from `n-2` down to `0`. This is because we already know the value of `f[i][i]` and want to calculate the score differences for larger subproblems.

4. **Choosing the Optimal Move**: Inside the nested loop, `j` goes from `i+1` to `n-1`. At this point, we consider two possible moves for a player:

   - **Taking the first pile**: If the player takes the pile at index `i`, the score difference would be `piles[i] - f[i + 1][j]`, indicating the player's score increases by `piles[i]` and then subtracting whatever the best result is for the opponent from the remaining piles `i+1` to `j`.
   - **Taking the last pile**: If the player takes the pile at index `j`, the score difference would be `piles[j] - f[i][j - 1]`, with similar logic. The player's score is increased by `piles[j]`, and then we subtract the opponent's optimal result from the remaining piles.
5. **Maximizing the Score Difference**: For each pair `(i, j)`, we assign `f[i][j]` the maximum value out of the two possible moves, which represents the best score difference the player can achieve with the current range of piles from `i` to `j`.

6. **Deciding the Winner**: After filling the table, `f[0][n - 1]` contains the maximum score difference that Alice can achieve playing with all the piles against an optimally playing Bob. Since we only return `true` or `false` based on whether Alice can win the game or not, and the game is biased such that Alice always has a winning strategy with the given conditions (even number of piles and total number of stones are odd), the solution function always returns `true`.

In conclusion, the algorithm efficiently calculates the optimal strategy by keeping track of the score differences that each player can achieve, and it leverages the properties of the dynamic programming to find the solution in a bottom-up manner.

Example Walkthrough

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

Example: Suppose there are 4 piles of stones with the following number of stones in each pile: `1, 2, 3, 4`.

Now let's apply the solution approach step by step:

1. **Initialization of the DP Table**: We initialize a 2D list `f` that will have `4 x 4` dimensions filled with zeros to store the maximum score difference.

   ```
   f = [
     [0, 0, 0, 0],
     [0, 0, 0, 0],
     [0, 0, 0, 0],
     [0, 0, 0, 0],
   ]
   ```

2. **Base Case**: We fill the diagonal of `f` with the number of stones in the corresponding pile:

   ```
   f = [
     [1, 0, 0, 0],
     [0, 2, 0, 0],
     [0, 0, 3, 0],
     [0, 0, 0, 4],
   ]
   ```

3. **Bottom-up Calculation**: Start from the second last diagonal because the last diagonal (the base case) is already filled.

   Subproblem `[0,1]`:

   - Alice takes the first pile: `piles[0] - f[1][1] = 1 - 2 = -1`
   - Alice takes the last pile: `piles[1] - f[0][0] = 2 - 1 = 1`

   Alice will take the last pile (for a better score difference of 1), so `f[0][1]` becomes `1`.

   Repeat this process for the subproblems `[1,2]` and `[2,3]`.

   ```
   f = [
     [1, 1, 0, 0],
     [0, 2, 1, 0],
     [0, 0, 3, -1],
     [0, 0, 0, 4],
   ]
   ```

Subproblem `[0,2]`:

- Alice takes the first pile: `piles[0] - f[1][2] = 1 - 1 = 0`
- Alice takes the last pile: `piles[2] - f[0][1] = 3 - 1 = 2`

Alice will take the last pile (better score difference of 2) so `f[0][2]` becomes `2`.

Repeat this process for the subproblem `[1,3]`.

```
  f = [
    [1, 1, 2, 0],
    [0, 2, 1, 3],
    [0, 0, 3, -1],
    [0, 0, 0, 4],
  ]
```

Subproblem `[0,3]`:

- Alice takes the first pile: `piles[0] - f[1][3] = 1 - 3 = -2`
- Alice takes the last pile: `piles[3] - f[0][2] = 4 - 2 = 2`

Alice will go for the last pile again (better score difference of 2), thus `f[0][3]` becomes `2`.

```
  f = [
    [1, 1, 2, 2],
    [0, 2, 1, 3],
    [0, 0, 3, -1],
    [0, 0, 0, 4],
  ]
```

4. **Deciding the Winner**: After completing the table, `f[0][3]` is the final maximum score difference that Alice can achieve. Since `f[0][3]` is `2` and it's a positive value, Alice can win the game. Therefore, for this example, the function would return `true`.

Concluding, Alice starts and can always opt to take the optimal move by choosing the first or last pile that maximizes her score difference based on the precomputed DP table, thus guaranteeing her victory with the optimal strategy in this example.

Python Solution

```python
from typing import List

class Solution:
    def stoneGame(self, piles: List[int]) -> bool:
        # Length of the piles array
        n = len(piles)
        # Initialize a 2D array to store the maximum difference in scores
        # between the player who starts and the other player, for any given pile range
        dp = [[0] * n for _ in range(n)]

        # Fill the array along the main diagonal, where each cell
        # on the diagonal represents only one pile
        for i, pile in enumerate(piles):
            dp[i][i] = pile

        # Dynamic programming - bottom-up approach, filling the table
        # Start from second last row and go upwards
        for i in range(n - 2, -1, -1):
            # For each row, start from the element right to the diagonal element and move rightwards
            for j in range(i + 1, n):
                # dp[i][j] will be the maximum difference in score achievable by the player who starts
                # i.e., max of choosing the left-most or right-most pile
                # The decision is whether to take the pile at the current left (piles[i])
                # minus what the opponent would get (dp[i + 1][j]),
                # or to take the pile at the current right (piles[j])
                # minus what the opponent would get (dp[i][j - 1])
                dp[i][j] = max(piles[i] - dp[i + 1][j], piles[j] - dp[i][j - 1])

        # The first player wins if the maximum difference in score is positive
        return dp[0][n - 1] > 0

# The code can be used like this:
# solution = Solution()
# result = solution.stoneGame([3, 7, 2, 10])  # Example call with some input
# print(result)  # Outputs True or False based on whether the first player has a winning strategy

```

Java Solution

```java
class Solution {
    public boolean stoneGame(int[] piles) {
        int totalPiles = piles.length;

        // Create a DP table to memorize the game's outcomes at different states.
        int[][] dp = new int[totalPiles][totalPiles];

        // Base case: When there's only one pile, the best score is the number of stones in it.
        for (int i = 0; i < totalPiles; ++i) {
            dp[i][i] = piles[i];
        }

        // Fill in the DP table, starting from the second last row, moving upwards.
        for (int startIndex = totalPiles - 2; startIndex >= 0; --startIndex) {
            // The column index starts from one place after the current row index,
            // since we are considering the game from start index to j.
            for (int endIndex = startIndex + 1; endIndex < totalPiles; ++endIndex) {
                // The current player can choose either the starting or ending pile,
                // and the score is the max of these two choices minus the score of
                // the next player's best choice.
                int pickStartPile = piles[startIndex] - dp[startIndex + 1][endIndex];
                int pickEndPile = piles[endIndex] - dp[startIndex][endIndex - 1];
                dp[startIndex][endIndex] = Math.max(pickStartPile, pickEndPile);
            }
        }

        // If the score accumulated from the first pile to the last pile is positive,
        // then the first player wins. This is the top-right corner of the DP matrix.
        return dp[0][totalPiles - 1] > 0;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Determines if the first player can win the stone game given the piles
    bool stoneGame(vector<int>& piles) {
        int n = piles.size(); // Get the number of piles
        vector<vector<int>> dp(n, vector<int>(n, 0)); // Create a 2D dp array filled with 0s

        // Initialize the diagonal of the dp array since a single pile is trivially the score
        for (int i = 0; i < n; ++i) {
            dp[i][i] = piles[i];
        }

        // Fill the dp array from the second-to-last row to the first row
        for (int i = n - 2; i >= 0; --i) {
            // For each row, iterate from the second column to the last column
            for (int j = i + 1; j < n; ++j) {
                // The dp formula to decide the best score:
                // the current player will either pick the left end or the right end.
                // The difference in scores is the remaining score minus the opponent's best response.
                dp[i][j] = max(piles[i] - dp[i + 1][j], piles[j] - dp[i][j - 1]);
            }
        }

        // If the score at dp[0][n - 1] is greater than 0, the first player wins
        return dp[0][n - 1] > 0;
    }
};

```

Typescript Solution

```typescript
function stoneGame(piles: number[]): boolean {
    const pileCount = piles.length;
    // Create a 2D array to store the scores differences,
    // initializing all elements to zero.
    const scoreDifferences: number[][] = new Array(pileCount)
        .fill(0)
        .map(() => new Array(pileCount).fill(0));

    // Initialize the diagonal elements of the array with the pile values
    // where only one pile is considered (i.e., when the interval [i, i] is considered).
    for (let i = 0; i < pileCount; ++i) {
        scoreDifferences[i][i] = piles[i];
    }

    // Loop through the array in a bottom-up manner.
    // i represents the starting index and j represents the ending index of the piles to consider.
    for (let i = pileCount - 2; i >= 0; --i) {
        for (let j = i + 1; j < pileCount; ++j) {
            // Apply the minimax strategy: the current player chooses the pile to maximize
            // their score (accounting for the score the other player can achieve).
            // Math.max chooses the better of two scenarios:
            //   1. Taking the pile at index i and subtracting the opponent's best score from (i+1) to j.
            //   2. Taking the pile at index j and subtracting the opponent's best score from i to (j-1).
            scoreDifferences[i][j] = Math.max(
                piles[i] - scoreDifferences[i + 1][j],
                piles[j] - scoreDifferences[i][j - 1]
            );
        }
    }

    // The game is won if the score difference when considering the whole array (0 to pileCount - 1)
    // is positive.
    return scoreDifferences[0][pileCount - 1] > 0;
}

```

Time and Space Complexity

The given Python code implements a dynamic programming solution to solve the Stone Game problem. Let's analyze both time complexity and space complexity:

- **Time Complexity**: The time complexity of this code can be determined by analyzing the nested loops and operations within them. The outer loop runs from `n-2` down to `0`, which is essentially `n` times. The inner loop runs from `i+1` to `n`, which also equates to `n` iterations in total (although it's fewer for each step of the outer loop). Within each iteration of the inner loop, the code performs a constant number of operations. Combining these factors, we get a time complexity of O(`n^2`), where `n` is the length of the `piles` list.

- **Space Complexity**: The space complexity is primarily dictated by the space needed to store the dynamic programming table `f`, which is a 2D array of size `n * n`, where `n` is the length of the `piles` list. Therefore the space complexity is O(`n^2`).

Overall, the time complexity is O(`n^2`) and the space complexity is O(`n^2`).