486. Predict the Winner

Medium Recursion Array Math Dynamic Programming Game Theory

Problem Description

In this problem, we are given an integer array called `nums`. The game involves two players, player 1 and player 2, who take turns to pick numbers from either end of the array. Each player accumulates points based on the value of the number they pick, and the size of the array decreases by one. Player 1 starts the game. The game is over when there are no more elements left in the array to pick from.

Our task is to determine if Player 1 can win the game under the condition that both players are playing optimally. This means that each player will choose their numbers in a way that maximizes their own score. If at the end of the game, the scores are tied, Player 1 is considered the winner.

Intuition

The solution revolves around the concept of dynamic programming, where we break down this problem into smaller subproblems and solve it for each sub-array of `nums`. The idea is to create a table `f` where `f[i][j]` represents the best score player 1 can achieve over player 2 (which might be a negative value if player 2 is leading) when considering a sub-array from index `i` to index `j`.

The intuition behind the dynamic programming approach is to consider each possible move for player 1 and predict the opponent's best response, which will be to minimize player 1's score. We proceed backwards, starting from the end of the array and move toward the front.

At each step, player 1 has two choices: to pick the number at the beginning or at the end of the sub-array. We simulate both choices. If player 1 picks beginning number, the score will be `nums[i] - f[i + 1][j]`. If player 1 picks the end number, the score will be `nums[j] - f[i][j - 1]`. The `-` sign is because player 1's choice leaves the remainder of the array to player 2, who will then be the one trying to maximize the score difference.

We then take the maximum of these two options as player 1's best strategy at this point, gradually filling up the dynamic programming table.

Finally, if `f[0][n - 1]` (the best outcome starting from the entire array) is greater or equal to 0, player 1 can win the game, and we return `true`.

Solution Approach

The provided code uses a 2-dimensional list (a matrix) `f` where `f[i][j]` will represent the best score player 1 can achieve more than player 2, from the sub-array of `nums` starting at index `i` and ending at index `j`.

Here's the step-by-step implementation of the solution:

1. **Initialization:** The variable `n` is set to the length of `nums`. Then, `f` is initialized as a 2-dimensional list with `n` lists each containing `n` zeros.
2. **Base Case:** The base case is filled out where there's only one element in the sub-array (i.e., `i == j`). In this case, the entire score is just the number itself. This is done using the loop:

```
for i, x in enumerate(nums):
    f[i][i] = x
```

3. **Dynamic Programming Table Filling:** The dynamic programming table is filled in a bottom-up manner. We iterate over the starting index of the sub-array `i` in reverse (from `n-2` down to `0`) and for each `i`, we iterate over the ending index `j` (from `i+1` up to `n-1`).

   Within the nested loops, we calculate `f[i][j]`, which is the best score player 1 can achieve from the current sub-array `nums[i:j+1]`. This is where player 1 may decide to pick the `i-th` element or the `j-th` element. The decision is modeled using:

```
f[i][j] = max(nums[i] - f[i + 1][j], nums[j] - f[i][j - 1])
```

   The first part of the `max` function, `nums[i] - f[i + 1][j]`, represents the scenario where player 1 picks the `i-th` element and thus the score is `nums[i]` minus the score that player 2 can subsequently achieve from the remaining sub-array (from `i+1` to `j`).

   The second part, `nums[j] - f[i][j - 1]`, handles the case where player 1 picks the `j-th` element and the score is `nums[j]` minus the score that player 2 can achieve from the sub-array (from `i` to `j-1`).

4. **Evaluation:** In the end, `f[0][n-1]` holds player 1's best score over player 2 for the entire array. If this score is non-negative, it means that player 1 can either win or tie the game, so the method returns `True`.

The key algorithmic patterns used in this approach are **dynamic programming** and **minimax**. Minimax is a recursive algorithm often used in decision-making and game theory, where players minimize the possible loss for a worst-case scenario. In conjunction to dynamic programming, it helps optimize the decisions by solving subproblems once and storing their solutions—providing an efficient way to tackle this kind of competitive game scenario.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Assume our `nums` array is `[1, 5, 2]`.

1. **Initialization:** We set `n` to be `3` because there are three elements in `nums`. Then, we initialize our table `f` with zeros. It will look like this:

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

2. **Base Case:** If there's only one element, player 1 will pick that element. So `f[0][0]`, `f[1][1]`, and `f[2][2]` will simply be `1`, `5`, and `2` respectively. The updated table `f` is:

```
f = [
    [1, 0, 0],
    [0, 5, 0],
    [0, 0, 2]
]
```

3. **Dynamic Programming Table Filling:** We fill the table `f` in a bottom-up manner.
   - For the sub-array starting at `i = 1` (the second element) and ending at `j = 2` (the last element):
     - The score if player 1 picks element at `i (5)`: `5 - f[2][2]` (`5 - 2 = 3`).
     - The score if player 1 picks element at `j (2)`: `2 - f[1][1]` (`2 - 5 = -3`).
     - Player 1 will pick the larger score, so `f[1][2] = max(3, -3) = 3`.
   - Now for the sub-array starting at `i = 0` and ending at `j = 1`:
     - Score if player 1 picks element at `i (1)`: `1 - f[1][1]` (`1 - 5 = -4`).
     - Score if player 1 picks element at `j (5)`: `5 - f[0][0]` (`5 - 1 = 4`).
     - Player 1 will choose the larger score, so `f[0][1] = max(-4, 4) = 4`.

   Our `f` table now looks like this:

```
f = [
    [1, 4, 0],
    [0, 5, 3],
    [0, 0, 2]
]
```

   - Lastly, for the entire array (`i = 0` to `j = 2`):
     - Score if player 1 picks first element (`1`): `1 - f[1][2]` (`1 - 3 = -2`).
     - Score if player 1 picks last element (`2`): `2 - f[0][1]` (`2 - 4 = -2`).
     - Both options result in `-2`, so `f[0][2] = max(-2, -2) = -2`.

   The final `f` table is:

```
f = [
    [1, 4, -2],
    [0, 5, 3],
    [0, 0, 2]
]
```

4. **Evaluation:** We examine `f[0][2]`, which is the score of player 1 after considering the entire array. Since `f[0][2] = -2`, player 1 is not guaranteed to win; player 2 has an advantage. Therefore, the answer is `False`.

In this specific example, no matter how optimally player 1 plays, player 2 will always have a strategy to ensure a higher score. Thus, player 1 cannot win in this scenario.

Solution Implementation

Python

```
class Solution:
    def PredictTheWinner(self, nums: List[int]) -> bool:
        # Determine the total number of elements in the nums list
        num_elements = len(nums)

        # Create a 2D list (dynamic programming table) with size num_elements x num_elements
        # where dp_table[i][j] will represent the maximum score the current player can achieve
        # from the subarray nums[i] to nums[j]
        dp_table = [[0 * num_elements for _ in range(num_elements)]

        # Base case: When i == j, the current player can only choose nums[i]
        for i in range(num_elements):
            dp_table[i][i] = nums[i]

        # Fill in the dp_table, starting from the end of the list and moving backwards
        for i in range(num_elements - 2, -1, -1):  # Start from second to last and go to start of the list
            for j in range(i + 1, num_elements):  # Start from just after i and move to the end of the list
                # The current player can choose either the start or end of the remaining nums list.
                # The score is the value chosen minus the result of the next turn (since the next turn, the opponent plays)
                dp_table[i][j] = max(nums[i] - dp_table[i + 1][j],  # If choosing the start
                                     nums[j] - dp_table[i][j - 1])  # If choosing the end

        # The player who starts (the one we are evaluating for winning condition) has the score
        # at dp_table[0][num_elements - 1]. This score needs to be non-negative for the player to win.
        return dp_table[0][num_elements - 1] >= 0
```

Java

```
class Solution {

    // Function to predict the winner of the game
    public boolean PredictTheWinner(int[] nums) {
        int length = nums.length; // The length of the input array
        int[][] dp = new int[length][length]; // Create a DP table to store the scores

        // Initialize the diagonal of the DP table where only one element is chosen
        for (int i = 0; i < length; ++i) {
            dp[i][i] = nums[i];
        }

        // Fill the DP table in a bottom-up manner
        for (int start = length - 2; start >= 0; --start) {
            for (int end = start + 1; end < length; ++end) {
                // For each interval [start, end], calculate the maximum score the player can achieve
                // by choosing either the start or the end element and subtract the opponent's optimal score
                dp[start][end] = Math.max(nums[start] - dp[start + 1][end], nums[end] - dp[start][end - 1]);
            }
        }

        // The game is winnable if the score at the full interval [0, length - 1] is non-negative
        return dp[0][length - 1] >= 0;
    }
}
```

C++

```
#include <vector>
#include <cstring>
#include <algorithm>

class Solution {
public:
    // Determine if the first player to move in the game can win when both players play optimally
    bool PredictTheWinner(vector<int>& nums) {
        int n = nums.size(); // The number of elements in the input array 'nums'
        vector<vector<int>> dp(n, vector<int>(n, 0)); // Initialize a 2D vector for dynamic programming

        // Base cases: when only one element is considered, the player who picks it wins with that value
        for (int i = 0; i < n; ++i) {
            dp[i][i] = nums[i];
        }

        // Fill the DP table in a bottom-up manner
        for (int start = n - 2; start >= 0; --start) {
            for (int end = start + 1; end < n; ++end) {
                // dp[start][end] represents the maximum score the player can achieve over the opponent,
                // starting from index 'start' to index 'end' inclusive
                dp[start][end] = std::max(nums[start] - dp[start + 1][end], nums[end] - dp[start][end - 1]);
            }
        }

        // The winning condition: the score is non-negative
        return dp[0][n - 1] >= 0;
    }
};
```

TypeScript

```
function predictTheWinner(nums: number[]): boolean {
    // Get the length of the nums array.
    const numLength = nums.length;

    // Create a 2D array 'dp' (short for dynamic programming)
    // to store the maximum score difference between the two players.
    const dp: number[][] = new Array(numLength).fill(0).map(() => new Array(numLength).fill(0));

    // Initialize the diagonal of the matrix where only one number is available to pick.
    for (let i = 0; i < numLength; ++i) {
        dp[i][i] = nums[i];
    }

    // Build the dp matrix from the bottom up, where 'i' represents the starting index
    // and 'j' represents the ending index of the subarray for the current game state.
    for (let i = numLength - 2; i >= 0; --i) {
        for (let j = i + 1; j < numLength; ++j) {
            // Determine the best score a player 1 can achieve from the current game state.
            // It's the maximum of choosing the 'i-th' or 'j-th' number,
            // and then subtracting the value of the dp state if the opponent picked next.
            dp[i][j] = Math.max(nums[i] - dp[i + 1][j], nums[j] - dp[i][j - 1]);
        }
    }

    // If the value of the dp table in the top right corner is non-negative,
    // player 1 can win or at least tie, so return true.
    // That value represents the score difference at the end if both play optimally.
    return dp[0][numLength - 1] >= 0;
}


class Solution:
    def PredictTheWinner(self, nums: List[int]) -> bool:
        # Determine the total number of elements in the nums list
        num_elements = len(nums)

        # Create a 2D list (dynamic programming table) with size num_elements x num_elements
        # where dp_table[i][j] will represent the maximum score the current player can achieve
        # from the subarray nums[i] to nums[j]
        dp_table = [[0 * num_elements for _ in range(num_elements)]

        # Base case: When i == j, the current player can only choose nums[i]
        for i in range(num_elements):
            dp_table[i][i] = nums[i]

        # Fill in the dp_table, starting from the end of the list and moving backwards
        for i in range(num_elements - 2, -1, -1):  # Start from second to last and go to start of the list
            for j in range(i + 1, num_elements):  # Start from just after i and move to the end of the list
                # The current player can choose either the start or end of the remaining nums list.
                # The score is the value chosen minus the result of the next turn (since the next turn, the opponent plays)
                dp_table[i][j] = max(nums[i] - dp_table[i + 1][j],  # If choosing the start
                                     nums[j] - dp_table[i][j - 1])  # If choosing the end

        # The player who starts (the one we are evaluating for winning condition) has the score
        # at dp_table[0][num_elements - 1]. This score needs to be non-negative for the player to win.
        return dp_table[0][num_elements - 1] >= 0
```

Time and Space Complexity

The given code implements a dynamic programming approach to solve the game prediction problem, where two players take turns to pick from a list of numbers from either end, and the goal is to predict whether the first player can win given the player can always pick optimally.

The time complexity of this approach comes from two nested loops that iterate over the elements of the array to fill in the dynamic programming table `f`:

- The outer loop runs `(n - 1)` times, where `n` is the length of the `nums` list (`for i in range(n - 2, -1, -1)`).
- The inner loop for each `i` runs `(n - i)` times (`for j in range(i + 1, n)`).

We can express the total number of iterations as the sum of an arithmetic series from `1` to `(n - 1)`, which gives us `(n(n - 1)) / 2`. Therefore, the time complexity is $O(n^2)$.

The space complexity is determined by the size of the dynamic programming table `f`, which is a 2D list of size `n * n`. Hence, the space complexity is also $O(n^2)$.

In conclusion, the time complexity is $O(n^2)$ and the space complexity is $O(n^2)$.