1908. Game of Nim

Medium | Bit Manipulation | Brainteaser | Array | Math | Dynamic Programming | Game Theory | Leetcode Link

Problem Description

In this game-themed problem, we have two players, Alice and Bob, who are playing a turn-based game involving piles of stones. There are `n` piles of stones, and on each player's turn, that player must remove any positive number of stones from any one pile. If a player can't make a move (which happens when all piles are empty), they lose. The twist in the problem comes from the assumption that both players will make the best possible move at any point in the game, meaning they play optimally. The task is to determine if Alice, who always goes first, will win the game given the initial number of stones in each pile represented by an integer array `piles`, where `piles[i]` specifies the number of stones in the ith pile.

Intuition

The problem is a variant of the classical game theory problem known as Nim. The key to solving this sort of problem lies in understanding the optimal strategies that players can employ and the winning positions in such a game.

A fundamental concept in playing Nim optimally is the Nim-sum, or the bitwise XOR sum of the number of stones in each pile. If the Nim-sum is 0 at the beginning of a player's turn, that player is in a losing position (assuming both players play perfectly). This is because no matter what move they make, the opponent can always return the game to a state where the Nim-sum is 0, putting the initial player back into a losing position on their next turn.

Therefore, if the Nim-sum of all the pile sizes is not 0 when it's Alice's turn (which is the case at the start of the game since she goes first), then Alice can make a move that forces a 0 Nim-sum on Bob's turn, putting her in a winning position from the start.

As the problem assumes optimal play from both Alice and Bob, we can deduce the winner by calculating the Nim-sum of all the pile sizes.

Although the provided solution code uses a depth-first search (DFS) approach, which can handle variations of the problem where the rules of play diverge from classical Nim, for this specific problem, a simplified solution would compute the Nim-sum, like so:

```python
class Solution:
    def nimGame(self, piles: List[int]) -> bool:
        xor_sum = 0
        for pile in piles:
            xor_sum ^= pile
        return xor_sum != 0
```

This code simply iterates over the piles and computes the cumulative XOR. If the result is non-zero, Alice wins, since she can always make a move to set the Nim-sum to 0 for Bob; if it is zero, Bob wins by maintaining the 0 Nim-sum position through each of his turns.

Solution Approach

The reference solution provided uses a recursive DFS approach to simulate every possible state of the game. Although for the specific game rules provided, there is a more efficient solution as explained earlier, this approach can handle a wider variety of game rules where the winning strategy might not be straightforward.

Here's a walk-through of the provided DFS solution:

1. The `dfs` function is defined within the context of the `nimGame` method. It is decorated with `@cache`, which is a Python decorator that automatically memoizes the results of the function calls. Memoization ensures that repeated states of the game are not re-evaluated, which optimizes the solution by reducing the number of recursive calls.

2. The `dfs` function is intended to be called with a tuple `st` representing the current state of the game, i.e., the number of stones in each pile. It converts `st` into a list `lst` for easier manipulation since tuples are immutable.

3. The `dfs` function then iterates over every pile `lst[i]` and for each pile tries removing every possible positive number of stones `j` (from 1 to the pile's size inclusive). This simulates all potential moves a player could make.

4. After simulating the move (removing `j` stones from pile `i`), `dfs` recursively checks whether this state leads to a losing state for the opponent by calling `not dfs(tuple(lst))`. If a losing state for the opponent is found, then the current state is winning, hence it returns `True`.

5. If no move leads to an immediate winning state, the `dfs` function returns `False`, indicating that any opponent's response to the current state can at most prolong the game, but eventually the current player will lose.

6. `nimGame` calls `dfs` using the initial state `tuple(piles)` and returns the result. If the result is `True`, it means Alice can win starting from the initial state; otherwise, if it's `False`, Bob will win.

The beauty of this approach is that it considers the entire space of possible moves and outcomes, and it ensures that we do not overlook any strategy that could lead to a win. However, as mentioned, the specific problem of Nim has a well-known mathematical solution, which is more efficient than the general DFS approach.

Nonetheless, the DFS approach is a powerful technique when dealing with complex decision-making problems in games or situations where optimal strategies are not known in advance. It provides a methodical way to explore all possible outcomes and determine the best sequence of actions when players are acting optimally.

Example Walkthrough

Let's consider a small example where there are 3 piles of stones with the following number of stones in each pile: `piles = [1, 3, 5]`. We will use this example to illustrate the solution approach.

1. At the beginning of the game, the Nim-sum is calculated by performing a bitwise XOR operation of the numbers of stones in each pile: `1 ^ 3 ^ 5`.

2. Performing the calculation gives us `1 ^ 3 = 2` and then `2 ^ 5 = 7`. Since 7 is not equal to 0, Alice can make a move to leave the Nim-sum at 0 for Bob.

3. To find Alice's optimal move, we can look at the binary representation of the pile sizes and compare it with the Nim-sum. Let's do this:

   - The binary representation of the piles: `1 -> 001`, `3 -> 011`, `5 -> 101`
   - The binary representation of the Nim-sum `7`: `111`

4. Alice will try to make a move that results in a new state with a Nim-sum of 0. She can do this by choosing a pile and removing enough stones to turn the current Nim-sum (`111`) to `000`. For example, Alice can remove 4 stones from the third pile (5 stones), which gives us a new state of `[1, 3, 1]`.

5. The new Nim-sum after Alice's move is `1 ^ 3 ^ 1`, which equals `0`, since `1 ^ 1 = 0` and `0 ^ 3 = 3` and finally `3 ^ 0 = 3,` and in binary, `3` is `011`, which will result back to `0` once XOR'ed with itself during Bob's turn.

6. Now it is Bob's turn, and since the Nim-sum is `0`, any move he makes will leave a non-zero Nim-sum for Alice to exploit.

7. If Bob removes 1 stone from any pile, the piles may look like `[1, 2, 1]`. Alice can then remove 2 stones from the second pile, leaving a single stone in two piles: `[1, 1, 1]`, which has a Nim-sum of `0` again.

8. Ultimately, no matter how Bob plays, if Alice starts with a non-zero Nim-sum and both play optimally henceforth, Alice will always be able to return the game to a state where the Nim-sum is `0` at Bob's turn.

Using this example, we can see that the decision on the first move is crucial: given a non-zero Nim-sum, there is always a winning strategy for the player making the first move. Alice wins this game if she follows the strategy of always making a move to return the game to a state with a Nim-sum of `0` at the end of her turn.

Python Solution

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

class Solution:
    def nimGame(self, piles: List[int]) -> bool:
        # Apply memoization to avoid recomputation of the same game states
        @lru_cache(maxsize=None)
        def can_win(st):
            # Convert tuple to list to modify the piles
            piles_list = list(st)
            # Iterate over each pile
            for i, pile in enumerate(piles_list):
                # Try removing 1 to pile stones from the current pile
                for stones_removed in range(1, pile + 1):
                    # Remove the stones for the current move
                    piles_list[i] -= stones_removed
                    # Check if opponent would lose from this state
                    if not can_win(tuple(piles_list)):
                        return True  # If opponent loses, current player wins
                    # Undo the move
                    piles_list[i] += stones_removed
            # If no winning move, return False
            return False

        # Start the game with the initial piles configuration
        return can_win(tuple(piles))

# Example usage:
# sol = Solution()
# result = sol.nimGame([1, 2, 3])  # Pass the initial piles as a list
# print(result)  # Outputs True or False depending on whether you can win the Nim game

```

Java Solution

```java
import java.util.Map;
import java.util.HashMap;

class Solution {
    // Initialize a memoization map to store computed game states
    private Map<Integer, Boolean> memoization = new HashMap<>();
    // 'powersOfEight' array will store the powers of 8 (1, 8, 64, ...) for encoding the game state
    private int[] powersOfEight = new int[8];

    public Solution() {
        // Precompute the powers of 8 up to 8^7
        powersOfEight[0] = 1;
        for (int i = 1; i < 8; ++i) {
            powersOfEight[i] = powersOfEight[i - 1] * 8;
        }
    }

    // Method to determine if the current player can win the nim game given the piles
    public boolean nimGame(int[] piles) {
        return depthFirstSearch(piles);
    }

    // Private method for a recursive depth-first search to simulate all possible moves
    private boolean depthFirstSearch(int[] piles) {
        // Calculate a unique state hash for the current state of the piles
        int stateHash = computeStateHash(piles);
        // If this state has been encountered before, return the stored result
        if (memoization.containsKey(stateHash)) {
            return memoization.get(stateHash);
        }

        // Try reducing each pile by a number of stones and perform DFS on the new state
        for (int i = 0; i < piles.length; ++i) {
            for (int stonesToRemove = 1; stonesToRemove <= piles[i]; ++stonesToRemove) {
                piles[i] -= stonesToRemove;
                // If the opponent cannot win after our move, we found a winning move
                if (!depthFirstSearch(piles)) {
                    piles[i] += stonesToRemove; // Undo the move
                    memoization.put(stateHash, true); // Memoize the winning result
                    return true;
                }
                piles[i] += stonesToRemove; // Undo the move before trying next move
            }
        }

        // If no winning move is found, this is a losing state
        memoization.put(stateHash, false);
        return false;
    }

    // Computes a unique hash for the current state of the piles using powers of 8
    private int computeStateHash(int[] piles) {
        int stateHash = 0;
        // Encode the piles' state into a single integer using base 8 representation
        for (int i = 0; i < piles.length; ++i) {
            stateHash += piles[i] * powersOfEight[i];
        }
        return stateHash;
    }
}

```

C++ Solution

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

using namespace std;

class Solution {
public:
    bool nimGame(vector<int>& piles) {
        unordered_map<int, int> memo; // Store calculated game results to avoid recomputation.
        int powerOfEight[8] = {1}; // Precomputed powers of 8 for state encoding.
        for (int i = 1; i < 8; ++i) {
            powerOfEight[i] = powerOfEight[i - 1] * 8;
        }

        // Function to encode the current state of piles as a single integer.
        auto encodeState = [&](vector<int>& piles) {
            int state = 0;
            for (int i = 0; i < piles.size(); ++i) {
                state += piles[i] * powerOfEight[i];
            }
            return state;
        };

        // Depth-First Search (DFS) algorithm to determine if current player can win.
        function<bool(vector<int>&)> dfs = [&](vector<int>& piles) {
            int currentState = encodeState(piles);
            if (memo.count(currentState)) {
                return memo[currentState]; // Return precomputed result if available.
            }
            for (int i = 0; i < piles.size(); ++i) { // Try each pile.
                for (int j = 1; j <= piles[i]; ++j) { // Try removing 1 to all stones from pile.
                    piles[i] -= j; // Remove stones from the current pile.
                    if (!dfs(piles)) { // If opponent loses from the resulting state...
                        piles[i] += j; // Backtrack to restore pile size.
                        return memo[currentState] = true; // Current player can win.
                    }
                    piles[i] += j; // Backtrack to restore pile size.
                }
            }
            return memo[currentState] = false; // If no winning move found, current player loses.
        };

        return dfs(piles); // Start the game with the initial piles.
    }
};

int main() {
    vector<int> piles = {3, 4, 5};
    Solution solution;
    bool canWin = solution.nimGame(piles);
    // canWin will be true or false depending on whether the player can win or not.
    return 0;
}

```

Typescript Solution

```typescript
// Represents the Nim game initial state
function nimGame(piles: number[]): boolean {
    // Base for representing the state
    const base: number[] = Array(8).fill(1);
    for (let i = 1; i < base.length; ++i) {
        base[i] = base[i - 1] * 8;
    }

    // Converts a pile array into a unique state representation
    const toState = (piles: number[]): number => {
        let state = 0;
        for (let i = 0; i < piles.length; ++i) {
            state += piles[i] * base[i];
        }
        return state;
    };

    // Memoization cache to store known game states
    const memo: Map<number, boolean> = new Map();

    // Recursive function to determine if the current player can win from the given pile configuration
    const canWin = (piles: number[]): boolean => {
        const currentState = toState(piles);
        if (memo.has(currentState)) {
            return memo.get(currentState)!;
        }
        for (let i = 0; i < piles.length; ++i) {
            for (let j = 1; j <= piles[i]; ++j) {
                piles[i] -= j; // Try reducing the current pile by 'j'
                if (!canWin(piles)) { // If the opponent can't win from here, current player wins
                    piles[i] += j; // Restore the pile before memoizing and returning
                    memo.set(currentState, true);
                    return true;
                }
                piles[i] += j; // Restore the pile for the next iteration
            }
        }
        memo.set(currentState, false); // Current player can't win from this state
        return false;
    };

    // Start the game by calling the recursive function with the initial piles configuration
    return canWin(piles);
}

```

Time and Space Complexity

The provided Python code implements a recursive solution to the Nim Game problem with memoization using the `cache` decorator, which caches results of subproblems to avoid repetitive calculations.

Time Complexity

The time complexity of the recursive solution is determined by the number of possible states and the transitions between these states. Since the state is represented by the `st` tuple, which is essentially the current configuration of `piles`, the number of distinct states can be vast. For each state, the `dfs` function iterates over each pile and attempts to reduce the number of stones in each pile by every possible number from `1` to `x` (the size of the pile).

Let's suppose:

- `n` is the number of piles.
- `m` is the maximum size of a single pile.

In the worst case, each pile can be reduced to zero in `m` different ways, which gives us `m^n` possible states (assuming piles can have different maximum sizes, in practice it's less due to combining identical pile sizes). The `dfs` function is called once for each transition between these states, and because of memoization, each state is computed only once. Hence, the **time complexity** is `O(m^n)` but in practice, it can be significantly less due to memoization and the fact that not all transitions occur due to pruning (when a winning state is found, further exploration of that branch is stopped).

Space Complexity

The **space complexity** is determined by the maximum size of the recursion stack and the space required to store the memoization cache. The recursion depth can go as deep as the number of possible transitions `m^n`, in the worst case. The cache will need to store a result for each unique state of the `piles`, thus requiring `O(m^n)` space as well. Therefore, the total **space complexity** is `O(m^n)`.

In summary:

- Time Complexity: `O(m^n)`
- Space Complexity: `O(m^n)`

Please note that in practice, the actual complexities may be lower due to memoization and pruning as explained above.