# 1510. Stone Game IV

Hard Math Dynamic Programming Game Theory

## Problem Description

Alice and Bob are playing a turn-based game where they remove stones from a pile. The game begins with `n` stones in the pile and in each turn, a player removes a square number of stones (like 1, 4, 9, 16, ...) from the pile. The players take alternate turns, with Alice always starting the game. The objective of the game is not to be the person who is unable to make a move (meaning that player can't remove a square number of stones because it doesn't exist in the pile). If a player can't make a move, they lose. The problem asks us to determine if Alice can win the game, assuming both Alice and Bob play optimally (making the best moves possible at every turn).

## Intuition

To solve this game algorithmically, we can analyze it as a recursive problem, where each state of the game (the number of stones left in the pile) can lead to several possible next states (depending on which square number of stones is taken). The key insight is to recognize that the game has optimal substructure, meaning the optimal decision at a given state depends only on the states reachable from it and not on the path taken to reach that state. We can use dynamic programming to avoid recomputing the outcomes of these substates.

We want to determine if Alice can guarantee a win given `n` stones. We can do this by simulating each player's moves recursively and memoizing (caching) the results to make the solution efficient. The base case of our recursive function is when there are 0 stones left, meaning the current player loses (since they can't make a move).

During Alice's turn (or Bob's), we check all possible moves (removing a square number of stones) and recursively call the function to simulate the opponent's turn with the remaining stones. If there's at least one move that leads Bob to a losing state (he can't force a win from that state), Alice wins by taking that move.

The `dfs` function in the provided solution is a typical depth-first search with memoization (caching) that implements the above logic. The `cache` decorator is used to memoize the results of the recursive calls, storing whether a particular number of stones leads to a win (True) or loss (False). During the recursion, if we find a state (number of stones) that makes the next player lose, we return True (indicating the current player can win).

In summary, we use depth-first search with memoization to check every possible move, remembering the outcomes of sub-problems to determine if Alice can win with `n` starting stones.

## Solution Approach

The solution uses a recursive function that employs depth-first search (DFS) to explore the state space of the game and memoization to save the results of subproblems. Let's walk through the implementation as found above in the `Solution` class:

1. The `winnerSquareGame` function is the starting point that takes an integer `n`, denoting the number of stones in the pile initially.
2. It defines a nested function `dfs` which is a recursive function designed to return True if the current player can force a win with `i` stones remaining, otherwise it returns False.
3. The `dfs` function is decorated with `@cache` from Python's standard library. This decorator automatically memoizes the results of the `dfs` function to avoid recalculating the same subproblems. Memoization is a key feature that improves the efficiency of the solution by storing the outcome of each state after it is computed for the first time.
4. Inside `dfs`, the base case is checked: if `i == 0`, the function returns False as the current player cannot make a move and thus loses.
5. If the base case is not met, the function enters a loop to try all possible square number moves (`j * j`) that are less than or equal to the current stone count (`i`).
6. For each possible move, it performs a recursive call: `dfs(i - j * j)`. This call represents the next player's turn with the remaining stones after the current player removes `j * j` stones.
7. The key part of the logic is checking if `not dfs(i - j * j)`. If the result is True, this means that the opponent (next player) will lose in the state after the current move. Since the opponent loses, the current player wins, so the function returns True.
8. If none of the potential moves lead to a winning state (the opponent can force a win no matter what the current player does), the loop finishes and the function returns False, indicating that the current player cannot force a win from this state.
9. Finally, the initial call `return dfs(n)` kicks off the recursive process for the initial state where Alice starts with `n` stones.

The recursive depth-first search, coupled with memoization, is a classic dynamic programming approach, and it is very effective in this case for solving combinatorial game problems. The solution systematically explores all possible outcomes and is able to avoid redundant work thanks to the caching mechanism, making it feasible to compute the answer for larger values of `n`.

### Example Walkthrough

Let's take `n = 7` stones as an example to illustrate how the solution algorithm works. According to the rules, Alice can only take square numbers of stones - like 1, 4, or 9, and so on. Since we are concerned with the number 7, the only square numbers less than or equal to 7 are 1 (1*1) and 4 (2*2).

1. Alice starts the game and has the option to take 1 or 4 stones.
2. If Alice takes 1 stone, we're left with 6 stones. Now, it's Bob's turn. The recursive function (`dfs`) will try to determine if Bob can win given that 6 stones remain.
3. If Alice takes 4 stones instead, 3 stones are left. Bob has only one choice here, which is to take 1 stone (as 4 is too many for the remaining 3), leaving 2 stones.
4. If at any point the `dfs` function is called with 0, it would return False, indicating the current player lost the game (as they can't make a move).

Let's trace the recursive calls from Alice's perspective when she takes 1 stone:

- `dfs(7)` - Alice's turn.
  - Alice takes 1: `dfs(6)` - Bob's turn.
    - Bob can take 1: `dfs(5)` - Alice's turn.
      - If Alice takes 1: `dfs(4)` - Bob's turn.
        - If Bob takes 1: `dfs(3)` - Alice's turn.
          - Alice will have no choice but to leave a losing state for Bob eventually, as none of the paths from this state lead to a win for her.
        - If Bob takes 4: `dfs(0)` - Alice wins (Bob loses since no stones are left).
      - If Alice takes 4: `dfs(1)` - Bob's turn.
        - No matter what Bob does (taking 1), he leaves a win for Alice.

From this example, with 7 stones, we can see that if Alice starts by taking 1 stone and at each step makes sure to leave a number of stones such that no square number can subtract to zero, she will win. The recursive calls will discover these winning strategies through its depth-first exploration, and the memoization will ensure we don't recompute the results for subproblems we've already seen.

Alice's winning strategy will look like this:

- Alice takes 1 (6 stones left).
- Bob could take 1 (5 stones left) or 4 (2 stones left).
- If 5 stones are left, Alice would take 4 (1 left), forcing Bob to take the last stone and lose.
- If 2 stones are left, Alice just needs to take 1, and Bob will be forced to take the last and lose.

By running through all possible scenarios like this, the algorithm concludes that Alice can win when the game starts with 7 stones.

## Solution Implementation

### Python

```python
class Solution:
    def winnerSquareGame(self, n: int) -> bool:
        from functools import lru_cache

        # Using memoization decorator to cache results of subproblems
        @lru_cache(maxsize=None)
        def can_win(remaining_stones: int) -> bool:
            # A game state where no stones a left means a losing state
            if remaining_stones == 0:
                return False

            square_root = 1
            # Iterate through all possible square numbers less than or equal to the remaining stones
            while square_root ** 2 <= remaining_stones:
                # If the opponent loses in any following state, the current player wins
                if not can_win(remaining_stones - square_root ** 2):
                    return True
                square_root += 1

            # If none of the moves lead to a win, then the current state is losing
            return False

        # Initiate the game with 'n' stones
        return can_win(n)
```

### Java

```java
class Solution {
    // This array is used for memoization to store the results of subproblems
    private Boolean[] memo;

    // This method starts the recursive depth-first search for finding the winner of the game
    public boolean winnerSquareGame(int n) {
        memo = new Boolean[n + 1]; // Initialize the memoization array
        return dfs(n); // Start the recursive DFS from the given number 'n'
    }

    // This helper method performs the depth-first search to determine if the current player can win
    private boolean dfs(int number) {
        // Base case: if the number is 0, the current player can't make a move and thus loses
        if (number == 0) {
            return false;
        }
        // Check if the result for the current number is already computed
        if (memo[number] != null) {
            return memo[number];
        }
        // Try all possible square numbers starting from 1 to the largest square number <= 'number'
        for (int squareRoot = 1; squareRoot * squareRoot <= number; ++squareRoot) {
            // Subtract the square of the current square root from 'number' to get the residual value
            // Pass the residual value to the recursive call for the opponent's turn
            int nextNumber = number - squareRoot * squareRoot;
            if (!dfs(nextNumber)) {
                // If the opponent loses on the residual value, the current player wins
                return memo[number] = true;
            }
        }
        // If after all attempts there's no winning strategy, the current player loses
        return memo[number] = false;
    }
}
```

### C++

```cpp
class Solution {
public:
    bool winnerSquareGame(int n) {
        // t represents the memoization array where
        // 0 means uncomputed, 1 means current player can win, -1 means current player can't win
        int memo[n + 1];
        memset(memo, 0, sizeof(memo)); // Initialize memoization array to 0

        // Define a recursive depth-first search function to determine if a player can win
        function<bool(int)> dfs = [&](int remainingStones) -> bool {
            if (remainingStones <= 0) { // Base case: no stones left
                return false;
            }
            if (memo[remainingStones] != 0) { // If already computed, return the stored result
                return memo[remainingStones] == 1;
            }
            // Try every square number less than or equal to the remaining stones
            for (int square = 1, squareRoot = 1; square <= remainingStones; ++squareRoot) {
                // If the opponent loses in the next state, the current player wins
                if (!dfs(remainingStones - square)) {
                    memo[remainingStones] = 1; // Mark the current state as winning for the current player
                    return true; // The current player can force a win
                }
                square += squareRoot * 2 + 1; // Update to the next square number
            }
            memo[remainingStones] = -1; // Mark the current state as losing for the current player
            return false; // The current player cannot force a win
        };

        // Call the dfs function with the total number of stones 'n' to determine if the player can win
        return dfs(n);
    }
};
```

### TypeScript

```typescript
// Determines if the player who starts with the game with 'n' stones is guaranteed to win.
// If a winning situation for any player occurs if they can force their opponent into a losing state.
// In other words, if there's any move available that leaves the opponent with a combination of stones
// that is already determined to be losing, the current player wins.
function winnerSquareGame(stoneCount: number): boolean {
    // Create an array to cache interim results where f[i] represents the winning
    // condition for a game starting with i stones. Initialize all values to false.
    const winningCache: boolean[] = new Array(stoneCount + 1).fill(false);

    // Iterate over each possible number of stones.
    for (let currentStoneCount = 1; currentStoneCount <= stoneCount; ++currentStoneCount) {
        // Check every square number up to the current stone count.
        for (let square = 1; square * square <= currentStoneCount; ++square) {
            // If the current player can make a move that puts the opponent in a losing situation
            // (winningCache[currentStoneCount - square * square] is false), then the current
            // player is in a winning situation for currentStoneCount.
            if (!winningCache[currentStoneCount - square * square]) {
                winningCache[currentStoneCount] = true;
                // As soon as a winning move is found, no need to check further moves.
                break;
            }
        }
    }
    // Return the winning indication for the game starting with the original 'stoneCount'.
    return winningCache[stoneCount];
}
```

```python
class Solution:
    def winnerSquareGame(self, n: int) -> bool:
        from functools import lru_cache

        # Using memoization decorator to cache results of subproblems
        @lru_cache(maxsize=None)
        def can_win(remaining_stones: int) -> bool:
            # A game state where no stones a left means a losing state
            if remaining_stones == 0:
                return False

            square_root = 1
            # Iterate through all possible square numbers less than or equal to the remaining stones
            while square_root ** 2 <= remaining_stones:
                # If the opponent loses in any following state, the current player wins
                if not can_win(remaining_stones - square_root ** 2):
                    return True
                square_root += 1

            # If none of the moves lead to a win, then the current state is losing
            return False

        # Initiate the game with 'n' stones
        return can_win(n)
```

## Time and Space Complexity

The given Python code aims to determine if a player can win a game where they can remove any square number of stones from a total of `n` stones. The code uses a depth-first search (DFS) algorithm with memoization to prevent recomputing states that have already been solved. The memoization is implemented using the `cache` decorator from Python's `functools`.

### Time Complexity

The time complexity of the solution depends on the number of distinct states `i` that will be passed to the `dfs` function and the number of iterations within each function call.

- Since each state from `0` to `i` is computed only once due to memoization, there will be `n` distinct states.
- For each state `i`, the inner while loop runs for $O(\sqrt{i})$ times since it enumerates through all possible square numbers that are less than or equal to `i`.

Combining these factors, the overall time complexity of the recursive calls can be represented by the sum of square roots for all numbers from 1 to `n`, which can be approximated by an integral from 1 to `n`, resulting in $O(n^{3/2})$.

### Space Complexity

The space complexity of the solution includes the space used by the recursion stack and the memoization cache.

- The recursion stack will use $O(\sqrt{n})$ space since the maximum depth of recursion is limited by the maximum number of perfect squares you can subtract from `n` consecutively before reaching zero.
- The memoization cache will hold a result for each state from `0` to `i`, therefore requiring $O(n)$ space.

As a result, the overall space complexity is $O(n)$ because the cache dominates the space complexity due to storing the result for each state compared to the stack space in depth.