464. Can I Win

Medium Bit Manipulation Memoization Math Dynamic Programming Bitmask Game Theory

Leetcode Link

Problem Description

In this leetcode problem, the "100 game" is used as an example of a two-player game where the players alternate turns, adding an integer from a specified range to a cumulative total. The objective of the game is to be the player who makes the running total reach or exceed a target value, known as the `desiredTotal`. The twist in this version of the game is that once a number has been used, it cannot be used again (without replacement). The task is to determine whether the first player can guarantee a win given complete access to the entire pool of numbers from 1 to the `maxChoosableInteger` and assuming both players make the most optimal move available to them at each turn. The problem asks to return `true` if the first player can ensure a victory and `false` otherwise.

Intuition

To determine if the first player has a winning strategy, we need a way to evaluate all possible game states without re-evaluating positions multiple times. We can use recursion to simulate each player's turns, where each state represents a unique combination of remaining numbers and the current total.

The state of the game is tracked using a bitmap where bits represent whether a particular number has been chosen already. Starting from an initial state where no numbers have been chosen and the current total is 0, we explore all possible moves for the first player, recursively simulating the opponent's response to each move.

If we can find any move such that the resulting state either:

1. Reaches/exceeds the `desiredTotal`—immediately winning the game, or
2. Forces the opponent into a position where they cannot win (every subsequent move leads to a loss),

then the first player can guarantee a win.

However, a naive recursive approach could re-compute the same state multiple times. To avoid this, we use memoization to cache the results of subproblems, which ensures each unique game state is only computed once. This is done using the `@cache` decorator in the Python code.

The base condition to end the recursion is reaching a state where the current total is greater than or equal to the `desiredTotal`, or if there are no moves left that could prevent the opponent from winning on their next turn. If no winning strategy is found for any possible move, the function returns `false`, indicating the first player cannot guarantee a win.

Before starting the recursive approach, there's an initial check to see if the sum of all chooseable integers is less than the `desiredTotal`. If so, it's impossible for either player to reach the target score, and the function immediately returns `false`.

Solution Approach

The implementation of the solution employs a Depth-First Search (DFS) algorithm combined with memoization. The DFS algorithm explores all possible game moves recursively, while memoization is used to cache results of previously encountered game states to avoid recalculations.

Here's a step-by-step explanation of the solution approach:

- **State Representation**: We use an integer (`state`) to represent the current state of the game, where each bit in `state` indicates whether a number has been chosen. If the i-th bit is set, the number i has already been selected.
- **Recursion with DFS**: The `dfs(state, t)` function is the heart of the DFS approach, where `state` represents the current game state and `t` the current sum. For each call of `dfs`, we loop through all integers from 1 to `maxChoosableInteger` to simulate choosing a new number.
- **Checking Valid Moves**: For each move, check if the corresponding bit in `state` is not set, meaning the number hasn't been used yet. (`state >> i) & 1` does this check.
- **Winning Conditions**: Upon choosing a number i, we add it to the current sum `t`. If this new sum `t + i` is greater than or equal to `desiredTotal`, we found a winning move. Alternatively, if the recursive call `dfs(state | 1 << i, t + i)` returns `false`, it means the opponent cannot win after we make this move, so it's also a winning move for us.
- **End of Recursion**: If none of the moves lead to a winning situation, the function will eventually return `false`, indicating that with optimal play from the opponent, the first player cannot win from this specific state.
- **Memoization**: We use the `@cache` decorator on the `dfs` function to cache the results. Whenever the `dfs` function is called with a state that has been computed before, it will return the cached result instead of recalculating.
- **Initial Check**: Before invoking the DFS, we calculate the sum s of all numbers that could be chosen. If s is less than `desiredTotal`, it's impossible for either player to reach the target, and the function immediately returns `false`.
- **Return Statement**: Finally, if the initial sum check passes, the function returns the result of the initial `dfs(0, 0)`, signifying whether the first player has a winning strategy starting from an empty state with a sum of 0.

Mathematically, the memoization ensures that the time complexity of the algorithm is reduced to $O(2^n)$, where n is `maxChoosableInteger`, because we only need to compute each of the 2^n possible states once.

Example Walkthrough

Let's use the "100 game" example and walk through a smaller case to understand the solution approach. Assume the `desiredTotal` is 10 and the `maxChoosableInteger` is 4. This means each player can choose from integers 1 to 4 without replacement to reach or exceed the total of 10.

1. **Initial Check**: We calculate the sum of all numbers from 1 to 4, which is $1+2+3+4=5+5=10$. Since 10 is greater than 10, it's possible to reach the desired total, so we proceed with the DFS approach.
2. **First Recursive Call**: We call `dfs(0, 0)` because initially, no numbers have been chosen (`state` is 0) and the running total (`t`) is also 0.
3. **Exploring Moves**: The first player begins by exploring all numbers from 1 to 4:
   - If Player 1 chooses 1, the state changes to `00001` ($1 << (1 - 1)$), and t becomes 1. The function then calls `dfs(00001, 1)`.
   - If Player 1 chooses 2, the state becomes `00010` and t becomes 2, leading to a call of `dfs(00010, 2)`.
   - This process continues until all possible first moves are explored.
4. **Recursing Deeper**: Let's consider Player 1 had chosen 1.
   - Now it's Player 2's turn with state `00001` and total 1. Player 2 also explores moves 2 through 4.
   - If Player 2 chooses 2 next, the state becomes `00011` (`00001 | 1 << (2 - 1)`) and t becomes 3. The function then calls `dfs(00011, 3)`.
5. **Checking Winning Condition**: Assume Player 1 plays 1, then Player 2 plays 2, and Player 1 follows with 3 on the next turn. Now, the state is `00011` for 1 and 2 being used, and the running total is $1+2+6 = 9$.
   - On the next turn, if there exists any number that Player 1 can choose that results in a total of 10 or more, Player 1 will win. Player 1 can choose 4 which results in a total of 10. Since `dfs` for this state returns `true`, Player 1 has a winning strategy.
6. **Memoization**: If at any point, the `dfs` function encounters a state that it has been before, instead of recomputing, it will use the cached result. This dramatically reduces the number of computations needed.
7. **Conclusion**: Using the DFS and memoization, we evaluate all possible sequences of moves. If Player 1 has a guaranteed strategy to reach or exceed the desired total of 10, `dfs(0, 0)` will ultimately return `true`. If each recursive branch eventually leads to a state where Player 1 cannot win, the function will return `false`.

For our small example, an optimal strategy exists for Player 1 to win by choosing the sequence 1, 4, and 3 (or several other combinations), resulting in reaching the `desiredTotal` of 10. So, the output for `dfs(0, 0)` would be `true`, indicating Player 1 can guarantee a win.

Python Solution

```python
from functools import lru_cache  # Import the lru_cache decorator for memoization

class Solution:
    def canIWin(self, max_choosable_integer: int, desired_total: int) -> bool:
        # Use memoization to store results of subproblems and avoid recomputation
        @lru_cache(maxsize=None)
        def can_win_state(state: int, current_total: int) -> bool:
            # Iterate through all possible integers that can be chosen
            for integer in range(1, max_choosable_integer + 1):
                # Check if 'integer' has already been chosen in the 'state'
                if state & (1 << (integer - 1)) == 0:
                    # Choose 'integer' if this iteration as 'integer' is already taken
                    # Check if choosing 'integer' now would win the game, or the opponent cannot win after we choose 'integer'
                    if current_total + integer >= desired_total or not can_win_state(state | (1 << integer), current_total + integer):
                        return True  # Winning move found, current player wins
            return False  # No winning move found; current player loses

        # Compute the sum of all integers that can be chosen
        sum_of_integers = (max_choosable_integer * (max_choosable_integer + 1)) // 2
        # If the sum is less than the desired total, no one can win
        if sum_of_integers < desired_total:
            return False

        # Start the game with the initial state (all zeros) and total 0
        return can_win_state(0, 0)

# Example of using the code:
# Create a Solution object
solution = Solution()

# Set the maximum choosable integer and desired total
max_choosable_integer = 10
desired_total = 11

# Call the canIWin method and print the result
print(solution.canIWin(max_choosable_integer, desired_total))
print("Can I win?", result)
```

Java Solution

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

class Solution {
    // A memoization map to store previously computed results for given states
    private Map<Integer, Boolean> memo = new HashMap<>();

    // Main method to determine if the player can win given the max number and desired total
    public boolean canIWin(int maxChoosableInteger, int desiredTotal) {
        // Calculate the sum of all choosable integers
        int sumOfAllIntegers = (1 + maxChoosableInteger) * maxChoosableInteger / 2;

        // If the sum is less than the desired total, no one can win
        if (sumOfAllIntegers < desiredTotal) {
            return false;
        }

        // Start the depth-first search to determine if the player can win
        return depthFirstSearch(0, 0, maxChoosableInteger, desiredTotal);
    }

    // Helper method for depth-first search
    private boolean depthFirstSearch(int usedNumbersState, int currentTotal, int maxChoosableInteger, int desiredTotal) {
        // If the result for this state has been computed before
        if (memo.containsKey(usedNumbersState)) {
            return memo.get(usedNumbersState);
        }

        // By default, assume the player cannot win with the current state
        boolean canWin = false;

        // Loop through all choosable integers
        for (int i = 1; i <= maxChoosableInteger; ++i) {
            // Check if the number has not been chosen yet (bit is not set)
            if ((usedNumbersState >> i) & 1) == 0) {
                // If choosing number i reaches or exceeds desiredTotal or the opponent cannot win,
                // then the current player can win by choosing i
                if (currentTotal + i >= desiredTotal
                    || !depthFirstSearch(usedNumbersState | (1 << i), currentTotal + i, maxChoosableInteger, desiredTotal)) {
                    canWin = true;
                    break;  // Terminate the loop as we found a winning situation
                }
            }
        }

        // Store the computed result in the memo map before returning it
        memo.put(usedNumbersState, canWin);
        return canWin;
    }
}
```

C++ Solution

```cpp
#include <unordered_map>
using namespace std;

class Solution {
public:
    // Determines if the current player can win the game.
    bool canIWin(int maxChoosableInteger, int desiredTotal) {
        // Calculate the sum of all choosable integers.
        int sum = maxChoosableInteger * (maxChoosableInteger + 1) / 2;
        // Check if the sum is less than the desired total, which means no one can win
        if (sum < desiredTotal) return false;

        // Prepare a memoization table to store intermediate results
        unordered_map<int, bool> memo;

        // Call to the recursive function to determine if we can win
        return dfs(0, 0, maxChoosableInteger, desiredTotal, memo);
    }

private:
    // Recursive function to check if we can reach the desired total from the current state.
    bool dfs(int state, int currentTotal, int maxChoosableInteger, int desiredTotal, unordered_map<int, bool>& memo) {
        // Check if the current state has been computed before
        if (memo.count(state)) return memo[state];

        // Initialize the result as false
        bool result = false;

        // Try every choosable integer to see if we can win
        for (int i = 1; i <= maxChoosableInteger; ++i) {
            // Skip if i is already used in the current state
            if ((state >> i) & 1) continue;

            // If adding i to currentTotal meets or exceeds desiredTotal, or the opponent cannot win,
            // then the current player can win from the current state.
            if (currentTotal + i >= desiredTotal || !dfs(state | (1 << i), currentTotal + i, maxChoosableInteger, desiredTotal, memo)) {
                result = true;
                break;
            }
        }

        // Memoize and return the result for the current state
        memo[state] = result;
        return result;
    }
};
```

Typescript Solution

```typescript
// Importing the necessary utilities from 'collections' module
import { HashMap } from 'collections';

// Global memoization table to store intermediate results
const memo: HashMap<number, boolean> = new HashMap<number, boolean>();

// Determines if the current player can win the game given the max choosable integer and the desired total
const canIWin = (maxChoosableInteger: number, desiredTotal: number): boolean => {
    // Calculate the sum of all choosable integers to check if it's possible to win
    const sum: number = ((1 + maxChoosableInteger) * maxChoosableInteger) / 2;
    // No one can win if the sum is less than the desired total
    if (sum < desiredTotal) return false;

    // Call to the recursive function to determine if the current player can win
    return dfs(0, 0, maxChoosableInteger, desiredTotal);
};

// Recursive function to check if the current player can reach the desired total from the current state
const dfs = (state: number, currentTotal: number, maxChoosableInteger: number, desiredTotal: number): boolean => {
    // Check for an existing computation for the current state
    if (memo.has(state)) return memo.get(state)!;

    // Initialize the result as false
    let result = false;

    // Iterate through every choosable integer to find a winning move
    for (let i = 1; i <= maxChoosableInteger; ++i) {
        // Check if 'i' has been used in the current state
        if ((state >> i) & 1) continue;

        // Check if adding 'i' to currentTotal wins the game, or if the opponent can win on the next turn
        if (currentTotal + i >= desiredTotal || !dfs(state | (1 << i), currentTotal + i, maxChoosableInteger, desiredTotal)) {
            result = true;
            break; // Current player wins, so we break the loop
        }
    }

    // Save the result to the memoization table for the current state
    memo.set(state, result);

    // Return the computed result
    return result;
};
```

The TypeScript code provided does not include a direct equivalent for `unordered_map` from C++ which is used for memoization; instead, I've used `HashMap` from the 'collectable' library which is a TypeScript-compatible library providing persistent immutable data structures. If `collectable` is not preferred, you may use a regular JavaScript `Map` object or a plain object with some adjustments to the hashing strategy for the state.

To use TypeScript's `Map` object instead:

```typescript
// Replace the import statement with the following line
const memo = new Map<number, boolean>();
// And replace each HashMap method as follows:
// memo.has(state) -> memo.has(state)
// memo.get(state)! -> memo.get(state)!
// memo.set(state, result) -> memo.set(state, result)
```

Time and Space Complexity

Time Complexity

The time complexity of the `canIWin` function is dependent on two factors: the number of recursive calls made by the `dfs(state, t)` function and the number of iterations within each call.

Each state in the `dfs` function represents a unique combination of chosen integers and can be represented by a bitmask, where the i-th bit is chosen if the i-th bit is set to 1. There are $2^{\text{maxChoosableInteger}}$ possible states because each of the `maxChoosableInteger` integers can be either chosen or not chosen.

For each call of `dfs`, we iterate from 1 to `maxChoosableInteger` to try all possibilities, which gives us $O(\text{maxChoosableInteger})$ for each call.

However, we don't visit all possible states of `dfs` due to the game ending once the `desiredTotal` is reached or exceeded. Plus, memoization caching `cache` ensures that we only compute each state once.

Considering this, the worst-case time complexity is $O(\text{maxChoosableInteger} \times 2^{\text{maxChoosableInteger}})$ since there are `maxChoosableInteger` operations done in each of the $2^{\text{maxChoosableInteger}}$ possible states.

Space Complexity

The space complexity is governed by the cache used to store the results for each state and the stack space used by the recursion.

1. **Cache**: Storage for each unique state requires $O(2^{\text{maxChoosableInteger}})$ space as there are that many possible states.
2. **Recursion Stack**: In the worst case, the recursion can go as deep as `maxChoosableInteger` levels if we continue to choose a new number until we run out, which is $O(\text{maxChoosableInteger})$ space.

Hence, the overall space complexity is also $O(\text{maxChoosableInteger} \times 2^{\text{maxChoosableInteger}})$, which simplifies to $O(2^{\text{maxChoosableInteger}})$ as the exponential term dominates the linear term.