# 294. Flip Game II

Medium | Memoization | Math | Dynamic Programming | Backtracking | Game Theory | Leetcode Link

## Problem Description

In the Flip Game, you are given a string `currentState` which consists of only '+' and '-'. A move in this game consists of turning two adjacent '+' into '--'. The game ends when a player can't make any more moves, implying the last person to make a valid move wins.

Your objective is to determine if the starting player can always win, regardless of the moves the opposing player makes. If the starting player can guarantee a win, your function should return `true`. If there's no strategy that guarantees a win for the starting player, return `false`.

It's a game of strategic choice, where each decision can affect the outcome. The challenge is to find whether there exists a sequence of moves that, if played optimally, ensures victory for the first player.

## Intuition

The solution to this problem lies in using backtracking to consider all possible moves of the game. This is done by using Depth First Search (DFS) to explore every potential game state that can result from a series of valid moves.

The backtracking approach iterates over the string, and for each pair of consecutive '+' symbols, flips them to '--' and then recursively checks if the other player would lose from the new game state. The key observation is that if there is at least one move after which the other player is guaranteed to lose (they cannot make any move that guarantees their win), then the current player is guaranteed to win.

The code uses a bitmask to represent the game state, where a '1' bit represents a '+', and '0' represents a '-'. A mask is created by setting bits for each '+' in the input `currentState`. The `dfs` function then checks each position in the string. If the current and next positions are '++', it flips them and recursively checks if this new state would lead to a losing state when it's the opponent's turn. If the recursive call returns `false`, meaning the opponent can't win from that state, then the current player can win by making this move.

The use of memoization (`@cache`) avoids re-computation of the states that have already been computed, significantly optimizing the solution. Without memoization, the solution would be too slow, as it would explore the same states multiple times.

Through this approach, we can determine whether the starting player can guarantee a win by meticulously searching through all possible game states and making the optimal move at each step.

## Solution Approach

The initial approach to solving the problem involves a recursive Depth First Search (DFS) strategy to explore each possible state of the game after a move has been made, combined with bitmasking to efficiently represent and manipulate the game states, and memoization to avoid recalculating results for previously explored game states.

Here's an in-depth walk-through of the code provided:

1. **Bitmask Representation**: A `mask` is created where each bit represents a position in the `currentState` string. A '1' in the bitmask corresponds to a '+' in the string, and a '0' corresponds to a '-'.
2. **Recursive DFS**: The core of the solution is the `dfs` function, which tries to simulate each possible move recursively.
   - For each call to `dfs`, we iterate through all positions in the string (except the last one since we are checking pairs of characters).
   - If the current bit and the next bit are set to '1' (indicating "++" in the original string), we flip these two bits to '0' (making them "--") and call `dfs` on the new mask.
   - The flip is executed using the XOR operation `mask ^ (1 << i) ^ (1 << (i + 1))` which flips the current bit and the bit next to it.
   - If the recursive `dfs` call returns `false`, it means that the opponent can't win from that modified game state, so the current player will indeed win if they make this move.
3. **Memoization**: The `@cache` decorator is Python's built-in way to employ memoization. With the help of this memoization, any call to the `dfs` function with the same mask is computed only once, and the result is stored for subsequent calls with the same argument. This dramatically reduces the number of computations required and is essential for making this recursive solution feasible.
4. **Winning Condition**: Once all possible moves are explored by recursive calls, if the function finds a move that guarantees a win (as the opponent loses), it returns `true`. If no such move exists (i.e., every move results in a winning position for the opponent), the `dfs` returns `false`.
5. **Result**: The `canWin` function initializes the setup (bitmask, string length) and makes the initial call to `dfs(mask)`. It returns the result of this call. If `dfs` returns `true`, it means the starting player can guarantee a victory with the right strategy.

In summary, this solution uses DFS to simulate the game, bitmasking for efficient state representation, and memoization to optimize the search process. It relies on the principle that if a player can force at least one scenario where the opponent cannot win after their next move, then the current player can secure a win.

### Example Walkthrough

Let's walk through a small example to illustrate the solution approach described in the content provided. Assume we are given the `currentState` string as `"++--+++"`.

1. **Bitmask Representation**: We begin by representing this `currentState` as a bitmask:
   - `++--+++` corresponds to the bitmask `1100111`. Each '1' represents a '+' and '0' represents a '-'.
2. **Initial Call**: Call the `dfs` function on this initial mask `1100111`.
3. **Recursive DFS**: Inside the `dfs` function, we explore all possible moves:
   - We check pairs of bits from left to right.
   - On encountering `11` (which means "++"), we flip them to `00` (creating "--") and recursively call `dfs`.
4. **Exploration**:
   - First Recursion: Let's flip the first two pluses. Our mask becomes `0000111`.
   - `dfs(0000111)` is called, and then it will check for moves within this new state.
   - Second Recursion: In the new state, we flip the last two pluses. Our mask becomes `0000001`.
   - `dfs(0000001)` is called but no moves can be made within this state, so this returns false, which implies that the opponent cannot win from this state.
5. **Backtracking**:
   - Since `dfs(0000001)` returned false, that means `dfs(0000111)` should return true, indicating that there is a move that ensures the parent state win.
   - The process continues by backtracking and exploring other paths to check if there is any sequence of moves that would lead to the starting player's loss.
   - However, because we can guarantee at least one winning move from our initial state, in this case, flipping the first two pluses, we already know the starting player can secure a win.
6. **Result**: After exploring all possible moves and finding that there is at least one sequence of moves leading the opposing player to a state where they cannot move, we determine that the starting player can indeed guarantee a win. Thus, the `dfs(mask)` will ultimately return `true` for the initial mask `1100111`.

In this case, the function `canWin` with the input `"++--+++"` would return `true`, signifying that the starting player has a winning strategy.

## Python Solution

```python
class Solution:
    def canWin(self, current_state: str) -> bool:
        from functools import lru_cache

        # Using lru_cache for memoization to optimize the recursive function
        # It will remember the results of function calls with particular arguments
        @lru_cache(maxsize=None)
        def can_opponent_win(flip_mask):
            # Iterate over all pairs of consecutive positions
            for i in range(n - 1):
                # If current position and the next are not both '+', skip this iteration
                if (flip_mask & (1 << i)) == 0 or (flip_mask & (1 << (i + 1)) == 0):
                    continue
                # Check if by flipping this pair of '+' the opponent can win
                if can_opponent_win(flip_mask ^ (1 << i) ^ (1 << (i + 1))):
                    # If the opponent can win with this flip, continue to the next iteration
                    continue
                # If we found a flip that doesn't let the opponent win, then the current player can win
                return True
            # If no flip makes the current player win, then they cannot win
            return False

        # Initialize the flip_mask and the length of the current state
        flip_mask, n = 0, len(current_state)
        # Build the initial mask to represent '++' as 1's in flip_mask
        for i, c in enumerate(current_state):
            if c == '+':
                flip_mask |= 1 << i

        # Start the recursive checking with the initial configuration of flip_mask
        return not can_opponent_win(flip_mask)

# An example of how to use the class.
solution_instance = Solution()
print(solution_instance.canWin("++++"))  # Output will depend on the state of the game.
```

## Java Solution

```java
class Solution {
    private int boardSize; // size of the string representing the game board
    private Map<Long, Boolean> memoization = new HashMap<>(); // memoization map for storing intermediate results

    // Main method that checks if the current player can win given the current state of the game board
    public boolean canWin(String currentState) {
        long bitMask = 0; // bitmask to represent the game board ('+' as 1 and '-' as 0)
        boardSize = currentState.length(); // store the length of the game board
        // Convert the game board string into a bitmask representation
        for (int i = 0; i < boardSize; ++i) {
            if (currentState.charAt(i) == '+') {
                bitMask |= 1L << i;
            }
        }
        return canPlayerWin(bitMask);
    }

    // Recursive depth-first search method to determine if the current player can win
    private boolean canPlayerWin(long bitMask) {
        // Check the memoization map for precomputed result for the current bitmask
        if (memoization.containsKey(bitMask)) {
            return memoization.get(bitMask);
        }
        // Check each pair of consective '+' signs for a possible move
        for (int i = 0; i < boardSize - 1; ++i) {
            // Check if two consecutive '+' signs are present at index i and i+1
            if ((bitMask & (1L << i)) != 0 && (bitMask & (1L << (i + 1))) != 0) {
                // Flip the two '+' to '-' (by toggling the bits) and continue the search
                if (!canPlayerWin(bitMask ^ (1L << i) ^ (1L << (i + 1)))) {
                    memoization.put(bitMask, true); // Memoize the winning move
                    return true; // Current player can win with this move
                }
            }
        }
        memoization.put(bitMask, false); // Memoize that the current player cannot win
        return false; // Current player cannot win with any move
    }
}
```

## C++ Solution

```cpp
using ll = long long; // Define 'll' as an alias for long long type

class Solution {
public:
    int boardSize; // This variable will hold the size of the game board
    unordered_map<ll, bool> memo; // A memoization map to store intermediate results

    // Main function to determine if we can win with the given board state
    bool canWin(string currentState) {
        boardSize = currentState.size(); // Set the size of the board based on the input string
        ll mask = 0; // This mask will represent our game board's state in binary

        // The loop below will convert the current board state into a binary representation
        for (int i = 0; i < boardSize; ++i) {
            if (currentState[i] == '+') {
                mask |= 1ll << i; // Set the bit to 1 at ith position if there is a '+' in the current state
            }
        }

        // Call the recursive function to determine if we can win from the initial state
        return dfs(mask);
    }

    // Recursive depth-first search (DFS) function to explore the game states and determine win possibility
    bool dfs(ll mask) {
        // If we already computed this state's result, return it to avoid recomputation
        if (memo.count(mask)) {
            return memo[mask];
        }

        // Iterate through possible moves.
        for (int i = 0; i < boardSize - 1; ++i) {
            // First check if current and next position are both '+' (represented by 1 in the mask)
            if ((mask & (1ll << i)) == 0 || (mask & (1ll << (i + 1))) == 0) {
                continue; // Skip if either position has '-', no valid move here
            }

            // Now we try to flip the two consecutive '+' to '-', so we toggle these bits in mask
            ll newMask = mask ^ (1ll << i) ^ (1ll << (i + 1));

            // If the opponent can win from the new state, we move on to next possibility
            if (dfs(newMask)) {
                continue;
            }

            // We found a move after which the opponent cannot win, so we can win from the current state
            memo[mask] = true;
            return true;
        }

        // If no move led to a winning scenario, then we can declare that we cannot win from this state
        memo[mask] = false;
        return false;
    }
};
```

## Typescript Solution

```typescript
type BitMask = number; // Define 'BitMask' as an alias for number type to represent the game board state in binary

let boardSize: number; // This variable will hold the size of the game board
let memo: Map<BitMask, boolean> = new Map(); // A memoization map to store intermediate results

// Main function to determine if we can win with the given board state
function canWin(currentState: string): boolean {
    boardSize = currentState.length; // Set the size of the board based on the input string
    let mask: BitMask = 0; // This mask will represent our game board's state in binary

    // The loop below will convert the current board state into a binary representation
    for (let i = 0; i < boardSize; ++i) {
        if (currentState[i] === '+') {
            mask |= 1 << i; // Set the bit to 1 at ith position if there is a '+' in the current state
        }
    }

    // Call the recursive function to determine if we can win from the initial state
    return dfs(mask);
}

// Recursive depth-first search (DFS) function to explore the game states and determine win possibility
function dfs(mask: BitMask): boolean {
    // If we already computed this state's result, return it to avoid recomputation
    if (memo.has(mask)) {
        return memo.get(mask)!;
    }

    // Iterate through possible moves.
    for (let i = 0; i < boardSize - 1; ++i) {
        // First check if current and next position are both '+' (represented by 1 in the mask)
        if ((mask & (1 << i)) === 0 || (mask & (1 << (i + 1))) === 0) {
            continue; // Skip if either position has '-', no valid move here
        }

        // Now we try to flip the two consecutive '+' to '-', so we toggle these bits in mask
        let newMask: BitMask = mask ^ (1 << i) ^ (1 << (i + 1));

        // If the opponent can win from the new state, we move on to the next possibility
        if (dfs(newMask)) {
            continue;
        }

        // We found a move after which the opponent cannot win, so we can win from the current state
        memo.set(mask, true);
        return true;
    }

    // If no move led to a winning scenario, then we can declare that we cannot win from this state
    memo.set(mask, false);
    return false;
}
```

## Time and Space Complexity

The given Python code uses a depth-first search algorithm with memoization to resolve a variant of the Nim game, where we check if we can remove two adjacent '+' symbols in a string representing a game state.

### Time Complexity

The time complexity of the DFS algorithm is generally `O(2^N)`, where `N` is the length of the input string `currentState`. This represents the number of possible states that the recursive function `dfs` might explore, as each position in the mask could be a '+' or a '-' after a move, effectively leading to a binary tree of depth `N`.

However, memoization is used, courtesy of the `@cache` decorator, which stores the result of each unique state of the `mask`. This means that each distinct game state would only be computed once. There are at most `2^N` different states for the mask (since each of the `N` positions in the mask can either be 0 or 1).

As a result, the total unique calls to the `dfs` function would not exceed `2^N`. Combined with the fact that for each call to `dfs` we iterate over the `N` positions for possible moves, the total time complexity is `O(N * 2^N)`.

### Space Complexity

The space complexity includes the storage for the recursive call stack and the memoization cache. In the worst case, the call stack could grow up to `N` as the depth of the recursion could be `N` in the case of a sequence of '+' signs. Therefore, the recursive call stack could contribute `O(N)`.

The memoization cache could hold up to `2^N` entries, each requiring a constant amount of space. Therefore, the space used by memoization is `O(2^N)`.

Combining the call stack space and memoization cache, the total space complexity is `O(N + 2^N)` which is dominated by `O(2^N)` for large `N`.

Hence, the overall space complexity of the code is `O(2^N)`.

Note: If we consider the length of the input string `currentState` constant or insignificant compared to the size of the state space, some may argue that the complexity can also be described as `O(1)` for time or space, as the number of operations or space doesn't increase with input size but rather with the number of possible states derived from the input.