294. Flip Game II **Medium** Memoization Math **Dynamic Programming Backtracking Game Theory Leetcode Link**

Problem Description

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

In the Flip Game, you are given a string currentState which consists of only '+' and '-'. A move in this game consists of turning two

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

Intuition

sequence of moves that, if played optimally, ensures victory for the first player.

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

The solution to this problem lies in using backtracking to consider all possible moves of the game. This is done by using Depth First

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

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.

the recursive call returns false, meaning the opponent can't win from that state, then the current player can win by making this

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.

1. Bitmask Representation: A mask is created where each bit represents a position in the currentState string. A '1' in the bitmask

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

them "--") and call dfs on the new mask.

corresponds to a '+' in the string, and a '0' corresponds to a '-'.

• 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

∘ 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

2. Recursive DFS: The core of the solution is the dfs function, which tries to simulate each possible move recursively.

- 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
- 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

to the dfs function with the same mask is computed only once, and the result is stored for subsequent calls with the same

(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.

Let's walk through a small example to illustrate the solution approach described in the content provided. Assume we are given the

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.

○ On encountering 11 (which means "++"), we flip them to 00 (creating "--") and recursively call dfs. 4. Exploration:

5. **Backtracking**:

strategy.

8

15

16

17

18

19

20

21

22

23

24

25

26

27

28

36

33

34

35

36

37

38

39

class Solution:

Example Walkthrough

currentState string as "++--++".

 First Recursion: Let's flip the first two pluses. Our mask becomes 0000111. o 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.

odfs (0000001) is called but no moves can be made within this state, so this returns false, which implies that the opponent

• The process continues by backtracking and exploring other paths to check if there is any sequence of moves that would

6. Result: After exploring all possible moves and finding that there is at least one sequence of moves leading the opposing player

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

to a state where they cannot move, we determine that the starting player can indeed guarantee a win. Thus, the dfs(mask) will

• Since dfs(0000001) returned false, that means dfs(0000111) should return true, indicating that there is a move that ensures the parent state win.

3. **Recursive DFS**: Inside the dfs function, we explore all possible moves:

We check pairs of bits from left to right.

cannot win from this state.

lead to the starting player's loss.

ultimately return true for the initial mask 1100111.

def canWin(self, current_state: str) -> bool:

from functools import lru_cache

def can_opponent_win(flip_mask):

@lru_cache(maxsize=None)

return True

flip_mask, n = 0, len(current_state)

for i, c in enumerate(current_state):

flip_mask |= 1 << i

return False

if c == '+':

34 solution_instance = Solution()

 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.

Using lru_cache for memoization to optimize the recursive function

It will remember the results of function calls with particular arguments

if can_opponent_win(flip_mask ^ (1 << i) ^ (1 << (i + 1))):</pre>

If no flip makes the current player win, then they cannot win

print(solution_instance.canWin("++++")) # Output will depend on the state of the game.

memoization.put(bitMask, false); // Memoize that the current player cannot win

return false; // Current player cannot win with any move

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

bool canWin(string currentState) {

int boardSize; // This variable will hold the size of the game board

// Main function to determine if we can win with the given board state

unordered_map<ll, bool> memo; // A memoization map to store intermediate results

boardSize = currentState.size(); // Set the size of the board based on the input string

Initialize the flip_mask and the length of the current state

Build the initial mask to represent '++' as 1's in flip_mask

- Python Solution
- # Iterate over all pairs of consecutive positions 9 for i in range(n - 1): 10 # If current position and the next are not both '+', skip this iteration 11 12 if (flip_mask & (1 << i)) == 0 or (flip_mask & (1 << (i + 1)) == 0):</pre> 13 continue 14 # Check if by flipping this pair of '+' the opponent can win

If the opponent can win with this flip, continue to the next iteration

If we found a flip that doesn't let the opponent win, then the current player can win

29 30 # Start the recursive checking with the initial configuration of flip_mask 31 return not can_opponent_win(flip_mask) 32 33 # An example of how to use the class.

Java Solution

```
1 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) {
 6
            long bitMask = 0; // bitmask to represent the game board ('+' as 1 and '-' as 0)
            boardSize = currentState.length(); // store the length of the game board
 8
            // Convert the game board string into a bitmask representation
 9
10
            for (int i = 0; i < boardSize; ++i) {</pre>
11
                if (currentState.charAt(i) == '+') {
12
                    bitMask |= 1L << i;
13
14
15
            return canPlayerWin(bitMask);
16
17
18
        // Recursive depth-first search method to determine if the current player can win
19
        private boolean canPlayerWin(long bitMask) {
            // Check the memoization map for precomputed result for the current bitmask
20
21
            if (memoization.containsKey(bitMask)) {
22
                return memoization.get(bitMask);
23
24
            // Check each pair of consective '+' signs for a possible move
25
            for (int i = 0; i < boardSize - 1; ++i) {</pre>
26
                // Check if two consecutive '+' signs are present at index i and i+1
27
                if ((bitMask & (1L << i)) != 0 && (bitMask & (1L << (i + 1))) != 0) {</pre>
28
                    // Flip the two '+' to '-' (by toggling the bits) and continue the search
29
                    if (!canPlayerWin(bitMask ^ (1L << i) ^ (1L << (i + 1)))) {</pre>
30
                        memoization.put(bitMask, true); // Memoize the winning move
31
                        return true; // Current player can win with this move
32
```

11 ll mask = 0; // This mask will represent our game board's state in binary 12 13 // The loop below will convert the current board state into a binary representation for (int i = 0; i < boardSize; ++i) {</pre> 14 if (currentState[i] == '+') { 15

C++ Solution

3 class Solution {

4 public:

6

8

9

10

8

10

11

12

13

14

15

16

17

18

20

21

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

49

50

51

53

52 }

```
mask |= 1ll << i; // Set the bit to 1 at ith position if there is a '+' in the current state
 16
 17
 18
 19
             // Call the recursive function to determine if we can win from the initial state
 20
 21
             return dfs(mask);
 22
 23
 24
         // Recursive depth-first search (DFS) function to explore the game states and determine win possibility
 25
         bool dfs(ll mask) {
 26
             // If we already computed this state's result, return it to avoid recomputation
 27
             if (memo.count(mask)) {
 28
                 return memo[mask];
 29
 30
 31
             // Iterate through possible moves.
 32
             for (int i = 0; i < boardSize - 1; ++i) {</pre>
 33
                 // First check if current and next position are both '+' (represented by 1 in the mask)
 34
                 if ((mask & (1ll << i)) == 0 || (mask & (1ll << (i + 1))) == 0) {</pre>
                     continue; // Skip if either position has '-', no valid move here
 35
 36
 37
                 // Now we try to flip the two consecutive '+' to '-', so we toggle these bits in mask
 38
 39
                 ll newMask = mask ^(1ll << i) ^(1ll << (i + 1));
 40
                 // If the opponent can win from the new state, we move on to next possibility
 42
                 if (dfs(newMask)) {
 43
                     continue;
 44
 45
                 // We found a move after which the opponent cannot win, so we can win from the current state
 46
 47
                 memo[mask] = true;
 48
                 return true;
 49
 50
 51
             // If no move led to a winning scenario, then we can declare that we cannot win from this state
 52
             memo[mask] = false;
 53
             return false;
 54
 55 };
 56
Typescript Solution
    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

mask |= 1 << i; // Set the bit to 1 at ith position if there is a '+' in the current state

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

// Call the recursive function to determine if we can win from the initial state

// If we already computed this state's result, return it to avoid recomputation

continue; // Skip if either position has '-', no valid move here

if ((mask & (1 << i)) === 0 || (mask & (1 << (i + 1))) === 0) {</pre>

let newMask: BitMask = mask $^(1 << i) ^(1 << (i + 1));$

22 // Recursive depth-first search (DFS) function to explore the game states and determine win possibility

// First check if current and next position are both '+' (represented by 1 in the mask)

// Now we try to flip the two consecutive '+' to '-', so we toggle these bits in mask

// If no move led to a winning scenario, then we can declare that we cannot win from this state

// If the opponent can win from the new state, we move on to the next possibility

for (let i = 0; i < boardSize; ++i) {</pre>

function dfs(mask: BitMask): boolean {

return memo.get(mask)!;

if (dfs(newMask)) {

continue;

memo.set(mask, false);

Time and Space Complexity

move, effectively leading to a binary tree of depth N.

// Iterate through possible moves.

for (let i = 0; i < boardSize - 1; ++i) {</pre>

return dfs(mask);

if (memo.has(mask)) {

if (currentState[i] === '+') {

43 44 // We found a move after which the opponent cannot win, so we can win from the current state memo.set(mask, true); 45 46 return true; 47 48

return false;

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 0(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

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).

Space Complexity

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 $O(N * 2^N)$.

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 $0(2^N)$.

Hence, the overall space complexity of the code is $0(2^N)$. Note: If we consider the length of the input string currentState constant or insignificant compared to the size of the state space,

Combining the call stack space and memoization cache, the total space complexity is 0(N + 2^N) which is dominated by 0(2^N) for large N.

some may argue that the complexity can also be described as 0(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.