Medium Bitmask Game Theory **Bit Manipulation** Memoization Math Dynamic Programming Leetcode Link

# **Problem Description**

464. Can I Win

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.

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

## Intuition

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.

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

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),

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

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

in the Python code.

then the first player can guarantee a win.

remaining numbers and the current total.

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 choosable 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.

there are no moves left that could prevent the opponent from winning on their next turn. If no winning strategy is found for any

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

• State Representation: We use an integer (state) to represent the current state of the game, where each bit in state indicates

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

whether a number has been chosen. If the 1-th bit is set, the number 1 has already been selected.

the opponent cannot win after we make this move, so it's also a winning move for us.

optimal play from the opponent, the first player cannot win from this specific state.

maxChoosableInteger, because we only need to compute each of the 2<sup>n</sup> possible states once.

to reach the desired total, so we proceed with the DFS approach.

This process continues until all possible first moves are explored.

4. Recursing Deeper: Let's consider Player 1 had chosen 1.

## • Recursion with DFS: The dfs(state, t) function is the heart of the DFS approach, where state represents the current game

exceed the total of 10.

dfs(000011, 3).

choosing a new number.

• Checking Valid Moves: For each potential move, we 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

• End of Recursion: If none of the moves lead to a winning situation, the function will eventually return false, indicating that with

state and t the current sum. For each call of dfs, we loop through all integers from 1 to maxChoosableInteger to simulate

• 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.

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<sup>n</sup>), where n is

• Return Statement: Finally, if the initial sum check passes, the function returns the result of the initial dfs (0, 0), signifying

Example Walkthrough Let's use the "100 game" example and walk through a smaller case to understand the solution approach. Assume the desiredTotal is

1. Initial Check: We calculate the sum of all numbers from 1 to 6, which is 1+2+3+4+5+6=21. Since 21 is greater than 10, it's possible

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

∘ If Player 1 chooses 1, the state changes to 000001 (1 << (1 - 1)), and t becomes 1. The function then calls dfs (000001, 1).

10 and the maxChoosableInteger is 6. This means each player can choose from integers 1 to 6 without replacement to reach or

also 0. 3. Exploring Moves: The first player begins by exploring all numbers from 1 to 6:

If Player 1 chooses 2, the state becomes 000010 and t becomes 2, leading to a call of dfs (000010, 2).

Now it's Player 2's turn with state 000001 and total 1. Player 2 also explores moves 2 through 6.

5. Checking Winning Condition: Assume Player 1 plays 1, then Player 2 plays 2, and Player 1 follows with 6 on the next turn. Now, the state is 000011 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 1 which results in a total of 10. Since dfs for this state returns true, Player 1 has a winning strategy.

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

For our small example, an optimal strategy exists for Player 1 to win by choosing the sequence 1, 6, 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

∘ If Player 2 chooses 2 next, the state becomes 000011 (000001 | 1 << (2 - 1)) and t becomes 3. The function then calls

6. Memoization: If at any point, the dfs function encounters a state that it has seen before, instead of recomputing, it will use the cached result. This dramatically reduces the number of computations needed.

1 from functools import lru\_cache # Import the lru\_cache decorator for memoization

def can\_win\_game(state: int, current\_total: int) -> bool:

if (state >> integer) & 1:

for integer in range(1, max\_choosable\_integer + 1):

def canIWin(self, max\_choosable\_integer: int, desired\_total: int) -> bool:

# Iterate through all possible integers that can be chosen

return False # No winning move found; current player loses

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

// A memoization map to store previously computed results for given states

public boolean canIWin(int maxChoosableInteger, int desiredTotal) {

// Main method to determine if the player can win given the max number and desired total

# Use memoization to store results of subproblems and avoid recomputation

# Check if 'integer' has already been chosen in the 'state'

continue # Skip this iteration as 'integer' is already taken

return True # Current player can win by choosing 'integer'

state where Player 1 cannot win, the function will return false.

- guarantee a win. Python Solution
- 17 18 # Compute the sum of all integers that can be chosen sum\_of\_integers = (1 + max\_choosable\_integer) \* max\_choosable\_integer // 2 19 20 # If the sum is less than the desired total, no one can win 21 if sum\_of\_integers < desired\_total:</pre> 22 return False 23

# 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\_game(state | (1 << integer), current\_total + integer):

```
38
Java Solution
```

import java.util.Map;

class Solution {

import java.util.HashMap;

using namespace std;

// Determines if the current player can win the game.

// Calculate sum of all choosable integers

if (memo.count(state)) return memo[state];

// Initialize the result as false

bool result = false;

if (sum < desiredTotal) return false;</pre>

unordered\_map<int, bool> memo;

bool canIWin(int maxChoosableInteger, int desiredTotal) {

int sum = (1 + maxChoosableInteger) \* maxChoosableInteger / 2;

// Prepare a memoization table to store intermediate results

// Call to the recursive function to determine if we can win

return dfs(0, 0, maxChoosableInteger, desiredTotal, memo);

// Check if the current state has been computed before

// Try every choosable integer to see if we can win

// Skip if i is already used in the current state

// Save the result to the memoization table for the current state

for (int i = 1; i <= maxChoosableInteger; ++i) {</pre>

// Check if the sum is less than the desired total, which means no one can win

// 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) {

class Solution {

public:

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

private:

class Solution:

8

9

10

11

12

13

14

15

16

24

25

26

30

34

8

9

10

@lru\_cache(maxsize=None)

return can\_win\_game(0, 0)

35 # Call the canIWin method and print the result

31 # Set the maximum choosable integer and desired total

result = solution.canIWin(max\_choosable\_integer, desired\_total)

private Map<Integer, Boolean> memo = new HashMap<>();

// Calculate the sum of all choosable integers

27 # Example of using the code:

28 # Create a Solution object

32 max\_choosable\_integer = 10

print("Can I win?", result)

29 solution = Solution()

desired\_total = 11

```
int sumOfAllIntegers = (1 + maxChoosableInteger) * maxChoosableInteger / 2;
11
12
13
           // If the sum is less than the desired total, no one can win
14
           if (sumOfAllIntegers < desiredTotal) {</pre>
                return false;
15
16
17
           // Start the depth-first search to determine if the player can win
18
19
            return depthFirstSearch(0, 0, maxChoosableInteger, desiredTotal);
20
21
22
       // Helper method for depth-first search
       private boolean depthFirstSearch(int usedNumbersState, int currentTotal, int maxChoosableInteger, int desiredTotal) {
23
24
           // Check if the state has already been computed
25
           if (memo.containsKey(usedNumbersState)) {
                return memo.get(usedNumbersState);
26
27
28
29
           // By default, assume the player cannot win with the current state
30
           boolean canWin = false;
31
32
           // Loop through all choosable integers
33
           for (int i = 1; i <= maxChoosableInteger; ++i) {</pre>
                // Check if the number has not been chosen yet (bit is not set)
34
35
                if (((usedNumbersState >> i) & 1) == 0) {
36
                    // If choosing number i reaches or exceeds desiredTotal or the opponent cannot win,
37
                    // it means the current player can win.
                    if (currentTotal + i >= desiredTotal
38
                        || !depthFirstSearch(usedNumbersState | (1 << i), currentTotal + i, maxChoosableInteger, desiredTotal)) {</pre>
39
40
                        canWin = true;
                        break; // Terminate the loop as we found a winning situation
41
42
43
44
45
46
           // Store the computed result in the memo map before returning it
            memo.put(usedNumbersState, canWin);
47
            return canWin;
48
49
50 }
51
C++ Solution
  1 #include <unordered_map>
```

36

37

38

39

40

41

42

43 };

memo.set(state, result);

return result;

the hashing strategy for the state.

// Return the computed result

```
32
                 if ((state >> i) & 1) continue;
 33
 34
                 // If adding i to currentTotal meets or exceeds desiredTotal, or the opponent cannot win,
 35
                 // set the result as true and break (current player wins)
                 if (currentTotal + i >= desiredTotal || !dfs(state | (1 << i), currentTotal + i, maxChoosableInteger, desiredTotal, mem
 36
 37
                     result = true;
 38
                     break;
 39
 40
 41
 42
             // Memoize and return the result for the current state
 43
             memo[state] = result;
 44
             return result;
 45
 46
    };
 47
Typescript Solution
   // Importing the necessary utilities from 'collections' module
  2 import { HashMap } from 'collectable';
    // Global memoization table to store intermediate results
    const memo: HashMap<number, boolean> = new HashMap<number, boolean>();
  7 // Determines if the current player can win the game given the max choosable integer and the desired total
  8 const canIWin = (maxChoosableInteger: number, desiredTotal: number): boolean => {
         // Calculate sum of all choosable integers to check if winning is possible
  9
 10
         const sum: number = (1 + maxChoosableInteger) * maxChoosableInteger / 2;
 11
         // No one can win if the sum is less than the desired total
         if (sum < desiredTotal) return false;</pre>
 12
 13
 14
         // Call to the recursive function to determine if the current player can win
         return dfs(0, 0, maxChoosableInteger, desiredTotal);
 15
 16 };
 17
    // 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 => {
 20
        // Check for an existing computation for the current state
         if (memo.has(state)) return memo.get(state)!;
 21
 22
 23
         // Initialize the result as false
 24
         let result = false;
 25
 26
         // Iterate through every choosable integer to find a winning move
         for (let i = 1; i <= maxChoosableInteger; ++i) {</pre>
 27
             // Check if 'i' has been used in the current state
 28
             if ((state >> i) & 1) continue;
 29
 30
 31
             // Check if adding 'i' to currentTotal wins the game, or if the opponent cannot win on the next turn
 32
             if (currentTotal + i >= desiredTotal || !dfs(state | (1 << i), currentTotal + i, maxChoosableInteger, desiredTotal)) {
 33
                 result = true;
                 break; // Current player wins, so we break the loop
 34
 35
```

To use TypeScript's Map object instead: 1 // Replace the import statement with the following line 2 const memo = new Map<number, boolean>(); // and replace each HashMap method as follows: 4 // memo.has(state) -> memo.has(state) 5 // memo.get(state)! -> memo.get(state)! // memo.set(state, result) -> memo.set(state, 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

### 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 integer i is chosen if the i-th bit is set to 1. There are 2^maxChoosableInteger possible states because each of the

maxChoosableInteger integers can be either chosen or not chosen.

Time and Space Complexity

## For each call of dfs, we iterate from 1 to maxChoosableInteger to try all possibilities, which gives us 0(maxChoosableInteger) for each call.

**Time Complexity** 

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

The time complexity of the canIWin function is dependent on two factors: the number of recursive calls made by the dfs(state, t)

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.

Considering this, the worst-case time complexity is O(maxChoosableInteger \* 2^maxChoosableInteger) since there are

number until we run out, which is O(maxChoosableInteger) space. Hence, the overall space complexity is also O(maxChoosableInteger + 2^maxChoosableInteger), which simplifies to

1. Cache: Storage for each unique state requires 0(2^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

maxChoosableInteger operations done in each of the 2^maxChoosableInteger possible states.

```
O(2^maxChoosableInteger) as the exponential term dominates the linear term.
```