1025. Divisor Game

Easy | Brainteaser | Math | Dynamic Programming | Game Theory

Problem Description

Alice and Bob are playing a game that involves numbers and strategy. In this game, they take turns with Alice starting first. They have a number `n` initialized on the chalkboard. Each turn, the current player must perform the following actions:

1. Choose any positive number `x` such that `x` is less than `n` and `n % x == 0`, which means `x` has to be a divisor of `n` (`n` is divisible by `x` without any remainder).
2. Then they must replace the number `n` on the chalkboard with `n - x`.

The game continues until a player can no longer make a move. That happens when `n` becomes 1, because there are no numbers less than 1 which divide 1, hence no legal move is possible. The player who cannot make a move loses the game.

The objective of this problem is to determine if Alice can win the game, given that both Alice and Bob play with optimal strategies. We return `true` if Alice can win and `false` if she cannot.

Intuition

To determine if Alice can win the game given the number `n`, we look for a pattern or strategy that can guarantee a win. Analyzing the outcomes of the first few even and odd values of `n` can give us some insight:

1. If `n` is an odd number, then all divisors of `n` are also odd. Subtracting an odd number from an odd number always results in an even number. So if Alice starts with an odd number, Bob will always receive an even number.
2. If `n` is an even number, Alice can always subtract 1 (which is a legal move because 1 is a divisor of all numbers), resulting in an odd number for Bob's turn.

Following these steps optimally, whenever Alice starts with an even number, she can always make a move that leaves an odd number to Bob. Since Bob then can only subtract odd divisors and return an even number back to Alice, this process repeats until Bob is eventually left with the number 1 and no legal moves, resulting in a loss for Bob.

Therefore, we can say that if `n` is even, Alice can win by following the strategy of always subtracting 1, ensuring Bob always gets an odd number. Conversely, if `n` is odd, Bob will be able to follow the same strategy against Alice when he gets his turn, and Alice will eventually lose. So the intuition behind the solution is that Alice wins if and only if she starts with an even number.

The solution is quite simple and elegant when we understand this pattern. The code checks whether the initial number `n` is even and returns `true` if it is and `false` otherwise, which can be achieved with a single line of code by checking `n % 2 == 0`.

Solution Approach

The implementation of the solution for Alice and Bob's game is surprisingly straightforward and does not require complex algorithms or data structures because the problem boils down to an insight into number parity (whether a number is even or odd) rather than an exhaustive search or some dynamic programming approach.

The pattern observed during the Intuition step is that if Alice starts with an even number (`n % 2 == 0`), she will win the game following the optimal strategy of reducing the number by 1 each time her turn arrives. This move ensures that Bob always gets an odd number and cannot force Alice into a losing position.

The solution code uses this insight directly:

```python
class Solution:
    def divisorGame(self, n: int) -> bool:
        return n % 2 == 0
```

In this code:

- The `divisorGame` method takes an integer `n`.
- It returns the Boolean value `True` if `n` is even, indicating that Alice can win.
- If `n` is odd, it returns `False`, indicating that Alice will lose if both players play optimally.

No additional data structures or algorithms are needed because the problem does not ask us to keep track of the game state or the sequence of moves, only to determine the winner based on the initial value of `n`.

By reducing the problem to a simple evaluation of whether the input number `n` is even or odd, we can implement a constant-time solution (`O(1)` time complexity) with constant space (`O(1)` space complexity), which is highly efficient.

Example Walkthrough

Let's take an example where `n = 4` which is an even number to illustrate the solution approach:

- **Turn 1 (Alice's turn)**: `n = 4`. Alice can choose `x = 1` (since 1 is a divisor of 4), and then replace `n` with `n - x`, which is `4 - 1 = 3`. Now `n = 3`.
- **Turn 2 (Bob's turn)**: `n = 3`. Bob can only choose odd divisors, and let's say he chooses `x = 1` again (the only divisor he can choose other than 3 itself), resulting in `n - x = 2`. Now `n = 2`.
- **Turn 3 (Alice's turn)**: `n = 2`. Alice, following the optimal strategy, chooses `x = 1`, leaving `n` at `2 - 1 = 1`. Now `n = 1`.
- **Turn 4 (Bob's turn)**: `n = 1`. Bob has no legal moves because the only divisor of 1 is 1 itself, but the rule states that the chosen number `x` must be less than `n`. Thus, Bob cannot make a move and loses the game.

As we can see, during her turn, Alice can always make sure to leave an odd number for Bob by choosing `x = 1`, following the optimal strategy. Since the only even divisor of an odd number is 1, and subtracting it will always result in an even number, Alice can repeat this process until Bob is left with the inevitable position of `n = 1`, causing him to lose.

This walkthrough demonstrates that for any even starting number `n`, Alice can win the game by making sure to subtract 1 on each of her turns. This strategy is fully encapsulated by the given code:

```python
class Solution:
    def divisorGame(self, n: int) -> bool:
        return n % 2 == 0
```

This code returns `True` for our example where `n = 4`, which indicates that Alice can indeed win if she starts with an even number and both players play optimally.

Python Solution

```python
class Solution:
    def divisor_game(self, n: int) -> bool:
        # Check if the given number n is even.
        # The main strategy relies on the fact that Alice will always win if she
        # starts with an even number. This happens because Alice can always subtract 1
        # to give Bob an odd number. Odd numbers always have odd divisors (except 1),
        # so Bob will return an even number back to Alice.
        # If Alice starts with an odd number, she can only make the number even,
        # and Bob will then be able to keep giving odd numbers back to her.
        # Eventually leading to Alice's loss when the number is reduced to 1.
        return n % 2 == 0

# The name of the method should match with the initial given name, which is `divisorGame`,
# so the method name remains unchanged to maintain compatibility with any code that might use this class.
# Note that the commentary provides an insight into why the method works and what the game strategy is,
# based on the problem statement likely provided in a corresponding LeetCode problem.

```

Java Solution

```java
class Solution {

    /**
     * Determines if the player who starts the game will win the divisor game given the number n.
     *
     * @param n The starting number for the divisor game.
     * @return True if the starting player wins the game; otherwise, false.
     */
    public boolean divisorGame(int n) {
        // In the divisor game, the player who starts with an even number wins.
        // This is because they can always subtract 1 (an optimal move),
        // giving the other player an odd number. Odd numbers always have odd divisors (except 1),
        // so the other player can only return an even number back.
        // The starting player can continue this strategy until they reach the number 2 and win the game.
        // If the starting number is odd, the starting player loses because they can only make
        // the number even for the other player, who will then start using the winning strategy.
        return n % 2 == 0;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Determines if the starting player of the Divisor Game will win.
    // The Divisor Game rules are as follows:
    // - Two players take turns picking a number x where 0 < x < n and n % x == 0.
    // - The player that chooses x subtracts it from n to form a new n.
    // - The player who cannot make a move (because no such x exists) loses.
    // According to the problem, Alice starts first, and if n is even, she can win by playing optimally.
    bool divisorGame(int N) {
        // The strategy here is based on the mathematical deduction that the
        // player who starts with an even number can always win by halving the number.
        // If 'N' is even, the current player will win.
        // If 'N' is odd, the next player will eventually get an even 'N' and win.
        return N % 2 == 0;
    }
};

```

Typescript Solution

```typescript
// Determines if the starting player of the Divisor Game will win.
// The game rules are as follows:
// - Two players take turns picking a number x where 0 < x < n and n % x == 0.
// - The player that chooses x subtracts it from n to form a new n.
// - The player who cannot make a move (because no such x exists) loses.
// Since Alice starts first, and if n is even, she can always win by playing optimally.
function divisorGame(N: number): boolean {
    // Optimally, the starting player (Alice) who begins with an even 'N'
    // can continue to subtract 1 (an optimal x) and hand over an odd number to the opponent.
    // This ensures that the opponent always gives back an even number
    // because odd - even = odd, and she can keep subtracting 1.
    // Eventually, Alice will give '2' to the opponent, who will then only
    // be able to subtract '1' and hand back '1', at which point they will lose
    // because no x exists that divides '1' (other than '1' itself, which is not allowed).
    // Thus, the initial parity of 'N' determines the game's outcome.
    return N % 2 === 0;
}

```

Time and Space Complexity

Time Complexity:

The given Python function `divisorGame` consists of a single operation that checks whether the input integer `n` is even. This check is performed in constant time. Therefore, the time complexity of this function is `O(1)`.

Space Complexity:

The function does not utilize any additional data structures that grow with the input size. Since it only uses a fixed amount of space to store the input and output values, the space complexity is also constant. Hence, the space complexity of the function is `O(1)`.