2029. Stone Game IX

Medium Greedy Array Math Counting Game Theory

Leetcode Link

Problem Description

In this game, Alice and Bob are playing with an array of stones in which each stone has a value. The objective of the game is to avoid making the sum of all removed stones divisible by 3. Alice starts first and both players alternatively remove stones from the array. If at any turn, after a player removes a stone, the sum of all removed stone values is divisible by 3, the player who made that move loses the game. If there are no stones left, Bob wins by default.

As both Alice and Bob play optimally, which means they always make the best possible move, the task is to determine if Alice can win the game and return `true` if she can, or `false` if Bob will win the game.

Intuition

To approach this problem, recognizing the significance of modulo 3 is crucial. When you take the modulo 3 of the stone values, each stone can only contribute a 0, 1, or 2 to the total sum. This greatly simplifies the problem since we only have to consider three cases for the sum of the removed stones modulo 3.

We can keep track of the count of stones for each modulo 3 result, which are `c[0]`, `c[1]`, and `c[2]`. A key insight is that adding a stone that has a value of 1 mod 3 or 2 mod 3 to a sum that is not divisible by 3 cannot make that sum divisible by 3. Removing a stone with a value of 0 mod 3 doesn't change the sum's modulo 3 result, but it does increase the number of moves played in the game.

The solution revolves around the strategic removal of stones such that the total sum remains non-divisible by 3. The `check(c)` function in the given solution code is designed to simulate the game by strategically reducing these counts and mimicking optimal play for both players. It adjusts the turn count and checks if the number of stones with remainder 1 mod 3 equals the number of stones with remainder 2 mod 3, which would make it easy for the next player to force a win.

We simulate two scenarios: one starting with a stone removed that had a remainder of 1 mod 3 (`c[1] -= 1`), and another where we start with a remainder of 2 mod 3 (`c1[1] -= 1`). That's because a different starting move might influence the game's outcome. By simulating the game for both starting moves and taking the logical OR (`check(c) or check(c1)`), we can determine whether Alice has a winning strategy by making either of these moves.

Solution Approach

The solution can be understood by looking into a couple of key elements: counting the stones based on their value modulo 3 and simulating two different scenarios corresponding to Alice's first removal being either a stone with value 1 mod 3 or 2 mod 3.

First, since only the sum modulo 3 matters, we start by categorizing the stones into three groups (`c[0]`, `c[1]`, `c[2]`) based on their modulo with 3. A stone with a value that `mod 3` equals 0 doesn't change the sum's mod3. Stones with values that `mod 3` equals 1 or 2 can only be added in such a way that the sum does not become 0 mod 3.

The solution uses a function `check(c)`, where `c` is the array with the counts of stones grouped by their modulo 3 value. The function first checks a special case where if there are no stones with a modulo value of 1 (`c[1]`), Alice has no winning move.

The main algorithm starts by simulating an initial move which removes a stone of 1 mod 3, updates the counts, then calculates the total turns played as `turn = 1 + min(c[1], c[2]) * 2 + c[0]`. This is based on players taking alternate turns and always removing stones in a way that does not make the sum mod 3 equal to 0. The minimum function is used to pair up stones with a mod 3 value of 1 and 2 since adding one of each maintains the sum non-divisible by 3. Stones with a mod 3 value of 0 are always added since they do not affect divisibility.

If, after the initial move of removing a stone of 1 mod 3, there are more stones in `c[1]` than in `c[2]`, we attempt to remove an additional stone from `c[1]`, and Alice again makes the next move (`turn += 1`). Finally, the condition `turn % 2 == 1 and c[1] != c[2]` checks if the move count is odd (meaning it is Bob's turn) and that there isn't a pair of stones that can be removed to make the sum divisible by 3, which would force Alice to lose.

The array `c1` is a clone of `c` but with the positions of `c[1]` and `c[2]` swapped. This simulates a different first move by Alice, where she begins by taking a stone with modulo 2. The `or` statement `check(c) or check(c1)` combines the results of both simulations to determine if there is a winning strategy for Alice.

This approach uses dynamic programming concepts to explore the possibilities in a reduced, elegant state space, which allows us to determine an optimal play for Alice.

Example Walkthrough

To illustrate the solution approach using a small example, let's consider an array `stones = [1, 2, 3, 4, 5, 6]`. Following the solution approach:

1. We first categorize these stone values based on their result when modulo 3 is applied:
   - `c[0]` (value % 3 == 0): `3, 6` (2 stones)
   - `c[1]` (value % 3 == 1): `1, 4` (2 stones)
   - `c[2]` (value % 3 == 2): `2, 5` (2 stones)
2. Looking at the counts, we can simulate Alice's first move:
   - Scenario A: Alice removes a stone from `c[1]`, leaving us with:
     - `c[0] = 2`, `c[1] = 1`, `c[2] = 2`
   - Scenario B: Alice removes a stone from `c[2]`, leaving us with:
     - `c[0] = 2`, `c[1] = 2`, `c[2] = 1`
3. We then call the function `check(c)` for both scenarios and if either returns true, Alice has a winning strategy.

For Scenario A:

- Alice's initial move: remove one stone from `c[1]`, turn = 1.
- Since `c[1]` and `c[2]` can pair up and each pair is taken in 2 turns, and `c[0]` always gets taken, we have:
  - `turns = 1 (Alice's first move) + min(1, 2) * 2 + c[0]`
  - `turns = 1 + 2 + 2`
  - `turns = 5`

After 5 turns, all stones are taken (`c[0]` stones are always safe to take because they don't affect the sum mod 3). The number of turns is odd, indicating it is Bob's turn, and the counts for `c[1]` and `c[2]` are different so Bob cannot force a loss for Alice.

For Scenario B:

- Alice's initial move: remove one stone from `c[2]`, turn = 1.
- Similar calculation for turns:
  - `turns = 1 (Alice's first move) + min(2, 1) * 2 + c[0]`
  - `turns = 1 + 2 + 2`
  - `turns = 5`

Similar to Scenario A, all stones are taken after 5 turns, with it being Bob's turn, and there is no way to force a loss on Alice.

Both scenarios result in a `turn` value that is odd, meaning it will be Bob's turn when all stones are taken, and he cannot force a loss for Alice. Since at least one scenario returns true, Alice has a winning strategy.

Hence, the simulation determines that Alice can win the game, and the function should return `true`.

Python Solution

```python
class Solution:
    def stoneGameIX(self, stones: List[int]) -> bool:
        # Helper function to check if Alice can win
        # starting with a stone that leaves a remainder of either 1 or 2 when divided by 3
        def can_alice_win(counts):
            # Alice loses immediately if there are no stones that leave a remainder of 1 when divided by 3
            if counts[1] == 0:
                return False
            counts[1] -= 1
            # Calculate the initial turn and simulate the game by adding stones
            # that leave a remainder of 0 when divided by 3 or twice the minimum
            # of stones leaving remainders of 1 or 2
            turn_count = 1 + min(counts[1], counts[2]) * 2 + counts[0]
            # If there are more stones with remainder 1, add another turn
            # (Alice picks one more of these stones)
            if counts[1] > counts[2]:
                turn_count += 1
                counts[1] -= 1
            # Alice wins if the final turn count is odd and there's no equal amount
            # of stones with remainders 1 and 2 after all possible selections
            return turn_count % 2 == 1 and counts[1] != counts[2]

        # Initialize an array to count stones based on their remainder when divided by 3
        remainder_counts = [0] * 3
        for stone in stones:
            remainder_counts[stone % 3] += 1
        # Create a variant of this array to simulate starting with a remainder of 2
        swapped_remainder_counts = [remainder_counts[0], remainder_counts[2], remainder_counts[1]]

        # Return True if Alice can win by starting with a remainder of 1 or 2
        return can_alice_win(remainder_counts) or can_alice_win(swapped_remainder_counts)

```

Java Solution

```java
class Solution {
    public boolean stoneGameIX(int[] stones) {
        // Counts for stones modulo 3 results are stored in counts array.
        // `counts[0]` will hold the count of stones that modulo 3 equals 0,
        // `counts[1]` is for stones that modulo 3 equals 1,
        // `counts[2]` is for stones that modulo 3 equals 2.
        int[] counts = new int[3];
        for (int stone : stones) {
            ++counts[stone % 3];
        }

        // Creating testCounts array to check the scenario starting with picking up
        // a stone that modulo 3 equals 2 (`counts[2]`), therefore flip counts[2] and counts[1].
        int[] testCounts = new int[]{counts[0], counts[2], counts[1]};

        // Check if Alice has a winning strategy for both scenarios: starting with picking up
        // a stone that modulo 3 equals 1, and then for one that modulo 3 equals 2.
        return hasWinningStrategy(counts) || hasWinningStrategy(testCounts);
    }

    // Helper method that checks if a winning strategy exists for
    // a given starting condition. The strategy depends on the relative counts
    // of the stones and the sequence of turns.
    private boolean hasWinningStrategy(int[] counts) {
        // If there are no stones that modulo 3 equals 1, Alice cannot win.
        if (counts[1] == 0) {
            return false;
        }

        // Decrement the count of stones that modulo 3 equals 1 since Alice is starting with this.
        --counts[1];

        // Calculate the initial turn number.
        int turn = 1 + Math.min(counts[1], counts[2]) * 2 + counts[0];

        // If the count of stones that modulo 3 equals 1 is greater than the count of stones
        // that modulo 3 equals 2, then we decrement the former and increment the turn number.
        if (counts[1] > counts[2]) {
            --counts[1];
            ++turn;
        }

        // Alice can win if the total turn counts are odd and the counts of stones that
        // modulo 3 equals 1 and 2 are not the same after her initial pick.
        return turn % 2 == 1 && counts[1] != counts[2];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Determines if the player starting the stone game will always win.
    bool stoneGameIX(vector<int>& stones) {
        // Count the occurrences of stones modulo 3.
        vector<int> counts(3, 0);
        for (int stone : stones) {
            ++counts[stone % 3];
        }
        // Swap counts of 1s and 2s for the second check.
        vector<int> swappedCounts = {counts[0], counts[2], counts[1]};

        // Check both scenarios: starting with a stone that leaves a remainder of 1 or 2 when divided by 3.
        return checkWinningScenario(counts) || checkWinningScenario(swappedCounts);
    }

private:
    // Helper function that checks if the player can win given a starting scenario.
    bool checkWinningScenario(vector<int>& counts) {
        // If there are no stones that leave a remainder of 1, Alice cannot win.
        if (counts[1] == 0) return false;

        // Pick one stone with a remainder of 1 to start.
        --counts[1];
        // Calculate the initial turn based on the stones picked.
        int turn = 1 + min(counts[1], counts[2]) * 2 + counts[0];

        // If there are more stones with a remainder of 1 than 2, pick another one to change turn count.
        if (counts[1] > counts[2]) {
            --counts[1];
            ++turn;
        }

        // Alice wins if the total number of turns is odd and there isn't an equal number of stones
        // that leave remainders of 1 and 2.
        return turn % 2 == 1 && counts[1] != counts[2];
    }
};

```

Typescript Solution

```typescript
// Type definition for the input stone array.
type StonesArray = number[];

// Counts occurrences of stones modulo 3.
const countStonesModuloThree = (stones: StonesArray): number[] => {
    const counts = [0, 0, 0];
    stones.forEach(stone => {
        counts[stone % 3]++;
    });
    return counts;
};

// Helper function that checks if the player can win given a starting scenario.
const checkWinningScenario = (counts: number[]): boolean => {
    // If there are no stones that leave a remainder of 1, the player cannot win.
    if (counts[1] === 0) return false;

    // Pick one stone with a remainder of 1 to start.
    counts[1]--;
    // Calculate the initial turn based on the stones picked.
    let turn = 1 + Math.min(counts[1], counts[2]) * 2 + counts[0];

    // If there are more stones with a remainder of 1 than 2, pick another one to change turn count.
    if (counts[1] > counts[2]) {
        counts[1]--;
        turn++;
    }

    // The player wins if the total number of turns is odd and there isn't an equal number of stones
    // that leave remainders of 1 and 2.
    return turn % 2 === 1 && counts[1] !== counts[2];
};

// Determines if the player starting the stone game will always win.
const stoneGameIX = (stones: StonesArray): boolean => {
    // Count the occurrences of stones modulo 3.
    const counts = countStonesModuloThree(stones);
    // Swap counts of 1s and 2s for the second check.
    const swappedCounts: number[] = [counts[0], counts[2], counts[1]];

    // Check both scenarios: starting with a stone that leaves a remainder of 1 or 2 when divided by 3.
    return checkWinningScenario(counts) || checkWinningScenario(swappedCounts);
};

// Example usage:
// const stones: StonesArray = [1, 1, 7, 10, 8, 17];
// console.log(stoneGameIX(stones)); // Outputs whether the starting player will always win.

```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be determined by analyzing the loop and the operations done within the loop and functions. There is a single loop iterating over the stones list, which introduces a linear complexity factor, `O(n)`, where `n` is the length of the stones. The rest of the operations and function calls inside and outside the loop run in constant time, `O(1)`.

Therefore, the total time complexity is `O(n)`.

Space Complexity

The space complexity is determined by the additional space used by the algorithm proportional to the input size. In this code, there is a constant amount of extra space used, such as the `c` list with a length of 3, which stores counts and the extra `c1` list, which is a rearranged version of `c`. There are no data structures used that grow with the input size.

Thus, the total space complexity is `O(1)` as only a constant amount of extra space is used regardless of the input size.