2038. Remove Colored Pieces if Both Neighbors are the Same Color

Medium Greedy Math String Game Theory

Leetcode Link

Problem Description

This LeetCode problem revolves around a game played on a line of colored pieces, where each piece is colored either 'A' or 'B'. The game is played with two players, Alice and Bob, who take turns. The key points for this game are:

- Alice goes first and can only remove a piece colored 'A' if it is sandwiched between two pieces also colored 'A'.
- Bob can only remove a piece colored 'B' that is likewise between two 'B' pieces.
- Neither player can remove pieces from the ends of the line.
- Players must pass their turn if they can't make a move, and if a player can't move, they lose the game.

The goal is to determine if Alice can win the game when both players are making their best possible moves.

Understanding the problem, it's clear that the game's outcome depends on the initial configuration of the colors. Furthermore, since only pieces not at the edges can be removed, and a piece must be surrounded by two of the same color, we can direct our attention to sequences of the same color, specifically sequences of three or more, as these will be the only configurations that can be acted upon.

Intuition

The solution approach is grounded in the concept that the ability to make a move is contingent on finding sequences of three like-colored pieces. The reason behind counting sequences of three is that it's the smallest number of pieces in a row that allows for a move (since each move requires a piece to be between two others of the same color). Our strategy, then, is to count the number of possible moves for Alice and Bob individually.

Here's the thinking process:

- Iterate through the string of 'A's and 'B's using a method that groups the same colors together and counts the size of these groups.
- For each group, if it's size is three or more, it can potentially provide at least one move for the corresponding player (size of group - 2).
- Tally the total possible moves for Alice and Bob separately.
- After processing the entire string of colors, compare the tallies. If Alice has more potential moves, she can win. If Bob has more or an equal number of moves, he can win.

By counting the excess in each group beyond two, we ensure that we are considering only valid moves. The final comparison of which player has more potential moves translates directly into who has the upper hand and is thus likely to win.

Solution Approach

The implementation employs the `groupby` function from Python's `itertools` module, which is a common pattern for grouping consecutive elements in an iterable that are the same. This pattern is useful for situations like this, where we need to consider consecutive runs of the same character.

The core idea is to traverse the string `colors` only once, and for each distinct group of consecutive characters ('A's or 'B's), we calculate how many valid moves can be made from that group. A valid move is one that involves removing a piece with the same color on both sides, which in our case means we are looking for a group of at least three same-colored pieces.

Here's a step-by-step breakdown of the solution approach:

1. Initialize two counters, `a` and `b`, to track the number of potential moves for Alice (`A`) and Bob (`B`) respectively.
2. Use the `groupby` function to iterate over the string `colors`. This will give us each group of consecutive characters along with an iterator to the group items.
3. For each group identified by `groupby`:
   - Determine the size of the group by converting the group iterator to a list and getting its length.
   - Since a move requires a piece to be surrounded by two others of the same color, any group of size `n` can provide `n - 2` potential moves.
   - If the group size minus 2 is positive, and it's a group of 'A', add the number of potential moves to Alice's counter (`a`).
   - Likewise, if it's a group of 'B' that allows for moves, add the number to Bob's counter (`b`).
4. After processing all groups, return whether Alice can win the game, which is the case if `a` is greater than `b`.

Below is the relevant code snippet demonstrating this:

```
a = b = 0
for c, v in groupby(colors):
    n = len(list(v)) - 2
    if n > 0 and c == 'A':
        a += n
    elif n > 0 and c == 'B':
        b += n
return a > b
```

This code snippet seamlessly integrates counting and determining the winner in a single pass. It is concise, efficient, and leverages the power of Python's standard library to perform the necessary task of grouping and counting.

Example Walkthrough

Let's consider a smaller example to illustrate the solution approach. Suppose the input string of colors is `"AABBBAA"`. Now we will walk through the given approach step-by-step:

1. Initialize counters for Alice (a) and Bob (b) to 0, where `a` will count potential moves for Alice and `b` will count potential moves for Bob.

2. Now, we start iterating over the string with the `groupby` function:

   - The first group is `"AA"`. This group is of size 2, so it does not allow for any moves (2-2=0). Alice's and Bob's counters remain at `a=0` and `b=0`.
   - Next, we encounter the group `"BBB"`. Since this is a group of size 3, it allows for 1 move for Bob (3-2=1). Now we have `a=0` and `b=1`.
   - Finally, we have another group `"AA"`. Like the first group, this is also of size 2, and no moves can be made from this group. The counters remain `a=0` and `b=1`.

3. After processing all groups, we compare Alice's and Bob's counters. Alice has `a=0` potential moves, and Bob has `b=1` potential move.

4. Since Bob has more or an equal number of moves compared to Alice, he can win if both players play optimally. Therefore, the final result is `False` as Alice cannot win the game.

The code corresponding to this approach would look like:

```
from itertools import groupby

colors = "AABBBAA"
a = b = 0
for c, v in groupby(colors):
    n = len(list(v)) - 2
    if n > 0 and c == 'A':
        a += n
    elif n > 0 and c == 'B':
        b += n
return a > b  # This will return False in this example
```

This walkthrough showcases how the strategy of counting the potential moves for each player can help to predict the game's outcome. By using the `groupby` function to identify the groups of consecutive 'A's or 'B's, we are able to perform the operations within a single iteration, resulting in an efficient solution.

Python Solution

```
from itertools import groupby

class Solution:
    def winnerOfGame(self, colors: str) -> bool:
        # Initialize counters for the number of times A and B can make a move.
        moves_a = moves_b = 0

        # Iterate through grouped sequences of 'A's and 'B's.
        for color, group in groupby(colors):
            # Compute the number of moves by subtracting 2 (minimum number to remove a color)
            # from the length of the sequence.
            moves = len(list(group)) - 2

            # A move is possible only if the length of consecutive colors is greater than 2.
            if moves > 0:
                if color == 'A':
                    # Accumulate the number of moves for A.
                    moves_a += moves
                elif color == 'B':
                    # Accumulate the number of moves for B.
                    moves_b += moves

        # Determine the winner. A wins if A has more moves than B.
        return moves_a > moves_b
```

Java Solution

```
class Solution {
    public boolean winnerOfGame(String colors) {
        // The length of the color sequence
        int sequenceLength = colors.length();

        // Counters for the moves available to A and B
        int movesForA = 0, movesForB = 0;

        // Traverse through the sequence
        for (int startIndex = 0, endIndex = 0; startIndex < sequenceLength; startIndex = endIndex) {

            // Find a contiguous block of the same color starting from startIndex
            while (endIndex < sequenceLength && colors.charAt(endIndex) == colors.charAt(startIndex)) {
                ++endIndex;
            }

            // Calculate the number of possible moves in the current contiguous block
            // We subtract 2 because we need at least three of the same colors in a row to make a move.
            int blockMoves = endIndex - startIndex - 2;

            // If there are possible moves from this block,
            // we assign them to the appropriate player based on the color.
            if (blockMoves > 0) {
                if (colors.charAt(startIndex) == 'A') {
                    movesForA += blockMoves;
                } else {
                    movesForB += blockMoves;
                }
            }
        }

        // If A has more moves than B, A wins; otherwise, B wins.
        // Return true if A wins, false if B wins.
        return movesForA > movesForB;
    }
}
```

C++ Solution

```
class Solution {
public:
    // Function to determine the winner of the game based on the color sequence
    bool winnerOfGame(string colors) {
        int length = colors.size(); // The total number of colors in the string
        int aliceCount = 0; // Count of the moves available to Alice
        int bobCount = 0; // Count of the moves available to Bob

        // Iterate over the string to count the moves for Alice and Bob
        for (int start = 0, end = 0; start < length; start = end) {
            // Find the sequence of same colors starting from 'start'
            while (end < length && colors[end] == colors[start]) {
                ++end;
            }
            // Calculate the number of valid moves for this continuous sequence
            // There must be at least 3 consecutive colors to form a valid move
            int moves = end - start - 2;
            if (moves > 0) {
                // If the current color is 'A', add the count to Alice's moves, otherwise to Bob's moves
                if (colors[start] == 'A') {
                    aliceCount += moves;
                } else {
                    bobCount += moves;
                }
            }
        }

        // Return true if Alice has more valid moves than Bob
        return aliceCount > bobCount;
    }
};
```

Typescript Solution

```
function winnerOfGame(colors: string): boolean {
    // Initialize the length of the colors string
    const colorsLength = colors.length;
    // Initialize counters for Alice (A) and Bob (B) to track their points
    let alicePoints = 0;
    let bobPoints = 0;

    // Iterate through the colors string to calculate points
    for (let startIndex = 0, endIndex = 0; startIndex < colorsLength; startIndex = endIndex) {
        // Find consecutive characters that are the same
        while (endIndex < colorsLength && colors[endIndex] === colors[startIndex]) {
            endIndex++;
        }

        // Calculate the count of the consecutive characters minus 2 (since 3 are needed to score a point)
        const consecutiveCount = endIndex - startIndex - 2;
        if (consecutiveCount > 0) {
            // Award points to Alice or Bob depending on the character ('A' or 'B')
            if (colors[startIndex] === 'A') {
                alicePoints += consecutiveCount;
            } else {
                bobPoints += consecutiveCount;
            }
        }
    }

    // Determine the winner by comparing the points
    // Alice wins if she has more points; otherwise, Bob wins or it's a draw
    return alicePoints > bobPoints;
}
```

Time and Space Complexity

The time complexity of the provided code is `O(n)`, where `n` is the length of the string `colors`. This is due to the fact that the `for` loop will iterate over each group of consecutive characters in the string exactly once, and the number of such groups is linearly proportional to the length of the string.

The space complexity of the code is actually `O(n)` in the worst case, due to the conversion of `v` (which is an iterator) to a list with `list(v)`. In the worst case, if all characters in `colors` are the same, `v` would be as large as the entire string, which would require additional space proportional to the size of the input string, hence `O(n)`. However, the reference answer considers space complexity to be `O(1)`, which would only be accurate if there was no conversion to a list, and the size of additional used memory did not depend on the size of the input.