1033. Moving Stones Until Consecutive

Medium Brainteaser Math Leetcode Link

Problem Description

In this LeetCode problem, we are tasked with finding the minimum and maximum number of moves to position three stones into three consecutive positions along the X-axis. We are given three integers `a`, `b`, and `c`, which represent the current positions of the stones. A move consists of picking up a stone that is at one of the endpoints (not in the middle) and moving it to a new position that is between the current two endpoints but not currently occupied by another stone.

The game ends when the stones are in consecutive positions, meaning they occupy three positions that are next to each other, and no more moves can be made. We need to calculate two things:

1. The minimum number of moves required to reach the end state.
2. The maximum number of moves that can be played before reaching the end state.

Intuition

To solve this problem, we first need to identify the positions of the stones in sorted order since their given positions could be in any order. We'll denote the smallest position as `x`, the largest as `z`, and the middle one as `y`. Once we have the stones sorted, we can determine possible moves.

For the minimum moves (`mi`):

- If the stones are already consecutive, no moves are required.
- If there is only one gap of one space between the stones, just one move is required, moving the endpoint stone into the one gap.
- If there are larger gaps, we may need two moves. One to move an outer stone next to the middle stone and another to fill in the remaining gap, unless one of the gaps is already just one space, in which case only one move is required.

For the maximum moves (`mx`):

- Since the maximum number of moves is made by putting the stones next to each other one step at a time, the number of moves is equal to the distance between `x` and `z`, minus the two positions where `y` and the stone being moved can be positioned (since the end state is consecutive positions).

The solution code reflects this logic by first finding `x`, `y`, and `z`, and then determining the minimum and maximum number of moves based on the distances between the stones.

Solution Approach

The implementation of the given solution involves determining the starting positions of the stones and calculating the minimum and maximum number of moves required to place them in consecutive positions.

Here is a breakdown of the approach:

1. Firstly, we identify the minimum and maximum values among `a`, `b`, and `c` to establish which stones are at the endpoints. We use the built-in functions `min()` and `max()` to find these values, assigning them to the local variables `x` and `z`.

2. The middle value `y` is then found by calculating the sum of all three positions and subtracting the values of `x` and `z`. This gives us the sorted order of the stones on the X-axis without actually sorting them.

3. With the stones now in order, we consider two scenarios for calculating minimum moves `mi`:

   - If `z - x` is less than or equal to 2, the stones are already in consecutive positions or only one move away from it. Therefore, we need either 0 or 1 move.
   - If `z - x` is greater than 2, we must consider the positions of `y`. If `y` is within one position of `x` or `z` (`y - x < 3` or `z - y < 3`), then only one move is required. Otherwise, we might need two moves: one to move an endpoint stone closer and another move to place it in the right position.
4. For maximum moves `mx`, if `z - x` is less than or equal to 2 again, we cannot make any moves, so `mx` is 0. If there is a bigger gap, then the maximum number of moves is the total distance between `x` and `z` minus 2 (for `x` and `z` themselves), since we are trying to fill this distance with consecutive positions.

5. At the end of this process, we `return [mi, mx]`, which represents the minimum and maximum number of moves respectively.

The algorithm uses constant space and operates in constant time, as all operations are basic arithmetic calculations. There are no complex data structures or patterns in use; it's straight-forward analysis of the positions of the stones.

Example Walkthrough

Let's take three stones at positions a = 1, b = 2, and c = 5.

1. First, we determine the minimum (`x`) and maximum (`z`) positions of the stones using the `min()` and `max()` functions. In our case, `x = 1` and `z = 5`.

2. We then find the middle position `y` by subtracting `x` and `z` from the sum `(a + b + c)`. That gives us `y = (1 + 2 + 5) - (1 + 5) = 8 - 6 = 2`.

3. Now we have the stones in sorted order: `x = 1`, `y = 2`, `z = 5`. To calculate the minimum number of moves (`mi`):

   - We check the distance between `x` and `z`. Since `z - x = 5 - 1 = 4`, which is more than 2, the stones are not consecutive yet.
   - We look at the gaps between `x, y` and `y, z`. Here, `y - x = 2 - 1 = 1` and `z - y = 5 - 2 = 3`. The gap between `x` and `y` is 1, which is less than 3, so moving the stone from position `z` to `y + 1` (3) will make them consecutive. Thus, only one move is needed.
4. To find the maximum number of moves (`mx`):

   - We have a larger gap so, `mx = z - x - 2 = 5 - 1 - 2 = 2`. This is the number of moves that can be played moving the stone from `z` to `y + 1` and then moving the stone from `x` to `x + 1` (or vice versa), making them consecutive one step at a time.
5. Finally, we return the calculated minimum and maximum moves as the output of the algorithm: `[mi, mx] = [1, 2]`.

Through this walkthrough with the example values, the stones would achieve consecutive positions in `1` minimum move and up to `2` maximum moves.

Python Solution

```python
from typing import List

class Solution:
    def numMovesStones(self, a: int, b: int, c: int) -> List[int]:
        # Find the minimum and maximum values among a, b, and c to identify
        # the positions of the leftmost (min_stone) and rightmost (max_stone) stones.
        min_stone, max_stone = min(a, b, c), max(a, b, c)

        # The middle stone will be the sum of a, b, c minus the leftmost and rightmost stones.
        middle_stone = a + b + c - min_stone - max_stone

        # Initializing the minimum and maximum moves to 0.
        min_moves = max_moves = 0

        # If the largest gap between the end stones is more than 2,
        # then moves are required.
        if max_stone - min_stone > 2:
            # If the gap between either the middle stone and one of the end stones
            # is less than 3, only one move is required (to move the middle stone).
            # Otherwise, two moves are required.
            min_moves = 1 if middle_stone - min_stone < 3 or max_stone - middle_stone < 3 else 2

            # The maximum moves are the largest gap minus two, because moving a stone
            # into either end of the gap will always reduce the remaining gap by one.
            max_moves = max_stone - min_stone - 2

        # Returning the minimum and maximum number of moves as a list.
        return [min_moves, max_moves]

```

Java Solution

```java
class Solution {
    public int[] numMovesStones(int a, int b, int c) {
        // Find the smallest value among a, b, c and assign it to minStone
        int minStone = Math.min(a, Math.min(b, c));

        // Find the largest value among a, b, c and assign it to maxStone
        int maxStone = Math.max(a, Math.max(b, c));

        // The middle stone is always the sum minus the smallest and largest
        int middleStone = a + b + c - minStone - maxStone;

        // Initialize the minimum and maximum moves to 0
        int minimumMoves = 0, maximumMoves = 0;

        // If the stones are not already consecutive
        if (maxStone - minStone > 2) {
            // If the gap between any two stones is less than 3, a single move is needed,
            // otherwise 2 moves are needed (move one stone next to an end stone and the other stone in between them)
            minimumMoves = (middleStone - minStone < 3 || maxStone - middleStone < 3) ? 1 : 2;

            // The maximum number of moves is the total gap minus the two spaces occupied by the two end stones
            maximumMoves = maxStone - minStone - 2;
        }

        // Return an array with the minimum and maximum moves
        return new int[] {minimumMoves, maximumMoves};
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    vector<int> numMovesStones(int a, int b, int c) {
        // Find the smallest and largest of the three stones.
        int minStone = min({a, b, c});
        int maxStone = max({a, b, c});

        // Calculate the middle stone by subtracting the smallest and largest from the sum.
        int midStone = a + b + c - minStone - maxStone;

        // Initialize the minimum and maximum moves to 0.
        int minMoves = 0, maxMoves = 0;

        // If the largest and smallest stone are not next to each other
        if (maxStone - minStone > 2) {
            // If either gap between min and mid or mid and max is less than 3,
            // one move is sufficient to place a stone in a position that bridges the gap.
            // Otherwise, two moves are needed (one for each gap).
            minMoves = (midStone - minStone < 3 || maxStone - midStone < 3) ? 1 : 2;

            // The maximum number of moves is the total span minus two (since two positions will be occupied
            // by the outer stones) which gives us the total number of free spots between them.
            maxMoves = maxStone - minStone - 2;
        }

        // Return a vector with the minimum and maximum number of moves.
        return {minMoves, maxMoves};
    }
};

```

Typescript Solution

```typescript
function numMovesStones(a: number, b: number, c: number): number[] {
    // First, find the minimum (left) value among a, b, and c.
    const left = Math.min(a, Math.min(b, c));
    // Next, find the maximum (right) value among a, b, and c.
    const right = Math.max(a, Math.max(b, c));
    // Calculate the middle value by subtracting the known left and right from the sum of all.
    const middle = a + b + c - left - right;

    let minMoves = 0; // To store the minimum number of moves.
    let maxMoves = 0; // To store the maximum number of moves.

    // If the stones are not consequtive (having at least two places gap).
    if (right - left > 2) {
        // If the gap from left to middle or middle to right is less than 3,
        // only one move is required as a stone can be placed in between the other two directly.
        // Otherwise, two moves are needed.
        minMoves = (middle - left < 3 || right - middle < 3) ? 1 : 2;
        // Maximum number of moves is the total gap between the outer stones minus the positions
        // occupied by the middle stone, as we can move one stone per turn.
        maxMoves = right - left - 2;
    }

    // Return an array of two numbers: minimum and maximum number of moves.
    return [minMoves, maxMoves];
}

```

Time and Space Complexity

The provided code is determining the minimum and maximum number of moves to position three stones (with unique positions `a`, `b`, and `c`) consecutively. For understanding the complexity, let's analyze the code step by step:

- The `min` and `max` functions are used to find the smallest (`x`) and largest (`z`) of the three stones' positions. This operation is `O(1)` because it deals with only three elements.

- Calculating `y` is done by summing the positions and subtracting `x` and `z` to find the remaining middle value. This is also `O(1)`.

- Setting `mi` and `mx` to zero is a constant operation, `O(1)`.

- The condition `if z - x > 2:` checks if the stones are not already consecutive. This comparison is `O(1)`.

- Inside the `if` statement, the code performs two further checks: `y - x < 3` and `z - y < 3`, each being `O(1)`. Based on these checks, `mi` is either set to 1 or 2 and `mx` is set to `z - x - 2`.

- The operation `z - x - 2` is also `O(1)` since it involves constant-time arithmetic operations.

- Returning the list `[mi, mx]` is `O(1)` as it simply packages two integers into a list.

Given that all operations are constant-time, the overall time complexity of the provided code is `O(1)`.

The space complexity is the amount of additional space or temporary space one needs to allocate in order to execute the code. Since the provided code only uses a fixed number of integer variables (`x`, `y`, `z`, `mi`, `mx`) and does not utilize any data structures that grow with input size:

- The overall space complexity of the provided code is also `O(1)` as it allocates a constant amount of space.