1040. Moving Stones Until Consecutive II

Medium Array Math Two Pointers Sorting

Problem Description

The problem presents a game scenario with stones placed at various positions on the X-axis. You are given an array, `stones`, representing the positions of the stones. A stone becomes an *endpoint stone* if it is at either end of the sequence (smallest or largest position). The objective is to make moves to remove a stone from being an endpoint stone by placing it at a new unoccupied position.

The rules of the game state that you cannot move an endpoint stone to a position where it continues to be an endpoint stone. The game ends when you cannot make any more moves and the stones are in three consecutive positions.

The problem asks for two outputs:

1. The minimum number of moves required, represented by `answer[0]`.
2. The maximum number of moves that can be made, represented by `answer[1]`.

Intuition

To figure out the solution, understand the following key points:

Minimum Moves: To compute the minimum number of moves (`mi`), we aim to bring the stones into a consecutive sequence with as few moves as possible. This usually entails moving endpoint stones closer to the center. The edge case is when there's exactly one gap between two stones that's equal to the number of stones minus two - this would require two moves to fill in the positions and make the sequence consecutive.

Maximum Moves: For the maximum number of moves (`mx`), think of moving the endpoint stones away from the center to the farthest position possible inside the current stone range, but one at a time, to ensure they are not considered an endpoint stone anymore. It's a game of maximizing the number of moves by utilizing the current gaps to the fullest.

The code's approach to solving for `mi` and `mx` involves sorting the stones array to process stone positions in a defined order. It then utilizes sliding window technique to find the minimum moves required to create a consecutive sequence. The maximum number of moves are computed considering the largest gaps on either end of the sorted array.

Solution Approach

The solution follows these steps:

1. **Sort the Array**: The first step requires sorting the `stones` array. Sorting helps identify the endpoints easily and arrange stones in an ascending order which is crucial for the next steps.
2. **Initialize Variables**: A variable `n` is used to store the number of stones, and `mi` is initialized with the value of `n`, to be later updated with the actual minimum moves.
3. **Compute the Maximum Moves (`mx`)**: Calculate the maximum moves which can be made by considering one of two scenarios, whichever yields a larger value:
   - Moving the stone from the start to just after the second stone, i.e., `stones[-1] - stones[1] + 1` minus the number of stones excluding the moved stone.
   - Moving the second last stone to just before the first stone, i.e., `stones[-2] - stones[0] + 1` minus the number of stones excluding the moved stone.

   The `+1` in the formulas accounts for the fact that we want to include both endpoints in the range. These operations represent the case where you move the first or last stone to get the most number of moves without being an endpoint stone in the next move.
4. **Sliding Window for Minimum Moves (`mi`)**: The next part of the code uses a sliding window approach to calculate the minimum moves. Two pointers `i` (start of the window) and `j` (end of the window) iterate over the array to determine the smallest movement of stones necessary to reach a position where stones are in a consecutive sequence.
   - The window size cannot be greater than `n`, so if the current window size exceeds `n`, the start of the window (`i`) is moved up.
   - If the end of the window minus the start of the window plus one (`j - i + 1`) equals `n - 1` and the range (`a - stones[i]`) is `n - 2`, this means there is exactly one gap that requires two moves to fill. Otherwise, the minimum moves (`mi`) are updated to `n` minus the size of the current window (`n - (j - i + 1)`).

The code uses these techniques and conditional checks to ensure we find the least and the maximum number of moves to reach the game's end goal.

Finally, the function returns the list `[mi, mx]` containing the minimum and maximum moves possible.

Example Walkthrough

Assuming we have an array of stone positions given as `stones = [1, 3, 5, 7]`. Let's apply the solution approach to this array to understand how we get our `answer` with the minimum and maximum number of moves.

1. **Sort the Array**: The initial array is already sorted, which looks like `[1, 3, 5, 7]`. The endpoints here are `1` and `7`.
2. **Initialize Variables**: The number of stones `n` is `4` in this case. We'll start with `mi = 4` for the minimum moves.
3. **Compute the Maximum Moves (`mx`)**:
   - Moving the stone from the start (`1`) to just after the second stone (`3`) would be: `7 - 3 + 1 = 5` minus the number of stones excluding the moved one, which is `3`, so `5 - 3 = 2`.
   - Moving the second last stone (`5`) to just before the first stone (`1`) would yield: `5 - 1 + 1 = 5` minus the remaining stones, again `3`, resulting in `2`.

   So, in either case, the `mx` is `2`.
4. **Sliding Window for Minimum Moves (`mi`)**:
   - Using a sliding window, we check consecutive positions in the sorted array to calculate the fewest moves to get stones in a consecutive sequence.
   - We start with the window `i = 0` and expand `j` till we reach `2`, so we look at `[1, 3, 5]`. The window size here is `3`, which is one less than `n`, and the range is `5 - 1 = 4`, which is not `n - 2`. The stones in this window already form a consecutive sequence, so no moves needed.
   - Now let's consider the window `[3, 5, 7]` with `i = 1` and `j = 3`. Similar to the first window, stones are already consecutive, and no moves are required.
   - Since the stones within these windows are consecutive and we never encounter a situation with a gap of `n - 2`, `mi` is `0`, as we don't need any moves to achieve the consecutive sequence.

Finally, we find that the minimum number of moves, `mi`, is `0` and the maximum number of moves, `mx`, is `2`. Hence, `answer = [mi, mx]` would return `[0, 2]`.

Solution Implementation

Python

```python
class Solution:
    def numMovesStonesII(self, stones: List[int]) -> List[int]:
        stones.sort()  # First, sort the stones in non-decreasing order.
        total_stones = len(stones)  # Find the total number of stones.

        # Initialize the minimum and maximum moves.
        min_moves = total_stones
        max_moves = max(stones[-1] - stones[1] - total_stones + 2,
                        stones[-2] - stones[0] - total_stones + 2)

        # 'i' will denote the start index of a window of stones
        i = 0
        for j, current_stone in enumerate(stones):
            # Slide the window towards the right if it's too long.
            # i.e., the length of the window is greater than the number of stones
            while current_stone - stones[i] + 1 > total_stones:
                i += 1

            # Check for a special case where there's exactly one spot open
            # within a group of consecutively placed stones - in this case,
            # two moves are always required.
            if j - i + 1 == total_stones - 1 and current_stone - stones[i] == total_stones - 2:
                min_moves = min(min_moves, 2)
            else:
                # Otherwise, the minimum moves are the total stones minus the size of the current window
                min_moves = min(min_moves, total_stones - (j - i + 1))

        # Return the results as a list containing the minimum and maximum moves.
        return [min_moves, max_moves]
```

Java

```java
class Solution {
    public int[] numMovesStonesII(int[] stones) {
        // Sort the stone positions
        Arrays.sort(stones);

        // Number of stones
        int numStones = stones.length;

        // Minimum moves initialized to the total number of stones
        int minMoves = numStones;

        // The maximum moves are calculated using the end positions of the stones minus their starting positions,
        // then subtracting the total number of stones minus one from it.
        int maxMoves = Math.max(stones[numStones - 1] - stones[1], stones[numStones - 2] - stones[0]) + 1 - numStones;

        // Initialize two pointers for the sliding window technique
        for (int i = 0, j = 0; j < numStones; ++j) {
            // While the difference between stones at j and i pointers plus one is greater than the total number of stones,
            // increment the i pointer.
            while (stones[j] - stones[i] + 1 > numStones) {
                ++i;
            }

            // Check if there's a gap for one stone to make two moves
            if (j - i + 1 == numStones - 1 && stones[j] - stones[i] == numStones - 2) {
                minMoves = Math.min(minMoves, 2);
            } else {
                // Otherwise, calculate the minimum moves as the difference between total stones and the number of stones
                // within the current range.
                minMoves = Math.min(minMoves, numStones - (j - i + 1));
            }
        }

        // Return the minimum and maximum moves as an array
        return new int[]{minMoves, maxMoves};
    }
}
```

C++

```cpp
class Solution {
public:
    vector<int> numMovesStonesII(vector<int>& stones) {
        // First, sort the positions of stones
        sort(stones.begin(), stones.end());

        // n represents the total number of stones
        int numStones = stones.size();

        // Initialize minimum moves to the number of stones, which will be updated later
        int minMoves = numStones;

        // The maximum moves are calculated by considering the empty spaces at the ends of the sequence
        int maxMoves = max(stones[numStones - 1] - stones[1], stones[numStones - 2] - stones[0]) - (numStones - 2);

        // using two-pointer technique to find the smallest window that can contain all the stones
        for (int start = 0, end = 0; end < numStones; ++end) {
            // Slide window if the current window cannot fit in a consecutive sequence of size numStones
            while (stones[end] - stones[start] + 1 > numStones) {
                ++start; // increment the start of the window
            }

            // checking for the case with n - 1 consecutive stones with one space in between them
            if (end - start + 1 == numStones - 1 && stones[end] - stones[start] == numStones - 2) {
                // 2 moves are required in this specific case
                minMoves = min(minMoves, 2);
            } else {
                // otherwise, the minimum moves are determined by the size of the current window
                minMoves = min(minMoves, numStones - (end - start + 1));
            }
        }

        // return the pair of minimum and maximum moves
        return {minMoves, maxMoves};
    }
};
```

TypeScript

```typescript
function numMovesStonesII(stones: number[]): number[] {
    // Sort stones array so stones are in non-decreasing order
    stones.sort((a, b) => a - b);
    const numberOfStones = stones.length;

    // Initialize minimum moves as number of stones
    let minimumMoves = numberOfStones;

    // Calculate maximum moves by considering the farthest two stones
    const maximumMoves = Math.max(
        stones[numberOfStones - 1] - stones[1],
        stones[numberOfStones - 2] - stones[0]
    ) - (numberOfStones - 2);

    // Sliding window to calculate minimum moves needed
    for (let startIdx = 0, endIdx = 0; endIdx < numberOfStones; ++endIdx) {
        // While the current window size is greater than slot size, increase the start index
        while (stones[endIdx] - stones[startIdx] + 1 > numberOfStones) {
            ++startIdx;
        }

        // Check for the special case where there's one space for the last stone
        if (endIdx - startIdx + 1 === numberOfStones - 1 && stones[endIdx] - stones[startIdx] === numberOfStones - 2) {
            minimumMoves = Math.min(minimumMoves, 2);
        } else {
            // Otherwise, calculate the moves needed by the number of stones outside the current window
            minimumMoves = Math.min(minimumMoves, numberOfStones - (endIdx - startIdx + 1));
        }
    }

    // Return an array of minimum and maximum moves
    return [minimumMoves, maximumMoves];
}
```

```python
class Solution:
    def numMovesStonesII(self, stones: List[int]) -> List[int]:
        stones.sort()  # First, sort the stones in non-decreasing order.
        total_stones = len(stones)  # Find the total number of stones.

        # Initialize the minimum and maximum moves.
        min_moves = total_stones
        max_moves = max(stones[-1] - stones[1] - total_stones + 2,
                        stones[-2] - stones[0] - total_stones + 2)

        # 'i' will denote the start index of a window of stones
        i = 0
        for j, current_stone in enumerate(stones):
            # Slide the window towards the right if it's too long.
            # i.e., the length of the window is greater than the number of stones
            while current_stone - stones[i] + 1 > total_stones:
                i += 1

            # Check for a special case where there's exactly one spot open
            # within a group of consecutively placed stones - in this case,
            # two moves are always required.
            if j - i + 1 == total_stones - 1 and current_stone - stones[i] == total_stones - 2:
                min_moves = min(min_moves, 2)
            else:
                # Otherwise, the minimum moves are the total stones minus the size of the current window
                min_moves = min(min_moves, total_stones - (j - i + 1))

        # Return the results as a list containing the minimum and maximum moves.
        return [min_moves, max_moves]
```

Time and Space Complexity

The time complexity of the given code is `O(n log n)` and the space complexity is `O(1)`.

Time Complexity

1. `stones.sort()`: Sorting the stones requires `O(n log n)` time, where `n` is the number of stones.
2. The `for` and `while` loops: The outer `for` loop runs `n` times, and the inner `while` loop might seem to increase the complexity, but it does not. The `while` loop will execute at most `n` times in total across all iterations of the `for` loop because each stone is only moved from the window once. Therefore, the loops combine for `O(n)` time.

Adding both complexities, the dominant term is the `O(n log n)` for the sort operation, leading to a total time complexity of `O(n log n)`.

Space Complexity

1. The space complexity remains `O(1)` since no additional space is used that scales with the input size. Only a fixed number of variables are used for tracking indices and the minimum and maximum moves, regardless of the number of stones.