390. Elimination Game

Medium Recursion Math

Leetcode Link

Problem Description

In this problem, you have a sorted list of integers from 1 to `n`. You apply a specific elimination algorithm to this list that consists of repeatedly removing numbers in a particular order until only one number remains. The rules for the removal process are as follows:

1. Begin with the first element in the list and then remove every other element. This is considered a left-to-right pass.
2. On the next iteration, start from the end of the remaining list and remove every other element, moving right to left.
3. Continue this process, alternating between left-to-right and right-to-left, until there is only one number left in the list.

The task is to determine what that last remaining number will be given the integer `n`.

Intuition

To find the solution without simulating the entire elimination process (which would be inefficient for large `n`), we can observe patterns in the list as we remove numbers. We notice that on each pass, the number of elements in the list is halved. Therefore, we can track the following variables throughout the process:

- `a1`: The first number left in the list after each pass.
- `an`: The last number left in the list after each pass.
- `i`: The index of the current pass, starting from 0.
- `step`: The distance between remaining numbers after each pass.
- `cnt`: The count of remaining numbers after each pass.

The critical insight is that we can mathematically determine the new values of `a1` and `an` without explicitly simulating each pass. Specifically, `a1` increments by `step` on every pass, and `an` decrements by `step` when the pass starts from the right and the count of remaining numbers is odd. We continue this process until `cnt` (the count of remaining numbers) is reduced to 1, which indicates only one number is left. At that point, `a1` will be the last remaining number, and we can return it.

By tracking these variables and updating them after each pass in an alternating fashion, we arrive at the last number without having to actually remove elements from the list, which significantly reduces the time complexity and allows us to solve the problem with a high efficiency.

Solution Approach

The solution implemented in Python uses a straightforward loop to simulate the process of elimination described above, without actually removing elements from a list. Instead, we use variables to keep track of the state of the list. Below is the detailed explanation of how the implementation corresponds to the solution approach:

We initially set `a1` to 1, `an` to `n`, `i` to 0, `step` to 1, and `cnt` to `n`. These variables represent the start and end of the list, the index of the pass (which determines the direction of elimination), the gap between numbers after each iteration, and the count of numbers remaining, respectively.

- `a1`: This variable represents the value of the first element in the remaining list.
- `an`: Represents the value of the last element in the remaining list.
- `i`: Used to track the number of completed passes, and it helps to determine the direction of the current pass.
- `step`: The distance between consecutive remaining numbers. On each pass, the `step` doubles since every second element is removed.
- `cnt`: The count of remaining numbers in the list. Initially the same as `n`, and it is halved after each pass (`cnt >>= 1` is equivalent to `cnt = cnt // 2`).

The loop continues until `cnt` becomes 1, which means there's only one number left. Inside the loop:

- On each iteration, we check if the pass is odd (`i % 2`) or even. For even passes (`i` is 0, 2, 4, ...), we always increase `a1` by `step`. For odd passes (`i` is 1, 3, 5, ...), we reduce `an` by `step` only.
- If the count of remaining numbers is odd (`cnt % 2`), then on every pass `an` is decremented for even `i` and `a1` is incremented for odd `i`.
- After processing the current pass, we double the step (`step <<= 1` is equivalent to `step = step * 2`) because every other number was removed during the pass.
- The count of remaining elements is halved (`cnt >>= 1`) as we effectively remove every second element in the list on each pass.
- We increment `i` after each pass to alternate the direction for the next pass.

At the end of the loop, `a1` is the last number remaining, which is what we return. The solution makes use of bitwise operations for efficiency, such as `>>=` for division by 2 and `<<=` for multiplication by 2. The algorithm efficiently tracks and updates the bounds of the remaining list without the need to maintain the list itself, resulting in a time complexity of O(log n).

Example Walkthrough

Let's walk through an example using the solution approach with `n = 10`. The sorted list from 1 to `n` will initially be `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`.

- **Initialization**: We set `a1` to 1, `an` to 10, `i` to 0, `step` to 1, and `cnt` to 10.
- **First Pass (i = 0, left-to-right)**: We increment `a1` by `step` since it's an even pass, so `a1` becomes 2. The `step` doubles to 2, and `cnt` is halved to 5.
  - Updated list would look like `[2, 4, 6, 8, 10]` if we were removing elements, but we're not.
- **Second Pass (i = 1, right-to-left)**: Now, it's an odd pass and `cnt` is odd, so `an` (last number) is decremented by `step`, becoming 10 - 2 = 8. We double the `step` to 4, and halve `cnt` to 2.
  - The conceptual list would now be `[2, 6]`.
- **Third Pass (i = 2, left-to-right)**: For this even pass, we increase `a1` again by `step`, resulting in `a1` = 2 + 4 = 6. The step is doubled to become 8, and `cnt` is halved to 1.

Since `cnt` is now 1, there's only one number left, and `a1` is that last remaining number which is `6`. Hence, the last remaining number when using this elimination algorithm on a sorted list from 1 to 10 is 6.

```

Let's illustrate the solution approach with a real example using `n = 10`. Initially, `a1` is set to 1, representing the first elemer

- In the first pass (left-to-right), we eliminate every other number, starting with the second one. We update `a1` to 2, double the `
- The second pass (right-to-left) starts with an odd count, so we decrease `an` by the current `step` value, which is 2, resulting ir
- On the third pass (left-to-right), since it's even, we increase `a1` by the step (now 4) to get `a1` = 6. Doubling the step to 8 ar

Therefore, the last standing number is `6` which we find by intelligently keeping track of the eliminated numbers' range and count wi
```

Python Solution

```python
class Solution:
    def lastRemaining(self, n: int) -> int:
        # Initialize the first and last elements
        first_element, last_element = 1, n

        # Initialize iteration count, step increment, and count of remaining elements
        iteration, step, remaining_count = 0, 1, n

        # Loop until only one element remains
        while remaining_count > 1:
            # If the iteration is odd, operate from the right (last to first)
            if iteration % 2:
                # If count is odd, decrement the last element
                last_element -= step
                # Adjust first element if the count is odd
                if remaining_count % 2:
                    first_element += step
            # If the iteration is even, operate from the left (first to last)
            else:
                # Increment the first element
                first_element += step
                # Adjust the last element if the count is odd
                if remaining_count % 2:
                    last_element -= step

            # Halve the remaining count as we remove half of the elements each iteration
            remaining_count >>= 1
            # Double the step size since every round we skip one more element
            step <<= 1
            # Move to the next iteration
            iteration += 1

        # The first element is the last remaining number
        return first_element

```

Java Solution

```java
class Solution {
    public int lastRemaining(int n) {
        int firstElement = 1;      // Initialize the first element of the sequence to 1
        int lastElement = n;       // Initialize the last element of the sequence to n
        int step = 1;              // Step size, which doubles each iteration

        // Loop until only one element remains. cnt keeps track of the remaining elements.
        for (int round = 0, remainingElements = n; remainingElements > 1;
             remainingElements >>= 1, step <<= 1, ++round) {

            // In odd rounds, move from right to left
            if (round % 2 == 1) {
                lastElement -= step;  // Subtract step from the last element

                // If the count is odd, increment the first element
                if (remainingElements % 2 == 1) {
                    firstElement += step;
                }
            }
            // In even rounds, move from left to right
            else {
                firstElement += step;  // Increment first element

                // If the count is odd, decrement the last element
                if (remainingElements % 2 == 1) {
                    lastElement -= step;
                }
            }
        }

        // Once the loop is finished, there's only one element left, which is the first element.
        return firstElement;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int lastRemaining(int n) {
        int first = 1, last = n; // Initialize the first and last elements
        int step = 1; // This will represent the step size after each round

        // Loop until there's only one number remaining
        for (int round = 0, remaining = n; remaining > 1; remaining >>= 1, step <<= 1, ++round) {
            // In every odd round, update the last element
            if (round % 2) {
                last -= step; // Move the last element backwards by the step size
                if (remaining % 2) {
                    first += step; // If there's an odd number of elements, move the first element forwards
                }
            } else {
                // In every even round, update the first element
                first += step; // Move the first element forwards by the step size
                if (remaining % 2) {
                    last -= step; // If there's an odd number of elements, move the last element backwards
                }
            }
        }
        return first; // Return the remaining number
    }
};

```

Typescript Solution

```typescript
// Function to determine the last remaining number after eliminating every second element in rounds
function lastRemaining(n: number): number {
    let first: number = 1; // Initialize the first element
    let last: number = n;  // Initialize the last element
    let step: number = 1;  // This will represent the step size after each round

    // Loop until there's only one number remaining
    for (let round: number = 0, remaining: number = n; remaining > 1; remaining >>= 1, step <<= 1, round++) {
        // In every odd round, update the last element
        if (round % 2 === 1) {
            last -= step; // Move the last element backwards by the step size
            if (remaining % 2 === 1) {
                first += step; // If there's an odd number of remaining elements, move the first element forwards
            }
        } else {
            // In every even round, update the first element
            first += step; // Move the first element forwards by the step size
            if (remaining % 2 === 1) {
                last -= step; // If there's an odd number of remaining elements, move the last element backwards
            }
        }
    }
    return first; // Return the remaining number
}

// Example usage
// let result = lastRemaining(10);
// console.log(result);  // This would compute the result for n=10 and print it to the console

```

Time and Space Complexity

The given Python code defines a method `lastRemaining` that calculates the last number remaining after repeatedly removing numbers from a list that has been recreated at each step of the problem. In the problem, numbers are removed alternately from the start and end, and every second number from the list is removed at each step.

Time Complexity

The time complexity of the function is determined by the while loop which runs until `cnt > 1`. The number `cnt` represents the remaining numbers in the sequence and is halved (`cnt >>= 1`) at each iteration of the while loop due to removing every second number. As a result, the time complexity of the loop, and hence the function, is `O(log n)`, as it reduces the sequence size by a factor of two at each step until only one number is left.

Space Complexity

The space complexity of the provided code is `O(1)`. This is because the algorithm uses a fixed number of variables (`a1`, `an`, `i`, `step`, `cnt`) and doesn't require any additional space that would grow with the input size `n`. It does not utilize any complex data structures that would require space proportional to the input size.