1006. Clumsy Factorial

Medium Stack Math Simulation

Problem Description

The **Clumsy Factorial** problem asks for the computation of a non-standard factorial of a given positive integer `n`. This non-standard factorial, termed as "clumsy factorial", is composed by performing a fixed rotation of mathematical operations—multiplication (`*`), division (`/`), addition (`+`), and subtraction (`-`)—on the series of integers starting from `n` and decrementing by 1 until 1 is reached. The order of these operations follows a pattern, starting with multiplication and cycling through division, addition, and subtraction before returning back to multiplication for the next set of numbers. It's important to note that the multiplications and divisions are processed before additions and subtractions due to the conventional order of operations in arithmetic, and that division is specifically floor division (which rounds down to the nearest integer). The challenge is to calculate this "clumsy factorial" following the stipulated rules.

Intuition

To approach this problem, one can simulate the calculations as described in the problem statement by sequentially applying the operations on the decreasing series of numbers. The solution keeps track of the next operation to perform using a rotating pattern with a simple counter variable that cycles through 0 to 3 (corresponding to multiplication, division, addition, and subtraction).

The solution uses a stack to keep intermediate results and to easily handle the priority of multiplication and division over addition and subtraction. This is crucial because, during the evaluation of the expression, results of multiplication and division must be computed immediately to respect the order of operations, while addition and subtraction can be deferred. When the operation to be performed is multiplication or division, the last number is popped from the stack, the operation is performed with the next number, and the result is pushed back onto the stack. For addition, the next number is directly pushed onto the stack, while for subtraction, the next number is negated and then pushed to the stack. In the end, the sum of numbers in the stack gives the final result of the clumsy factorial.

This approach bypasses the need to handle parentheses or keep track of different precedence levels beyond the immediate multiplication/division versus addition/subtraction; the use of a stack neatly manages intermediary values and negation, allowing the running sum to be calculated directly at the end.

Solution Approach

The solution utilizes a simple yet effective approach to implement the clumsy factorial. Here's a step-by-step breakdown of the algorithm, referring to the code provided:

1. **Initialization**: A stack, `s`, is created with a single element, `N`, which is the starting value of the clumsy factorial.
2. **Loop Through Numbers**:
   - A loop is constructed to iterate through the numbers starting from `N-1` down to `1`, inclusive. This loop decrements the value of `i` on each iteration, which represents the next number involved in the current operation.
3. **Cycling Through Operations**:
   - A variable `op` is used as an operation counter that cycles through `0` to `3`, determining the operation to perform (multiplication, division, addition, subtraction).
   - The `op` value is incremented and wrapped back to `0` after reaching `3` using `op = (op + 1) % 4` to reset the cycle.
4. **Performing Operations**:
   - The code checks the value of `op` to decide which operation to perform.
   - **Multiplication** (`op == 0`): Pops the last number from the stack, multiplies it with `i`, and pushes the result back onto the stack.
   - **Division** (`op == 1`): Similar to multiplication, but performs an integer division (floor division) with `i`, and pushes the result back.
   - **Addition** (`op == 2`): Directly pushes `i` onto the stack, as addition can be deferred without affecting the outcome.
   - **Subtraction** (`op == 3`): Pushes `-i` onto the stack to represent the subtraction. The negation allows us to turn the final evaluation into a simple summation process.
5. **Calculating the Result**:
   - Once all numbers and operations have been processed and represented on the stack, the final value of the clumsy factorial is obtained by summing all elements in the stack using `sum(s)`.

This solution uses a stack to hold intermediate values, which elegantly handles the different precedences of operations. It takes advantage of the associative property of addition and subtraction, simply negating numbers for subtraction and deferring the addition until the end. By doing this, the result can be calculated in a single pass through the numbers with a straightforward summation, avoiding the complexity of managing different levels of operation precedence or parentheses. The judicious use of the stack as per the operation precedence rule makes this approach efficient and easy to implement.

Example Walkthrough

Let's illustrate the solution approach using a small example. Suppose we want to calculate the clumsy factorial for `N = 4`. The series of operations will start at 4 and include the next integers in descending order applying multiplication (`*`), division (`/`), addition (`+`), and subtraction (`-`) in a repeating cycle until we reach 1.

Here is how the computation would proceed step by step:

1. Initialize the stack with the starting value `s=[4]`.
2. The next number `i` initialized at `N-1`, which is `3`.
3. The first operation to be applied is multiplication (`op == 0`), so we pop `4` from the stack, multiply it by `3`, and push the result `12` back onto the stack. Now the stack contains `[12]`.
4. Decrement `i` to `2`. Now, it's time for division (`op == 1`). We pop `12` from the stack, perform floor division by `2` which is `6`, and push this result back onto the stack. The stack now contains `[6]`.
5. Decrement `i` to `1`. The next operation is addition (`op == 2`). This time, we push the value `1` directly onto the stack as addition can be deferred without issue. The stack now has `[6, 1]`.
6. At this point, we have no more numbers to decrement to, so we do not process subtraction which would be the next operation in the cycle (`op == 3`).
7. Finally, we simply calculate the sum of the elements in the stack to get the result of the clumsy factorial. So the result is `sum([6, 1])` which equals `7`.

Therefore, the clumsy factorial of `4` is `7`.

This simple example demonstrates the algorithm's effectiveness. It efficiently manages the operation precedence by using the stack to perform immediate multiplications and divisions, while at the same time reducing the addition and subtraction to a final summation over the stack's contents without the need for managing different precedence levels or using parentheses.

Solution Implementation

Python

```python
class Solution:
    def clumsy(self, N: int) -> int:
        # Initialize operation counter (0: multiply, 1: divide, 2: add, 3: subtract)
        operation = 0

        # Start with a stack containing the starting number
        stack = [N]

        # Iterate backwards from N-1 to 1
        for number in range(N - 1, 0, -1):
            if operation == 0:
                # For multiplication, pop the last element from stack, multiply it by the current number, and push the result back
                stack.append(stack.pop() * number)
            elif operation == 1:
                # For division, pop the last element from stack, perform integer division with the current number, and push the resul
                # Python 3 requires explicit integer division using "//"
                stack.append(stack.pop() // number)
            elif operation == 2:
                # For addition, simply push the current number onto the stack
                stack.append(number)
            else:
                # For subtraction, push the current number as a negative onto the stack
                stack.append(-number)

            # Move to the next operation in the sequence, wrapping back to 0 after 3 (subtraction)
            operation = (operation + 1) % 4

        # Return the sum of the numbers in the stack
        return sum(stack)

# Example of usage:
# sol = Solution()
# result = sol.clumsy(10)
# print(result)  # This would output the clumsy factorial result of 10
```

Java

```java
class Solution {
    public int clumsy(int N) {
        // Use a deque to implement the stack for numbers operations
        Deque<Integer> stack = new ArrayDeque<>();
        stack.push(N); // Push the first number onto the stack

        int operation = 0; // To keep track of which operation to perform

        // Iterate from N - 1 down to 1
        for (int i = N - 1; i > 0; --i) {
            if (operation == 0) {
                // Perform multiplication and push the result onto the stack
                stack.push(stack.pop() * i);
            } else if (operation == 1) {
                // Perform division and push the result onto the stack
                stack.push(stack.pop() / i);
            } else if (operation == 2) {
                // Perform addition and push the number onto the stack
                stack.push(i);
            } else {
                // Perform subtraction and push the negated number onto the stack
                stack.push(-i);
            }
            // Cycle through the operations: multiply, divide, add, subtract
            operation = (operation + 1) % 4;
        }

        // Accumulate the sum of all numbers in the stack
        int result = 0;
        while (!stack.isEmpty()) {
            result += stack.pop();
        }

        // Return the final accumulated result
        return result;
    }
}
```

C++

```cpp
#include <stack>

class Solution {
public:
    int clumsy(int N) {
        std::stack<int> numStack; // A stack to keep track of numbers and calculations
        numStack.push(N); // Push the initial number onto the stack

        int operation = 0; // Variable to cycle through the operations: 0 for multiply, 1 for divide, 2 for addition, 3 for subtracti

        // Iterate from N-1 down to 1
        for (int i = N - 1; i > 0; --i) {
            switch (operation) {
                case 0: // Multiply
                    numStack.top() *= i;
                    break;
                case 1: // Divide
                    numStack.top() /= i;
                    break;
                case 2: // Addition
                    numStack.push(i); // We push the number directly onto the stack for addition
                    break;
                case 3: // Subtraction
                    numStack.push(-i); // We push the negative number for subtraction
                    break;
            }
            // Cycle through the operations by using modulo 4
            operation = (operation + 1) % 4;
        }

        // Compute the sum of all numbers in the stack
        int result = 0; // Variable to store the final result
        while (!numStack.empty()) {
            result += numStack.top(); // Add the top element of the stack to result
            numStack.pop(); // Remove the top element from the stack
        }

        return result; // Return the computed result
    }
};
```

TypeScript

```typescript
// Method to simulate 'clumsy factorial' operation on a given integer N.
function clumsy(N: number): number {
    // Use an array as a stack to perform number operations.
    const stack: number[] = [];
    stack.push(N); // Push the first number onto the stack.

    // Variable to keep track of the current operation.
    let operation = 0;

    // Iterate from N - 1 down to 1.
    for (let i = N - 1; i > 0; i--) {
        if (operation === 0) {
            // Multiply the top of the stack with the current number and push the result back.
            stack.push(stack.pop()! * i);
        } else if (operation === 1) {
            // Divide the top of the stack by the current number and push the result back.
            stack.push(Math.floor(stack.pop()! / i));
        } else if (operation === 2) {
            // Add the current number to the stack.
            stack.push(i);
        } else {
            // Subtract the current number by pushing its negation onto the stack.
            stack.push(-i);
        }
        // Proceed to the next operation in the cycle: multiply, divide, add, subtract.
        operation = (operation + 1) % 4;
    }

    // Sum up all the numbers in the stack.
    let result = 0;
    while (stack.length) {
        result += stack.pop()!;
    }

    // Return the final result of the clumsy operation.
    return result;
}

// Example usage:
// const myResult = clumsy(4); // Expected output is 7;
```

```python
class Solution:
    def clumsy(self, N: int) -> int:
        # Initialize operation counter (0: multiply, 1: divide, 2: add, 3: subtract)
        operation = 0

        # Start with a stack containing the starting number
        stack = [N]

        # Iterate backwards from N-1 to 1
        for number in range(N - 1, 0, -1):
            if operation == 0:
                # For multiplication, pop the last element from stack, multiply it by the current number, and push the result back
                stack.append(stack.pop() * number)
            elif operation == 1:
                # For division, pop the last element from stack, perform integer division with the current number, and push the resul
                # Python 3 requires explicit integer division using "//"
                stack.append(stack.pop() // number)
            elif operation == 2:
                # For addition, simply push the current number onto the stack
                stack.append(number)
            else:
                # For subtraction, push the current number as a negative onto the stack
                stack.append(-number)

            # Move to the next operation in the sequence, wrapping back to 0 after 3 (subtraction)
            operation = (operation + 1) % 4

        # Return the sum of the numbers in the stack
        return sum(stack)

# Example of usage:
# sol = Solution()
# result = sol.clumsy(10)
# print(result)  # This would output the clumsy factorial result of 10
```

Time and Space Complexity

The time complexity of the code provided is `O(N)`. This is because there is a single loop that iterates over `N` elements, decrementing at each step until it reaches `0`. Within this loop, each operation (multiplication, division, addition, and subtraction) is performed at most once per iteration, and these operations can be considered to have a constant time complexity.

The space complexity of the code provided is `O(N)` as well. This stems from the use of the stack `s` which, in the worst case, could have as many as `N` elements if all operations except for subtraction were somehow skipped (which actually doesn't occur in the given algorithm, but this is a theoretical upper bound). A more accurate assessment taking into account the algorithm's actual behavior would still lead to `O(N)` space complexity since each operator reduces the stack size except for addition and subtraction but these are always followed by multiplications and divisions which reduce it.