224. Basic Calculator

Hard | Stack | Recursion | Math | String

Leetcode Link

Problem Description

The problem tasks us with creating a program that can evaluate a string expression containing integers, '+', '-', '(', and ')', representing the addition and subtraction of integers as well as grouping with parentheses. No built-in functions or libraries for evaluating mathematical expressions are allowed. The objective is to parse and calculate the result of the given string as the expression would be evaluated in arithmetic.

Intuition

To solve this challenge, we simulate the mechanics of an arithmetic expression evaluator by manually parsing the string. This involves the following steps:

1. **Linear Scanning**: We process the string from left to right, character by character, to evaluate the numbers and the operators.
2. **Dealing with Numbers**: Since the numbers may have more than one digit, we need to convert the sequence of digit characters into actual numerical values. This is done by initializing a number `x` to `0` and then for each digit `'d'` we find, multiplying `x` by `10` and adding `d` to `x`, effectively constructing the number in decimal.
3. **Handling Operators**: We keep track of the last seen operator which will be either a '+' or a '-', through a `sign` variable which is `1` or `-1`, respectively. When a number is completely parsed, we combine it with the current `sign` to add or subtract from the accumulated answer.
4. **Using Stack for Parentheses**: To evaluate expressions within parentheses, we use a stack. When we encounter an opening parenthesis '(', we push the current accumulated answer and the current sign onto the stack, then reset the answer and sign to begin evaluating the expression inside the parentheses. When we encounter a closing parenthesis ')', we pop the sign and the accumulated answer before the parenthesis from the stack, and combine them with the current answer inside the parentheses (apply the sign and then add the two numbers).
5. **Final Computation**: After we process all characters, the result of the entire expression will be in the `answer` variable. Parentheses are handled during the process as they are encountered, ensuring that the subexpressions are calculated at the correct times.

This approach systematically breaks down the string into components that we can evaluate in isolation, handling the precedence of operations in expressions with parentheses appropriately, leading to the correct final answer.

Solution Approach

The solution's approach primarily involves a while loop that iterates through the string `s`, evaluating expressions and handling arithmetic operations and parentheses. The core components of the solution include:

- **Stack (`stk`)**: Used to store previous results and signs when an opening parenthesis is found. We push the current `ans` and `sign` and restart their values for evaluating the new embedded expression.
- **Current answer (`ans`)**: This is the running total of the expression evaluated so far, excluding the content within parentheses which haven't been closed yet. When we find a closing parenthesis, we update it to include the result of expression inside the just-closed parenthesis.
- **Current sign (`sign`)**: This variable holds the last seen sign of '+' or '-', initialized as `1` (representing '+'). This is used to multiply with the current number found to add or subtract it from `ans`.
- **Index (`i`)**: To keep track of the current position within the string.

The algorithm proceeds as follows:

- Iterate over each character in the string until the end is reached.
- If a digit is encountered:
  - A separate while loop gathers the entire number (as numbers could have multiple digits), constructing the value by multiplying the previously accumulated value by `10` and adding the digit to it.
  - After the whole number is determined, multiply it by the current `sign` and add it to the current `ans`.
  - Update the index `i` to the position after the last digit of the number.
- When a '+' or '-' is encountered:
  - The `sign` is updated to `1` for '+' or `-1` for '-'.
- On encountering an opening parenthesis '(':
  - Push the current `ans` and `sign` onto the stack (to 'remember' them for after the close parenthesis).
  - Reset `ans` and `sign` for the next calculation within the new context (inside the parentheses).
- On encountering a closing parenthesis ')':
  - The expression inside the parenthesis is complete, so compute its value by multiplying it with the `sign` popped from the stack.
  - Then add the result to the `ans` before the parenthesis (also popped from the stack).
- The loop increments the index `i` at each step, except when processing a whole number, where `i` is updated accordingly.

After the while loop completes, `ans` contains the result of evaluating the entire expression, which is then returned.

Example Walkthrough

Let's use a simple expression to demonstrate the approach: `1 + (2 - (3 + 4))`.

1. Initialize `ans = 0`, `sign = 1`, `stk = []`, and start at index `i = 0`.
2. `i = 0`: The first character is `1`, a digit. So we form the number `1`.
3. Update `ans` by adding `1 * sign` which is `1`. Now `ans = 1`.
4. `i = 2`: The character is `+`. Set `sign` to `1`.
5. `i = 4`: Encounter `(`. Push the current `ans` (`1`) and `sign` (`1`) onto `stk` and reset `ans` to `0`, `sign` to `1`.
6. `i = 5`: Next is digit `2`. Update `ans` to `2`.
7. `i = 7`: Encounter `-`. Set `sign` to `-1`.
8. `i = 9`: Encounter `(` again. Push the current `ans` (`2`) and `sign` (`-1`) onto `stk` and reset `ans` to `0`, `sign` to `1`.
9. `i = 10`: Digit `3` is seen. Update `ans` to `3`.
10. `i = 12`: The character is `+`. Set `sign` to `1`.
11. `i = 14`: Digit `4` is seen. Update `ans` by adding `4 * sign` which is `4`. Now `ans = 3 + 4 = 7`.
12. `i = 15`: Encounter `)`. Pop `stk` which has `2` and `-1`. Update `ans`: `ans = 2 - (7) = -5`.
13. `i`: Move forward to the next character, but there is none immediately after the closing parenthesis, so continue.
14. `i = 17`: Now, encounter another `)`. Pop `stk` which has `1` and `1`. Update `ans`: `ans = 1 + (-5) = -4`.

At the end of the string, we have `ans = -4`. Since we've processed every character according to the rules, and handled the nested parentheses correctly, the resultant `ans` is the evaluation of the entire expression `1 + (2 - (3 + 4))`, which is indeed `-4`.

Python Solution

```python
class Solution:
    def calculate(self, s: str) -> int:
        stack = []  # Initialize an empty list to be used as a stack
        result, operator = 0, 1  # Initialize result to 0 and operator to 1 ('+' sign)
        index, length = 0, len(s)  # Initialize loop variables

        # Iterate over the input string
        while index < length:
            # If the current character is a digit
            if s[index].isdigit():
                number = 0
                # Continue until a non-digit is found, building the number
                while index < length and s[index].isdigit():
                    number = number * 10 + int(s[index])
                    index += 1
                # Update the result with the current number and the preceding operator
                result += operator * number
                # Compensate for the index increment in the loop
                index -= 1
            # If the current character is a plus, set operator to 1
            elif s[index] == "+":
                operator = 1
            # If the current character is a minus, set operator to -1
            elif s[index] == "-":
                operator = -1
            # Handle opening parentheses
            elif s[index] == "(":
                # Push the current result and operator to the stack
                stack.append(result)
                stack.append(operator)
                # Reset the result and operator for the new expression within the parentheses
                result, operator = 0, 1
            # Handle closing parentheses
            elif s[index] == ")":
                # The result inside the parentheses is multiplied by the operator before the parentheses
                result *= stack.pop()
                # Add the result inside the parentheses to the result before the parentheses
                result += stack.pop()
            # Move to the next character
            index += 1

        return result  # Return the evaluated result

```

Java Solution

```java
class Solution {
    public int calculate(String s) {
        // Stack to hold the intermediate results and signs
        Deque<Integer> stack = new ArrayDeque<>();
        // Initialize the sign as positive
        int sign = 1;
        // This will hold the final result of the evaluation
        int result = 0;
        // Length of the input string for iteration
        int length = s.length();

        // Iterating over each character in the string
        for (int i = 0; i < length; ++i) {
            char ch = s.charAt(i);
            // Check if the current char is a digit
            if (Character.isDigit(ch)) {
                int startIndex = i;
                int number = 0;
                // Build the number till we encounter a non-digit
                while (startIndex < length && Character.isDigit(s.charAt(startIndex))) {
                    number = number * 10 + s.charAt(startIndex) - '0';
                    startIndex++;
                }
                // Update the result with the current number and sign
                result += sign * number;
                // Move the pointer to the end of the number
                i = startIndex - 1;
            } else if (ch == '+') {
                // Set sign as positive for addition
                sign = 1;
            } else if (ch == '-') {
                // Set sign as negative for subtraction
                sign = -1;
            } else if (ch == '(') {
                // Push the result and sign on the stack before the parenthesis
                stack.push(result);
                stack.push(sign);
                // Reset result and sign for the expression inside the parenthesis
                result = 0;
                sign = 1;
            } else if (ch == ')') {
                // After closing parenthesis, result is the evaluated value inside the parenthesis
                // Pop the sign before the parenthesis and multiply it with the current result,
                // then add the result before the parenthesis
                result = stack.pop() * result + stack.pop();
            }
        }
        // Return the final result of the evaluation
        return result;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int calculate(string s) {
        stack<int> operands;  // Stack to hold numbers and signs (as integer)
        int result = 0;       // Accumulator for the current result
        int sign = 1;         // Current sign, starts with positive
        int n = s.size();     // Size of the input string

        for (int i = 0; i < n; ++i) {
            // If the character is a digit, it could be part of a multi-digit number.
            if (isdigit(s[i])) {
                int num = 0; // Initialize the number to be 0
                // Construct the number from the subsequent digits
                while (i < n && isdigit(s[i])) {
                    num = num * 10 + (s[i] - '0');
                    ++i;
                }
                // Add/Subtract the number to/from the current result
                result += sign * num;
                // Since the loop will increment i after, we decrease it here
                --i;
            } else if (s[i] == '+') {
                sign = 1;  // Set sign to positive for the next number
            } else if (s[i] == '-') {
                sign = -1; // Set sign to negative for the next number
            } else if (s[i] == '(') {
                // When encountering '(', push the current result and sign onto the stack
                operands.push(result);
                operands.push(sign);
                // Reset result and sign for the new sub-expression inside the parentheses
                result = 0;
                sign = 1;
            } else if (s[i] == ')') {
                // Ending a sub-expression, pop sign from stack and multiply it with the current result
                result *= operands.top();
                operands.pop();
                // Add the result of the sub-expression to the result up to the '('
                result += operands.top();
                operands.pop();
            }
            // Ignore whitespace and other non-math characters
        }
        return result; // Return the final result
    }
};

```

Typescript Solution

```typescript
function calculate(s: string): number {
    const stack: number[] = [];
    let currentSign = 1;      // This will hold the current sign, 1 for '+' and -1 for '-'
    let result = 0;           // This will accumulate the result of the arithmetic expression
    const length = s.length;  // Length of the input string

    // Iterate through each character of the input string
    for (let i = 0; i < length; ++i) {
        // Skip spaces in the expression
        if (s[i] === ' ') {
            continue;
        }

        // Update the sign for the next number
        if (s[i] === '+') {
            currentSign = 1;
        } else if (s[i] === '-') {
            currentSign = -1;
        } else if (s[i] === '(') {
            // Push the result and sign onto the stack before resetting them
            stack.push(result);
            stack.push(currentSign);
            result = 0;
            currentSign = 1;
        } else if (s[i] === ')') {
            // Pop the sign then the result from the stack and combine them
            result *= stack.pop() as number;
            result += stack.pop() as number;
        } else {
            // Parse the number and aggregate the result
            let value = 0;
            let j = i;
            // Continue for all subsequent digits to form the full number
            for (; j < length && !isNaN(Number(s[j])) && s[j] !== ' '; ++j) {
                value = value * 10 + (s[j].charCodeAt(0) - '0'.charCodeAt(0));
            }
            result += currentSign * value;
            i = j - 1; // Update the outer loop's index after processing the number
        }
    }

    // Return the computed result
    return result;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by the number of operations needed to parse and evaluate the expression.

- Looping through each character in the input string `s` of length `n` accounts for `O(n)` because each character is considered exactly once.
- Each digit in the string is processed and converted to an integer, which in the worst case, every character could be a digit, resulting in `O(n)` for the digit conversion process.
- The operations related to stack (pushing and popping) occur in constant time `O(1)` for each operation, but in the worst case, the total time complexity for all such operations is proportional to the number of parentheses in the input string. Since parentheses pairs cannot exceed `n/2`, the complexity due to stack operations is also `O(n)`.

Therefore, the overall time complexity of the algorithm is `O(n)`.

Space Complexity

The space complexity of the code is determined by the amount of additional memory needed to store intermediate results and the call stack (if applicable).

- The stack `stk` is used to store the result and sign at each level of parentheses, and in the worst case, where we have a pair of parentheses for every two characters (like "((...))"), the stack size could grow up to `n/2`. Therefore, the space complexity due to the stack is `O(n/2)`, which simplifies to `O(n)`.
- Variables `ans`, `sign`, `i`, `x`, `j`, and `n` use a constant amount of space, contributing `O(1)`.

Hence, the overall space complexity of the algorithm is `O(n)` (where `n` is the length of the string `s`).