150. Evaluate Reverse Polish Notation

Medium Stack Array Math

Problem Description

You are asked to evaluate an arithmetic expression provided as an array of strings, `tokens`, which uses Reverse Polish Notation (RPN). This notation places the operator after the operands. For example, the expression "3 4 +" in RPN is equivalent to "3 + 4" in standard notation. Your task is to calculate the result of the expression and return the resulting integer value.

Several points to consider for this problem are:

- The expression only contains the operators `+`, `-`, `*`, and `/`.
- Operands could be either integers or sub-expressions.
- When performing division, the result is always truncated towards zero.
- The expression does not contain any division by zero.
- The expression given is always valid and can be evaluated without error.
- The result of the evaluated expression and any intermediate operations will fit within a 32-bit integer.

Intuition

To solve this problem, we need to understand the nature of Reverse Polish Notation. In RPN, every time we encounter an operator, it applies to the last two operands that were seen. A `stack` is the perfect data structure for this evaluation because it allows us to keep track of the operands as they appear and then apply the operators in the correct order.

The intuition for the solution is as follows:

1. We iterate through each string (`token`) in the `tokens` array.
2. If the token is a number (single digit or multiple digits), we convert it to an integer and push it onto a `stack`.
3. If the token is an operator, we pop the last two numbers from the stack and apply this operator; these two numbers are the operands for the operator.
4. The result of the operation is then pushed back onto the stack.
5. After applying an operator, the stack should be in a state that is ready to process the next token.
6. When we've processed all the tokens, the stack will contain only one element, which is the final result of the expression.

Division in Python by default results in a floating-point number. Since the problem states that the division result should truncate toward zero, we explicitly convert the result to an `int`, which discards the decimal part.

The key here is to iterate through the tokens once and perform operations in order, ensuring the `stack`'s top two elements are the operands for any operator we come across.

Solution Approach

The solution makes use of a very simple yet powerful algorithm that utilizes a `stack` data structure to process the given tokens one by one. Here are the steps it follows:

1. Initialize an empty list `nums` that will act as a `stack` to store the operands.
2. Iterate over each token in the `tokens` array.
   - If the token is a numeric value (identified by either being a digit or having more than one character, which accounts for numbers like "-2"), we convert the token to an integer and push it onto the stack.
   - If the token is an operator (`+`, `-`, `*`, `/`), we perform the following:
     - Pop the top two elements from the stack. Since the last element added is at the top of the stack, we'll refer to these as the second operand (at `nums[-1]`) and the first operand (at `nums[-2]`) in that order.
     - Apply the operator to these two operands. For addition, subtraction, and multiplication, this is straightforward.
     - For division, we apply integer division which is the same as dividing and then applying the `int` function to truncate towards zero. This is important as it handles the truncation towards zero for negative numbers correctly. The simple floor division operator `//` in Python truncates towards negative infinity, which can give incorrect results for negative quotients.
     - The result of the operation is then placed back into the stack at the position of the first operand (`nums[-2]`).
     - The second operand (`nums[-1]`) which has already been used, is removed from the stack.
3. After processing all the tokens, there should be a single element left in the `nums` stack. This element is the result of evaluating the expression.

The algorithm used here is particularly efficient because it has a linear time complexity, processing each token exactly once. The space complexity of this approach is also linear, as it depends on the number of tokens that are pushed into the `stack`. The use of the stack ensures that the operands for any operator are always readily available at the top of the stack.

Here is a snippet of how the arithmetic operations are processed:

- Addition: `nums[-2] += nums[-1]`
- Subtraction: `nums[-2] -= nums[-1]`
- Multiplication: `nums[-2] *= nums[-1]`
- Division: `nums[-2] = int(nums[-2] / nums[-1])`

Once finished, the program returns `nums[0]` as the result of the expression.

Example Walkthrough

Let's use the following RPN expression as our example: `"2 1 + 3 *"` which, in standard notation, translates to `(2 + 1) * 3`.

By following the solution approach:

1. We initialize an empty list `nums` to serve as our stack: `nums = []`.
2. We iterate through the tokens: `["2", "1", "+", "3", "*"]`.
   a. The first token is `"2"`, which is a number. We push it onto the stack: `nums = [2]`.
   b. The second token is `"1"`, which is also a number. We push it onto the stack: `nums = [2, 1]`.
   c. The third token is `"+"`, which is an operator. We need to pop the top two numbers and apply the operator: - We pop the first operand: `secondOperand = nums.pop()`, which is `1`. Now `nums = [2]`.- We pop the second operand: `firstOperand = nums.pop()`, which is `2`. Now `nums = []`.- We add the two operands: `stackResult = firstOperand + secondOperand`, which equals `2 + 1 = 3`.- We push the result onto the stack: `nums = [3]`.
   d. The fourth token is `"3"`, a number. We push it onto the stack: `nums = [3, 3]`.
   e. The fifth token is `"*"`, an operator.- We pop the second operand, `secondOperand = nums.pop()`, which is `3`, leaving `nums = [3]`.- We pop the first operand, `firstOperand = nums.pop()`, which is `3`, leaving `nums = []`.- We multiply the two operands: `stackResult = firstOperand * secondOperand`, which equals `3 * 3 = 9`.- We push the result onto the stack: `nums = [9]`.
3. After processing all the tokens, we are left with a single element in our stack `nums = [9]`, which is our result.

So, the given RPN expression `"2 1 + 3 *"` evaluates to `9`. Thus, the function would return `9` as the result of the expression.

Python Solution

```python
from typing import List

class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        # Stack for storing numbers
        number_stack = []

        # Loop over each token in the input list
        for token in tokens:
            # If the token represents a number (accounting for negative numbers)
            if len(token) > 1 or token.isdigit():
                # Convert the token to an integer and push it onto the stack
                number_stack.append(int(token))
            else:
                # Perform the operation based on the operator
                if token == "+":
                    # Pop the last two numbers, add them, and push the result back
                    number_stack[-2] += number_stack[-1]
                elif token == "-":
                    # Pop the last two numbers, subtract the second from the first, and push back
                    number_stack[-2] -= number_stack[-1]
                elif token == "*":
                    # Pop the last two numbers, multiply, and push the result back
                    number_stack[-2] *= number_stack[-1]
                else:
                    # Division
                    # Ensure integer division for negative numbers too
                    number_stack[-2] = int(float(number_stack[-2]) / number_stack[-1])
                # Pop the last number (second operand) from the stack as it's been used
                number_stack.pop()

        # Return the result which is the only number left in the stack
        return number_stack[0]
```

Java Solution

```java
class Solution {
    public int evalRPN(String[] tokens) {
        Deque<Integer> stack = new ArrayDeque<>(); // Create a stack to hold integer values

        // Iterate over each token in the input array
        for (String token : tokens) {
            // Check if the token is a number (either single digit or multi-digit)
            if (token.length() > 1 || Character.isDigit(token.charAt(0))) {
                // Push the number onto the stack
                stack.push(Integer.parseInt(token));
            } else {
                // Pop the top two elements for the operator
                int secondOperand = stack.pop();
                int firstOperand = stack.pop();

                // Apply the operator on the two operands based on the token
                switch (token) {
                    case "+":
                        stack.push(firstOperand + secondOperand); // Add and push the result
                        break;
                    case "-":
                        stack.push(firstOperand - secondOperand); // Subtract and push the result
                        break;
                    case "*":
                        stack.push(firstOperand * secondOperand); // Multiply and push the result
                        break;
                    case "/":
                        stack.push(firstOperand / secondOperand); // Divide and push the result
                        break;
                }
            }
        }

        // The final result is the only element in the stack, return it
        return stack.pop();
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <stack>

class Solution {
public:
    int evalRPN(vector<string>& tokens) {
        // Create a stack to keep track of integers for evaluation
        stack<int> numbers;

        // Iterate over each token in the Reverse Polish Notation expression
        for (const string& token : tokens) {
            // If the token represents a number (can be multiple digits or negative)
            if (token.size() > 1 || isdigit(token[0])) {
                // Convert the string token to an integer and push onto the stack
                numbers.push(stoi(token));
            } else {
                // Pop the second operand from the stack
                int operand2 = numbers.top();
                numbers.pop();

                // Pop the first operand from the stack
                int operand1 = numbers.top();
                numbers.pop();

                // Perform the operation based on the type of operator
                switch (token[0]) {
                    case '+': // Addition
                        numbers.push(operand1 + operand2);
                        break;
                    case '-': // Subtraction
                        numbers.push(operand1 - operand2);
                        break;
                    case '*': // Multiplication
                        numbers.push(operand1 * operand2);
                        break;
                    case '/': // Division
                        numbers.push(operand1 / operand2);
                        break;
                }
            }
        }

        // The final result is the only number remaining on the stack
        return numbers.top();
    }
};
```

Typescript Solution

```typescript
// Evaluates the value of an arithmetic expression in Reverse Polish Notation.
// Input is an array of tokens, where each token is either an operator or an operand.
// Valid operators are: '+', '-', '*', and '/'.
// Assumes the RPN expression is valid.
function evalRPN(tokens: string[]): number {
    // Stack to hold the operands for evaluation.
    const stack: number[] = [];

    // Helper function to check if a string is a numeric value.
    function isNumber(token: string): boolean {
        return !isNaN(parseFloat(token)) && isFinite(Number(token));
    }

    // Process each token in the RPN expression.
    for (const token of tokens) {
        // If the token is a number, push it onto the stack.
        if (isNumber(token)) {
            stack.push(Number(token));
        } else {
            // Since we know the token is an operator, pop two operands from the stack.
            const secondOperand = stack.pop();
            const firstOperand = stack.pop();

            // Safety check: Ensure operands are valid numbers to avoid runtime errors.
            if (typeof firstOperand === 'undefined' || typeof secondOperand === 'undefined') {
                throw new Error("Invalid Expression: Insufficient operands for the operator.");
            }

            // Perform the operation according to the current token and push the result onto the stack.
            switch (token) {
                case '+':
                    stack.push(firstOperand + secondOperand);
                    break;
                case '-':
                    stack.push(firstOperand - secondOperand);
                    break;
                case '*':
                    stack.push(firstOperand * secondOperand);
                    break;
                case '/':
                    // Use truncation to conform to the requirements of integer division in RPN.
                    // The '~~' is a double bitwise NOT operator, used here as a substitute for Math.trunc
                    // which is not available in the browser specified in the problem statement.
                    stack.push(~~(firstOperand / secondOperand));
                    break;
                default:
                    throw new Error("Invalid token: Encountered an unknown operator.");
            }
        }
    }

    // The result of the expression is the last element of the stack.
    // Safety check: there should only be one element left in the stack.
    if (stack.length !== 1) {
        throw new Error("Invalid Expression: The final stack should only contain one element.");
    }

    return stack[0];
}
```

Time and Space Complexity

The given Python function `evalRPN` evaluates Reverse Polish Notation (RPN) expressions. The time complexity and space complexity analysis are as follows:

Time Complexity

The time complexity of this function is $O(n)$, where n is the number of tokens in the input list `tokens`. This is because the function iterates through each token exactly once. Each operation within the loop, including arithmetic operations and stack operations (append and pop), can be considered to have constant time complexity, $O(1)$.

Space Complexity

The space complexity of the code is $O(n)$ in the worst case, where n is the number of tokens in the list. This worst-case scenario occurs when all tokens are numbers and are thus pushed onto the stack. In the best-case scenario, where the input is balanced with numbers and operators, the space complexity could be better than $O(n)$, but the upper bound remains $O(n)$. The auxiliary space required is for the `nums` stack used to perform the calculations. There are no other data structures that use significant memory.