2232. Minimize Result by Adding Parentheses to Expression

Medium · String · Enumeration · Leetcode Link

Problem Description

You are given an expression in the form of <num1>+<num2>, where both <num1> and <num2> are positive integers. The goal is to add a single pair of parentheses to the expression to minimize its value after the addition is performed. There is a requirement that the left parenthesis has to be placed somewhere to the left of the '+' sign, and the right parenthesis has to be placed somewhere to the right of the '+' sign.

The task is to modify the expression by adding these parentheses such that the computed result is the smallest possible value, and return the modified expression as a string. If there are several ways to insert the parentheses that yield the same minimum value, any of those valid expressions can be returned.

The problem ensures that the result before and after inserting the parentheses will both fit within the bounds of a signed 32-bit integer.

Intuition

The intuition behind the solution stems from the fundamental properties of arithmetic, particularly the effect of the order of operations (parentheses, exponents, multiplication and division, addition and subtraction - PEMDAS) on the outcome of expressions. Placing parentheses around a part of an expression changes the order in which operations are performed.

The given expression <num1>+<num2> will always result in addition. However, by strategically placing a pair of parentheses, we can create a situation where a multiplication occurs before the addition, which can potentially lower the total value. For example, if num1 = 2 and num2 = 3, the expression 2+3 would normally evaluate to 5. But by changing it to 2+(3), it still evaluates to 5. However, changing it to (2+3) would still evaluate to 5. The point here is to see if, by splitting either num1 or num2 and performing multiplication with one of those split parts, we can reduce the overall value.

The solution involves iterating through all possible positions where the parentheses could be placed in the expression, calculating the result for each arrangement, and keeping track of the smallest result along with the corresponding arrangement. Specifically, we split the num1 and num2 into two parts at different positions: num1 is split into a and b, and num2 is split into c and d. We then calculate the result by computing (b+c)*a*d. We compare this result to the minimum value we've seen so far and update the minimum and the answer when we find a smaller value.

By iterating through all possible splits for num1 and num2, we ensure that we cover all possible combinations of parentheses placements. This brute-force approach guarantees that we find the smallest possible expression value.

Solution Approach

The implementation uses a brute-force strategy to enumerate all possibilities for where to place a pair of parentheses in the given expression. It does so by splitting the expression around the '+' sign to yield two substrings: l which corresponds to <num1>, and r which corresponds to <num2>. These substrings represent the left and right parts of the expression around the '+' sign, respectively.

The algorithm iteratively partitions each of these substrings l and r into two parts: one up to a certain position, which will be inside the parentheses and be added to each other, and another part from that position to the end of the string, which will be used for multiplication.

A couple of nested loops are used to test every possible pair of positions to place the parentheses. For the string l, valid split positions range from [0, m) where m is the length of l. For the string r, valid split positions range from [0, n) where n is the length of r. For each combination of i in l and j in r, we consider the calculation:

- c is the sum of the integers formed by the substrings l[i:] and r[:j+1].
- a multiplies the result with the integer formed by substring l[:i], defaulting to 1 if i is 0 (meaning no substring).
- b multiplies the result with the integer formed by substring r[j+1:], defaulting to 1 if j is n-1 (meaning no substring).

The total expression value for the current placement of parentheses is then a * b * c. This value is compared to the current minimum value mi, and if it is smaller, the minimum value is updated, and the corresponding string representation ans is formed using f-string formatting. This representation inserts parentheses at the calculated positions within l and r.

After all possible placements for the parentheses have been tested, the ans that corresponds to the smallest found value mi is returned.

This approach guarantees that all placements are considered and that the optimal result is always found. The choice of brute-force is suitable here due to the small size of the problem space, and such an approach is straightforward to implement and understand, despite not being the most efficient in terms of computational complexity.

Example Walkthrough

To illustrate the solution approach with an example, let's take the expression 123+4567. We want to insert a pair of parentheses in such a way that the sum of the numbers is minimized after performing the operations within parentheses first.

Following the described brute-force strategy, we'll split the expression at the '+' sign. This gives us num1 = "123" and num2 = "4567". We will consider all possible ways to split these strings into two parts and compute the resulting expression as per the given approach.

For num1, we have 3 split positions:

1. ("","123") → We can't place parentheses as there is no number before the '+'.
2. ("1","23") → The calculation would be 1 * ((23+4567)).
3. ("12","3") → The calculation would be 12 * ((3+4567)).
4. ("123","") → The calculation would be 123 * ((+4567)), which is essentially 123 + 4567, the original expression without any meaningful parentheses.

For num2, we also have 4 split positions:

1. ("","4567") → We can't place parentheses as there is no number after the '+'.
2. ("4","567") → The calculation would be ((123+4)) * 567.
3. ("45","67") → The calculation would be ((123+45)) * 67.
4. ("456","7") → The calculation would be ((123+456)) * 7.
5. ("4567","") → Same as the case with num1, no meaningful parentheses can be inserted.

Now we evaluate each combination to find the minimum value. For example, let's consider splitting num1 at 2 ("12","3") and num2 at 2 ("45","67"), which implies we insert parentheses to get (12+45)*3*67. Performing the operations inside the parentheses first gives us (57)*3*67.

Let's compute the result:

- Inside parentheses 12+45 = 57
- Multiplying by the remaining parts: 57 * 3 * 67 = 11457

As we loop through all combinations, we find the one that gives us the smallest result. For simplicity, let's assume that in our example (12+45)*3*67 gives us the smallest result. Then, we return this manipulated string representation as our answer: (12+45)*3*67.

We perform a similar brute-force test for all the other possible splits of num1 and num2, keep track of the smallest result, and update our answer each time we find a new smallest result. By the end of our nested loops, we will have considered every possible placement for parentheses and arrive at the minimum possible evaluated expression, which we return.

Python Solution

```python
class Solution:
    def minimizeResult(self, expression: str) -> str:
        # Split the input string into two parts separated by the '+'
        left_part, right_part = expression.split("+")

        # Get the length of the left and right parts
        left_len = len(left_part)
        right_len = len(right_part)

        # Initialize minimum value to positive infinity
        min_value = float('inf')  # or use `math.inf`

        # Initialize the answer string
        answer = ""

        # Iterate through all possible non-empty prefixes of left_part
        for i in range(left_len):
            # Iterate through all possible non-empty suffixes of right_part
            for j in range(right_len):
                # Calculate the sum of the chosen parts and evaluate the expression
                current_sum = int(left_part[i:]) + int(right_part[:j+1])

                # Take the prefix of left_part and the suffix of right_part;
                # If empty (at the extremities), their value should be 1 (neutral for multiplication)
                left_prefix_value = 1 if i == 0 else int(left_part[:i])
                right_suffix_value = 1 if j == right_len - 1 else int(right_part[j+1:])

                # Calculate the temporary result of the current configuration
                current_result = left_prefix_value * right_suffix_value * current_sum

                # If the current result is less than the previous minimum, update it
                if current_result < min_value:
                    min_value = current_result

                    # Format and update the answer string to represent the current minimum expression
                    # Surround the chosen parts with parentheses to indicate their placement in the expression
                    answer = f"{left_part[:i]}({left_part[i:]}+{right_part[:j+1]}){right_part[j+1:]}"

        # Return the formatted answer string
        return answer
```

Java Solution

```java
class Solution {
    public String minimizeResult(String expression) {
        // Find the index of the '+' in the expression
        int plusIndex = expression.indexOf('+');

        // Split the expression into left and right parts
        String leftPart = expression.substring(0, plusIndex);
        String rightPart = expression.substring(plusIndex + 1);

        // Get the lengths of the left and right parts
        int leftLength = leftPart.length();
        int rightLength = rightPart.length();

        // Initialize minimum value to the largest possible integer
        int minResult = Integer.MAX_VALUE;

        // Variable to store the answer string with minimum value
        String answer = "";

        // Check every possible pair of parentheses
        for (int i = 0; i < leftLength; ++i) {
            for (int j = 0; j < rightLength; ++j) {

                // Calculate the sum inside the parentheses
                int parenthesizedSum = Integer.parseInt(leftPart.substring(i)) +
                                       Integer.parseInt(rightPart.substring(0, j + 1));

                // Compute the left multiplication factor
                int leftMulFactor = i == 0 ? 1 : Integer.parseInt(leftPart.substring(0, i));

                // Compute the right multiplication factor
                int rightMulFactor = j == rightLength - 1 ? 1 : Integer.parseInt(rightPart.substring(j + 1));

                // Calculate the total for the current partition
                int total = leftMulFactor * rightMulFactor * parenthesizedSum;

                // Check if the total is the new minimum
                if (total < minResult) {
                    // Update the minimum result
                    minResult = total;

                    // Format answer string with the current minimum partition
                    answer = String.format("%s(%s+%s)%s",
                                leftPart.substring(0, i),
                                leftPart.substring(i),
                                rightPart.substring(0, j + 1),
                                rightPart.substring(j + 1));
                }
            }
        }

        // Return the answer string with the minimum result after placing parentheses
        return answer;
    }
}
```

C++ Solution

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

// Function to convert a vector of char digits to an integer, or return 1 if the vector is empty
int GetNum(const std::vector<char>& digits) {
    if (digits.empty()) {
        return 1;
    }
    return std::stoi(std::string(digits.begin(), digits.end()));
}

// Function to find the optimal insertion of parentheses in an addition expression to minimize result
std::string MinimizeResult(const std::string& expression) {
    // Find the position of the plus sign
    size_t plusPosition = expression.find('+');
    // Split the expression into two parts based on the plus sign
    std::string leftPart = expression.substr(0, plusPosition);
    std::string rightPart = expression.substr(plusPosition + 1);

    // Initialize minimum sum to a very large number
    int minimumSum = INT_MAX;
    // String to hold the answer
    std::string answer;

    // Vectors to hold the digits of the current partition
    std::vector<char> prefixArray, currentLeftArray(leftPart.begin(), leftPart.end()),
                      currentRightArray(rightPart.begin(), rightPart.end()), suffixArray;

    // Iterate through the left part of the equation character by character
    for (size_t i = 0; i <= leftPart.size(); ++i) {
        // Reset the current right array and suffix array for each iteration of the left array
        currentRightArray.assign(rightPart.begin(), rightPart.end());
        suffixArray.clear();

        // Iterate through the right part of the equation character by character
        for (size_t j = 0; j <= rightPart.size(); ++j) {
            // Calculate the current value with the parentheses inserted
            int currentValue = (GetNum(currentLeftArray) + GetNum(currentRightArray)) *
                               GetNum(prefixArray) * GetNum(suffixArray);
            // If the current value is less than the minimum sum found so far
            if (currentValue < minimumSum) {
                // Update the minimum sum and the answer string
                minimumSum = currentValue;
                answer = std::string(prefixArray.begin(), prefixArray.end()) +
                         "(" + std::string(currentLeftArray.begin(), currentLeftArray.end()) +
                         "+" + std::string(currentRightArray.begin(), currentRightArray.end()) +
                         ")" + std::string(suffixArray.begin(), suffixArray.end());
            }
            // If there are still characters in the right array, relocate the last one to the suffix array
            if (!currentRightArray.empty()) {
                suffixArray.insert(suffixArray.begin(), currentRightArray.back());
                currentRightArray.pop_back();
            }
        }
        // If there are still characters in the left array, relocate the first one to the prefix array
        if (!currentLeftArray.empty()) {
            prefixArray.push_back(currentLeftArray.front());
            currentLeftArray.erase(currentLeftArray.begin());
        }
    }

    // Return the answer
    return answer;
}
```

Typescript Solution

```typescript
// Function to find the optimal insertion of parentheses in an addition expression to minimize result
function minimizeResult(expression: string): string {
    // Split the expression into two numbers based on the plus sign
    const [leftPart, rightPart] = expression.split('+');
    // Initialize minimum sum to a very large number
    let minimumSum = Number.MAX_SAFE_INTEGER;
    // String to hold the answer
    let answer = '';
    // Split the left and right parts of the equation into arrays
    let prefixArray = [], currentLeftArray = leftPart.split(''), currentRightArray = rightPart.split(''), suffixArray = [];

    // Iterate through the left part of the equation
    while (currentLeftArray.length) {
        // Reset the current right array and suffix array for each iteration of the left array
        currentRightArray = rightPart.split('');
        suffixArray = [];

        // Iterate through the right part of the equation
        while (currentRightArray.length) {
            // Calculate the current value with the parentheses inserted
            let currentValue = (getNum(currentLeftArray) + getNum(currentRightArray)) * getNum(prefixArray) * getNum(suffixArray);
            // If the current value is less than the minimum sum found so far
            if (currentValue < minimumSum) {
                // Update the minimum sum and the answer string
                minimumSum = currentValue;
                answer = `${prefixArray.join('')}(${currentLeftArray.join('')}+${currentRightArray.join('')})${suffixArray.join('')}`
            }
            // Move the last element from right to suffix for next iteration
            suffixArray.unshift(currentRightArray.pop());
        }
        // Move the first element from left to prefix for next iteration
        prefixArray.push(currentLeftArray.shift());
    }

    // Return the answer
    return answer;
}

// Helper function to convert an array of string digits to a number, or return 1 if the array is empty
function getNum(digitsArray: Array<string>): number {
    return digitsArray.length ? Number(digitsArray.join('')) : 1;
}
```

Time and Space Complexity

Time Complexity

The given code snippet involves a nested loop to check every possible way to insert parentheses into a mathematical expression to minimize its value. Here's a step-by-step analysis:

- We are using two nested loops. The outer loop runs m times, and the inner loop runs n times, where m is the length of the string l before the "+" operator, and n is the length of the string r after the "+" operator.
- Within the inner loop, the operations performed are constant time operations, which includes slicing of strings (which is O(k), where k is the slice length), integer conversion and arithmetic operations.
- The string formatting inside the loop is also a constant time operation relative to the loop iterations, as it depends only on the lengths of the strings involved.

Given that the two loops are nested, the total time complexity is $O(m * n * k)$, where k is the maximum slice length, which would be at most $\max(m, n)$ in this scenario.

Because slicing lengths vary with each loop iteration but are bounded by the lengths of l and r, the factor of k in the complexity can be subsumed under m and n, leading to a simplified time complexity of $O(m * n)$.

Space Complexity

The space complexity of the code can be broken down as follows:

- Constant space to store integers (mi for the minimum result, m and n for the lengths of l and r, a, b, c, and t for intermediate calculations).
- Variable space for the string ans - since it stores strings of length proportional to the input expression, its space requirement will be $O(m + n)$.
- Additional space for the slices of the strings within the loop. These strings will also have their lengths bounded by m and n, however, this space is reused in each iteration and does not cumulatively add to the complexity.

Taking into account the largest factors, the overall space complexity is $O(m + n)$. This includes the space needed to store the input and the space for the intermediate string formed while concatenating the result.