Documentation for the "Different Ways to Add Parentheses" Problem

Overview

The "Different Ways to Add Parentheses" problem is a computational challenge that involves evaluating a mathematical expression containing numbers and operators (specifically +, -, and *). The task is to return all possible results from computing the expression in different ways by adding parentheses. This problem is categorized as a dynamic programming and recursion problem due to its nature of breaking down a complex expression into simpler sub-expressions. The solution requires considering each operator in the expression as a potential point of division, thereby allowing for multiple interpretations of the expression based on how the numbers and operations are grouped.

Input and Output Specifications

The input to the function is a string representing a mathematical expression, which consists of integers and the operators +, -, and *. The constraints specify that the length of the expression can range from 1 to 20 characters, and the integer values are guaranteed to be within the range of 0 to 99 without leading signs. The output is a list of integers representing all possible results that can be obtained by evaluating the expression in every possible way using different placements of parentheses. The results can appear in any order, and the total number of unique results is constrained not to exceed 10,000.

Approach to the Solution

The solution employs a divide-and-conquer approach facilitated by recursion. The primary strategy is to iterate through each character in the expression and identify the operators. For each operator encountered, the expression is divided into two parts: the left side (everything before the operator) and the right side (everything after the operator). The function then recursively computes the possible results for both sides. Once the results for the left and right sub-expressions are computed, they are combined according to the operator's nature (e.g., addition, subtraction, multiplication) to yield new possible outcomes.

Memoization for Optimization

Given the potential for overlapping subproblems in this recursive approach (where the same sub-expression might be evaluated multiple times), memoization is employed. This technique involves storing the results of previously computed expressions in a dictionary, which allows for quick retrieval when the same sub-expression is encountered again. By using memoization, the overall efficiency of the solution is significantly improved, reducing redundant calculations and thereby optimizing the time complexity of the algorithm.

Complexity Analysis and Practical Applications

The time complexity of the solution can vary depending on the structure of the expression and the number of operators present. In the worst case, the complexity can approach exponential growth relative to the number of operators due to the myriad ways in which they can be grouped. However, the use of memoization effectively mitigates this issue. The space complexity is primarily driven by the storage of computed results in the memoization dictionary. This problem is not only a great exercise in recursion and dynamic programming but also has real-world applications in areas such as expression parsing, compiler design, and symbolic computation, where understanding how different parenthesizations can alter the meaning or value of expressions is critical.