Documentation

Problem Overview

The problem, Expression Add Operators, requires constructing all possible valid expressions by inserting binary operators (+, -, *) between digits in a given string num. These expressions should evaluate a given integer target. The constraints emphasize that operands should not have leading zeros (e.g., "05") unless the operand is exactly "0". This problem involves recursion and backtracking to explore all possible combinations of operators and operands.

Approach and Methodology

The problem is best solved using a backtracking technique. This approach systematically explores all possible expressions by trying each operator between every digit split of the string. The recursive function tracks the current state of the expression, the previous operand, and the total computed value of the expression so far. The use of prev_operand is crucial for handling the precedence of the multiplication operator (*), as it needs to "undo" the previously added value before multiplying and re-adding to maintain the correct evaluation order.

Each recursive call considers three possible operations:

1. Addition (+) adds the current operand to the total value.
2. Subtraction (-) subtracts the current operand from the total value.
3. Multiplication (*) adjusts the result by combining the previous operand with the current operand.

Edge Cases and Considerations

Handling leading zeros is a key challenge. For example, the string "105" should allow expressions like "10-5" but not "1*05". To manage this, substrings with leading zeros are skipped unless the substring is a single "0". Another important consideration is managing the size of the input. Since the string length is constrained to at most 10 digits, the backtracking approach remains feasible despite the exponential number of possible combinations.

Edge cases also include scenarios where no valid expressions exist, such as when the target is unreachable given the digits in num. For example, if num = "3456237490" and target = 9191, no combination of operators can produce the desired result, and the output should be an empty list.

Efficiency and Complexity

The solution has a time complexity of $(O(4^n))$, where (n) is the length of the input string. This arises because, at each step, we may explore up to three operators and one skip. While this exponential growth seems significant, the constraints ensure that the input size remains manageable. The space complexity is (O(n)), primarily due to the recursion stack used during backtracking and the storage of intermediate expressions.

Practical Applications

This problem has practical relevance in evaluating arithmetic expressions and validating parsing logic in compilers or calculators. Additionally, it serves as an excellent exercise for understanding recursion, backtracking, and precedence rules in mathematical computations. By solving this, programmers can strengthen their ability to design solutions for similar combinatorial optimization problems, such as generating valid parentheses or evaluating mathematical expressions with operator precedence.