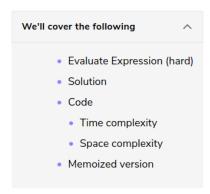
# Solution Review: Problem Challenge 1



# **Evaluate Expression (hard)**

Given an expression containing digits and operations (+, -, \*), find all possible ways in which the expression can be evaluated by grouping the numbers and operators using parentheses.

#### Example 1:

```
Input: "1+2*3"
Output: 7, 9
Explanation: 1+(2*3) => 7 and (1+2)*3 => 9
```

#### Example 2:

```
Input: "2*3-4-5"
Output: 8, -12, 7, -7, -3
Explanation: 2*(3-(4-5)) => 8, 2*(3-4-5) => -12, 2*3-(4-5) => 7, 2*(3-4)-5 => -7, (2*3)-4-5 => -3
```

### Solution #

This problem follows the Subsets pattern and can be mapped to Balanced Parentheses. We can follow a similar BFS approach.

Let's take Example-1 mentioned above to generate different ways to evaluate the expression.

- 1. We can iterate through the expression character-by-character.
- 2. we can break the expression into two halves whenever we get an operator (+, -, \*).
- 3. The two parts can be calculated by recursively calling the function.
- Once we have the evaluation results from the left and right halves, we can combine them to produce all results.

## Code #

Here is what our algorithm will look like:



```
# break the equation here into two parts and make recursively calls

leftParts = diff ways_to_evaluate_expression(input[0:i])

rightParts = diff_ways_to_evaluate_expression(input[i+1:])

for part1 in leftParts:

for part2 in rightParts:

if char == '+':

result.append(part1 + part2)

elif char == '-':

result.append(part1 - part2)

elif char == '*':

result.append(part1 * part2)

return result

def main():

print("Expression evaluations: " +

| str(diff_ways_to_evaluate_expression("1+2*3")))

print("Expression evaluations: " +

| str(diff_ways_to_evaluate_expression("2*3-4-5")))

Run

Run

Save Reset
```

#### Time complexity

The time complexity of this algorithm will be exponential and will be similar to Balanced Parentheses. Estimated time complexity will be  $O(N*2^N)$  but the actual time complexity ( $O(4^n/\sqrt{n})$ ) is bounded by the Catalan number and is beyond the scope of a coding interview. See more details here.

## Space complexity #

The space complexity of this algorithm will also be exponential, estimated at  $O(2^N)$  though the actual will be  $O(4^n/\sqrt{n})$ .

### Memoized version #

The problem has overlapping subproblems, as our recursive calls can be evaluating the same sub-expression multiple times. To resolve this, we can use memoization and store the intermediate results in a **HashMap**. In each function call, we can check our map to see if we have already evaluated this sub-expression before. Here is the memoized version of our algorithm; please see highlighted changes:

```
🤑 Python3
                                     Js JS
👙 Java
                         ⊚ C++
        diff ways to evaluate expression(input)
      return diff_ways_to_evaluate_expression_rec({}, input)
    def diff_ways_to_evaluate_expression_rec(map, input):
        result.append(int(input))
           if not char.isdigit():
            leftParts = diff_ways_to_evaluate_expression_rec(
             rightParts = diff_ways_to_evaluate_expression_rec(
             for part1 in leftParts:
              for part2 in rightParts:
  if char == '+':
                   result.append(part1 + part2)
                   result.append(part1 - part2)
                 elif char == '*':
                  result.append(part1 * part2)
```

