

## Grokking the Coding Interview: Patterns for Coding Questions

63% completed


 Search Course

 Solution Review: Problem Challenge 1

### Pattern: Subsets

-  Introduction
-  Subsets (easy)
-  Subsets With Duplicates (easy)
-  Permutations (medium)
-  String Permutations by changing case (medium)
-  Balanced Parentheses (hard)
-  Unique Generalized Abbreviations (hard)
-  Problem Challenge 1
-  Solution Review: Problem Challenge 1
-  Problem Challenge 2
-  Solution Review: Problem Challenge 2
-  Problem Challenge 3
-  Solution Review: Problem Challenge 3

### Pattern: Modified Binary Search

-  Introduction
-  Order-agnostic Binary Search (easy)
-  Ceiling of a Number (medium)
-  Next Letter (medium)
-  Number Range (medium)
-  Search in a Sorted Infinite Array (medium)
-  Minimum Difference Element (medium)
-  Bitonic Array Maximum (easy)
-  Problem Challenge 1
-  Solution Review: Problem Challenge 1
-  Problem Challenge 2
-  Solution Review: Problem Challenge 2
-  Problem Challenge 3
-  Solution Review: Problem Challenge 3

### Pattern: Bitwise XOR

-  Introduction
-  Single Number (easy)
-  Two Single Numbers (medium)
-  Complement of Base 10 Number (medium)
-  Problem Challenge 1
-  Solution Review: Problem Challenge 1

### Pattern: Top 'K' Elements

-  Introduction
-  Top 'K' Numbers (easy)
-  Kth Smallest Number (easy)

## Solution Review: Problem Challenge 1

### We'll cover the following

- Evaluate Expression (hard)
- Solution
- Code
- Time complexity
- Space complexity
- Memoized version

### 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.
4. 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:

 Java	 Python3	 C++	 JS
--	---	---	--

```

7   for (let i = 0; i < input.length; i++) {
8     const char = input[i];
9     if (isNaN(parseInt(char, 10))) { // if not a digit
10       // break the equation here into two parts and make recursively calls
11       const leftParts = diff_ways_to_evaluate_expression(input.substring(0, i));
12       const rightParts = diff_ways_to_evaluate_expression(input.substring(i + 1));
13       for (let l = 0; l < leftParts.length; l++) {
14         for (let r = 0; r < rightParts.length; r++) {
15           let part1 = leftParts[l],
16               part2 = rightParts[r];
17           if (char === '+') {
18             result.push(part1 + part2);
19           } else if (char === '-') {
20             result.push(part1 - part2);
21           } else if (char === '*') {
22             result.push(part1 * part2);
23           }
24         }
25       }
26     }
27   }
28 }
29
30 return result;
31 }
32
33 console.log(`Expression evaluations: ${diff_ways_to_evaluate_expression('1+2*3')}`);
34 console.log(`Expression evaluations: ${diff_ways_to_evaluate_expression('2*3-4-5')}`);

```

RUN
SAVE
RESET


Output

```
Expression evaluations: 7,9
```

1.957s

Expression evaluations: 8,-12,7,-7,-3

### 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:

Java	Python3	C++	JS JS
------	---------	-----	-------

```
function diff_ways_to_evaluate_expression(input) {
    return diff_ways_to_evaluate_expression_rec({}, input);
}

function diff_ways_to_evaluate_expression_rec(map, input) {
    if (input in map) {
        return map[input];
    }

    const result = [];
    // base case: if the input string is a number, parse and add it to output.
    if (!(input.includes('+')) && !(input.includes('-')) && !(input.includes('*'))) {
        result.push(parseInt(input));
    } else {
        for (let i = 0; i < input.length; i++) {
            const char = input[i];
            if (isNaN(parseInt(char, 10))) { // if not a digit
                // break the equation here into two parts and make recursively calls
                const leftParts = diff_ways_to_evaluate_expression_rec(map, input.substring(0, i));
                const rightParts = diff_ways_to_evaluate_expression_rec(map, input.substring(i + 1));
                for (let l = 0; l < leftParts.length; l++) {
                    for (let r = 0; r < rightParts.length; r++) {
                        let part1 = leftParts[l],
                            part2 = rightParts[r];
                        if (char === '+') {
                            result.push(part1 + part2);
                        } else if (char === '-') {
                            result.push(part1 - part2);
                        }
                    }
                }
            }
        }
    }
    map[input] = result;
    return result;
}
```

**RUN** SAVE RESET Close 1.578s

Output  
Expression evaluations: 7,9  
Expression evaluations: 8,-12,7,-7,-3

### Pattern: K-way merge ^

- Introduction
- Merge K Sorted Lists (medium)
- Kth Smallest Number in M Sorted Lists (Medium)
- Kth Smallest Number in a Sorted Matrix (Hard)
- Smallest Number Range (Hard)
- Problem Challenge 1
- Solution Review: Problem Challenge 1

### Pattern : 0/1 Knapsack ^ (Dynamic Programming)

- Introduction
- 0/1 Knapsack (medium)
- Equal Subset Sum Partition (medium)
- Subset Sum (medium)
- Minimum Subset Sum Difference (hard)
- Problem Challenge 1
- Solution Review: Problem Challenge 1

### Pattern: Topological Sort (Graph) ^

- Introduction
- Topological Sort (medium)
- Tasks Scheduling (medium)
- Tasks Scheduling Order (medium)
- All Tasks Scheduling Orders (hard)
- Alien Dictionary (hard)
- Problem Challenge 1
- Solution Review: Problem Challenge 1
- Problem Challenge 2

Interviewing soon? We've partnered with Hired so that companies apply to you instead of you applying to them. [See how](#) X

**Back** Next →

Problem Challenge 1 Mark as Complete

Report an Issue Ask a Question