<https://leetcode.com/problems/expression-add-operators/description/>

Given a string num that contains only digits and an integer target, return all possibilities to insert the binary operators '+', '-', and/or '\*' between the digits of num so that the resultant expression evaluates to the target value.

Note that operands in the returned expressions should not contain leading zeros.

**Example 1:**

**Input:** num = "123", target = 6

**Output:** ["1\*2\*3","1+2+3"]

**Explanation:** Both "1\*2\*3" and "1+2+3" evaluate to 6.

**Example 2:**

**Input:** num = "232", target = 8

**Output:** ["2\*3+2","2+3\*2"]

**Explanation:** Both "2\*3+2" and "2+3\*2" evaluate to 8.

**Example 3:**

**Input:** num = "3456237490", target = 9191

**Output:** []

**Explanation:** There are no expressions that can be created from "3456237490" to evaluate to 9191.

**Constraints:**

1 <= num.length <= 10

num consists of only digits.

-2^31 <= target <= 2^31 - 1

**Attempt 1: 2024-07-05**

**Solution 1: Backtracking (120 min)**

class Solution {

    public List<String> addOperators(String num, int target) {

        List<String> result = new ArrayList<>();

        helper(num, target, 0, 0, 0, new StringBuilder(), result);

        return result;

    }

    private void helper(String num, int target, int index, long preOperand, long curTotal, StringBuilder sb, List<String> result) {

        if(index == num.length() && curTotal == target) {

            result.add(sb.toString());

            return;

        }

        for(int i = index; i < num.length(); i++) {

            // Skip numbers with leading zeros (unless the number itself is zero)

            /\*\*

             \* Refer to

             \* https://discuss.leetcode.com/topic/24523/java-standard-backtrace-ac-solutoin-short-and-clear/63?page=4

             \* explain this condition:

                if(i != pos && num.charAt(pos) == '0') break;

                I think this condition it to exclude numbers with leading zeros

                Valid answers are:

                "105", 5 -> ["1\*0+5","10-5"]

                "00", 0 -> ["0+0", "0-0", "0\*0"]

                but with leading zeros we will have

                "105", 5 -> ["1\*0+5","10-5", "1\*05"]

                "00", 0 -> ["0+0", "0-0", "0\*0", "00"]

            \*/

            // Refer to

            // https://leetcode.com/problems/expression-add-operators/discuss/71895/Java-Standard-Backtrace-AC-Solutoin-short-and-clear/239225

            // corner case: if current position is 0, we can only use it as a single digit number, should be 0

            // if it is not a single digit number with leading 0, it should be considered as an invalid number

            if(i != index && num.charAt(index) == '0') {

                break;

            }

            // Reserve current StringBuilder length, prepare for backtracking

            /\*\*

             \* Refer to

             \* https://discuss.leetcode.com/topic/32068/java-simple-solution-beats-96-56

             \* What's different is, I use backtracking with StringBuilder instead of directly String addition.

             \* This increase speed by 20%.

            \*/

            int len = sb.length();

            // Parse the current number substring

            long curOperand = Long.parseLong(num.substring(index, i + 1));

            // If this is the first operand (no operator before it)

            if(index == 0) {

                helper(num, target, i + 1, curOperand, curOperand, sb.append(curOperand), result);

                sb.setLength(len);

            } else {

                // Try adding the '+' operator

                helper(num, target, i + 1, curOperand, curTotal + curOperand, sb.append("+").append(curOperand), result);

                sb.setLength(len);

                // Try adding the '-' operator

                helper(num, target, i + 1, -curOperand, curTotal - curOperand, sb.append("-").append(curOperand), result);

                sb.setLength(len);

                // Try adding the '\*' operator

                /\*\*

                 \* Refer to

                 \* https://discuss.leetcode.com/topic/24523/java-standard-backtrace-ac-solutoin-short-and-clear/12?page=1

                 \* for example, if you have a sequence of 12345 and you have proceeded to 1 + 2 + 3,

                   now your eval is 6 right? If you want to add a \* between 3 and 4, you would take 3

                   as the digit to be multiplied, so you want to take it out from the existing eval.

                   You have 1 + 2 + 3 \* 4 and the eval now is (1 + 2 + 3) - 3 + (3 \* 4). Hope this could help : )

                \*/

                helper(num, target, i + 1, preOperand \* curOperand, curTotal - preOperand + preOperand \* curOperand, sb.append("\*").append(curOperand), result);

                sb.setLength(len);

            }

        }

    }

}

Time Complexity: O(3^N)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/expression-add-operators/solutions/71895/java-standard-backtrace-ac-solutoin-short-and-clear/>

<https://leetcode.com/problems/expression-add-operators/solutions/71895/java-standard-backtrace-ac-solutoin-short-and-clear/comments/74405>

<https://leetcode.com/problems/expression-add-operators/solutions/71895/java-standard-backtrace-ac-solutoin-short-and-clear/comments/74388>

<https://leetcode.com/problems/expression-add-operators/solutions/71895/java-standard-backtrace-ac-solutoin-short-and-clear/comments/239225>

<https://leetcode.com/problems/expression-add-operators/solutions/71895/java-standard-backtrace-ac-solutoin-short-and-clear/comments/74377>

<https://leetcode.com/problems/expression-add-operators/solutions/71921/java-simple-solution-beats-96-56/>

This problem has a lot of edge cases to be considered:

overflow: we use a long type once it is larger than Integer.MAX\_VALUE or minimum, we get over it.

0 sequence: because we can't have numbers with multiple digits started with zero, we have to deal with it too.

a little trick is that we should save the value that is to be multiplied in the next recursion.

However, adding String is extremely expensive. Speed can be increased by 20% if you use StringBuilder instead. Before: 238ms (67.31%), Now: 189ms (96.56%)

class Solution {

/\*\*

\* Refer to

\* https://discuss.leetcode.com/topic/24523/java-standard-backtrace-ac-solutoin-short-and-clear?page=1

\* This problem has a lot of edge cases to be considered:

overflow: we use a long type once it is larger than Integer.MAX\_VALUE or minimum, we get over it.

0 sequence: because we can't have numbers with multiple digits started with zero, we have to deal with it too.

a little trick is that we should save the value that is to be multiplied in the next recursion.

\*/

public List<String> addOperators(String num, int target) {

List<String> result = new ArrayList<String>();

if(num == null || num.length() == 0) {

return result;

}

StringBuilder sb = new StringBuilder();

dfs(result, sb, num, 0, target, 0, 0);

return result;

}

private void dfs(List<String> result, StringBuilder sb, String num, int pos, int target, long prev, long multi) {

if(pos == num.length() && target == prev) {

result.add(sb.toString());

return;

}

for(int i = pos; i < num.length(); i++) {

/\*\*

\* Refer to

\* https://discuss.leetcode.com/topic/24523/java-standard-backtrace-ac-solutoin-short-and-clear/63?page=4

\* explain this condition:

if(i != pos && num.charAt(pos) == '0') break;

I think this condition it to exclude numbers with leading zeros

Valid answers are:

"105", 5 -> ["1\*0+5","10-5"]

"00", 0 -> ["0+0", "0-0", "0\*0"]

but with leading zeros we will have

"105", 5 -> ["1\*0+5","10-5", "1\*05"]

"00", 0 -> ["0+0", "0-0", "0\*0", "00"]

\*/

// Refer to

// https://leetcode.com/problems/expression-add-operators/discuss/71895/Java-Standard-Backtrace-AC-Solutoin-short-and-clear/239225

// corner case: if current position is 0, we can only use it as a single digit number, should be 0

// if it is not a single digit number with leading 0, it should be considered as an invalid number

if(num.charAt(pos) == '0' && i != pos) {

break;

}

// This process is quite similar to 93. Restore IP Address

long curr = Long.parseLong(num.substring(pos, i + 1));

/\*\*

\* Refer to

\* https://discuss.leetcode.com/topic/32068/java-simple-solution-beats-96-56

\* What's different is, I use backtracking with StringBuilder instead of directly String addition.

\* This increase speed by 20%.

\*/

int len = sb.length();

if(pos == 0) {

dfs(result, sb.append(curr), num, i + 1, target, curr, curr);

sb.setLength(len);

} else {

dfs(result, sb.append("+").append(curr), num, i + 1, target, prev + curr, curr);

sb.setLength(len);

dfs(result, sb.append("-").append(curr), num, i + 1, target, prev - curr, -curr);

sb.setLength(len);

/\*\*

\* Refer to

\* https://discuss.leetcode.com/topic/24523/java-standard-backtrace-ac-solutoin-short-and-clear/12?page=1

\* for example, if you have a sequence of 12345 and you have proceeded to 1 + 2 + 3,

now your eval is 6 right? If you want to add a \* between 3 and 4, you would take 3

as the digit to be multiplied, so you want to take it out from the existing eval.

You have 1 + 2 + 3 \* 4 and the eval now is (1 + 2 + 3) - 3 + (3 \* 4). Hope this could help : )

\*/

dfs(result, sb.append("\*").append(curr), num, i + 1, target, prev - multi + multi \* curr, multi \* curr);

sb.setLength(len);

}

}

}

**Refer to**

<https://algo.monster/liteproblems/282>

**Problem Description**

The task is to take a string num that is composed exclusively of digits and an integer target, then find all the different ways to insert the binary operators '+', '-', and '\*' between the digits in num so that the resulting expression equals the target. For example, if num = "123" and target = 6, one way to reach the target is to insert the '+' operator as "1+2+3".

A few important points to note for this problem:

You can insert an operator between any two digits in the string num.

The expressions that you create should not have numbers with leading zeroes.

The order of digits in the num string must remain the same.

The challenge is to generate all such valid expressions that evaluate to target.

**Intuition**

To solve the puzzle efficiently, we use a Depth First Search (DFS) approach.

**Breaking Down the Problem:** Since you have to evaluate all possible combinations of digits and operators, this naturally forms a tree structure with each node representing a decision to include a '+', '-', '\*', or no operator before the current digit.

**Handling '0's:** No number in the expression can have leading zeroes. This is managed by ensuring that whenever a '0' appears as a current digit, it doesn't process further if it is the start of a new operand, thus avoiding expressions with numbers like "01" or "002".

**Evaluating Expressions:** Keep track of the current expression's value and update it as you insert new operators. This involves tracking the last operand to correctly evaluate multiplication which has higher precedence than addition and subtraction.

**Depth First Search (DFS) Algorithm:** The key to the solution is a recursive DFS function that tries all possible combinations of operators and operand lengths. At each recursive call, the current value of the expression, path (the actual expression built so far), value of the previous operand (to handle multiplication correctly), and position in the string are updated.

**Base Case:** Once the end of the string is reached, check whether the current value of the expression is equal to the target. If it is, add the path to the list of possible solutions.

**DFS Recursion:** The DFS function:

Includes the current digit as is and moves to the next step without adding any operators if it's the first digit (to avoid leading '+', '-', or '\*' characters).

Tries adding each of the operators (+, -, \*) between the current and the next digit if it's not the first digit, carefully updating the expression's value and path. It is also critical to adjust the value of the previous operand when a multiplication is encountered due to precedence.

By implementing this DFS approach, we exhaustively search through all the possible expressions and return those that meet the target value.

**Solution Approach**

The implementation of the solution uses a recursive function to perform a depth-first search through the possible combinations of binary operators and operands. The DFS searches through the num string, trying to add an operator or not at each step, to explore all potential valid expressions that can be formed. Here is the breakdown of the key parts of the implementation:

**Recursive Depth-First Search (DFS):** A recursive function dfs is defined which facilitates the depth-first search. This recursive function takes four parameters:

u: The current index in the string num.

prev: The value of the last operand included in the expression (necessary for handling precedence in multiplication).

curr: The current value of the expression as it's been evaluated so far.

path: The current form of the expression as a string, which shows exactly how the numbers and operators are combined.

**Base Case:** The base case for recursion occurs when the end of the string num is reached (i.e., u == len(num)). Here, if the current value equals the target, the current path (the expression) is added to the answer list ans.

**Loop Through Possible Operands:** The function loops from the current index u to the end of the num string, incrementing i at each iteration. This loop determines the next operand by considering one digit at a time and then multiple digits as a single operand until a leading zero would be created, which is avoided.

**Skip Leading Zeros:** If the current digit is '0' and it's the start of a new operand, it will break the loop to avoid operands with leading zeroes.

**Operand Conversion:** The substring from the current index u up to and including i is converted to an integer named next. This integer will become the next operand for the operators to act upon.

**No Operator for First Digit:** If we are at the start of the string (when u is 0), no operator is added. The DFS function is called with the next index and values are set to include the first operand.

**Adding Operators:** If not at the start, three recursive calls are made, one for each operator ('+', '-', '\*'):

**Addition (+):** The DFS is called with the cumulative value updated by adding next.

**Subtraction (-):** The DFS is called with the cumulative value updated by subtracting next.

**Multiplication (\*):** The DFS is called with a more complex value update, where the last operand is first removed from the cumulative value (curr - prev) and then the multiplication of prev and next is added back to it to maintain the correct order of operations.

The dfs function is initially called with starting values, including an empty path string. Finally, the list ans is returned, containing all the paths (expressions) that evaluate to the target.

By methodically and recursively exploring each combination and path, we can generate a comprehensive list of all expressions that meet the criteria. The provided solution is both elegant and efficient in its approach to solving this problem.

**Example Walkthrough**

Let's walk through a simple example to illustrate the solution approach. Consider num = "105" and target = 5. We want to find all valid expressions that can be created from "105" which evaluate to 5.

Now, we'll go through each step as per the DFS algorithm strategy:

**Initialization:** Start with an empty path, the curr value as 0, prev as 0, and an index u at 0.

**First Call to DFS:** Since u is at the start, no operator is added. The first digit, '1', is converted to an integer and added to prev and curr. The new path becomes "1", and recursion continues to the next digit with u incremented.

**Second Digit - '0':** At u = 1, we have to consider that '0' might be a leading zero for an operand. Since it's not the first digit, we explore adding '+', '-', and '\*':

No Operator: Skipping any operator, prev and curr remain the same, path becomes "10", and u goes to the next digit.

Add '+': The function is called with curr as 10, prev as 10, and path as "10+".

Add '-': The function is called with curr as 10, prev as -10, and path as "10-".

Add '\*': We don't consider multiplication here because it would lead to a result that is certainly not going to match our target of 5 (since we'd be multiplying by 0).

**Third Digit - '5':** Now, u = 2 and the possible operands are '05' which we ignore to avoid leading zero, or just '5'.

We again try all three possibilities from the conditions met, now with the path that already includes '10':

Path "10+5": For the '+' operator from the previous state "10+", adding 5 results in 15, which does not match the target. This path is abandoned.

Path "10-5": For the '-' operator from the previous state "10-", subtracting 5 results in 5, which matches our target. We add "10-5" to our list of answers.

Multiplied by '5': Multiplication would only happen if the previous operator was a '\*', which wasn't a valid option in step 3, so we don't consider it here.

Since we managed to find one valid solution, "10-5" evaluates to the target of 5, which means our answer list would be ["10-5"].

By iterating through each digit in num and exploring all possible operator insertions in this manner, we systematically discover all expressions that evaluate to the given target. The DFS algorithm ensures we explore every possibility, and our logic for handling leading zeroes and operator precedence ensures all expressions are valid and correctly evaluated.

**Solution Implementation**

import java.util.ArrayList;

import java.util.List;

public class Solution {

private List<String> answers; // Holds the valid expressions

private String number; // The input number as a String

private int targetValue; // The target value for the expressions

// Main function to find expressions that evaluate to the target value

public List<String> addOperators(String num, int target) {

answers = new ArrayList<>();

number = num;

targetValue = target;

recursiveSearch(0, 0, 0, "");

return answers;

}

// Helper function to perform depth-first search

private void recursiveSearch(int index, long prevOperand, long currentTotal, String expression) {

// If we've reached the end of the string, check if the currentTotal equals the target value

if (index == number.length()) {

if (currentTotal == targetValue) {

answers.add(expression);

}

return;

}

// Try all possible splits of the remainder of the string

for (int i = index; i < number.length(); i++) {

// Skip numbers with leading zeros (unless the number itself is zero)

if (i != index && number.charAt(index) == '0') {

break;

}

// Parse the current number substring

long nextOperand = Long.parseLong(number.substring(index, i + 1));

// If this is the first operand (no operator before it)

if (index == 0) {

recursiveSearch(i + 1, nextOperand, nextOperand, expression + nextOperand);

} else {

// Try adding the '+' operator

recursiveSearch(i + 1, nextOperand, currentTotal + nextOperand, expression + "+" + nextOperand);

// Try adding the '-' operator

recursiveSearch(i + 1, -nextOperand, currentTotal - nextOperand, expression + "-" + nextOperand);

// Try adding the '\*' operator

recursiveSearch(i + 1, prevOperand \* nextOperand, currentTotal - prevOperand + prevOperand \* nextOperand, expression + "\*" + nextOperand);

}

}

}

}

**Time and Space Complexity**

The provided Python function addOperators seeks to insert arithmetic operators into a string representing a number in all possible ways such that the resulting expression evaluates to a given target value. To analyze its time and space complexity, we have to understand how the recursion is structured.

The recursion explores adding '+', '-', or '\*' between every two digits in the string. For a string of length n, there are n-1 interstitial positions where operators can be placed. Since there are 3 different operators to choose from at each interstitial position, in the worst case, the number of different expressions generated is up to O(3^(n-1)). Thus, the time complexity of this recursive solution can be expressed as O(3^n) to account for this explosive growth due to branching at each character position. This is assuming the time taken to compute the next component of the expression and concatenate strings is negligible in comparison to the recursion itself.

Now, considering the space complexity, there are two aspects to consider:

The recursion stack depth, which in the worst case can go up to n - the depth equals the length of the string if the recursion explores adding an operator after every digit. This contributes O(n) to the space complexity.

The space for storing the paths (partial expressions). Since the function concatenates strings to build up expressions, the maximum length of a path could be 2n-1 (for n digits, and n-1 operators). Since it only keeps a single path in memory at any one time during the recursion, the space taken by the path does not have a multiplicative effect based on the number of function calls.

Taking into account both the recursion stack and the path length, the space complexity of the algorithm can be concluded to be O(n).

In summary:

Time Complexity: O(3^n)

Space Complexity: O(n)

**Refer to**

[L93.Restore IP Addresses (Ref.L282)](note://WEBa03a392cd2882182805c026c8844ef29)