<https://leetcode.com/problems/different-ways-to-add-parentheses/description/>

Given a string expression of numbers and operators, return all possible results from computing all the different possible ways to group numbers and operators. You may return the answer in any order.

The test cases are generated such that the output values fit in a 32-bit integer and the number of different results does not exceed 10^4.

**Example 1:**

**Input:** expression = "2-1-1"

**Output:** [0,2]

**Explanation:**((2-1)-1) = 0 (2-(1-1)) = 2

**Example 2:**

**Input:** expression = "2\*3-4\*5"

**Output:** [-34,-14,-10,-10,10]

**Explanation:**(2\*(3-(4\*5))) = -34 ((2\*3)-(4\*5)) = -14 ((2\*(3-4))\*5) = -10 (2\*((3-4)\*5)) = -10 (((2\*3)-4)\*5) = 10

**Constraints:**

1 <= expression.length <= 20

expression consists of digits and the operator '+', '-', and '\*'.

All the integer values in the input expression are in the range [0, 99].

**Attempt 1: 2024-07-10**

**Solution 1: DFS + Divide and Conquer (360 min)**

**Style 1: Split the original expression into tokens list**

class Solution {

    public List<Integer> diffWaysToCompute(String expression) {

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

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

            return result;

        }

        // Split string into numbers and operations

        // be careful the numbers could be continuous digits

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

        for(int i = 0; i < expression.length(); i++) {

            String token = "";

            while(i < expression.length() && Character.isDigit(expression.charAt(i))) {

                token += expression.charAt(i);

                i++;

            }

            // Add number section

            tokens.add(token);

            // Add operate (and it could not be the last character in string)

            if(i < expression.length()) {

                tokens.add(String.valueOf(expression.charAt(i)));

            }

        }

        result = helper(tokens, 0, tokens.size() - 1);

        return result;

    }

    private List<Integer> helper(List<String> tokens, int lo, int hi) {

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

        // Base case: only one number left when lo = hi

        if(lo == hi) {

            result.add(Integer.valueOf(tokens.get(lo)));

            return result;

        }

        // Same process as 95.Unique Binary Search Tree II

        // Scan all combinations between left and right sub-list divide by operate

        for(int i = lo; i <= hi; i++) {

            if(i % 2 == 1) {

                String ops = tokens.get(i);

                List<Integer> left = helper(tokens, lo, i - 1);

                List<Integer> right = helper(tokens, i + 1, hi);

                for(int l : left) {

                    for(int r : right) {

                        if(ops.equals("+")) {

                            result.add(l + r);

                        }

                        if(ops.equals("\*")) {

                            result.add(l \* r);

                        }

                        if(ops.equals("-")) {

                            result.add(l - r);

                        }

                    }

                }

            }

        }

        return result;

    }

}

**Style 1's Memoization (10 min)**

class Solution {

    public List<Integer> diffWaysToCompute(String expression) {

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

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

            return result;

        }

        // Split string into numbers and operations

        // be careful the numbers could be continuous digits

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

        for(int i = 0; i < expression.length(); i++) {

            String token = "";

            while(i < expression.length() && Character.isDigit(expression.charAt(i))) {

                token += expression.charAt(i);

                i++;

            }

            // Add number section

            tokens.add(token);

            // Add operate (and it could not be the last character in string)

            if(i < expression.length()) {

                tokens.add(String.valueOf(expression.charAt(i)));

            }

        }

        // memo stores all possible results from the i-th

        // integer to the j-th integer (inclusive) in the list.

        Map<String, List<Integer>> memo = new HashMap<String, List<Integer>>();

        result = helper(tokens, 0, tokens.size() - 1, memo);

        return result;

    }

    private List<Integer> helper(List<String> tokens, int lo, int hi, Map<String, List<Integer>> memo) {

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

        // If already stored result, return quickly

        String str = lo + "\_" + hi;

        if(memo.containsKey(str)) {

            return memo.get(str);

        }

        // Base case: only one number left when lo = hi

        if(lo == hi) {

            result.add(Integer.valueOf(tokens.get(lo)));

            return result;

        }

        // Same process as 95.Unique Binary Search Tree II

        // Scan all combinations between left and right sub-list divide by operate

        for(int i = lo; i <= hi; i++) {

            if(i % 2 == 1) {

                String ops = tokens.get(i);

                List<Integer> left = helper(tokens, lo, i - 1, memo);

                List<Integer> right = helper(tokens, i + 1, hi, memo);

                for(int l : left) {

                    for(int r : right) {

                        if(ops.equals("+")) {

                            result.add(l + r);

                        }

                        if(ops.equals("\*")) {

                            result.add(l \* r);

                        }

                        if(ops.equals("-")) {

                            result.add(l - r);

                        }

                    }

                }

            }

        }

        // map.put() automatically contains update operation

        memo.put(str, result);

        return result;

    }

}

Time Complexity: O(n) ~ O(n^2 \* 2^n)

Space Complexity: O(2^n)

**Refer to**

<https://leetcode.com/problems/different-ways-to-add-parentheses/solutions/66333/java-recursive-9ms-and-dp-4ms-solution/>

I think it's more efficient to pre-parse the string because String.substring() is costly. I store the parsed string in a list, for example, if the string is 1+2+3+4, then the list will contain:

"1", "+", "2", "+", "3", "+", "4"

Personally I feel this is also more convenient because all integers occurs at even indices (0, 2, 4, 6) and all operators are at odd indices (1, 3, 5).

Then the problem is very similar to "Unique Binary Search Trees II". For each operator in the list, we compute all possible results for entries to the left of that operator, which is List<Integer> left, and also all possible results for entries to the right of that operator, namely List<Integer> right, and combine the results. It can be achieved by recursion or more efficiently by dp.

Recursion:

public List<Integer> diffWaysToCompute(String input) {

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

if(input==null||input.length()==0) return result;

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

for(int i=0; i<input.length(); i++){

int j=i;

while(j<input.length()&&Character.isDigit(input.charAt(j)))

j++;

String num=input.substring(i, j);

ops.add(num);

if(j!=input.length()) ops.add(input.substring(j, j+1));

i=j;

}

result=compute(ops, 0, ops.size()-1);

return result;

}

private List<Integer> compute(List<String> ops, int lo, int hi){

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

if(lo==hi){

Integer num=Integer.valueOf(ops.get(lo));

result.add(num);

return result;

}

for(int i=lo+1; i<=hi-1; i=i+2){

String operator=ops.get(i);

List<Integer> left=compute(ops,lo, i-1), right=compute(ops, i+1, hi);

for(int leftNum:left)

for(int rightNum: right){

if(operator.equals("+"))

result.add(leftNum+rightNum);

else if(operator.equals("-"))

result.add(leftNum-rightNum);

else

result.add(leftNum\*rightNum);

}

}

return result;

}

And DP, where dp[i][j] stores all possible results from the i-th integer to the j-th integer (inclusive) in the list.

public List<Integer> diffWaysToCompute(String input) {

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

if(input==null||input.length()==0) return result;

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

for(int i=0; i<input.length(); i++){

int j=i;

while(j<input.length()&&Character.isDigit(input.charAt(j)))

j++;

ops.add(input.substring(i, j));

if(j!=input.length()) ops.add(input.substring(j, j+1));

i=j;

}

int N=(ops.size()+1)/2; //num of integers

ArrayList<Integer>[][] dp=(ArrayList<Integer>[][]) new ArrayList[N][N];

for(int d=0; d<N; d++){

if(d==0){

for(int i=0; i<N; i++){

dp[i][i]=new ArrayList<>();

dp[i][i].add(Integer.valueOf(ops.get(i\*2)));

}

continue;

}

for(int i=0; i<N-d; i++){

dp[i][i+d]=new ArrayList<>();

for(int j=i; j<i+d; j++){

ArrayList<Integer> left=dp[i][j], right=dp[j+1][i+d];

String operator=ops.get(j\*2+1);

for(int leftNum:left)

for(int rightNum:right){

if(operator.equals("+"))

dp[i][i+d].add(leftNum+rightNum);

else if(operator.equals("-"))

dp[i][i+d].add(leftNum-rightNum);

else

dp[i][i+d].add(leftNum\*rightNum);

}

}

}

}

return dp[0][N-1];

}

**Style 2: Keep the original expression as single string**

class Solution {

    // This method starts the process by calling the 'helper' helper method.

    public List<Integer> diffWaysToCompute(String expression) {

        return helper(expression);

    }

    // Recursive function to compute all possible results from the input expression.

    private List<Integer> helper(String expression) {

        List<Integer> results = new ArrayList<>();

        // Base case: if expression is a single number, return it as the only result.

        if (!expression.contains("+") && !expression.contains("-") && !expression.contains("\*")) {

            results.add(Integer.parseInt(expression));

            return results;

        }

        // Iterate through each character of the expression string.

        for (int i = 0; i < expression.length(); i++) {

            char op = expression.charAt(i);

            // When an operator is found, divide the expression into two parts.

            if (op == '-' || op == '+' || op == '\*') {

                List<Integer> left = helper(expression.substring(0, i));

                List<Integer> right = helper(expression.substring(i + 1));

                // Compute all combinations of results from left and right sub-expressions.

                for (int l : left) {

                    for (int r : right) {

                        if (op == '-') {

                            results.add(l - r);

                        } else if (op == '+') {

                            results.add(l + r);

                        } else if (op == '\*') {

                            results.add(l \* r);

                        }

                    }

                }

            }

        }

        // Return all the computed results.

        return results;

    }

}

**Style 2's Memoization (10 min)**

class Solution {

    // This method starts the process by calling the 'helper' helper method.

    public List<Integer> diffWaysToCompute(String expression) {

        // Cache to store already computed results for expressions.

        Map<String, List<Integer>> memo = new HashMap<>();

        return helper(expression, memo);

    }

    // Recursive function to compute all possible results from the input expression.

    private List<Integer> helper(String expression, Map<String, List<Integer>> memo) {

        // Check if the result for this expression is cached.

        if (memo.containsKey(expression)) {

            return memo.get(expression);

        }

        List<Integer> results = new ArrayList<>();

        // Base case: if expression is a single number, return it as the only result.

        if (!expression.contains("+") && !expression.contains("-") && !expression.contains("\*")) {

            results.add(Integer.parseInt(expression));

            return results;

        }

        // Iterate through each character of the expression string.

        for (int i = 0; i < expression.length(); i++) {

            char op = expression.charAt(i);

            // When an operator is found, divide the expression into two parts.

            if (op == '-' || op == '+' || op == '\*') {

                List<Integer> left = helper(expression.substring(0, i), memo);

                List<Integer> right = helper(expression.substring(i + 1), memo);

                // Compute all combinations of results from left and right sub-expressions.

                for (int l : left) {

                    for (int r : right) {

                        if (op == '-') {

                            results.add(l - r);

                        } else if (op == '+') {

                            results.add(l + r);

                        } else if (op == '\*') {

                            results.add(l \* r);

                        }

                    }

                }

            }

        }

        // Cache the computed results for the current expression.

        memo.put(expression, results);

        // Return all the computed results.

        return results;

    }

}

Time Complexity: O(n) ~ O(n^2 \* 2^n)

Space Complexity: O(2^n)

**Refer to**

<https://leetcode.com/problems/different-ways-to-add-parentheses/solutions/1294189/easy-solution-faster-than-100-explained-with-diagrams/>

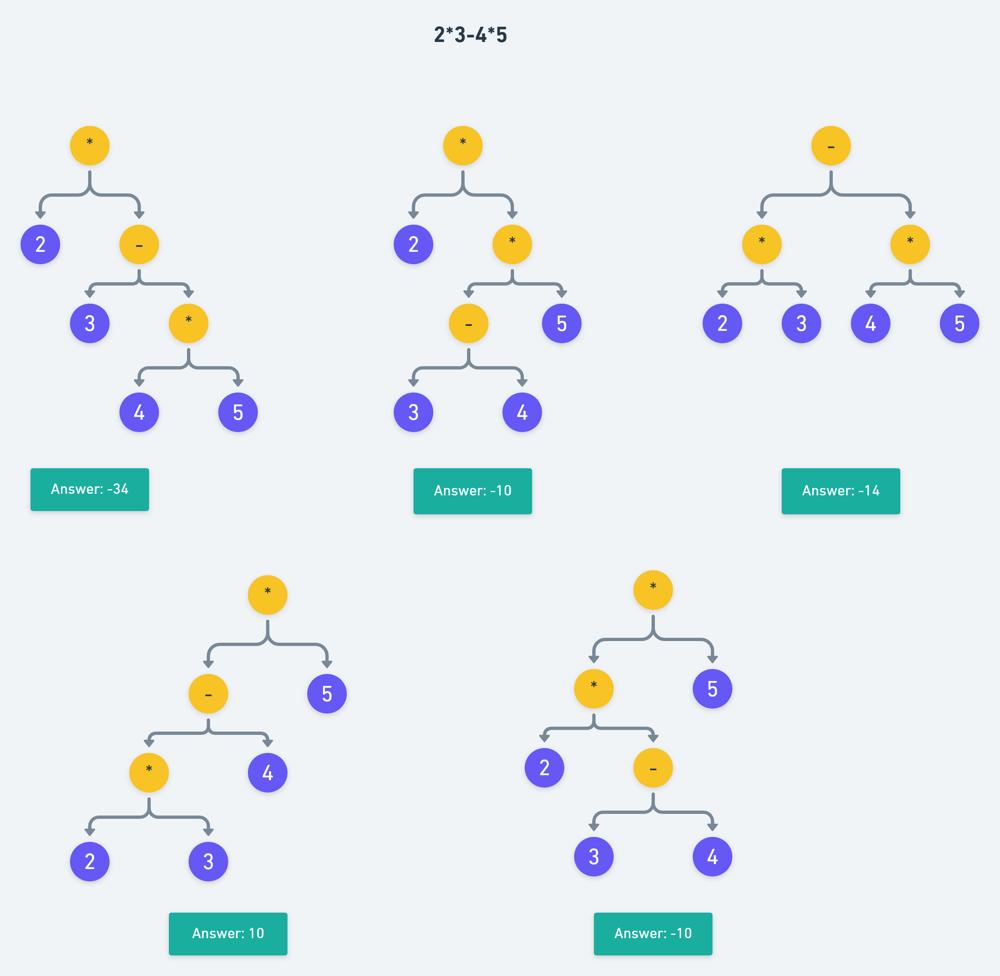
The problem becomes easier when we think about these expressions as expression trees.

We can traverse over the experssion and whenever we encounter an operator, we recursively divide the expression into left and right part and evaluate them seperately until we reach a situation where our expression is purely a number and in this case we can simply return that number.

Since there can be multiple ways to evaluate an expression (depending on which operator you take first) we will get a list of reults from left and the right part.

Now that we have all the possible results from the left and the right part, we can use them to find out all the possible results for the current operator.

The following image shows all the possible expression trees for the expression 2 \* *3 - 4 \** 5



C++ implementation

class Solution {

public:

// function to get the result of the operation

int perform(int x, int y, char op) {

if(op == '+') return x + y;

if(op == '-') return x - y;

if(op == '\*') return x \* y;

return 0;

}

vector<int> diffWaysToCompute(string exp) {

vector<int> results;

bool isNumber = 1;

for(int i = 0; i < exp.length(); i++) {

// check if current character is an operator

if(!isdigit(exp[i])) {

// if current character is not a digit then

// exp is not purely a number

isNumber = 0;

// list of first operands

vector<int> left = diffWaysToCompute(exp.substr(0, i));

// list of second operands

vector<int> right = diffWaysToCompute(exp.substr(i + 1));

// performing operations

for(int x : left) {

for(int y : right) {

int val = perform(x, y, exp[i]);

results.push\_back(val);

}

}

}

}

if(isNumber == 1) results.push\_back(stoi(exp));

return results;

}

};

**Refer to**

<https://algo.monster/liteproblems/241>

**Problem Description**

The problem in question requires us to compute all possible results from a given string expression which includes numbers and arithmetic operators ('+', '-', '\*'). The key challenge is to consider all the different groupings of the numbers and operators to find every possible evaluation outcome.

Imagine you have an expression like "2-1-1". If you group the numbers and operators differently, you get different results:

((2-1)-1) = 0

(2-(1-1)) = 2

The goal is to find all such distinct values that can be obtained by all the different possible ways to group the numbers and operators. This problem is inherently recursive, as each time you encounter an operator, you have two sub-expressions (left and right) which can themselves be broken down further.

**Intuition**

The provided solution uses a recursive approach with memoization to solve the problem efficiently. The basic idea is to use recursion to break down the expression at every operator, compute all possible results for the left and right sub-expressions separately, and then combine these results according to the current operator.

Here's a step-by-step breakdown of the solution approach:

Define a recursive function dfs that will compute and return all possible outcomes of a given expression.

Base Case: If the input to dfs is a single number, we return a list containing just that number.

Recursive Case: If the input contains operators, iterate through each character in the expression.

When an operator is encountered, split the expression into left and right sub-expressions at that operator.

Recursively call dfs on the left and right sub-expressions to compute their results separately.

Once we have the results from the left and right sides, combine each pair of results from the left and right using the current operator.

Cache the results for each sub-expression to avoid re-computation (this is done using the @cache decorator). This step makes the solution much faster, as the number of unique sub-expressions is significantly less than the total number of calls made during the recursive process.

Initiate the recursive function with the full expression to get all possible results and return them.

This approach leverages the power of recursion to solve a complex problem by breaking it down into simpler sub-problems, solving each sub-problem once, caching the solution, and combining these solutions to get the final result set.

**Solution Approach**

The implementation of the solution adopts a divide-and-conquer strategy and utilizes Python's built-in memorization technique using the cache decorator from the functools module. This strategy is laid out in the recursive dfs function. Here is how the solution unfolds:

**Memoization:** The @cache decorator is used to memoize the dfs function. It helps in storing the results of expensive function calls and returning the cached result when the same inputs occur again. This is essential since many sub-expressions will repeat in the evaluation process and we do not want to recompute them each time.

**Recursive Function (dfs):** This function is the heart of the solution. It takes as input a sub-expression (exp) of the entire expression and returns all outcomes of this sub-expression as a list of integers.

**Base Case:** If the exp is solely composed of digits with no operators, we convert it to an integer and return it in a list format. This signifies the end of the recursion for that path, i.e., when we're down to a single numerical value.

if exp.isdigit():

return [int(exp)]

**Recursive Case:** When exp contains operators, we iterate over each character:

for i, c in enumerate(exp):

if c in '-+\*':

For each operator we find (c), we divide the exp into two parts: left of the operator (exp[:i]) and right of the operator (exp[i + 1 :]).

We then recursively call dfs on these parts, which gives us all possible outcomes for both left and right sub-expressions.

left, right = dfs(exp[:i]), dfs(exp[i + 1 :])

After that, we perform calculations based on the type of operator. We run through every combination of results from left and right, performing the operation dictated by c on each pair:

for a in left:

for b in right:

if c == '-':

ans.append(a - b)

elif c == '+':

ans.append(a + b)

else:

ans.append(a \* b)

The results for each sub-expression, after combining with the operator, are stored in a temporary list called ans.

**Collecting Results:** Once all possible operations on the sub-expressions are complete, ans is returned to the caller. This process continues recursively, gathering results from sub-expressions, combining them, and passing them up the call stack.

**Initiate and Return:** Finally, dfs is initiated with the full initial expression as the parameter.

return dfs(expression)

All the recursive calls to dfs, and the memoization through @cache, ensures that every unique sub-expression is evaluated only once, and the results are reused wherever necessary. This brings down the time complexity significantly, which is important given the statement that the number of results does not exceed 10^4.

By systematically considering every split at each operator and recursively solving smaller problems, the function finally returns a list of all possible results for the initial expression.

**Example Walkthrough**

Let's walk through a small example using the solution approach.

Consider the expression "2 \* 3 - 4 \* 5". We want to calculate all possible results from different combinations of parentheses placement.

We begin with the dfs function, which is a recursive function that helps compute all possible outcomes:

The initial call is with the whole expression dfs("2 \* 3 - 4 \* 5").

The function iterates through the expression character by character. When it encounters operators, it divides the expression and performs recursive calls.

For example, it first encounters the operator \* between "2" and "3". It will then make two recursive calls: dfs("2") for the left sub-expression and dfs("3-4\*5") for the right sub-expression.

The dfs("2") is a base case which is a single number and returns [2].

dfs("3-4\*5") will further break down into other expressions when it encounters the - and \* operators respectively.

This process of breaking down expressions continues until all sub-expressions are single numbers.

During the recursive calls, each dfs will eventually return a list of results.

As the recursion unwinds, the function starts combining the results from left and right sub-expressions with the operators that led to their division. The combine operation follows based on the type of the operator:

If the expression was split by a \*, the results from the left and right are multiplied.

If the split was by a +, the results from each side are added.

If the division happened due to a -, the results from the left are subtracted from the results from the right.

For "2 \* 3 - 4 \* 5", the recursive calls will eventually perform operations like:

The results of dfs("2") ([2]) will be multiplied with the results of dfs("3") ([3]), yielding 6.

Then, dfs("4") ([4]) will be multiplied with dfs("5") ([5]), yielding 20.

Ultimately, dfs("3 - 4 \* 5") will evaluate to [3-20], which equals [-17].

Finally, the results of dfs("2") and dfs("3 - 4 **\*** 5") will be combined to give the final result as [2\*-17], which is [-34].

Since the memoization is enabled with @cache, the results for each sub-expression are stored and reused, saving a significant amount of computation.

By following this method recursively and combining the left and right sub-expression results using the operators, we construct all possible evaluation results.

In this example, dfs would be called like:

dfs("2 \* 3 - 4 \* 5") -> combines results from dfs("2") \* dfs("3 - 4 \* 5")

dfs("3 - 4 \* 5") -> combines results from dfs("3") - dfs("4 \* 5")

dfs("4 \* 5") -> combines results from dfs("4") \* dfs("5")

And so on for each operator encountered...

When computation finishes, dfs("2 \* 3 - 4 **\*** 5") will return a list of all the possible outcomes that can be achieved by adding parentheses in our example expression in different ways. These outcomes will be the complete set of distinct values that can be obtained for the expression "2 \* 3 - 4 \* 5".

**Solution Implementation**

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

public class Solution {

// Cache to store already computed results for expressions.

private static Map<String, List<Integer>> memoizationCache = new HashMap<>();

// This method starts the process by calling the 'computeAllPossibleResults' helper method.

public List<Integer> diffWaysToCompute(String expression) {

return computeAllPossibleResults(expression);

}

// Recursive function to compute all possible results from the input expression.

private List<Integer> computeAllPossibleResults(String expression) {

// Check if the result for this expression is cached.

if (memoizationCache.containsKey(expression)) {

return memoizationCache.get(expression);

}

List<Integer> results = new ArrayList<>();

// Base case: if expression is a single number, return it as the only result.

if (!expression.contains("+") && !expression.contains("-") && !expression.contains("\*")) {

results.add(Integer.parseInt(expression));

return results;

}

// Iterate through each character of the expression string.

for (int i = 0; i < expression.length(); i++) {

char operation = expression.charAt(i);

// When an operator is found, divide the expression into two parts.

if (operation == '-' || operation == '+' || operation == '\*') {

List<Integer> resultsLeft = computeAllPossibleResults(expression.substring(0, i));

List<Integer> resultsRight = computeAllPossibleResults(expression.substring(i + 1));

// Compute all combinations of results from left and right sub-expressions.

for (int leftResult : resultsLeft) {

for (int rightResult : resultsRight) {

if (operation == '-') {

results.add(leftResult - rightResult);

} else if (operation == '+') {

results.add(leftResult + rightResult);

} else if (operation == '\*') {

results.add(leftResult \* rightResult);

}

}

}

}

}

// Cache the computed results for the current expression.

memoizationCache.put(expression, results);

// Return all the computed results.

return results;

}

}

**Time and Space Complexity**

The given code is a Divide and Conquer algorithm that computes all possible results from a given arithmetic expression with numbers and operators. The memoization via @cache helps avoid repeated computation for the same sub-expressions.

**Time Complexity:**

**There are 2n+1 possibilities for sub-expressions from the given expression of length n, where n is the number of operators.**

**Each sub-expression can be split into two parts at each operator, generating a combination of left and right results which can be anywhere from 1 to Catalan(n) (which is an upper bound on the number of unique BSTs that can be formed with n nodes). The Catalan number grows approximately as 4^n / (n^(3/2) sqrt(pi)), which is less than 2^n.**

**Considering the above facts and the memoization, the upper bound on the time complexity is O(n \* 2^n \* n) = O(n^2 \* 2^n) since for each operator we do two recursive calls and combine their results, plus the iteration for combining results that take O(n) time.**

**Space Complexity:**

Due to memoization, we are storing the intermediate results for each sub-expression. The space usage would depend on the number of unique sub-expressions, which is at most O(2^n).

Recursive calls will use additional stack space, which could be as deep as the number of operators, adding another O(n) factor.

Taking into account the memoization storage and recursive call stack, the space complexity would be O(2^n + n) = O(2^n), since 2^n should dominate for large n.

**Refer to**

[L95.Unique Binary Search Trees II](note://WEBf95fafbd7e21c2466d68ef518b0685d4)