Boolean Parenthesization Problem - GeeksforGeeks

Source: https://www.geeksforgeeks.org/boolean-parenthesization-problem-dp-37/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Boolean Parenthesization Problem Last Updated : 20 Aug, 2025 Given a boolean expression s that contain symbols and operators. The task is to count the number of ways we can parenthesize the expression so that the value of the expression evaluates to true . Symbols T ---> true F ---> false Operators & ---> boolean AND | ---> boolean OR ^ ---> boolean XOR Examples: Input: s = T|T&F^T Output: 4 Explanation: The expression evaluates to true in 4 ways ((T|T)&(F^T)), (T|(T&(F^T))), (((T|T)&F)^T) and (T|((T&F)^T)). Input: s = T^F|F Output: 2 Explanation: ((T^F)|F) and (T^(F|F)) are the only ways. Try it on GfG Practice Important Point: When calculating ways to make an expression evaluate to true , we also need to consider combinations where subexpressions evaluate to fals e because operators like XOR can produce true when one operand is false . For example, if we have "F^T", even though the left subexpression evaluates to false, the XOR operation with true on the right gives us true as the final result. Therefore, we need to keep track of both true and false counts f or subexpressions to handle all possible combinations correctly. Table of Content [Naive Approach] - Using Recursion - O(2^n) Time and O(n^2) Space [Expected Approach 1]- Using Top-Down DP - O(n^3) Time and O(n^2) Space [Expected Approach 2]- Using Bottom-Up DP - O(n^3) Time and O(n^2) Space [Naive Approach] - Using Recursion - O(2^n) Time and O(n^2) Space The idea is to solve this recursively by splitting the expression at each operator and evaluating all possible combinations of true/false values from the left and right s ubexpressions. For each operator position k, we parenthesize the expression into two parts: (i, k-1) and (k+1, j) . We then recursively calculate how many ways each part can evaluate to true and false. Once we have these counts, we can combine them based on the operator at position k. C++ #include <bits/stdc++.h> using namespace std ; // Function to evaluate a boolean expression given two operands and an operator bool evaluate (bool left , bool right , char op) { if (op == '&') return left & right ; if (op == '|') return left | right ; return left ^ right ; // for '^' } // Recursive function to count the number of ways s[i...j] can evaluate to 'req' int countRecur (int i , int j , bool req , string & s) { // Base case: if the substring is a single operand (T/F) if (i == j) { return (req == (s [i] == 'T')) ? 1 : 0 ; } int ways = 0 ; // Partition only at operator positions → they lie at odd indices for (int k = i + 1 ; k < j ; k += 2) { // Count ways for left and right subexpressions int leftTrue = countRecur (i , k - 1 , true , s); int leftFalse = countRecur (i , k - 1 , false , s); int rightTrue = countRecur (k + 1 , j , true , s); int rightFalse = countRecur (k + 1 , j , false , s); // Combine results based on operator at position k if (evaluate (true , true , s [k]) == req) ways += leftTrue * rightTrue ; if (evaluate (true , false , s [k]) == req) ways += leftTrue * rightFalse ; if (evaluate (false , true , s [k]) == req) ways += leftFalse * rightTrue ; if (evaluate (false , false , s [k]) == req) ways += leftFalse * rightFalse ; } return ways ; } // Wrapper function: returns number of ways to evaluate entire expression to True int countWays (string s) { int n = s . length (); return countRecur (0 , n - 1 , true , s); } int main () { string s = "T|T&F^T" ; cout << countWays (s); return 0 ; } Java import java.util.* ; class GfG { static boolean evaluate (boolean b1 , boolean b2 , char op) { if (op == '&') { return b1 & b2 ; } else if (op == '|') { return b1 | b2 ; } return b1 ^ b2 ; } // Function which returns the number of ways // s[i:j] evaluates to req. static int countRecur (int i , int j , boolean req , String s) { // Base case: if (i == j) { return (req == (s . charAt (i) == 'T')) ? 1 : 0 ; } int ans = 0 ; for (int k = i + 1 ; k < j ; k += 2) { // Count Ways in which left substring // evaluates to true and false. int leftTrue = countRecur (i , k - 1 , true , s); int leftFalse = countRecur (i , k - 1 , false , s); // Count Ways in which right substring // evaluates to true and false. int rightTrue = countRecur (k + 1 , j , true , s); int rightFalse = countRecur (k + 1 , j , false , s); // Check if the combinations results // to req.

if (evaluate (true , true , s . charAt (k)) == req) { ans += leftTrue * rightTrue ; } if (evaluate (true , false , s . charAt (k)) == req) { ans += leftTrue * rightFalse ; } if (evaluate (false , true , s . charAt (k)) == req) { ans += leftFalse * rightTrue ; } if (evaluate (false , false , s . charAt (k)) == req) { ans += leftFalse * rightFalse ; } } return ans ; } static int countWays (String s) { int n = s . length (); return countRecur (0 , n - 1 , true , s); } public static void main (String [] args) { String s = "T|T&F^T" ; System . out . println (countWays (s)); } } Python def evaluate (b1 , b2 , op): if op == '&' : return b1 & b2 elif op == '|' : return b1 | b2 return b1 ^ b2 # Function which returns the number of ways # s[i:j] evaluates to req def countRecur (i , j , req , s): # Base case: When we have only one character if i == j : return 1 if req == (s [i] == 'T') else 0 ans = 0 # Loop through the operators (every other character is an operator) for k in range (i + 1 , j , 2): # Count ways in which left and right substrings # evaluate to True or False leftTrue = countRecur (i , k - 1 , True , s) leftFalse = countRecur (i , k - 1 , False , s) rightTrue = countRecur (k + 1 , j , True , s) rightFalse = countRecur (k + 1 , j , False , s) # Check the result of applying the operator at position # `k` to the subproblems if evaluate (True , True , s [k]) == req : ans += leftTrue * rightTrue if evaluate (True , False , s [k]) == req : ans += leftTrue * rightFalse if evaluate (False , True , s [k]) == req : ans += leftFalse * rightTrue if evaluate (False , False , s [k]) == req : ans += leftFalse * rightFalse return ans # Function to count the number of ways the expression # evaluates to True def countWays (s): n = len (s) return countRecur (0 , n - 1 , True , s) if __name__ == "__main__" : s = "T|T&F^T" print (countWays (s)) C# // C# program to implement Boolean // Parenthesization Problem using recursion using System ; class GfG { static bool evaluate (bool b1 , bool b2 , char op) { if (op == '&') { return b1 & b2 ; } else if (op == '|') { return b1 | b2 ; } return b1 ^ b2 ; } // Function which returns the number of ways // s[i:j] evaluates to req. static int countRecur (int i , int j , bool req , string s) { // Base case: if (i == j) { return (req == (s [i] == 'T')) ? 1 : 0 ; } int ans = 0 ; for (int k = i + 1 ; k < j ; k += 2) { // Count Ways in which left substring // evaluates to true and false. int leftTrue = countRecur (i , k - 1 , true , s); int leftFalse = countRecur (i , k - 1 , false , s); // Count Ways in which right substring // evaluates to true and false. int rightTrue = countRecur (k + 1 , j , true , s); int rightFalse = countRecur (k + 1 , j , false , s); // Check if the combinations results // to req. if (evaluate (true , true , s [k]) == req) { ans += leftTrue * rightTrue ; } if (evaluate (true , false , s [k]) == req) { ans += leftTrue * rightFalse ; } if (evaluate (false , true , s [k]) == req) { ans += leftFalse * rightTrue ; } if (evaluate (false , false , s [k]) == req) { ans += leftFalse * rightFalse ; } } return ans ; } static int countWays (string s) { int n = s . Length ; return countRecur (0 , n - 1 , true , s); } static void Main (string [] args) { string s = "T|T&F^T" ; Console . WriteLine (countWays (s)); } } JavaScript // JavaScript program to implement Boolean // Parenthesization Problem using recursion // Function to evaluate a // boolean condition. function evaluate (b1 , b2 , op) { if (op === "&") { return b1 & b2 ; } else if (op === "|") { return b1 | b2 ; } return b1 ^ b2 ; } // Function which returns the number of ways // s[i:j] evaluates to req. function countRecur (i , j , req , s) { // Base case: if (i === j) { return (req === 1) === (s [i] === "T") ? 1 : 0 ; } let ans = 0 ; for (let k = i + 1 ; k < j ; k += 2) { // Count Ways in which left substring // evaluates to true and false. const leftTrue = countRecur (i , k - 1 , 1 , s); const leftFalse = countRecur (i , k - 1 , 0 , s); // Count Ways in which right substring // evaluates to true and false. const rightTrue = countRecur (k + 1 , j , 1 , s); const rightFalse = countRecur (k + 1 , j , 0 , s); // Check if the combinations results // to req. if (evaluate (1 , 1 , s [k]) === req) { ans += leftTrue * rightTrue ; } if (evaluate (1 , 0 , s [k]) === req) { ans += leftTrue * rightFalse ; } if (evaluate (0 , 1 , s [k]) === req) { ans += leftFalse * rightTrue ; } if (evaluate (0 , 0 , s [k]) === req) { ans += leftFalse * rightFalse ; } } return ans ; } function countWays (s) { let n = s . length ; return countRecur (0 , n - 1 , 1 , s); } // driver code const s = "T|T&F^T" ; console . log (countWays (s)); Output 4 [Expected Approach 1]- Using Top-Down DP - O(n^3) Time and O(n^2) Space If we notice carefully, we can observe that the above recursive solution holds the following two properties of Dynamic Programming : 1. Optimal Substructure : Number of ways to make expression s[i, j] evaluate to req depends on the optimal solutions of countWays(i, k-1, 0), countWays(i, k-1, 1), countWays(k+1, j, 0) and countWays(k+1, j, 1) where k lies between i and j. 2. Overlapping Subproblems : While applying a recursive approach in this problem, we notice that certain subproblems are computed multiple times. For example, countWays(0, 4, 1) and countWays(0, 7, 1) will call the same subproblem countWays(0, 2, 0) twice. There are three parameters : i, j, req that changes in the recursive solution. So we create a 3D matrix of size n*n*2 for memoization . We initialize this matrix as -1 to indicate nothing is computed initially. Now we modify our recursive solution to first check if the value is -1, then only make recursive calls. This way, we avoid re-computations of the same subproblems. C++ #include <bits/stdc++.h> using namespace std ; // Function to evaluate a // boolean condition. bool evaluate (int b1 , int b2 , char op) { if (op == '&') { return b1 & b2 ; } else if (op == '|') { return b1 | b2 ; } return b1 ^ b2 ; } // Function which returns the

number of ways // s[i:j] evaluates to req. int countRecur (int i , int j , int req , string & s , vector < vector < vector < int >>> & memo) { // Base case: if (i == j) { return (req == (s [i] == 'T')) ? 1 : 0 ; } // If value is memoized if (memo [i][j][req] != -1) return memo [i][j][req]; int ans = 0 ; for (int k = i + 1 ; k < j ; k += 2) { // Count Ways in which left substring // evaluates to true and false. int leftTrue = countRecur (i , k - 1 , 1 , s , memo); int leftFalse = countRecur (i , k - 1 , 0 , s , memo); // Count Ways in which right substring // evaluates to true and false. int rightTrue = countRecur (k + 1 , j , 1 , s , memo); int rightFalse = countRecur (k + 1 , j , 0 , s , memo); // Check if the combinations results // to req. if (evaluate (1 , 1 , s [k]) == req) { ans += leftTrue * rightTrue ; } if (evaluate (1 , 0 , s [k]) == req) { ans += leftTrue * rightFalse ; } if (evaluate (0 , 1 , s [k]) == req) { ans += leftFalse * rightTrue ; } if (evaluate (0 , 0 , s [k]) == req) { ans += leftFalse * rightFalse ; } } return memo [i][j][req] = ans ; } int countWays (string s) { int n = s . length (); vector < vector < vector < int >>> memo (n , vector < vector < int >> (n , vector < int > (2 , -1))); return countRecur (0 , n - 1 , 1 , s , memo); } int main () { string s = "T|T&F^T" ; cout << countWays (s); return 0 ; } Java import java.util.Arrays ; class GfG { // Function to evaluate a // boolean condition. static boolean evaluate (int b1 , int b2 , char op) { if (op == '&') { return (b1 & b2) == 1 ; } else if (op == '|') { return (b1 | b2) == 1 ; } return (b1 ^ b2) == 1 ; } // Function which returns the number of ways // s[i:j] evaluates to req. static int countRecur (int i , int j , int req , String s , int [][][] memo) { // Base case: if (i == j) { return (req == (s . charAt (i) == 'T' ? 1 : 0)) ? 1 : 0 ; } // If value is memoized if (memo [i][j][req] != - 1) { return memo [i][j][req] ; } int ans = 0 ; for (int k = i + 1 ; k < j ; k += 2) { // Count Ways in which left substring // evaluates to true and false. int leftTrue = countRecur (i , k - 1 , 1 , s , memo); int leftFalse = countRecur (i , k - 1 , 0 , s , memo); // Count Ways in which right substring // evaluates to true and false. int rightTrue = countRecur (k + 1 , j , 1 , s , memo); int rightFalse = countRecur (k + 1 , j , 0 , s , memo); // Check if the combinations result // to req. if (evaluate (1 , 1 , s . charAt (k)) == (req == 1)) { ans += leftTrue * rightTrue ; } if (evaluate (1 , 0 , s . charAt (k)) == (req == 1)) { ans += leftTrue * rightFalse ; } if (evaluate (0 , 1 , s . charAt (k)) == (req == 1)) { ans += leftFalse * rightTrue ; } if (evaluate (0 , 0 , s . charAt (k)) == (req == 1)) { ans += leftFalse * rightFalse ; } } return memo [i][j][req] = ans ; } static int countWays (String s) { int n = s . length (); int [][][] memo = new int [n][n][2] ; for (int [][] mat : memo) { for (int [] row : mat) { Arrays . fill (row , - 1); } } return countRecur (0 , n - 1 , 1 , s , memo); } public static void main (String [] args) { String s = "T|T&F^T" ; System . out . println (countWays (s)); } } Python # Function to evaluate a # boolean condition. def evaluate (b1 , b2 , op): if op == '&' : return b1 & b2 elif op == '|' : return b1 | b2 return b1 ^ b2 # Function which returns the number of ways # s[i:j] evaluates to req. def countRecur (i , j , req , s , memo): # Base case: if i == j : return 1 if req == (1 if s [i] == 'T' else 0) else 0 # If value is memoized if memo [i][j][req] != - 1 : return memo [i][j][req] ans = 0 for k in range (i + 1 , j , 2): # Count Ways in which left substring # evaluates to true and false. left_true = countRecur (i , k - 1 , 1 , s , memo) left_false = countRecur (i , k - 1 , 0 , s , memo) # Count Ways in which right substring # evaluates to true and false. right_true = countRecur (k + 1 , j , 1 , s , memo) right_false = countRecur (k + 1 , j , 0 , s , memo) # Check if the combinations result # to req. if evaluate (1 , 1 , s [k]) == req : ans += left_true * right_true if evaluate (1 , 0 , s [k]) == req : ans += left_true * right_false if evaluate (0 , 1 , s [k]) == req : ans += left_false * right_true if evaluate (0 , 0 , s [k]) == req : ans += left_false * right_false memo [i][j][req] = ans return ans def countWays (s): n = len (s) memo = [[[-1 for _ in range (2)] for _ in range (n)] for _ in range (n)] return countRecur (0 , n - 1 , 1 , s , memo) if __name__ == "__main__" : s = "T|T&F^T" print (countWays (s)) C# using System ; class GfG { // Function to evaluate a // boolean condition. static bool evaluate (int b1 , int b2 , char op) { if (op == '&') { return (b1 & b2) == 1 ; } else if (op == '|') { return (b1 | b2) == 1 ; } return (b1 ^ b2) == 1 ; } // Function which returns the number of ways // s[i:j] evaluates to req. static int countRecur (int i , int j , int req , string s , int [, ,] memo) { // Base case: if (i == j) { return (req == (s [i] == 'T' ? 1 : 0)) ? 1 : 0 ; } // If value is memoized if (memo [i , j , req] != - 1) { return memo [i , j , req]; } int ans = 0 ; for (int k = i + 1 ; k < j ; k += 2) { // Count Ways in which left substring // evaluates to true and false. int leftTrue = countRecur (i , k - 1 , 1 , s , memo); int leftFalse = countRecur (i , k - 1 , 0 , s , memo); // Count Ways in which right substring // evaluates to true and false. int rightTrue = countRecur (k + 1 , j , 1 , s , memo); int rightFalse = countRecur (k + 1 , j , 0 , s , memo); // Check if the combinations result // to req. if (evaluate (1 , 1 , s [k]) == (req == 1)) { ans += leftTrue * rightTrue ; } if (evaluate (1 , 0 , s [k]) == (req == 1)) { ans += leftTrue * rightFalse ; } if (evaluate (0 , 1 , s [k]) == (req == 1)) { ans += leftFalse * rightTrue ; } if (evaluate (0 , 0 , s [k]) == (req == 1)) { ans += leftFalse * rightFalse ; } } return memo [i , j , req] = ans ; } static int countWays (string s) { int n = s . Length ; int [, ,] memo = new int [n , n , 2]; for (int i = 0 ; i < n ; i ++) { for (int j = 0 ; j < n ; j ++) { for (int k = 0 ; k < 2 ; k ++) { memo [i , j , k] = - 1 ; } } } return countRecur (0 , n - 1 , 1 , s , memo); } static void Main (string [] args) { string s =

"T|T&F^T" ; Console . WriteLine (countWays (s)); } } JavaScript // Function to evaluate a // boolean condition. function evaluate (b1 , b2 , op) { if (op === "&") { return b1 & b2 ; } else if (op === "|") { return b1 | b2 ; } return b1 ^ b2 ; } // Function which returns the number of ways // s[i:j] evaluates to req. function countRecur (i , j , req , s , memo) { // Base case: if (i === j) { return req === (s [i] === "T" ? 1 : 0) ? 1 : 0 ; } // If value is memoized if (memo [i][j][req] !== - 1) { return memo [i][j][req]; } let ans = 0 ; for (let k = i + 1 ; k < j ; k += 2) { // Count Ways in which left substring // evaluates to true and false. let leftTrue = countRecur (i , k - 1 , 1 , s , memo); let leftFalse = countRecur (i , k - 1 , 0 , s , memo); // Count Ways in which right substring // evaluates to true and false. let rightTrue = countRecur (k + 1 , j , 1 , s , memo); let rightFalse = countRecur (k + 1 , j , 0 , s , memo); // Check if the combinations result // to req. if (evaluate (1 , 1 , s [k]) === req) { ans += leftTrue * rightTrue ; } if (evaluate (1 , 0 , s [k]) === req) { ans += leftTrue * rightFalse ; } if (evaluate (0 , 1 , s [k]) === req) { ans += leftFalse * rightTrue ; } if (evaluate (0 , 0 , s [k]) === req) { ans += leftFalse * rightFalse ; } } return (memo [i][j][req] = ans); } function countWays (s) { let n = s . length ; let memo = Array . from ({ length : n }, () => Array . from ({ length : n }, () => Array (2). fill (- 1))); return countRecur (0 , n - 1 , 1 , s , memo); } // driver code const s = "T|T&F^T" ; console . log (countWays (s)); Output 4 [Expected Approach 2]- Using Bottom-Up DP - O(n^3) Time and O(n^2) Space This problem is solved using tabulation (bottom-up DP) . We define a 3D DP table where: dp[i][j][1] stores the number of ways the subexpression s[i:j] evaluates to True . dp[i][j][0] stores the number of ways the subexpression s[i:j] evaluates to False . To fill the DP table, we iterate over all possible substrings s[i:j] and break them into two parts at every operator s[k] . The left part is s[i:k-1] , and the right part is s[k+1:j] . Based on the operator (& , | , ^), we compute the number of ways to get True and False . The final result is stored in dp[0][n-1][1] , which gives the total ways to evaluate the full expression to True . Illustration: The iterative implementation is going to be tricky here we initially know diagonal values (which are 0), our result is going to be at the top right corner (or dp[0][n-1][1]) and we never access lower diagonal values. So we cannot fill the matrix with a normal traversal, we rather need to fill in diagonal manner. We fill the matrix using a variable len that stores differences between row and column indexes. We keep increasing len by 2 until it becomes n-1 (for the top right element) C++ #include <bits/stdc++.h> using namespace std ; // Function to evaluate a boolean condition. bool evaluate (int b1 , int b2 , char op) { if (op == '&') return b1 & b2 ; if (op == '|') return b1 | b2 ; return b1 ^ b2 ; } // Function to count ways to parenthesize the expression to get 'True' int countWays (string s) { int n = s . length (); vector < vector < vector < int >>> dp (n , vector < vector < int >> (n , vector < int > (2 , 0))); // Base case: Single operands (T or F) for (int i = 0 ; i < n ; i += 2) { dp [i][i][1] = (s [i] == 'T'); dp [i][i][0] = (s [i] == 'F'); } // Iterate over different substring lengths for (int len = 2 ; len < n ; len += 2) { // len increases by 2 (odd indices are operators) for (int i = 0 ; i < n - len ; i += 2) { int j = i + len ; // Reset values for the current subproblem dp [i][j][0] = dp [i][j][1] = 0 ; for (int k = i + 1 ; k < j ; k += 2) { // Iterate over operators char op = s [k]; int leftTrue = dp [i][k - 1][1], leftFalse = dp [i][k - 1][0]; int rightTrue = dp [k + 1][j][1], rightFalse = dp [k + 1][j][0]; // Count ways to get True or False if (evaluate (1 , 1 , op)) dp [i][j][1] += leftTrue * rightTrue ; if (evaluate (1 , 0 , op)) dp [i][j][1] += leftTrue * rightFalse ; if (evaluate (0 , 1 , op)) dp [i][j][1] += leftFalse * rightTrue ; if (evaluate (0 , 0 , op)) dp [i][j][1] += leftFalse * rightFalse ; if (! evaluate (1 , 1 , op)) dp [i][j][0] += leftTrue * rightTrue ; if (! evaluate (1 , 0 , op)) dp [i][j][0] += leftTrue * rightFalse ; if (! evaluate (0 , 1 , op)) dp [i][j][0] += leftFalse * rightTrue ; if (! evaluate (0 , 0 , op)) dp [i][j][0] += leftFalse * rightFalse ; } } } // Return ways to make entire expression True return dp [0][n - 1][1]; } int main () { string s = "T|T&F^T" ; cout << countWays (s) << endl ; return 0 ; } Java /*package whatever //do not write package name here */ import java.io.* ; class GfG { static boolean evaluate (int b1 , int b2 , char op) { if (op == '&') return (b1 & b2) == 1 ; if (op == '|') return (b1 | b2) == 1 ; return (b1 ^ b2) == 1 ; } static int countWays (String s) { int n = s . length (); int [][][] dp = new int [n][n][2] ; for (int i = 0 ; i < n ; i += 2) { dp [i][i][1] = (s . charAt (i) == 'T') ? 1 : 0 ; dp [i][i][0] = (s . charAt (i) == 'F') ? 1 : 0 ; } for (int len = 2 ; len < n ; len += 2) { for (int i = 0 ; i < n - len ; i += 2) { int j = i + len ; dp [i][j][0] = dp [i][j][1] = 0 ; for (int k = i + 1 ; k < j ; k += 2) { char op = s . charAt (k); int leftTrue = dp [i][k - 1][1] , leftFalse = dp [i][k - 1][0] ; int rightTrue = dp [k + 1][j][1] , rightFalse = dp [k + 1][j][0] ; if (evaluate (1 , 1 , op)) dp [i][j][1] += leftTrue * rightTrue ; if (evaluate (1 , 0 , op)) dp [i][j][1] += leftTrue * rightFalse ; if (evaluate (0 , 1 , op)) dp [i][j][1] += leftFalse * rightTrue ; if (evaluate (0 , 0 , op)) dp [i][j][1] += leftFalse * rightFalse ; if (! evaluate (1 , 1 , op)) dp [i][j][0] += leftTrue * rightTrue ; if (! evaluate (1 , 0 , op)) dp [i][j][0] += leftTrue * rightFalse ; if (! evaluate (0 , 1 , op)) dp [i][j][0] += leftFalse * rightTrue ; if (! evaluate (0 , 0 , op)) dp [i][j][0] += leftFalse * rightFalse ; } } } return dp [0][n - 1][1] ; } public static void main (String [] args) { String s = "T|T&F^T" ; System . out . println (countWays (s

)); } } Python def evaluate (b1 , b2 , op): if op == '&' : return b1 & b2 if op == '|' : return b1 | b2 return b1 ^ b2 def countWays (s): n = len (s) dp = [[[0 , 0] for _ in range (n)] for _ in range (n)] # Base case: Single operands (T or F) for i in range (0 , n , 2): dp [i][i][1] = 1 if s [i] == 'T' else 0 dp [i][i][0] = 1 if s [i] == 'F' else 0 # Iterate over different substring lengths for length in range (2 , n , 2): # length increases by 2 (odd indices are operators) for i in range (0 , n - length , 2): j = i + length # Reset values for the current subproblem dp [i][j][0] = dp [i][j][1] = 0 for k in range (i + 1 , j , 2): # Iterate over operators op = s [k] leftTrue , leftFalse = dp [i][k - 1][1], dp [i][k - 1][0] rightTrue , rightFalse = dp [k + 1][j][1], dp [k + 1][j][0] # Count ways to get True or False if evaluate (1 , 1 , op): dp [i][j][1] += leftTrue * rightTrue if evaluate (1 , 0 , op): dp [i][j][1] += leftTrue * rightFalse if evaluate (0 , 1 , op): dp [i][j][1] += leftFalse * rightTrue if evaluate (0 , 0 , op): dp [i][j][1] += leftFalse * rightFalse if not evaluate (1 , 1 , op): dp [i][j][0] += leftTrue * rightTrue if not evaluate (1 , 0 , op): dp [i][j][0] += leftTrue * rightFalse if not evaluate (0 , 1 , op): dp [i][j][0] += leftFalse * rightTrue if not evaluate (0 , 0 , op): dp [i][j][0] += leftFalse * rightFalse return dp [0][n - 1][1] # Return ways to make entire expression True if __name__ == "__main__" : s = "T|T&F^T" print (countWays (s)) C# using System ; class GfG { // Function to evaluate a boolean condition. static bool Evaluate (int b1 , int b2 , char op) { if (op == '&') return (b1 & b2) == 1 ; if (op == '|') return (b1 | b2) == 1 ; return (b1 ^ b2) == 1 ; } // Function to count ways to parenthesize the expression // to get 'True' static int CountWays (string s) { int n = s . Length ; int [, ,] dp = new int [n , n , 2]; // Base case: Single operands (T or F) for (int i = 0 ; i < n ; i += 2) { dp [i , i , 1] = (s [i] == 'T') ? 1 : 0 ; dp [i , i , 0] = (s [i] == 'F') ? 1 : 0 ; } // Iterate over different substring lengths for (int len = 2 ; len < n ; len += 2) { for (int i = 0 ; i < n - len ; i += 2) { int j = i + len ; dp [i , j , 0] = dp [i , j , 1] = 0 ; for (int k = i + 1 ; k < j ; k += 2) { char op = s [k]; int leftTrue = dp [i , k - 1 , 1], leftFalse = dp [i , k - 1 , 0]; int rightTrue = dp [k + 1 , j , 1], rightFalse = dp [k + 1 , j , 0]; if (Evaluate (1 , 1 , op)) dp [i , j , 1] += leftTrue * rightTrue ; if (Evaluate (1 , 0 , op)) dp [i , j , 1] += leftTrue * rightFalse ; if (Evaluate (0 , 1 , op)) dp [i , j , 1] += leftFalse * rightTrue ; if (Evaluate (0 , 0 , op)) dp [i , j , 1] += leftFalse * rightFalse ; if (! Evaluate (1 , 1 , op)) dp [i , j , 0] += leftTrue * rightTrue ; if (! Evaluate (1 , 0 , op)) dp [i , j , 0] += leftTrue * rightFalse ; if (! Evaluate (0 , 1 , op)) dp [i , j , 0] += leftFalse * rightTrue ; if (! Evaluate (0 , 0 , op)) dp [i , j , 0] += leftFalse * rightFalse ; } } } return dp [0 , n - 1 , 1]; } static void Main () { string s = "T|T&F^T" ; Console . WriteLine (CountWays (s)); } } JavaScript // Function to evaluate a boolean condition. function evaluate (b1 , b2 , op) { if (op === "&") return b1 & b2 ; if (op === "|") return b1 | b2 ; return b1 ^ b2 ; } // Function to count ways to parenthesize the expression to // get 'True' function countWays (s) { let n = s . length ; let dp = Array . from ({ length : n }, () => Array . from ({ length : n }, () => [0 , 0])); // Base case: Single operands (T or F) for (let i = 0 ; i < n ; i += 2) { dp [i][i][1] = s [i] === "T" ? 1 : 0 ; dp [i][i][0] = s [i] === "F" ? 1 : 0 ; } // Iterate over different substring lengths for (let len = 2 ; len < n ; len += 2) { for (let i = 0 ; i < n - len ; i += 2) { let j = i + len ; dp [i][j][0] = dp [i][j][1] = 0 ; for (let k = i + 1 ; k < j ; k += 2) { let op = s [k]; let leftTrue = dp [i][k - 1][1], leftFalse = dp [i][k - 1][0]; let rightTrue = dp [k + 1][j][1], rightFalse = dp [k + 1][j][0]; if (evaluate (1 , 1 , op)) dp [i][j][1] += leftTrue * rightTrue ; if (evaluate (1 , 0 , op)) dp [i][j][1] += leftTrue * rightFalse ; if (evaluate (0 , 1 , op)) dp [i][j][1] += leftFalse * rightTrue ; if (evaluate (0 , 0 , op)) dp [i][j][1] += leftFalse * rightFalse ; if (! evaluate (1 , 1 , op)) dp [i][j][0] += leftTrue * rightTrue ; if (! evaluate (1 , 0 , op)) dp [i][j][0] += leftTrue * rightFalse ; if (! evaluate (0 , 1 , op)) dp [i][j][0] += leftFalse * rightTrue ; if (! evaluate (0 , 0 , op)) dp [i][j][0] += leftFalse * rightFalse ; } } } return dp [0][n - 1][1]; } // Driver Code let s = "T|T&F^T" ; console . log (countWays (s)); Output 4 Comment Article Tags: Article Tags: Dynamic Programming DSA Microsoft Amazon Linkedin + 1 More