Count Ways To Assign Unique Cap To Every Person - GeeksforGeeks

Source: https://www.geeksforgeeks.org/bitmasking-and-dynamic-programming-set-1-count-ways-to-assign-unique-cap-to-every-person/

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 Count Ways To Assign Unique Cap To Every Person Last Updated : 23 Jul, 2025 Given n people and 100 types of caps labelled from 1 to 100 , along with a 2D integer array caps where caps[i] represents the list of caps preferred by the i-th person , the task is to determine the number of ways the n people can wear different caps. Example: Input: caps = [[3, 4], [4, 5], [5]] Output: 1 Explanation: First person choose cap 3, Second person choose cap 4 and last one cap 5. Input: caps = [[3, 5, 1], [3, 5]] Output: 4 Explanation: There are 4 ways to choose hats: (3, 5), (5, 3), (1, 3) and (1, 5) Table of Content Using Recursion Using Top-Down DP (Memoization) - O(n * 2^n) Time and O(2^n) Space Using Bottom-Up DP (Tabulation) – O(n * 2^n) Time and O(2^n) Space Using Recursion In the recursive approach, there are two cases for each cap: Skip the current cap, this means we move to the next cap without assigning the current cap to any person, keeping the assigned count unchanged. The recursive call will look like: dfs(assignedCount, cap+1) Assign the current cap to each person who prefers it: For each person who likes the current cap, if they do not already have a cap assigned, we assign this cap to them and move to the next cap with the assigned count incremented by 1 . After the recursive call, we backtrack by unassigning the cap to explore other possibilities. The recurrence relation is as follows: Base Cases: If assignedCount == totalPeople , return 1, as this indicates all people have a unique cap assigned. If cap > 100 , return 0, as this means there are no more caps left to assign but not all people have received a cap. dfs(assignedCount, cap) = dfs(assignedCount, cap+1) + ∑person in capToPeople[cap] dfs(assignedCount+1, cap+1) if assignedPeople[person] == false , we loop through each person who prefers the current cap, check if they have already been assigned a cap, and if not, assign it to them and recurse to explore further assignments with the updated assignedCount. C++ // C++ Code to Assign Unique Cap To Every Person // using Recursion #include <bits/stdc++.h> using namespace std ; // Recursive function to calculate the number of ways // to assign caps to people such that each person has // a unique cap int dfs (int assignedCount , vector < bool >& assignedPeople , int cap , vector < vector < int >>& capToPeople , int totalPeople) { // Base case: if all people have a cap assigned, return 1 if (assignedCount == totalPeople) { return 1 ; } // If we've considered all caps and not everyone // has a cap, return 0 if (cap > 100) { return 0 ; } // Case: skip the current cap int ways = dfs (assignedCount , assignedPeople , cap + 1 , capToPeople , totalPeople); // Assign the current cap to each person who likes it for (int person : capToPeople [cap]) { // Check if the person already has a cap assigned if (! assignedPeople [person]) { // Assign current cap to the person assignedPeople [person] = true ; // Recurse with increased assigned count ways = ways + dfs (assignedCount + 1 , assignedPeople , cap + 1 , capToPeople , totalPeople); // Backtrack: unassign the cap for other possibilities assignedPeople [person] = false ; } } return ways ; } // Main function to calculate the number of ways to assign caps int numberWays (vector < vector < int >>& caps) { int n = caps . size (); // Map each cap to the list of people who prefer it vector < vector < int >> capToPeople (101); for (int i = 0 ; i < n ; ++ i) { for (int cap : caps [i]) { capToPeople [cap]. push_back (i); } } // Initialize assignedPeople vector to track // assigned caps vector < bool > assignedPeople (n , false); // Call the recursive function starting from the first cap return dfs (0 , assignedPeople , 1 , capToPeople , n); } int main () { vector < vector < int >> caps = {{ 1 , 2 , 3 }, { 1 , 2 }, { 3 , 4 }, { 4 , 5 }}; cout << numberWays (

caps) << endl ; return 0 ; } Java // Java Code to Assign Unique Cap To Every Person // using Recursion import java.util.ArrayList ; import java.util.List ; class GfG { // Recursive function to calculate the number of ways // to assign caps to people such that each person has // a unique cap static int dfs (int assignedCount , ArrayList < Boolean > assignedPeople , int cap , ArrayList < ArrayList < Integer >> capToPeople , int totalPeople) { // Base case: if all people have a cap assigned, return 1 if (assignedCount == totalPeople) { return 1 ; } // If we've considered all caps and not everyone // has a cap, return 0 if (cap > 100) { return 0 ; } // Case: skip the current cap int ways = dfs (assignedCount , assignedPeople , cap + 1 , capToPeople , totalPeople); // Assign the current cap to each person who likes it for (int person : capToPeople . get (cap)) { // Check if the person already has a cap assigned if (! assignedPeople . get (person)) { // Assign current cap to the person assignedPeople . set (person , true); // Recurse with increased assigned count ways = ways + dfs (assignedCount + 1 , assignedPeople , cap + 1 , capToPeople , totalPeople); // Backtrack: unassign the cap for other possibilities assignedPeople . set (person , false); } } return ways ; } // Main function to calculate the number of ways to assign caps static int numberWays (ArrayList < ArrayList < Integer >> caps) { int n = caps . size (); // Map each cap to the list of people who prefer it ArrayList < ArrayList < Integer >> capToPeople = new ArrayList <> (101); for (int i = 0 ; i <= 100 ; i ++) { capToPeople . add (new ArrayList <> ()); } for (int i = 0 ; i < n ; ++ i) { for (int cap : caps . get (i)) { capToPeople . get (cap). add (i); } } // Initialize assignedPeople list to track assigned caps ArrayList < Boolean > assignedPeople = new ArrayList <> (n); for (int i = 0 ; i < n ; i ++) { assignedPeople . add (false); } // Call the recursive function starting from the first cap return dfs (0 , assignedPeople , 1 , capToPeople , n); } public static void main (String [] args) { ArrayList < ArrayList < Integer >> caps = new ArrayList <> (); caps . add (new ArrayList <> (List . of (1 , 2 , 3))); caps . add (new ArrayList <> (List . of (1 , 2))); caps . add (new ArrayList <> (List . of (3 , 4))); caps . add (new ArrayList <> (List . of (4 , 5))); System . out . println (numberWays (caps)); } } Python # Python Code to Assign Unique Cap To Every Person # using Recursion def dfs (assigned_count , assigned_people , cap , cap_to_people , total_people): # Base case: if all people have a cap assigned, return 1 if assigned_count == total_people : return 1 # If we've considered all caps and not everyone # has a cap, return 0 if cap > 100 : return 0 # Case: skip the current cap ways = dfs (assigned_count , assigned_people , cap + 1 , cap_to_people , total_people) # Assign the current cap to each person who likes it for person in cap_to_people [cap]: # Check if the person already has a cap assigned if not assigned_people [person]: # Assign current cap to the person assigned_people [person] = True # Recurse with increased assigned count ways += dfs (assigned_count + 1 , assigned_people , cap + 1 , cap_to_people , total_people) # Backtrack: unassign the cap for other possibilities assigned_people [person] = False return ways # Main function to calculate the number # of ways to assign caps def number_ways (caps): n = len (caps) # Map each cap to the list of people who prefer it cap_to_people = [[] for _ in range (101)] for i in range (n): for cap in caps [i]: cap_to_people [cap] . append (i) # Initialize assigned_people list to track assigned caps assigned_people = [False] * n # Call the recursive function starting from the first cap return dfs (0 , assigned_people , 1 , cap_to_people , n) if __name__ == "__main__" : caps = [[1 , 2 , 3], [1 , 2], [3 , 4], [4 , 5]] print (number_ways (caps)) C# // C# Code to Assign Unique Cap To Every Person // using Recursion using System ; using System.Collections.Generic ; class GfG { // Recursive function to calculate the number of ways // to assign caps to people such that each person has // a unique cap static int Dfs (int assignedCount , bool [] assignedPeople , int cap , List < List < int >> capToPeople , int totalPeople) { // Base case: if all people have a cap assigned, return 1 if (assignedCount == totalPeople) { return 1 ; } // If we've considered all caps and not everyone // has a cap, return 0 if (cap > 100) { return 0 ; } // Case: skip the current cap int ways = Dfs (assignedCount , assignedPeople , cap + 1 , capToPeople , totalPeople); // Assign the current cap to each person who likes it foreach (int person in capToPeople [cap]) { // Check if the person already has a cap assigned if (! assignedPeople [person]) { // Assign current cap to the person assignedPeople [person] = true ; // Recurse with increased assigned count ways += Dfs (assignedCount + 1 , assignedPeople , cap + 1 , capToPeople , totalPeople); // Backtrack: unassign the cap for other // possibilities assignedPeople [person] = false ; } } return ways ; } // Main function to calculate the number of ways // to assign caps static int NumberWays (List < List < int >> caps) { int n = caps . Count ; // Map each cap to the list of people who prefer it List < List < int >> capToPeople = new List < List < int >> (); for (int i = 0 ; i <= 100 ; i ++) { capToPeople . Add (new List < int > ()); } for (int i = 0 ; i < n ; i ++) { foreach (int cap in caps [i]) { capToPeople [cap]. Add (i); } } // Initialize assignedPeople array to // track assigned caps bool [] assignedPeople = new bool [n]; // Call the recursive function starting // from the first cap return Dfs (0 , assignedPeople , 1 , capToPeople , n); }

static void Main () { List < List < int >> caps = new List < List < int >> { new List < int > { 1 , 2 , 3 }, new List < int > { 1 , 2 }, new List < int > { 3 , 4 }, new List < int > { 4 , 5 } }; Console . WriteLine (NumberWays (caps)); } } JavaScript // Javascript Code to Assign Unique Cap To Every Person // using Recursion function dfs (assignedCount , assignedPeople , cap , capToPeople , totalPeople) { // Base case: if all people have a cap assigned, return 1 if (assignedCount === totalPeople) { return 1 ; } // If we've considered all caps and not everyone // has a cap, return 0 if (cap > 100) { return 0 ; } // Case: skip the current cap let ways = dfs (assignedCount , assignedPeople , cap + 1 , capToPeople , totalPeople); // Assign the current cap to each person who likes it for (let person of capToPeople [cap]) { // Check if the person already has a cap assigned if (! assignedPeople [person]) { // Assign current cap to the person assignedPeople [person] = true ; // Recurse with increased assigned count ways += dfs (assignedCount + 1 , assignedPeople , cap + 1 , capToPeople , totalPeople); // Backtrack: unassign the cap for other possibilities assignedPeople [person] = false ; } } return ways ; } // Main function to calculate the number of // ways to assign caps function numberWays (caps) { const n = caps . length ; // Map each cap to the list of people who prefer it const capToPeople = Array . from ({ length : 101 }, () => []); for (let i = 0 ; i < n ; i ++) { for (let cap of caps [i]) { capToPeople [cap]. push (i); } } // Initialize assignedPeople array to track assigned caps const assignedPeople = new Array (n). fill (false); // Call the recursive function starting from the first cap return dfs (0 , assignedPeople , 1 , capToPeople , n); } const caps = [[1 , 2 , 3], [1 , 2], [3 , 4], [4 , 5]]; console . log (numberWays (caps)); Output 8 The above solution will have a exponential time complexity. Using Top-Down DP (Memoization) - O(n * 2^n) Time and O(2^n) Space If we notice carefully, we can observe that the above recursive solution holds the following two properties of Dynamic Programming . 1. Optimal Substructure: The problem exhibits optimal substructure , meaning that the solution to the problem can be derived from the solutions of smaller subproblems. The recursive relation is: ways = dfs(allMask, assignedPeople, cap+1) (skip current cap) ways = ways + dfs(allMask, assignedPeople ■ (1 << person), cap+1) (assign current cap to a person) 2. Overlapping Subproblems: Many subproblems are computed multiple times with the same parameters (cap, assignedPeople) . To avoid recomputing the same subproblems , we store the result in a memoization table memo[cap][assignedPeople], which stores the number of ways to assign caps for a given cap and assignedPeople combination. C++ // C++ Code to Assign Unique Cap To Every Person // using Memoization and Bitmasking #include <bits/stdc++.h> using namespace std ; // Recursive function to count ways to assign // caps with memoization int dfs (int allMask , int assignedPeople , int cap , vector < vector < int >>& capToPeople , vector < vector < int >>& memo) { // Base case: if all people have hats assigned if (assignedPeople == allMask) { return 1 ; } // If we've considered all caps and not everyone // has a cap, return 0 if (cap > 100) { return 0 ; } // Return memoized result if already computed if (memo [cap][assignedPeople] != -1) { return memo [cap][assignedPeople]; } // Case: skip the current cap int ways = dfs (allMask , assignedPeople , cap + 1 , capToPeople , memo); // Try assigning the current cap to each person // who can wear it for (int person : capToPeople [cap]) { // Check if the person hasn't been assigned a cap yet if ((assignedPeople & (1 << person)) == 0) { // Assign current cap to the person ways = ways + dfs (allMask , assignedPeople | (1 << person), cap + 1 , capToPeople , memo); } } // Memoize and return the result return memo [cap][assignedPeople] = ways ; } // Main function to calculate the number of // ways to assign caps int numberWays (vector < vector < int >>& caps) { int n = caps . size (); int allMask = (1 << n) - 1 ; // Create adjacency matrix for // cap-to-people distribution vector < vector < int >> capToPeople (101); for (int i = 0 ; i < n ; ++ i) { for (int cap : caps [i]) { capToPeople [cap]. push_back (i); } } // Memo array to store computed results vector < vector < int >> memo (101 , vector < int > (1 << n , -1)); // Call the recursive function starting // from the first cap return dfs (allMask , 0 , 1 , capToPeople , memo); } int main () { vector < vector < int >> caps = {{ 1 , 2 , 3 }, { 1 , 2 }, { 3 , 4 }, { 4 , 5 }}; cout << numberWays (caps) << endl ; return 0 ; } Java // Java Code to Assign Unique Cap To Every Person // using Memoization and Bitmasking import java.util.ArrayList ; import java.util.List ; class GfG { // Recursive function to calculate the number of ways // to assign caps to people such that each person has // a unique cap static int dfs (int assignedCount , ArrayList < Boolean > assignedPeople , int cap , ArrayList < ArrayList < Integer >> capToPeople , int totalPeople) { // Base case: if all people have a cap assigned, return 1 if (assignedCount == totalPeople) { return 1 ; } // If we've considered all caps and not everyone // has a cap, return 0 if (cap > 100) { return 0 ; } // Case: skip the current cap int ways = dfs (assignedCount , assignedPeople , cap + 1 , capToPeople , totalPeople); // Assign the current cap to each person who likes it for (int person : capToPeople . get (cap)) { // Check if the person already has a cap assigned if (! assignedPeople . get (person)) { // Assign current cap to the person assignedPeople . set (person , true); // Recurse with increased assigned count ways = ways +

dfs (assignedCount + 1 , assignedPeople , cap + 1 , capToPeople , totalPeople); // Backtrack: unassign the cap for other possibilities assignedPeople . set (person , false); } } return ways ; } // Main function to calculate the number of ways to assign caps static int numberWays (ArrayList < ArrayList < Integer >> caps) { int n = caps . size (); // Map each cap to the list of people who prefer it ArrayList < ArrayList < Integer >> capToPeople = new ArrayList <> (101); for (int i = 0 ; i <= 100 ; i ++) { capToPeople . add (new ArrayList <> ()); } for (int i = 0 ; i < n ; ++ i) { for (int cap : caps . get (i)) { capToPeople . get (cap). add (i); } } // Initialize assignedPeople list to track assigned caps ArrayList < Boolean > assignedPeople = new ArrayList <> (n); for (int i = 0 ; i < n ; i ++) { assignedPeople . add (false); } // Call the recursive function starting from the first cap return dfs (0 , assignedPeople , 1 , capToPeople , n); } public static void main (String [] args) { ArrayList < ArrayList < Integer >> caps2 = new ArrayList <> (); caps2 . add (new ArrayList <> (List . of (1 , 2 , 3))); caps2 . add (new ArrayList <> (List . of (1 , 2))); caps2 . add (new ArrayList <> (List . of (3 , 4))); caps2 . add (new ArrayList <> (List . of (4 , 5))); System . out . println (numberWays (caps2)); } } Python # Python Code to Assign Unique Cap To Every Person # using Memoization and Bitmasking # Recursive function to count ways to assign # caps with memoization def dfs (all_mask , assigned_people , cap , cap_to_people , memo): # Base case: if all people have hats assigned if assigned_people == all_mask : return 1 # If we've considered all caps and not everyone # has a cap, return 0 if cap > 100 : return 0 # Return memoized result if already computed if memo [cap][assigned_people] != - 1 : return memo [cap][assigned_people] # Case: skip the current cap ways = dfs (all_mask , assigned_people , cap + 1 , cap_to_people , memo) # Try assigning the current cap to each person # who can wear it for person in cap_to_people [cap]: # Check if the person hasn't been assigned a cap yet if (assigned_people & (1 << person)) == 0 : # Assign current cap to the person ways += dfs (all_mask , assigned_people | (1 << person), \ cap + 1 , cap_to_people , memo) # Memoize and return the result memo [cap][assigned_people] = ways return ways # Main function to calculate the number of # ways to assign caps def number_ways (caps): n = len (caps) all_mask = (1 << n) - 1 # Create adjacency matrix for # cap-to-people distribution cap_to_people = [[] for _ in range (101)] for i in range (n): for cap in caps [i]: cap_to_people [cap] . append (i) # Memo array to store computed results memo = [[- 1] * (1 << n) for _ in range (101)] # Call the recursive function starting # from the first cap return dfs (all_mask , 0 , 1 , cap_to_people , memo) if __name__ == "__main__" : caps = [[1 , 2 , 3], [1 , 2], [3 , 4], [4 , 5]] print (number_ways (caps)) C# // C# Code to Assign Unique Cap To Every Person // using Memoization and Bitmasking using System ; using System.Collections.Generic ; class GfG { // Recursive function to count ways to assign // caps with memoization static int dfs (int allMask , int assignedPeople , int cap , List < List < int >> capToPeople , List < List < int >> memo) { // Base case: if all people have hats assigned if (assignedPeople == allMask) { return 1 ; } // If we've considered all caps and not everyone // has a cap, return 0 if (cap > 100) { return 0 ; } // Return memoized result if already computed if (memo [cap][assignedPeople] != - 1) { return memo [cap][assignedPeople]; } // Case: skip the current cap int ways = dfs (allMask , assignedPeople , cap + 1 , capToPeople , memo); // Try assigning the current cap to each person // who can wear it foreach (int person in capToPeople [cap]) { // Check if the person hasn't been assigned a cap yet if ((assignedPeople & (1 << person)) == 0) { // Assign current cap to the person ways += dfs (allMask , assignedPeople | (1 << person), cap + 1 , capToPeople , memo); } } // Memoize and return the result memo [cap][assignedPeople] = ways ; return ways ; } // Main function to calculate the number of // ways to assign caps static int NumberWays (List < List < int >> caps) { int n = caps . Count ; int allMask = (1 << n) - 1 ; // Create adjacency matrix for // cap-to-people distribution List < List < int >> capToPeople = new List < List < int >> (new List < int > [101]); for (int i = 0 ; i < 101 ; i ++) { capToPeople [i] = new List < int > (); } for (int i = 0 ; i < n ; i ++) { foreach (int cap in caps [i]) { capToPeople [cap]. Add (i); } } // Memo array to store computed results List < List < int >> memo = new List < List < int >> (new List < int > [101]); for (int i = 0 ; i < 101 ; i ++) { memo [i] = new List < int > (new int [1 << n]); for (int j = 0 ; j < (1 << n); j ++) { memo [i][j] = - 1 ; } } // Call the recursive function starting // from the first cap return dfs (allMask , 0 , 1 , capToPeople , memo); } static void Main (string [] args) { List < List < int >> caps = new List < List < int >> () { new List < int > { 1 , 2 , 3 }, new List < int > { 1 , 2 }, new List < int > { 3 , 4 }, new List < int > { 4 , 5 } }; Console . WriteLine (NumberWays (caps)); } } JavaScript // Javascript Code to Assign Unique Cap To Every Person // using Memoization and Bitmasking // Recursive function to count ways to assign // caps with memoization function dfs (allMask , assignedPeople , cap , capToPeople , memo) { // Base case: if all people have hats assigned if (assignedPeople === allMask) { return 1 ; } // If we've considered all caps and not everyone // has a cap, return 0 if (cap > 100) { return 0 ; } // Return memoized result if already computed if (memo [cap][assignedPeople] !== - 1) { return memo [cap][

assignedPeople]; } // Case: skip the current cap let ways = dfs (allMask , assignedPeople , cap + 1 , capToPeople , memo); // Try assigning the current cap to each person // who can wear it for (let person of capToPeople [cap]) { // Check if the person hasn't been assigned a cap yet if ((assignedPeople & (1 << person)) === 0) { // Assign current cap to the person ways += dfs (allMask , assignedPeople | (1 << person), cap + 1 , capToPeople , memo); } } // Memoize and return the result memo [cap][assignedPeople] = ways ; return ways ; } // Main function to calculate the number of // ways to assign caps function numberWays (caps) { let n = caps . length ; let allMask = (1 << n) - 1 ; // Create adjacency matrix for // cap-to-people distribution let capToPeople = Array . from ({ length : 101 }, () => []); for (let i = 0 ; i < n ; i ++) { for (let cap of caps [i]) { capToPeople [cap]. push (i); } } // Memo array to store computed results let memo = Array . from ({ length : 101 }, () => Array (1 << n). fill (- 1)); // Call the recursive function starting // from the first cap return dfs (allMask , 0 , 1 , capToPeople , memo); } let caps = [[1 , 2 , 3], [1 , 2], [3 , 4], [4 , 5]]; console . log (numberWays (caps)); Output 8 Using Bottom-Up DP (Tabulation) – O(n * 2^n) Time and O(2^n) Space We use a 2D DP table of size (number of caps + 1) * (2^n). The state dp[cap][assignedPeople] represents the number of ways to assign caps to people considering the first cap caps and assigning caps to the people represented by the bitmask assignedPeople. Dynamic Programming Relation: Base Case: dp[0][0] = 1, If no caps are assigned and no people are assigned any caps, there is 1 way to do nothing. Skip the current cap: dp[cap][assignedPeople] += dp[cap-1][assignedPeople] Assign the current cap to a person who hasn't been assigned a cap: dp[cap][assignedPeople | (1 << person)] += dp[cap - 1][assignedPeople] After filling the DP table, the final result will be stored in dp[100][allMask], which represents the number of ways to assign all caps to all people. C++ // C++ code to calculate the number of ways // to assign caps to people using Tabulation #include <bits/stdc++.h> using namespace std ; int numberWays (vector < vector < int >>& caps) { int n = caps . size (); int allMask = (1 << n) - 1 ; // Create adjacency matrix for cap-to-people distribution vector < vector < int >> capToPeople (101); for (int i = 0 ; i < n ; ++ i) { for (int cap : caps [i]) { capToPeople [cap]. push_back (i); } } // DP table: dp[cap][assignedPeople] // stores the number of ways // to assign caps for the first 'cap' // caps with 'assignedPeople' bitmask vector < vector < int >> dp (102 , vector < int > (1 << n , 0)); // Base case: With 0 caps, no people assigned, // there's 1 way (do nothing) dp [0][0] = 1 ; // Fill the DP table for (int cap = 1 ; cap <= 100 ; ++ cap) { for (int assignedPeople = 0 ; assignedPeople <= allMask ; ++ assignedPeople) { // If there are no ways to assign caps for // this state, continue if (dp [cap - 1][assignedPeople] == 0) { continue ; } // Case 1: Skip the current cap dp [cap][assignedPeople] += dp [cap - 1][assignedPeople]; // Case 2: Assign current cap to each person who can wear it for (int person : capToPeople [cap]) { // If the person hasn't been assigned a cap yet if ((assignedPeople & (1 << person)) == 0) { dp [cap][assignedPeople | (1 << person)] += dp [cap - 1][assignedPeople]; } } } } // The result will be in dp[100][allMask], // as we have considered all caps and assigned all people return dp [100][allMask]; } int main () { vector < vector < int >> caps = {{ 1 , 2 , 3 }, { 1 , 2 }, { 3 , 4 }, { 4 , 5 }}; cout << numberWays (caps) << endl ; return 0 ; } Java // Java code to calculate the number of ways // to assign caps to people using Tabulation import java.util.* ; class GfG { // Method to calculate the number of ways // to assign caps using Tabulation static int numberWays (List < List < Integer >> caps) { int n = caps . size (); int allMask = (1 << n) - 1 ; // Create adjacency matrix for cap-to-people distribution List < Integer >[] capToPeople = new ArrayList [101] ; for (int i = 0 ; i < 101 ; i ++) { capToPeople [i] = new ArrayList <> (); } // Fill the cap-to-people adjacency list for (int i = 0 ; i < n ; ++ i) { for (int cap : caps . get (i)) { capToPeople [cap] . add (i); } } // DP table: dp[cap][assignedPeople] // stores the number of ways to // assign caps for the first 'cap' caps // with 'assignedPeople' bitmask int [][] dp = new int [102][1 << n] ; // Base case: With 0 caps, no people assigned, // there's 1 way (do nothing) dp [0][0] = 1 ; // Fill the DP table for (int cap = 1 ; cap <= 100 ; ++ cap) { for (int assignedPeople = 0 ; assignedPeople <= allMask ; ++ assignedPeople) { // If there are no ways to assign caps for // this state, continue if (dp [cap - 1][assignedPeople] == 0) { continue ; } // Case 1: Skip the current cap dp [cap][assignedPeople] += dp [cap - 1][assignedPeople] ; // Case 2: Assign current cap to each person who can wear it for (int person : capToPeople [cap]) { // If the person hasn't been assigned a cap yet if ((assignedPeople & (1 << person)) == 0) { dp [cap][assignedPeople | (1 << person)] += dp [cap - 1][assignedPeople] ; } } } } // The result will be in dp[100][allMask], // as we have considered all caps and assigned all people return dp [100][allMask] ; } public static void main (String [] args) { List < List < Integer >> caps = new ArrayList <> (); caps . add (Arrays . asList (1 , 2 , 3)); caps . add (Arrays . asList (1 , 2)); caps . add (Arrays . asList (3 , 4)); caps . add (Arrays . asList (4 , 5)); System . out . println (numberWays (caps)); } } Python # Python code to calculate the number of ways # to assign caps to people using Tabulation def numberWays (

caps): n = len (caps) allMask = (1 << n) - 1 # Create adjacency list for cap-to-people # distribution capToPeople = [[] for _ in range (101)] # Fill the cap-to-people adjacency list for i in range (n): for cap in caps [i]: capToPeople [cap] . append (i) # DP table: dp[cap][assignedPeople] # stores the number of ways to assign # caps for the first 'cap' caps # with 'assignedPeople' bitmask dp = [[0] * (1 << n) for _ in range (102)] # Base case: With 0 caps, no people assigned, # there's 1 way (do nothing) dp [0][0] = 1 # Fill the DP table for cap in range (1 , 101): for assignedPeople in range (allMask + 1): # If there are no ways to assign caps for # this state, continue if dp [cap - 1][assignedPeople] == 0 : continue # Case 1: Skip the current cap dp [cap][assignedPeople] += dp [cap - 1][assignedPeople] # Case 2: Assign current cap to each person who can wear it for person in capToPeople [cap]: # If the person hasn't been assigned a cap yet if (assignedPeople & (1 << person)) == 0 : dp [cap][assignedPeople | (1 << person)] \ += dp [cap - 1][assignedPeople] # The result will be in dp[100][allMask], # as we have considered all caps and assigned all people return dp [100][allMask] if __name__ == "__main__" : caps = [[1 , 2 , 3], [1 , 2], [3 , 4], [4 , 5]] print (numberWays (caps)) C# // C# code to calculate the number of ways // to assign caps to people using Tabulation using System ; using System.Collections.Generic ; class GfG { static int NumberWays (List < List < int >> caps) { int n = caps . Count ; int allMask = (1 << n) - 1 ; // Create adjacency list for cap-to-people // distribution List < List < int >> capToPeople = new List < List < int >> (new List < int > [101]); for (int i = 0 ; i < 101 ; ++ i) { capToPeople [i] = new List < int > (); } // Fill the cap-to-people adjacency list for (int i = 0 ; i < n ; ++ i) { foreach (int cap in caps [i]) { capToPeople [cap]. Add (i); } } // DP table: dp[cap][assignedPeople] // stores the number of ways to assign // caps for the first 'cap' caps // with 'assignedPeople' bitmask int [,] dp = new int [102 , 1 << n]; // Base case: With 0 caps, no people assigned, // there's 1 way (do nothing) dp [0 , 0] = 1 ; // Fill the DP table for (int cap = 1 ; cap <= 100 ; ++ cap) { for (int assignedPeople = 0 ; assignedPeople <= allMask ; ++ assignedPeople) { // If there are no ways to assign caps // for this state, continue if (dp [cap - 1 , assignedPeople] == 0) { continue ; } // Case 1: Skip the current cap dp [cap , assignedPeople] += dp [cap - 1 , assignedPeople]; // Case 2: Assign current cap to each person who can wear it foreach (int person in capToPeople [cap]) { // If the person hasn't been assigned a cap yet if ((assignedPeople & (1 << person)) == 0) { dp [cap , assignedPeople | (1 << person)] += dp [cap - 1 , assignedPeople]; } } } } // The result will be in dp[100, allMask], // as we have considered all caps and assigned all people return dp [100 , allMask]; } static void Main () { List < List < int >> caps = new List < List < int >> { new List < int > { 1 , 2 , 3 }, new List < int > { 1 , 2 }, new List < int > { 3 , 4 }, new List < int > { 4 , 5 } }; Console . WriteLine (NumberWays (caps)); } } JavaScript // JavaScript code to calculate the number of ways // to assign caps to people using Tabulation function numberWays (caps) { const n = caps . length ; const allMask = (1 << n) - 1 ; // Create adjacency list for cap-to-people distribution const capToPeople = Array . from ({ length : 101 }, () => []); // Fill the cap-to-people adjacency list for (let i = 0 ; i < n ; ++ i) { for (let cap of caps [i]) { capToPeople [cap]. push (i); } } // DP table: dp[cap][assignedPeople] // stores the number of ways to assign caps // for the first 'cap' caps with 'assignedPeople' bitmask const dp = Array . from ({ length : 102 }, () => Array (1 << n). fill (0)); // Base case: With 0 caps, no people // assigned, there's 1 way (do nothing) dp [0][0] = 1 ; // Fill the DP table for (let cap = 1 ; cap <= 100 ; ++ cap) { for (let assignedPeople = 0 ; assignedPeople <= allMask ; ++ assignedPeople) { // If there are no ways to assign caps // for this state, continue if (dp [cap - 1][assignedPeople] === 0) { continue ; } // Case 1: Skip the current cap dp [cap][assignedPeople] += dp [cap - 1][assignedPeople]; // Case 2: Assign current cap to each person who can wear it for (let person of capToPeople [cap]) { // If the person hasn't been assigned a cap yet if ((assignedPeople & (1 << person)) === 0) { dp [cap][assignedPeople | (1 << person)] += dp [cap - 1][assignedPeople]; } } } } // The result will be in dp[100][allMask], // as we have considered all caps and assigned all people return dp [100][allMask]; } const caps = [[1 , 2 , 3], [1 , 2], [3 , 4], [4 , 5]]; console . log (numberWays (caps)); Output 8 Comment Article Tags: Article Tags: Dynamic Programming DSA