Coin Change - Count Ways to Make Sum - GeeksforGeeks

Source: https://www.geeksforgeeks.org/coin-change-dp-7/

Coin Change - Count Ways to Make Sum Last Updated : 24 Jan, 2026 Given an integer array coins[ ] representing different denominations of currency and an integer sum. We need to find the number of ways we can make sum by using different combinations from coins[ ]. Note: Assume that we have an infinite supply of each type of coin. Therefore, we can use any coin as many times as we want. Examples: Input: sum = 4, coins[] = [1, 2, 3] Output: 4 Explanation: There are four solutions: [1, 1, 1, 1], [1, 1, 2], [2, 2] and [1, 3] Input: sum = 10, coins[] = [2, 5, 3, 6] Output: 5 Explanation: There are five solutions: [2, 2, 2, 2, 2], [2, 2, 3, 3], [2, 2, 6], [2, 3, 5] and [5, 5] Try it on GfG Practice Table of Content [Naive Approach] Using Recursion - O(2^sum) time and O(sum) space [Better Approach 1] Using Top-Down DP (Memoization) - O(sum*n) time and O(sum*n) space [Better Approach 2] Using Bottom-Up DP (Tabulation) – O(sum*n) time and O(sum*n) space [Expected Approach] Using Space Optimized DP – O(sum*n) time and O(sum) space [Naive Approach] Using Recursion - O(2^sum) time and O(sum) space To solve this problem initially, we use recursion because at every step we have a choice: either we include the current coin or we do not include it. For each coin, there are two possibilities: Include the current coin If we pick the current coin, then its value reduces the remaining target sum. Because coins are available in infinite supply, we can include the same coin again. So the recursive call becomes: count(coins, n, sum - coins[n-1]) Exclude the current coin If we decide not to pick the current coin, then we move to the previous coin while keeping the target sum unchanged. So the recursive call becomes: count(coins, n-1, sum) Since we are looking for all different combinations that form the given sum, the final answer will be the sum of both possibilities (include + exclude). C++ //Driver Code Starts #include <iostream> #include <vector> using namespace std ; //Driver Code Ends int countRecur (vector < int >& coins , int n , int sum) { // If sum is 0 then there is 1 solution if (sum == 0) return 1 ; if (sum < 0 || n == 0) return 0 ; // count is sum of solutions // (i)including coins[n-1] (ii) excluding coins[n-1] return countRecur (coins , n , sum - coins [n - 1]) + countRecur (coins , n - 1 , sum); } int count (vector < int > & coins , int sum) { return countRecur (coins , coins . size (), sum); } //Driver Code Starts int main () { vector < int > coins = { 1 , 2 , 3 }; int sum = 5 ; cout << count (coins , sum); return 0 ; } //Driver Code Ends Java //Driver Code Starts import java.util.Arrays ; public class GFG { //Driver Code Ends static int countRecur (int [] coins , int n , int sum) { // If sum is 0 then there is 1 solution if (sum == 0) return 1 ; if (sum < 0 || n == 0) return 0 ; // count is sum of solutions // (i) including coins[n-1] (ii) excluding coins[n-1] return countRecur (coins , n , sum - coins [n - 1]) + countRecur (coins , n - 1 , sum); } static int count (int [] coins , int sum) { return countRecur (coins , coins . length , sum); } //Driver Code Starts public static void main (String [] args) { int [] coins = { 1 , 2 , 3 }; int sum = 5 ; System . out . println (count (coins , sum)); } } //Driver Code Ends Python def countRecur (coins , n , sum): # If sum is 0 then there is 1 solution if sum == 0 : return 1 if sum < 0 or n == 0 : return 0 # count is sum of solutions # (i) including coins[n-1] (ii) excluding coins[n-1] return countRecur (coins , n , sum - coins [n - 1]) + \ countRecur (coins , n - 1 , sum) def count (coins , sum): return countRecur (coins , len (coins), sum) #Driver Code Starts if __name__ == "__main__" : coins = [1 , 2 , 3] sum = 5 print (count (coins , sum)) #Driver Code Ends C# //Driver Code Starts using System ; class GFG { //Driver Code Ends static int countRecur (int [] coins , int n , int sum) { // If sum is 0 then there is 1 solution if (sum == 0) return 1 ; if (sum < 0 || n == 0) return 0 ; // count is sum of solutions // (i) including coins[n-1] (ii) excluding coins[n-1] return countRecur (coins , n , sum - coins [n - 1]) + countRecur (coins , n - 1 , sum); } static int count (int [] coins , int sum) { return countRecur (coins , coins . Length , sum); } //Driver Code Starts static void Main () { int [] coins = { 1 , 2 , 3 }; int sum = 5 ; Console . WriteLine (count (coins , sum)); } } //Driver Code Ends JavaScript function countRecur (coins , n , sum) { // If sum is 0 then there is 1 solution if (sum === 0) return 1 ; if (sum < 0 || n === 0) return 0 ; // count is sum of solutions // (i) including coins[n-1] (ii) excluding coins[n-1] return countRecur (coins , n , sum - coins [n - 1]) + countRecur (coins , n - 1 , sum); } function count (coins , sum) { return countRecur (coins , coins . length , sum); } //Driver Code Starts //Driver Code let coins = [1 , 2 , 3]; let sum = 5 ; console . log (count (coins , sum)); //Driver Code Ends Output 5 [Expected Approach 1] Using Top-Down DP (Memoization) - O(sum*n) time and O(sum*n) space In the previous recursive approach, we observed that many subproblems are solved repeatedly. This repetition increases time complexity. To handle this, we use Memoization. We create a DP table of size n × (sum + 1), because the result of the recursion depends on two changing parameters: the number of coins considered and the remaining target sum So whenever we compute a subproblem count(i, sum), we store the result in the DP table. Next time when the same subproblem appears, instead of computing it again, we directly fetch it from the DP table, which saves a lot of time. C++ //Driver Code Starts #include <iostream> #include <vector> using namespace std ; //Driver Code Ends int countRecur (vector < int >& coins , int n , int sum , vector < vector < int >> & dp) { // If sum is 0 then there is 1 solution if (sum == 0) return 1 ; if (sum < 0 || n == 0) return 0 ; // If the subproblem is previously calculated then // simply return the result if (dp [n -1][sum] != -1) return dp [n -1][sum]; // count is sum of solutions (i) // including coins[n-1] (ii) excluding coins[n-1] return dp [n -1][sum] = countRecur (coins , n , sum - coins [n -1], dp) + countRecur (coins , n - 1 , sum , dp); } int count (vector < int > & coins , int sum) { vector < vector < int >> dp (coins . size (), vector < int > (sum + 1 , -1)); return countRecur (coins , coins . size (), sum , dp); } //Driver Code Starts int main () { vector < int > coins = { 1 , 2 , 3 }; int sum = 5 ; cout << count (coins , sum); return 0 ; } //Driver Code Ends Java //Driver Code Starts import java.util.Arrays ; class GFG { //Driver Code Ends static int countRecur (int [] coins , int n , int sum , int [][] dp) { // If sum is 0 then there is 1 solution if (sum == 0) return 1 ; if (sum < 0 || n == 0) return 0 ; // If the subproblem is previously calculated then // simply return the result if (dp [n - 1][sum] != - 1) return dp [n - 1][sum] ; // count is sum of solutions (i) // including coins[n-1] (ii) excluding coins[n-1] return dp [n - 1][sum] = countRecur (coins , n , sum - coins [n - 1] , dp) + countRecur (coins , n - 1 , sum , dp); } static int count (int [] coins , int sum) { int [][] dp = new int [coins . length][sum + 1] ; for (int [] row : dp) Arrays . fill (row , - 1); return countRecur (coins , coins . length , sum , dp); } //Driver Code Starts public static void main (String [] args) { int [] coins = { 1 , 2 , 3 }; int sum = 5 ; System . out . println (count (coins , sum)); } } //Driver Code Ends Python def countRecur (coins , n , sum , dp): # If sum is 0 then there is 1 solution if sum == 0 : return 1 if sum < 0 or n == 0 : return 0 # If the subproblem is previously calculated then # simply return the result if dp [n - 1][sum] != - 1 : return dp [n - 1][sum] # count is sum of solutions (i) # including coins[n-1] (ii) excluding coins[n-1] dp [n - 1][sum] = (countRecur (coins , n , sum - coins [n - 1], dp) + countRecur (coins , n - 1 , sum , dp)) return dp [n - 1][sum] def count (coins , sum): dp = [[- 1 for _ in range (sum + 1)] for _ in range (len (coins))] return countRecur (coins , len (coins), sum , dp) #Driver Code Starts if __name__ == "__main__" : coins = [1 , 2 , 3] sum = 5 print (count (coins , sum)) #Driver Code Ends C# //Driver Code Starts using System ; class GFG { //Driver Code Ends static int countRecur (int [] coins , int n , int sum , int [,] dp) { // If sum is 0 then there is 1 solution if (sum == 0) return 1 ; if (sum < 0 || n == 0) return 0 ; // If the subproblem is previously calculated then // simply return the result if (dp [n - 1 , sum] != - 1) return dp [n - 1 , sum]; // count is sum of solutions (i) // including coins[n-1] (ii) excluding coins[n-1] dp [n - 1 , sum] = countRecur (coins , n , sum - coins [n - 1], dp) + countRecur (coins , n - 1 , sum , dp); return dp [n - 1 , sum]; } static int count (int [] coins , int sum) { int [,] dp = new int [coins . Length , sum + 1]; for (int i = 0 ; i < coins . Length ; i ++) for (int j = 0 ; j <= sum ; j ++) dp [i , j] = - 1 ; return countRecur (coins , coins . Length , sum , dp); } //Driver Code Starts static void Main () { int [] coins = { 1 , 2 , 3 }; int sum = 5 ; Console . WriteLine (count (coins , sum)); } } //Driver Code Ends JavaScript function countRecur (coins , n , sum , dp) { // If sum is 0 then there is 1 solution if (sum === 0) return 1 ; if (sum < 0 || n === 0) return 0 ; // If the subproblem is previously calculated then // simply return the result if (dp [n - 1][sum] !== - 1) return dp [n - 1][sum]; // count is sum of solutions (i) // including coins[n-1] (ii) excluding coins[n-1] dp [n - 1][sum] = countRecur (coins , n , sum - coins [n - 1], dp) + countRecur (coins , n - 1 , sum , dp); return dp [n - 1][sum]; } function count (coins , sum) { const dp = Array . from ({ length : coins . length }, () => Array (sum + 1). fill (- 1)); return countRecur (coins , coins . length , sum , dp); } //Driver Code Starts //Driver Code const coins = [1 , 2 , 3]; const sum = 5 ;

console . log (count (coins , sum)); //Driver Code Ends Output 5 [Better Approach 2] Using Bottom-Up DP (Tabulation) – O(sum*n) time and O(sum*n) space In the memoization approach we solved each subproblem top-down using recursion, but in the tabulation approach we build the solution in a bottom-up manner by filling a DP table iteratively. We first define the base cases for the DP table. dp[0][0] = 1, meaning if we have 0 coins and target sum is 0, there is exactly one way — choose nothing. For dp[0][j] where j > 0, the value is 0 because with zero coins we cannot make a positive sum. For dp[i][0], the value is 1 for all i because there is only one way to make sum 0 — by not selecting any coin. After the base initialization, we fill the table iteratively using the same idea. C++ //Driver Code Starts #include <iostream> using namespace std ; //Driver Code Ends int count (vector < int >& coins , int sum) { int n = coins . size (); vector < vector < int > > dp (n + 1 , vector < int > (sum + 1 , 0)); dp [0][0] = 1 ; for (int i = 1 ; i <= n ; i ++) { for (int j = 0 ; j <= sum ; j ++) { // Add the number of ways to make change without // using the current coin, dp [i][j] += dp [i - 1][j]; if ((j - coins [i - 1]) >= 0) { // Add the number of ways to make change // using the current coin dp [i][j] += dp [i][j - coins [i - 1]]; } } } return dp [n][sum]; } //Driver Code Starts int main () { vector < int > coins = { 1 , 2 , 3 }; int sum = 5 ; cout << count (coins , sum); return 0 ; } //Driver Code Ends Java //Driver Code Starts import java.util.Arrays ; public class GFG { //Driver Code Ends static int count (int [] coins , int sum) { int n = coins . length ; int [][] dp = new int [n + 1][sum + 1] ; dp [0][0] = 1 ; for (int i = 1 ; i <= n ; i ++) { for (int j = 0 ; j <= sum ; j ++) { // Add the number of ways to make change without // using the current coin, dp [i][j] += dp [i - 1][j] ; if (j - coins [i - 1] >= 0) { // Add the number of ways to make change // using the current coin dp [i][j] += dp [i][j - coins [i - 1]] ; } } } return dp [n][sum] ; } //Driver Code Starts public static void main (String [] args) { int [] coins = { 1 , 2 , 3 }; int sum = 5 ; System . out . println (count (coins , sum)); } } //Driver Code Ends Python def count (coins , sum): n = len (coins) # Initialize DP table dp = [[0] * (sum + 1) for _ in range (n + 1)] dp [0][0] = 1 for i in range (1 , n + 1): for j in range (sum + 1): # Add the number of ways to make change without # using the current coin, dp [i][j] += dp [i - 1][j] if j - coins [i - 1] >= 0 : # Add the number of ways to make change # using the current coin dp [i][j] += dp [i][j - coins [i - 1]] return dp [n][sum] #Driver Code Starts if __name__ == "__main__" : coins = [1 , 2 , 3] sum = 5 print (count (coins , sum)) #Driver Code Ends C# //Driver Code Starts using System ; class GFG { //Driver Code Ends static int count (int [] coins , int sum) { int n = coins . Length ; int [,] dp = new int [n + 1 , sum + 1]; dp [0 , 0] = 1 ; for (int i = 1 ; i <= n ; i ++) { for (int j = 0 ; j <= sum ; j ++) { // Add the number of ways to make change without // using the current coin, dp [i , j] += dp [i - 1 , j]; if (j - coins [i - 1] >= 0) { // Add the number of ways to make change // using the current coin dp [i , j] += dp [i , j - coins [i - 1]]; } } } return dp [n , sum]; } //Driver Code Starts static void Main () { int [] coins = { 1 , 2 , 3 }; int sum = 5 ; Console . WriteLine (count (coins , sum)); } } //Driver Code Ends JavaScript function count (coins , sum) { let n = coins . length ; let dp = Array . from ({ length : n + 1 }, () => Array (sum + 1). fill (0)); dp [0][0] = 1 ; for (let i = 1 ; i <= n ; i ++) { for (let j = 0 ; j <= sum ; j ++) { // Add the number of ways to make change without // using the current coin, dp [i][j] += dp [i - 1][j]; if (j - coins [i - 1] >= 0) { // Add the number of ways to make change // using the current coin dp [i][j] += dp [i][j - coins [i - 1]]; } } } return dp [n][sum]; } //Driver Code Starts // Driver Code let coins = [1 , 2 , 3]; let sum = 5 ; console . log (count (coins , sum)); //Driver Code Ends Output 5 [Expected Approach] Using Space Optimized DP – O(sum*n) time and O(sum) space In previous approach of dynamic programming we have derive the relation between states as given below: if (sum-coins[i]) is greater than 0, then dp[i][sum] = dp[i][sum-coins[i]] + dp[i+1][sum]. else dp[i][sum] = dp[i+1][sum]. If we observe that for calculating current dp[i][sum] state we only need previous row dp[i-1][sum] or current row dp[i][sum-coins[i]]. There is no need to store all the previous states just one previous state is used to compute result. C++ //Driver Code Starts #include <iostream> #include <vector> using namespace std ; //Driver Code Ends int count (vector < int > & coins , int sum) { int n = coins . size (); // dp[i] will be storing the number of solutions for // value i. vector < int > dp (sum + 1); dp [0] = 1 ; // Pick all coins one by one and update the table[] // values after the index greater than or equal to the // value of the picked coin for (int i = 0 ; i < n ; i ++) for (int j = coins [i]; j <= sum ; j ++) dp [j] += dp [j - coins [i]]; return dp [sum]; } //Driver Code Starts int main () { vector < int > coins = { 1 , 2 , 3 }; int sum = 5 ; cout << count (coins , sum); return 0 ; } //Driver Code Ends Java //Driver Code Starts public class GFG { //Driver Code Ends static int count (int [] coins , int sum) { int n = coins . length ; // dp[i] will be storing the number of solutions for // value i. int [] dp = new int [sum + 1] ; dp [0] = 1 ; // Pick all coins one by one and update the table[] // values after the index greater than or equal to the // value of the picked coin for (int i = 0 ; i < n ; i ++) for (int j = coins [i] ; j <= sum ; j ++) dp [j] += dp [j - coins [i]] ; return dp [sum] ; } //Driver Code Starts public static void main (String [] args) { int [] coins = { 1 , 2 , 3 }; int sum = 5 ; System . out . println (count (coins , sum

)); } } //Driver Code Ends Python def count (coins , sum): n = len (coins) # dp[i] will be storing the number of solutions for # value i. dp = [0] * (sum + 1) dp [0] = 1 # Pick all coins one by one and update the table[] # values after the index greater than or equal to the # value of the picked coin for i in range (n): for j in range (coins [i], sum + 1): dp [j] += dp [j - coins [i]] return dp [sum] // Driver Code Starts if __name__ == "__main__" : coins = [1 , 2 , 3] sum = 5 print (count (coins , sum)) // Driver Code Ends C# //Driver Code Starts using System ; class GFG { //Driver Code Ends static int count (int [] coins , int sum) { int n = coins . Length ; // dp[i] will be storing the number of solutions for // value i. int [] dp = new int [sum + 1]; dp [0] = 1 ; // Pick all coins one by one and update the table[] // values after the index greater than or equal to the // value of the picked coin for (int i = 0 ; i < n ; i ++) for (int j = coins [i]; j <= sum ; j ++) dp [j] += dp [j - coins [i]]; return dp [sum]; } //Driver Code Starts static void Main () { int [] coins = { 1 , 2 , 3 }; int sum = 5 ; Console . WriteLine (count (coins , sum)); } } //Driver Code Ends JavaScript function count (coins , sum) { let n = coins . length ; // dp[i] will be storing the number of solutions for // value i. let dp = Array (sum + 1). fill (0); dp [0] = 1 ; // Pick all coins one by one and update the table[] // values after the index greater than or equal to the // value of the picked coin for (let i = 0 ; i < n ; i ++) for (let j = coins [i]; j <= sum ; j ++) dp [j] += dp [j - coins [i]]; return dp [sum]; } //Driver Code Starts // Driver Code let coins = [1 , 2 , 3]; let sum = 5 ; console . log (count (coins , sum)); //Driver Code Ends Output 5 Related articles: Coin Change – Minimum Coins to Make Sum Understanding The Coin Change Problem With Dynamic Programming Comment Article Tags: Article Tags: Dynamic Programming Greedy Mathematical DSA Microsoft Morgan Stanley Samsung Snapdeal Paytm Accolite + 6 More