# Travelling Salesman Problem - GeeksforGeeks

**Source:** https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/

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 Travelling Salesman Problem Last Updated : 2 Dec, 2025 Given a 2d matrix cost[][] of dimensions n * n where cost[i][j] denotes the cost of moving from city i to city j. Find the minimum cost to complete a tour from city 0 (0-based index) to all other cities such that we visit each city exactly once and then at the end come back to city 0. Examples: Input: cost[][] = [[0, 111], [112, 0]] Output: 223 Explanation: We can visit 0->1->0 and cost = 111 + 112 = 223. Input: cost[][] = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]] Output: 80 Explanation: We can visit 0->2->3->1->0 and cost = 15 + 30 + 25+ 10= 80. Try it on GfG Practice Table of Content [Naive Approach] Using DFS - O(n!) Time and O(n) Space [Better Approach] Using Recursion - O(n!) Time and O(n) Space [Expected Approach 1] Using Top-Down DP (Memoization) - O((n^2)*(2^n)) Time and O(n*(2^n)) Space [Expected Approach 2] Using Bottom-Up DP (Tabulation) - O((n^2)*(2^n)) Time and O(n*(2^n)) Space [Naive Approach] Using DFS - O(n!) Time and O(n) Space The idea is to consider all possible routes that start at city 0, visit every other city exactly once, and finally return to city 0. For each route, we keep track of which cities have been visited and the current city, then accumulate the travel cost as we move to the next city. After evaluating every valid route, we choose the one with the smallest total cost. C++ //Driver Code Starts #include <iostream> #include <vector> #include <algorithm> using namespace std ; //Driver Code Ends int dfs ( vector < vector < int >>& cost , vector < bool >& vis , int last , int cnt ) { int n = cost . size (); // If all visited -> return cost back to start (0) if ( cnt == n ) return cost [ last ][ 0 ]; int minCost = ( int ) 1e9 ; for ( int city = 1 ; city < n ; city ++ ) { if ( ! vis [ city ]) { // mark the city as visited and explore // all possible paths from there vis [ city ] = true ; minCost = min ( minCost , cost [ last ][ city ] + dfs ( cost , vis , city , cnt + 1 )); // backtrack vis [ city ] = false ; } } // return min cost among all possible paths return minCost ; } int tsp ( vector < vector < int >>& cost ) { int n = cost . size (); // to make sure that a city is visited exactly once vector < bool > vis ( n , false ); vis [ 0 ] = true ; return dfs ( cost , vis , 0 , 1 ); } //Driver Code Starts int main () { vector < vector < int >> cost = { { 0 , 10 , 15 , 20 }, { 10 , 0 , 35 , 25 }, { 15 , 35 , 0 , 30 }, { 20 , 25 , 30 , 0 } }; int res = tsp ( cost ); cout << res ; } //Driver Code Ends Java //Driver Code Starts class GFG { //Driver Code Ends static int dfs ( int [][] cost , boolean [] vis , int last , int cnt ) { int n = cost . length ; // If all visited -> return cost back to start (0) if ( cnt == n ) return cost [ last ][ 0 ] ; int minCost = ( int ) 1e9 ; for ( int city = 1 ; city < n ; city ++ ) { if ( ! vis [ city ] ) { // mark the city as visited and explore // all possible paths from there vis [ city ] = true ; minCost = Math . min ( minCost , cost [ last ][ city ]+ dfs ( cost , vis , city , cnt + 1 )); // backtrack vis [ city ] = false ; } } // return min cost among all possible paths return minCost ; } static int tsp ( int [][] cost ) { int n = cost . length ; // to make sure that a city is visited exactly once boolean [] vis = new boolean [ n ] ; vis [ 0 ] = true ; return dfs ( cost , vis , 0 , 1 ); //Driver Code Starts } public static void main ( String [] args ) { int [][] cost = {{ 0 , 10 , 15 , 20 }, { 10 , 0 , 35 , 25 }, { 15 , 35 , 0 , 30 }, { 20 , 25 , 30 , 0 }}; int res = tsp ( cost ); System . out . println ( res ); } } //Driver Code Ends Python def dfs ( cost , vis , last , cnt ): n = len ( cost ) # If all visited -> return cost back to start (0) if cnt == n : return cost [ last ][ 0 ] minCost = int ( 1e9 ) for city in range ( 1 , n ): if not vis [ city ]: # mark the city as visited and explore # all possible paths from there vis [ city ] = True minCost = min ( minCost , cost [ last ][ city ] + dfs ( cost , vis , city , cnt + 1 )) # backtrack vis [ city ] = False # return min cost among all possible paths return minCost def tsp ( cost ): n = len ( cost ) # to make sure that a city is visited exactly once vis = [ False ] * n vis [ 0 ] = True return dfs ( cost , vis , 0 , 1 ) #Driver Code Starts if __name__ == "__main__" : cost = [ [ 0 , 10 , 15 , 20 ], [ 10 , 0 , 35 , 25 ], [ 15 , 35 , 0 , 30 ], [ 20 , 25 , 30 , 0 ] ] result = tsp ( cost ) print ( result ) #Driver Code Ends C# //Driver Code

Starts using System ; class GFG { //Driver Code Ends static int dfs ( int [,] cost , bool [] vis , int last , int cnt ) { int n = cost . GetLength ( 0 ); // If all visited -> return cost back to start (0) if ( cnt == n ) return cost [ last , 0 ]; int minCost = ( int ) 1 e9 ; for ( int city = 1 ; city < n ; city ++ ) { if ( ! vis [ city ]) { // mark the city as visited and explore // all possible paths from there vis [ city ] = true ; minCost = Math . Min ( minCost , cost [ last , city ] + dfs ( cost , vis , city , cnt + 1 )); // backtrack vis [ city ] = false ; } } // return min cost among all possible paths return minCost ; } static int tsp ( int [,] cost ) { int n = cost . GetLength ( 0 ); // to make sure that a city is visited exactly once bool [] vis = new bool [ n ]; vis [ 0 ] = true ; return dfs ( cost , vis , 0 , 1 ); } //Driver Code Starts public static void Main () { int [,] cost = { { 0 , 10 , 15 , 20 }, { 10 , 0 , 35 , 25 }, { 15 , 35 , 0 , 30 }, { 20 , 25 , 30 , 0 } }; int res = tsp ( cost ); Console . WriteLine ( res ); } } //Driver Code Ends JavaScript function dfs ( cost , vis , last , cnt ) { const n = cost . length ; // If all visited -> return cost back to start (0) if ( cnt === n ) return cost [ last ][ 0 ]; let minCost = 1e9 ; for ( let city = 1 ; city < n ; city ++ ) { if ( ! vis [ city ]) { // mark the city as visited and explore // all possible paths from there vis [ city ] = true ; minCost = Math . min ( minCost , cost [ last ][ city ] + dfs ( cost , vis , city , cnt + 1 )); // backtrack vis [ city ] = false ; } } // return min cost among all possible paths return minCost ; } function tsp ( cost ) { const n = cost . length ; // to make sure that a city is visited exactly once const vis = Array ( n ). fill ( false ); vis [ 0 ] = true ; return dfs ( cost , vis , 0 , 1 ); } //Driver Code Starts // Driver code const cost = [ [ 0 , 10 , 15 , 20 ], [ 10 , 0 , 35 , 25 ], [ 15 , 35 , 0 , 30 ], [ 20 , 25 , 30 , 0 ] ]; const res = tsp ( cost ); console . log ( res ); //Driver Code Ends Output 80 Bitmask Representation for Tracking Visited Cities :- To keep track of which cities have been visited, we use an integer called mask , where each bit represents a city. If the i-th bit is set, it means the i-th city has been visited; if it is not set, the city is still unvisited. The maximum possible value of the mask is 2 n-1 + 2 n-2 + ... 2 1 +2 0 = 2 n -1, when all cities have been visited. This representation ensures that each city is visited exactly once, and a path is considered complete when all bits are set and we return to city 0. [Better Approach] Using Recursion - O(n!) Time and O(n) Space The idea is to try every possible way of visiting all cities exactly once and returning to city 0. We use a bitmask to keep track of which cities have been visited so far, and from the current city, we recursively try going to any city that is still unvisited. By exploring all valid routes in this way and adding up their travel costs, we will find the minimum cost among them. C++ //Driver Code Starts #include <iostream> #include <vector> using namespace std ; //Driver Code Ends int totalCost ( int mask , int pos , vector < vector < int >>& cost ) { int n = cost . size (); // Base case: if all cities are visited, return the // cost to return to the starting city (0) if ( mask == ( 1 << n ) - 1 ) { return cost [ pos ][ 0 ]; } int ans = INT_MAX ; // Try visiting every city that // has not been visited yet for ( int i = 0 ; i < n ; i ++ ) { if (( mask & ( 1 << i )) == 0 ) { // If city i is not visited, visit it and // update the mask ans = min ( ans , cost [ pos ][ i ] + totalCost ( mask | ( 1 << i ), i , cost )); } } return ans ; } int tsp ( vector < vector < int >>& cost ) { // Start from city 0, and only city 0 is // visited initially (mask = 1) int mask = 1 , pos = 0 ; return totalCost ( mask , pos , cost ); } //Driver Code Starts int main () { vector < vector < int >> cost = { { 0 , 10 , 15 , 20 }, { 10 , 0 , 35 , 25 }, { 15 , 35 , 0 , 30 }, { 20 , 25 , 30 , 0 } }; int res = tsp ( cost ); cout << res ; } //Driver Code Ends Java //Driver Code Starts class GFG { //Driver Code Ends static int totalCost ( int mask , int pos , int [][] cost ) { int n = cost . length ; // Base case: if all cities are visited, return the // cost to return to the starting city (0) if ( mask == ( 1 << n ) - 1 ) { return cost [ pos ][ 0 ] ; } int ans = Integer . MAX_VALUE ; // Try visiting every city that // has not been visited yet for ( int i = 0 ; i < n ; i ++ ) { if (( mask & ( 1 << i )) == 0 ) { // If city i is not visited, visit it and // update the mask ans = Math . min ( ans , cost [ pos ][ i ] + totalCost ( mask | ( 1 << i ), i , cost )); } } return ans ; } static int tsp ( int [][] cost ) { // Start from city 0, and only city 0 is // visited initially (mask = 1) int mask = 1 , pos = 0 ; return totalCost ( mask , pos , cost ); } //Driver Code Starts public static void main ( String [] args ) { int [][] cost = {{ 0 , 10 , 15 , 20 }, { 10 , 0 , 35 , 25 }, { 15 , 35 , 0 , 30 }, { 20 , 25 , 30 , 0 }}; int res = tsp ( cost ); System . out . println ( res ); } } //Driver Code Ends Python def totalCost ( mask , pos , cost ): n = len ( cost ) # Base case: if all cities are visited, return the # cost to return to the starting city (0) if mask == ( 1 << n ) - 1 : return cost [ pos ][ 0 ] ans = float ( 'inf' ) # Try visiting every city that # has not been visited yet for i in range ( n ): if ( mask & ( 1 << i )) == 0 : # If city i is not visited, visit it and # update the mask ans = min ( ans , cost [ pos ][ i ] + totalCost ( mask | ( 1 << i ), i , cost )) return ans def tsp ( cost ): mask = 1 pos = 0 # Start from city 0, and only city 0 is # visited initially (mask = 1) return totalCost ( mask , pos , cost ) #Driver Code Starts if __name__ == "__main__" : cost = [ [ 0 , 10 , 15 , 20 ], [ 10 , 0 , 35 , 25 ], [ 15 , 35 , 0 , 30 ], [ 20 , 25 , 30 , 0 ] ] result = tsp ( cost ) print ( result ) #Driver Code Ends C# //Driver Code Starts using System ; class GFG { //Driver Code Ends static int totalCost ( int mask , int pos , int [,] cost ) { int n = cost . GetLength ( 0 ); // Base case: if all cities are visited, return the // cost to return to the starting city (0) if ( mask == ( 1 << n ) - 1 ) { return cost [ pos , 0 ]; } int ans = int . MaxValue ; // Try visiting every city that // has not been visited yet for ( int i = 0 ; i < n ; i ++ ) { if (( mask & ( 1 << i )) == 0 ) { // If city i is

not visited, visit it and // update the mask ans = Math . Min ( ans , cost [ pos , i ] + totalCost ( mask | ( 1 << i ), i , cost )); } } return ans ; } static int tsp ( int [,] cost ) { int mask = 1 , pos = 0 ; // Start from city 0, and only city 0 is // visited initially (mask = 1) return totalCost ( mask , pos , cost ); } //Driver Code Starts public static void Main () { int [,] cost = { { 0 , 10 , 15 , 20 }, { 10 , 0 , 35 , 25 }, { 15 , 35 , 0 , 30 }, { 20 , 25 , 30 , 0 } }; int res = tsp ( cost ); Console . WriteLine ( res ); } } //Driver Code Ends JavaScript function totalCost ( mask , pos , cost ) { const n = cost . length ; // Base case: if all cities are visited, return the // cost to return to the starting city (0) if ( mask === ( 1 << n ) - 1 ) { return cost [ pos ][ 0 ]; } let ans = Number . MAX_SAFE_INTEGER ; // Try visiting every city that // has not been visited yet for ( let i = 0 ; i < n ; i ++ ) { if (( mask & ( 1 << i )) === 0 ) { // If city i is not visited, visit it and // update the mask ans = Math . min ( ans , cost [ pos ][ i ] + totalCost ( mask | ( 1 << i ), i , cost )); } } return ans ; } function tsp ( cost ) { const mask = 1 , pos = 0 ; // Start from city 0, and only city 0 is // visited initially (mask = 1) return totalCost ( mask , pos , cost ); } //Driver Code Starts // Driver code const cost = [ [ 0 , 10 , 15 , 20 ], [ 10 , 0 , 35 , 25 ], [ 15 , 35 , 0 , 30 ], [ 20 , 25 , 30 , 0 ] ]; const res = tsp ( cost ); console . log ( res ); //Driver Code Ends Output 80 [Expected Approach 1] Using Top-Down DP (Memoization ) - O(n 2 *2 n ) Time and O(n*2 n ) Space In the above recursive solution, we can see that the same subproblem is computed multiple times. The subproblem is fully defined by two things: the current city the set of cities already visited (represented by the mask) Different routes can lead to the same combination of these two values. For example, consider a state where the mask shows that cities 0, 1, 2, and 4 have been visited, and the current city is 4. This state might be reached by: 0 -> 1 -> 2 -> 4 0 -> 2 -> 1 -> 4 Even though the arrival order differs, once we are at city 4 with exactly these cities visited, the remaining task is identical. The set of unvisited cities is the same, and the next decisions do not depend on the earlier path. To optimize this, we store the answer for each (current city, mask) state in a DP table. If the same state appears again, we directly use the stored value instead of recomputing it. C++ //Driver Code Starts #include <iostream> #include <vector> #include <algorithm> #include <climits> using namespace std ; //Driver Code Ends int totalCost ( int mask , int curr , vector < vector < int >>& cost , vector < vector < int >>& dp ) { int n = cost . size (); // Base case: if all cities are visited, return the // cost to return to the starting city (0) if ( mask == ( 1 << n ) - 1 ) { return cost [ curr ][ 0 ]; } // If the value has already been computed, return it // from the dp table if ( dp [ curr ][ mask ] != -1 ) { return dp [ curr ][ mask ]; } int ans = INT_MAX ; // Try visiting every city that has not been visited yet for ( int i = 0 ; i < n ; i ++ ) { if (( mask & ( 1 << i )) == 0 ) { // If city i is not visited // Visit city i and update the mask ans = min ( ans , cost [ curr ][ i ] + totalCost ( mask | ( 1 << i ), i , cost , dp )); } } return dp [ curr ][ mask ] = ans ; } int tsp ( vector < vector < int >>& cost ) { int n = cost . size (); vector < vector < int >> dp ( n , vector < int > ( 1 << n , -1 )); // Start from city 0, with only city 0 // visited initially (mask = 1) int mask = 1 , curr = 0 ; return totalCost ( mask , curr , cost , dp ); } //Driver Code Starts int main () { vector < vector < int >> cost = { { 0 , 10 , 15 , 20 }, { 10 , 0 , 35 , 25 }, { 15 , 35 , 0 , 30 }, { 20 , 25 , 30 , 0 } }; int res = tsp ( cost ); cout << res ; } //Driver Code Ends Java //Driver Code Starts import java.util.Arrays ; class GFG { //Driver Code Ends static int totalCost ( int mask , int curr , int [][] cost , int [][] dp ) { int n = cost . length ; // Base case: if all cities are visited, return the // cost to return to the starting city (0) if ( mask == ( 1 << n ) - 1 ) { return cost [ curr ][ 0 ] ; } // If the value has already been computed, return it // from the dp table if ( dp [ curr ][ mask ] != - 1 ) { return dp [ curr ][ mask ] ; } int ans = Integer . MAX_VALUE ; // Try visiting every city that has not been visited yet for ( int i = 0 ; i < n ; i ++ ) { if (( mask & ( 1 << i )) == 0 ) { // If city i is not visited // Visit city i and update the mask ans = Math . min ( ans , cost [ curr ][ i ] + totalCost ( mask | ( 1 << i ), i , cost , dp )); } } return dp [ curr ][ mask ] = ans ; } static int tsp ( int [][] cost ) { int n = cost . length ; int [][] dp = new int [ n ][ 1 << n ] ; // Initialize the dpization table with -1 // (indicating uncomputed states) int mask = 1 , curr = 0 ; for ( int i = 0 ; i < n ; i ++ ) { Arrays . fill ( dp [ i ] , - 1 ); } // Start from city 0, with only city 0 // visited initially (mask = 1) return totalCost ( mask , curr , cost , dp ); } //Driver Code Starts public static void main ( String [] args ) { int [][] cost = {{ 0 , 10 , 15 , 20 }, { 10 , 0 , 35 , 25 }, { 15 , 35 , 0 , 30 }, { 20 , 25 , 30 , 0 }}; int res = tsp ( cost ); System . out . println ( res ); } } //Driver Code Ends Python def totalCost ( mask , curr , cost , dp ): n = len ( cost ) # Base case: if all cities are visited, return the # cost to return to the starting city (0) if mask == ( 1 << n ) - 1 : return cost [ curr ][ 0 ] # If the value has already been computed, return it # from the dp table if dp [ curr ][ mask ] != - 1 : return dp [ curr ][ mask ] ans = float ( 'inf' ) # Try visiting every city that has not been visited yet for i in range ( n ): if ( mask & ( 1 << i )) == 0 : # If city i is not visited # Visit city i and update the mask ans = min ( ans , cost [ curr ][ i ] + totalCost ( mask | ( 1 << i ), i , cost , dp )) dp [ curr ][ mask ] = ans return ans def tsp ( cost ): n = len ( cost ) dp = [[ - 1 ] * ( 1 << n ) for _ in range ( n )] # Initialize the dpization table with -1 # (indicating uncomputed states) for i in range ( n ): dp [ i ] = [ - 1 ] * ( 1 << n ) # Start from city 0, with only city 0 # visited initially (mask = 1) mask = 1 pos = 0 return totalCost ( mask , pos , cost , dp ) #Driver

Code Starts if __name__ == "__main__" : cost = [ [ 0 , 10 , 15 , 20 ], [ 10 , 0 , 35 , 25 ], [ 15 , 35 , 0 , 30 ], [ 20 , 25 , 30 , 0 ] ] res = tsp ( cost ) print ( res ) #Driver Code Ends C# //Driver Code Starts using System ; class GFG { //Driver Code Ends static int totalCost ( int mask , int curr , int [,] cost , int [,] dp ) { int n = cost . GetLength ( 0 ); // Base case: if all cities are visited, return the // cost to return to the starting city (0) if ( mask == ( 1 << n ) - 1 ) { return cost [ curr , 0 ]; } // If the value has already been computed, return it // from the dp table if ( dp [ curr , mask ] != - 1 ) { return dp [ curr , mask ]; } int ans = int . MaxValue ; // Try visiting every city that has not been visited yet for ( int i = 0 ; i < n ; i ++ ) { if (( mask & ( 1 << i )) == 0 ) { // If city i is not visited // Visit city i and update the mask ans = Math . Min ( ans , cost [ curr , i ] + totalCost ( mask | ( 1 << i ), i , cost , dp )); } } return dp [ curr , mask ] = ans ; } static int tsp ( int [,] cost ) { int n = cost . GetLength ( 0 ); // dp[curr,mask] stores the minimum cost to visit // remaining cities starting from city 'curr' int [,] dp = new int [ n , 1 << n ]; // Initialize the dp table with -1 // (indicating uncomputed states) for ( int i = 0 ; i < n ; i ++ ) { for ( int j = 0 ; j < ( 1 << n ); j ++ ) { dp [ i , j ] = - 1 ; } } int mask = 1 , curr = 0 ; // Start from city 0, with only city 0 // visited initially (mask = 1) return totalCost ( mask , curr , cost , dp ); } //Driver Code Starts public static void Main () { int [,] cost = { { 0 , 10 , 15 , 20 }, { 10 , 0 , 35 , 25 }, { 15 , 35 , 0 , 30 }, { 20 , 25 , 30 , 0 } }; int res = tsp ( cost ); Console . WriteLine ( res ); } } //Driver Code Ends JavaScript function totalCost ( mask , curr , cost , dp ) { const n = cost . length ; // Base case: if all cities are visited, return the // cost to return to the starting city (0) if ( mask === ( 1 << n ) - 1 ) { return cost [ curr ][ 0 ]; } // If the value has already been computed, return it // from the dp table if ( dp [ curr ][ mask ] !== - 1 ) { return dp [ curr ][ mask ]; } let ans = Number . MAX_SAFE_INTEGER ; // Try visiting every city that has not been visited yet for ( let i = 0 ; i < n ; i ++ ) { if (( mask & ( 1 << i )) === 0 ) { // If city i is not visited // Visit city i and update the mask ans = Math . min ( ans , cost [ curr ][ i ] + totalCost ( mask | ( 1 << i ), i , cost , dp )); } } return dp [ curr ][ mask ] = ans ; } function tsp ( cost ) { const n = cost . length ; const dp = Array . from ({ length : n }, () => Array ( 1 << n ). fill ( - 1 )); // Initialize the dpization table with -1 // (indicating uncomputed states) for ( let i = 0 ; i < n ; i ++ ) { dp [ i ]. fill ( - 1 ); } const mask = 1 , curr = 0 ; // Start from city 0, with only city 0 // visited initially (mask = 1) return totalCost ( mask , curr , cost , dp ); } //Driver Code Starts // Driver code const cost = [ [ 0 , 10 , 15 , 20 ], [ 10 , 0 , 35 , 25 ], [ 15 , 35 , 0 , 30 ], [ 20 , 25 , 30 , 0 ] ]; const res = tsp ( cost ); console . log ( res ); //Driver Code Ends Output 80 [Expected Approach 2] Using Bottom-Up DP (Tabulation ) - O(n 2 *2 n ) Time and O(n*2 n ) Space Instead of using recursion we can use iterative DP. We define dp[mask][i] as the minimum cost to visit all cities in the subset represented by mask and end at city i. Initially, only city 0 is visited, so dp[1][0] = 0. We iterate over all masks representing subsets of cities. For each mask and each city i already included in mask, we try visiting every city j not yet in mask. The new mask nxt = mask | (1 << j) represents the updated set of visited cities, and we update dp[nxt][j] as: dp[nxt][j] = min(dp[nxt][j], dp[mask][i] + cost[i][j]) This ensures that we consider the minimum cost to reach city j after visiting the cities in mask. After filling the DP table, the answer is the minimum cost to visit all cities (fullMask) and return to city 0: ans = min(ans, dp[fullMask][i] + cost[i][0]) for all i from 0 to n - 1 C++ //Driver Code Starts #include <iostream> #include <vector> #include <algorithm> #include <climits> using namespace std ; //Driver Code Ends int tsp ( vector < vector < int >>& cost ) { int n = cost . size (); if ( n <= 1 ) return n == 1 ? cost [ 0 ][ 0 ] : 0 ; // maximum cost to visit all cities const int INF = INT_MAX ; int FULL = 1 << n , fullMask = FULL - 1 ; // dp[mask][i] represents the minimum cost to visit all cities // corresponding to the set bits in 'mask', ending at city 'i' vector < vector < int >> dp ( FULL , vector < int > ( n , INF )); dp [ 1 ][ 0 ] = 0 ; // iterate over all subsets of cities for ( int mask = 1 ; mask < FULL ; mask ++ ) { for ( int i = 0 ; i < n ; i ++ ) { // skip if city i is not included in mask if ( ! ( mask & ( 1 << i ))) continue ; if ( dp [ mask ][ i ] == INF ) continue ; // try to go to every unvisited city j for ( int j = 0 ; j < n ; j ++ ) { // skip if city j is already visited if ( mask & ( 1 << j )) continue ; // cost to visit new city j from city i // such that previously visited cities // remain visited int nxt = mask | ( 1 << j ); dp [ nxt ][ j ] = min ( dp [ nxt ][ j ], dp [ mask ][ i ] + cost [ i ][ j ]); } } } int ans = INF ; for ( int i = 0 ; i < n ; i ++ ) { // if last city on path is i and // cost of path is not infinity if ( dp [ fullMask ][ i ] != INF ) // update net cost such that city 0 is visited in last ans = min ( ans , dp [ fullMask ][ i ] + cost [ i ][ 0 ]); } return ans ; } //Driver Code Starts int main () { vector < vector < int >> cost = { { 0 , 10 , 15 , 20 }, { 10 , 0 , 35 , 25 }, { 15 , 35 , 0 , 30 }, { 20 , 25 , 30 , 0 } }; int res = tsp ( cost ); cout << res ; } //Driver Code Ends Java //Driver Code Starts import java.util.Arrays ; class GFG { //Driver Code Ends static int tsp ( int [][] cost ) { int n = cost . length ; if ( n <= 1 ) return n == 1 ? cost [ 0 ][ 0 ] : 0 ; // maximum cost to visit all cities final int INF = Integer . MAX_VALUE ; int FULL = 1 << n , fullMask = FULL - 1 ; // dp[mask][i] represents the minimum cost to visit all cities // corresponding to the set bits in 'mask', ending at city 'i' int [][] dp = new int [ FULL ][ n ] ; for ( int [] row : dp ) Arrays . fill ( row , INF ); dp [ 1 ][ 0 ] = 0 ; // iterate over all subsets of cities for ( int mask = 1 ; mask < FULL ; mask ++ ) { for ( int i = 0 ; i < n ; i ++ ) { // skip if city i is not included in mask if

(( mask & ( 1 << i )) == 0 ) continue ; if ( dp [ mask ][ i ] == INF ) continue ; // try to go to every unvisited city j for ( int j = 0 ; j < n ; j ++ ) { // skip if city j is already visited if (( mask & ( 1 << j )) != 0 ) continue ; // cost to visit new city j from city i // such that previously visited cities // remain visited int nxt = mask | ( 1 << j ); dp [ nxt ][ j ] = Math . min ( dp [ nxt ][ j ] , dp [ mask ][ i ] + cost [ i ][ j ] ); } } } int ans = INF ; for ( int i = 0 ; i < n ; i ++ ) { // if last city on path is i and // cost of path is not infinity if ( dp [ fullMask ][ i ] != INF ) // update net cost such that city 0 is visited in last ans = Math . min ( ans , dp [ fullMask ][ i ] + cost [ i ][ 0 ] ); } return ans ; } //Driver Code Starts public static void main ( String [] args ) { int [][] cost = { { 0 , 10 , 15 , 20 }, { 10 , 0 , 35 , 25 }, { 15 , 35 , 0 , 30 }, { 20 , 25 , 30 , 0 } }; int res = tsp ( cost ); System . out . println ( res ); } } //Driver Code Ends Python #Driver Code Starts import sys #Driver Code Ends def tsp ( cost ): n = len ( cost ) if n <= 1 : return cost [ 0 ][ 0 ] if n == 1 else 0 # maximum cost to visit all cities INF = sys . maxsize FULL = 1 << n fullMask = FULL - 1 # dp[mask][i] represents the minimum cost to visit all cities # corresponding to the set bits in 'mask', ending at city 'i' dp = [[ INF ] * n for _ in range ( FULL )] dp [ 1 ][ 0 ] = 0 # iterate over all subsets of cities for mask in range ( 1 , FULL ): for i in range ( n ): # skip if city i is not included in mask if not ( mask & ( 1 << i )): continue if dp [ mask ][ i ] == INF : continue # try to go to every unvisited city j for j in range ( n ): # skip if city j is already visited if mask & ( 1 << j ): continue # cost to visit new city j from city i # such that previously visited cities # remain visited nxt = mask | ( 1 << j ) dp [ nxt ][ j ] = min ( dp [ nxt ][ j ], dp [ mask ][ i ] + cost [ i ][ j ]) ans = INF for i in range ( n ): # if last city on path is i and # cost of path is not infinity if dp [ fullMask ][ i ] != INF : # update net cost such that city 0 is visited in last ans = min ( ans , dp [ fullMask ][ i ] + cost [ i ][ 0 ]) return ans #Driver Code Starts if __name__ == "__main__" : cost = [ [ 0 , 10 , 15 , 20 ], [ 10 , 0 , 35 , 25 ], [ 15 , 35 , 0 , 30 ], [ 20 , 25 , 30 , 0 ] ] res = tsp ( cost ) print ( res ) #Driver Code Ends C# //Driver Code Starts using System ; class GFG { //Driver Code Ends static int tsp ( int [,] cost ) { int n = cost . GetLength ( 0 ); if ( n <= 1 ) return n == 1 ? cost [ 0 , 0 ] : 0 ; // maximum cost to visit all cities int INF = int . MaxValue ; int FULL = 1 << n , fullMask = FULL - 1 ; // dp[mask][i] represents the minimum cost to visit all cities // corresponding to the set bits in 'mask', ending at city 'i' int [,] dp = new int [ FULL , n ]; for ( int i = 0 ; i < FULL ; i ++ ) for ( int j = 0 ; j < n ; j ++ ) dp [ i , j ] = INF ; dp [ 1 , 0 ] = 0 ; // iterate over all subsets of cities for ( int mask = 1 ; mask < FULL ; mask ++ ) { for ( int i = 0 ; i < n ; i ++ ) { // skip if city i is not included in mask if (( mask & ( 1 << i )) == 0 ) continue ; if ( dp [ mask , i ] == INF ) continue ; // try to go to every unvisited city j for ( int j = 0 ; j < n ; j ++ ) { // skip if city j is already visited if (( mask & ( 1 << j )) != 0 ) continue ; // cost to visit new city j from city i // such that previously visited cities // remain visited int nxt = mask | ( 1 << j ); dp [ nxt , j ] = Math . Min ( dp [ nxt , j ], dp [ mask , i ] + cost [ i , j ]); } } } int ans = INF ; for ( int i = 0 ; i < n ; i ++ ) { // if last city on path is i and // cost of path is not infinity if ( dp [ fullMask , i ] != INF ) // update net cost such that city 0 is visited in last ans = Math . Min ( ans , dp [ fullMask , i ] + cost [ i , 0 ]); } return ans ; } //Driver Code Starts static void Main () { int [,] cost = { { 0 , 10 , 15 , 20 }, { 10 , 0 , 35 , 25 }, { 15 , 35 , 0 , 30 }, { 20 , 25 , 30 , 0 } }; Console . WriteLine ( tsp ( cost )); } } //Driver Code Ends JavaScript function tsp ( cost ) { const n = cost . length ; if ( n <= 1 ) return n === 1 ? cost [ 0 ][ 0 ] : 0 ; // maximum cost to visit all cities const INF = Number . MAX_SAFE_INTEGER ; const FULL = 1 << n ; const fullMask = FULL - 1 ; // dp[mask][i] represents the minimum cost to visit all cities // corresponding to the set bits in 'mask', ending at city 'i' const dp = Array . from ({ length : FULL }, () => Array ( n ). fill ( INF )); dp [ 1 ][ 0 ] = 0 ; // iterate over all subsets of cities for ( let mask = 1 ; mask < FULL ; mask ++ ) { for ( let i = 0 ; i < n ; i ++ ) { // skip if city i is not included in mask if (( mask & ( 1 << i )) === 0 ) continue ; if ( dp [ mask ][ i ] === INF ) continue ; // try to go to every unvisited city j for ( let j = 0 ; j < n ; j ++ ) { // skip if city j is already visited if (( mask & ( 1 << j )) !== 0 ) continue ; // cost to visit new city j from city i // such that previously visited cities // remain visited const nxt = mask | ( 1 << j ); dp [ nxt ][ j ] = Math . min ( dp [ nxt ][ j ], dp [ mask ][ i ] + cost [ i ][ j ]); } } } let ans = INF ; for ( let i = 0 ; i < n ; i ++ ) { // if last city on path is i and // cost of path is not infinity if ( dp [ fullMask ][ i ] !== INF ) // update net cost such that city 0 is visited in last ans = Math . min ( ans , dp [ fullMask ][ i ] + cost [ i ][ 0 ]); } return ans ; } //Driver Code Starts // Driver code const cost = [ [ 0 , 10 , 15 , 20 ], [ 10 , 0 , 35 , 25 ], [ 15 , 35 , 0 , 30 ], [ 20 , 25 , 30 , 0 ] ]; console . log ( tsp ( cost )); //Driver Code Ends Output 80 Comment Article Tags: Article Tags: Bit Magic Graph Dynamic Programming DSA Microsoft Google NPHard Opera + 4 More