The Knight's Tour - GeeksforGeeks

Source: https://www.geeksforgeeks.org/the-knights-tour-problem/

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 The Knight's Tour Last Updated : 29 Sep, 2025 Given an integer n , consider an n × n chessboard. A Knight starts at the top-left corner (0, 0) and must visit every cell exactly once following the Knight’s standard moves in chess (two steps in one direction and one step perpendicular). Return the n × n grid where each cell contains the step number (starting from 0) at which the Knight visits that cell. If no valid tour exists, return -1 . Examples: Input: n = 5 Output: [[0, 5, 14, 9, 20], [13, 8, 19, 4, 15], [18, 1, 6, 21, 10], [7, 12, 23, 16, 3], [24, 17, 2, 11, 22]] Explanation: Each number represents the step at which the Knight visits that cell, starting from (0, 0) as step 0. The output shows one valid Knight’s Tour on a 5×5 board. Input: n = 3 Output: [-1] Explanation: It is not possible to find a valid Knight's Tour on a 3x3 chessboard since the Knight cannot visit all 9 cells exactly once without revisiting or getting stuck. Try it on GfG Practice Table of Content [Approach -1] Using Recursion + Backtracking - O(8^(n*n)) Time and O(n^2) Space [Approach -2] Using Warnsdorff's Algorithm - O(n^3) Time and O(n^2) Space [Approach -1] Using Recursion + Backtracking - O(8 n*n) Time and O(n 2) Space We will use recursion and backtracking to build a sequence of knight moves that visits every cell once. Start at (0, 0), mark each visited cell with the move number, and try all 8 knight moves from the current cell. If a move leads to a dead end, undo it (backtrack) and try the next move. Stop when you have placed n*n moves (success) or exhausted all options (failure). C++ #include <iostream> #include <vector> using namespace std ; // 8 directions of knight moves int dx [8] = { 2 , 1 , -1 , -2 , -2 , -1 , 1 , 2 }; int dy [8] = { 1 , 2 , 2 , 1 , -1 , -2 , -2 , -1 }; // Utility function to check if the // move is valid bool isSafe (int x , int y , int n , vector < vector < int >> & board) { return (x >= 0 && y >= 0 && x < n && y < n && board [x][y] == -1); } // Recursive function to solve Knight's Tour bool knightTourUtil (int x , int y , int step , int n , vector < vector < int >> & board) { // If all squares are visited if (step == n * n) { return true ; } // Try all 8 possible knight moves for (int i = 0 ; i < 8 ; i ++) { int nx = x + dx [i]; int ny = y + dy [i]; if (isSafe (nx , ny , n , board)) { board [nx][ny] = step ; if (knightTourUtil (nx , ny , step + 1 , n , board)) { return true ; } // Backtrack board [nx][ny] = -1 ; } } return false ; } // Function to start Knight's Tour vector < vector < int >> knightTour (int n) { vector < vector < int >> board (n , vector < int > (n , -1)); // Start from top-left corner board [0][0] = 0 ; if (knightTourUtil (0 , 0 , 1 , n , board)) { return board ; } return {{ -1 }}; } int main () { int n = 5 ; vector < vector < int >> res = knightTour (n); for (auto & row : res) { for (int val : row) { cout << val << " " ; } cout << endl ; } return 0 ; } Java import java.util.ArrayList ; import java.util.Arrays ; class GFG { // 8 directions of knight moves static int [] dx = { 2 , 1 , - 1 , - 2 , - 2 , - 1 , 1 , 2 }; static int [] dy = { 1 , 2 , 2 , 1 , - 1 , - 2 , - 2 , - 1 }; // Utility function to check if the // move is valid static boolean isSafe (int x , int y , int n , ArrayList < ArrayList < Integer >> board) { return (x >= 0 && y >= 0 && x < n && y < n && board . get (x). get (y) == - 1); } // Recursive function to solve Knight's Tour static boolean knightTourUtil (int x , int y , int step , int n , ArrayList < ArrayList < Integer >> board) { // If all squares are visited if (step == n * n) { return true ; } // Try all 8 possible knight moves for (int i = 0 ; i < 8 ; i ++) { int nx = x + dx [i] ; int ny = y + dy [i] ; if (isSafe (nx , ny , n , board)) { board . get (nx). set (ny , step); if (knightTourUtil (nx , ny , step + 1 , n , board)) { return true ; } // Backtrack board . get (nx). set (ny , - 1); } } return false ; } // Function to start Knight's Tour static ArrayList < ArrayList < Integer >> knightTour (int n) { ArrayList < ArrayList < Integer >> board = new ArrayList <> (); for (int i = 0 ; i < n ; i ++) { ArrayList < Integer > row = new ArrayList <> (); for (int j = 0 ; j < n ; j ++) { row . add (- 1); } board . add (row); } // Start from top-left corner board . get (0). set (0 , 0); if (knightTourUtil (0 , 0 , 1 , n , board)) { return board ; } // If no solution exists, return board with -1 only ArrayList < ArrayList < Integer

>> noSol = new ArrayList <> (); ArrayList < Integer > row = new ArrayList <> (); row . add (- 1); noSol . add (row); return noSol ; } public static void main (String [] args) { int n = 5 ; ArrayList < ArrayList < Integer >> res = knightTour (n); for (ArrayList < Integer > row : res) { for (int val : row) { System . out . print (val + " "); } System . out . println (); } } } Python dx = [2 , 1 , - 1 , - 2 , - 2 , - 1 , 1 , 2] dy = [1 , 2 , 2 , 1 , - 1 , - 2 , - 2 , - 1] def isSafe (x , y , n , board): return x >= 0 and y >= 0 and x < n and y < n and board [x][y] == - 1 def knightTourUtil (x , y , step , n , board): # If all squares are visited if step == n * n : return True # Try all 8 possible knight moves for i in range (8): nx = x + dx [i] ny = y + dy [i] if isSafe (nx , ny , n , board): board [nx][ny] = step if knightTourUtil (nx , ny , step + 1 , n , board): return True # Backtrack board [nx][ny] = - 1 return False def knightTour (n): board = [[- 1 for _ in range (n)] for _ in range (n)] # Start from top-left corner board [0][0] = 0 if knightTourUtil (0 , 0 , 1 , n , board): return board return [[- 1]] if __name__ == "__main__" : n = 5 res = knightTour (n) for row in res : for val in row : print (val , end = " ") print () C# using System ; using System.Collections.Generic ; class GFG { static int [] dx = { 2 , 1 , - 1 , - 2 , - 2 , - 1 , 1 , 2 }; static int [] dy = { 1 , 2 , 2 , 1 , - 1 , - 2 , - 2 , - 1 }; static bool isSafe (int x , int y , int n , List < List < int >> board) { return x >= 0 && y >= 0 && x < n && y < n && board [x][y] == - 1 ; } static bool knightTourUtil (int x , int y , int step , int n , List < List < int >> board) { // If all squares are visited if (step == n * n) return true ; // Try all 8 possible knight moves for (int i = 0 ; i < 8 ; i ++) { int nx = x + dx [i]; int ny = y + dy [i]; if (isSafe (nx , ny , n , board)) { board [nx][ny] = step ; if (knightTourUtil (nx , ny , step + 1 , n , board)) return true ; // Backtrack board [nx][ny] = - 1 ; } } return false ; } static List < List < int >> knightTour (int n) { var board = new List < List < int >> (); for (int i = 0 ; i < n ; i ++) { var row = new List < int > (); for (int j = 0 ; j < n ; j ++) row . Add (- 1); board . Add (row); } // Start from top-left corner board [0][0] = 0 ; if (knightTourUtil (0 , 0 , 1 , n , board)) return board ; return new List < List < int >> { new List < int > { - 1 } }; } static void Main () { int n = 5 ; var res = knightTour (n); foreach (var row in res) { foreach (var val in row) Console . Write (val + " "); Console . WriteLine (); } } } JavaScript // 8 possible knight moves const dx = [2 , 1 , - 1 , - 2 , - 2 , - 1 , 1 , 2]; const dy = [1 , 2 , 2 , 1 , - 1 , - 2 , - 2 , - 1]; // Check if a cell is valid function isSafe (x , y , n , board) { return x >= 0 && y >= 0 && x < n && y < n && board [x][y] === - 1 ; } // Recursive utility function for Knight's Tour function knightTourUtil (x , y , step , n , board) { if (step === n * n) return true ; for (let i = 0 ; i < 8 ; i ++) { const nx = x + dx [i]; const ny = y + dy [i]; if (isSafe (nx , ny , n , board)) { board [nx][ny] = step ; if (knightTourUtil (nx , ny , step + 1 , n , board)) return true ; // Backtrack board [nx][ny] = - 1 ; } } return false ; } // Function to start Knight's Tour function knightTour (n) { const board = Array . from ({ length : n }, () => Array (n). fill (- 1)); // Start from top-left corner board [0][0] = 0 ; if (knightTourUtil (0 , 0 , 1 , n , board)) return board ; return [[- 1]]; } // Driver code const n = 5 ; const res = knightTour (n); for (const row of res) { console . log (row . join (' ')); } Output 0 5 14 9 20 13 8 19 4 15 18 1 6 21 10 7 12 23 16 3 24 17 2 11 22 [Approach -2] Using Warnsdorff's Algorithm - O(n 3) Time and O(n 2) Space When solving the Knight's Tour problem, backtracking works but is inefficient because it explores many unnecessary paths. If the correct move happens to be the last option, the algorithm wastes time trying all the wrong ones first. A smarter strategy is Warnsdorff's Algorithm , which uses a heuristic to reduce backtracking. Instead of trying moves in random order, it always chooses the next move with the fewest onward moves (the cell with the smallest degree). This prevents the knight from getting stuck early and greatly improves efficiency. Illustration C++ #include <iostream> #include <vector> #include <algorithm> using namespace std ; // Define 8 knight moves globally int dir [8][2] = { { 2 , 1 }, { 1 , 2 }, { -1 , 2 }, { -2 , 1 }, { -2 , -1 }, { -1 , -2 }, { 1 , -2 }, { 2 , -1 } }; // Count the number of onward moves from position (x, y) int countOptions (vector < vector < int >>& board , int x , int y) { int count = 0 ; int n = board . size (); for (int i = 0 ; i < 8 ; i ++) { int nx = x + dir [i][0]; int ny = y + dir [i][1]; if (nx >= 0 && ny >= 0 && nx < n && ny < n && board [nx][ny] == -1) { count ++ ; } } return count ; } // Generate valid knight moves from (x, y), sorted by fewest onward moves vector < vector < int >> getSortedMoves (vector < vector < int >>& board , int x , int y) { vector < vector < int >> moveList ; for (int i = 0 ; i < 8 ; i ++) { int nx = x + dir [i][0]; int ny = y + dir [i][1]; if (nx >= 0 && ny >= 0 && nx < board . size () && ny < board . size () && board [nx][ny] == -1) { int options = countOptions (board , nx , ny); moveList . push_back ({ options , i }); } } // Sort using default vector<int> lexicographic comparison sort (moveList . begin (), moveList . end ()); return moveList ; } // Recursive function to solve the Knight's Tour bool knightTourUtil (int x , int y , int step , int n , vector < vector < int >>& board) { if (step == n * n) return true ; vector < vector < int >> moves = getSortedMoves (board , x , y); for (vector < int > move : moves) { int dirIdx = move [1]; int nx = x + dir [dirIdx][0]; int ny = y + dir [dirIdx][1]; board [nx][ny] = step ; if (knightTourUtil (nx , ny , step + 1 , n , board)) return true ; // Backtrack board [nx][ny] = -1 ; } return false ; } // Function to start Knight's Tour vector < vector < int >> knightTour (int n) { vector < vector < int >> board (n , vector < int

> (n , -1)); // Start from top-left corner board [0][0] = 0 ; if (knightTourUtil (0 , 0 , 1 , n , board)) { return board ; } return {{ -1 }}; } int main () { int n = 5 ; vector < vector < int >> result = knightTour (n); for (vector < int > row : result) { for (int val : row) { cout << val << " " ; } cout << endl ; } return 0 ; } Java import java.util.ArrayList ; import java.util.Arrays ; import java.util.Collections ; import java.util.Comparator ; class GFG { // Define 8 knight moves globally static int [][] dir = { { 2 , 1 }, { 1 , 2 }, { - 1 , 2 }, { - 2 , 1 }, { - 2 , - 1 }, { - 1 , - 2 }, { 1 , - 2 }, { 2 , - 1 } }; // Count the number of onward moves from position (x, y) static int countOptions (ArrayList < ArrayList < Integer >> board , int x , int y) { int count = 0 , n = board . size (); for (int i = 0 ; i < 8 ; i ++) { int nx = x + dir [i][0] ; int ny = y + dir [i][1] ; if (nx >= 0 && ny >= 0 && nx < n && ny < n && board . get (nx). get (ny) == - 1) count ++ ; } return count ; } // Generate valid knight moves from (x, y), sorted by fewest onward moves static ArrayList < ArrayList < Integer >> getSortedMoves (ArrayList < ArrayList < Integer >> board , int x , int y) { ArrayList < ArrayList < Integer >> moveList = new ArrayList <> (); int n = board . size (); for (int i = 0 ; i < 8 ; i ++) { int nx = x + dir [i][0] ; int ny = y + dir [i][1] ; if (nx >= 0 && ny >= 0 && nx < n && ny < n && board . get (nx). get (ny) == - 1) { int options = countOptions (board , nx , ny); ArrayList < Integer > move = new ArrayList <> (); move . add (options); move . add (i); moveList . add (move); } } // Sort by options count Collections . sort (moveList , new Comparator < ArrayList < Integer >> () { public int compare (ArrayList < Integer > a , ArrayList < Integer > b) { if (! a . get (0). equals (b . get (0))) return a . get (0) - b . get (0); return a . get (1) - b . get (1); } }); return moveList ; } // Recursive function to solve the Knight's Tour static boolean knightTourUtil (int x , int y , int step , int n , ArrayList < ArrayList < Integer >> board) { if (step == n * n) return true ; ArrayList < ArrayList < Integer >> moves = getSortedMoves (board , x , y); for (ArrayList < Integer > move : moves) { int dirIdx = move . get (1); int nx = x + dir [dirIdx][0] ; int ny = y + dir [dirIdx][1] ; board . get (nx). set (ny , step); if (knightTourUtil (nx , ny , step + 1 , n , board)) return true ; // Backtrack board . get (nx). set (ny , - 1); } return false ; } // Function to start Knight's Tour static ArrayList < ArrayList < Integer >> knightTour (int n) { ArrayList < ArrayList < Integer >> board = new ArrayList <> (); for (int i = 0 ; i < n ; i ++) { ArrayList < Integer > row = new ArrayList <> (Collections . nCopies (n , - 1)); board . add (row); } board . get (0). set (0 , 0); if (knightTourUtil (0 , 0 , 1 , n , board)) return board ; ArrayList < ArrayList < Integer >> fail = new ArrayList <> (); fail . add (new ArrayList <> (Arrays . asList (- 1))); return fail ; } public static void main (String [] args) { int n = 5 ; ArrayList < ArrayList < Integer >> result = knightTour (n); for (ArrayList < Integer > row : result) { for (int val : row) System . out . print (val + " "); System . out . println (); } } } Python # Define 8 knight moves globally dir = [[2 , 1], [1 , 2], [- 1 , 2], [- 2 , 1], [- 2 , - 1], [- 1 , - 2], [1 , - 2], [2 , - 1]] # Count the number of onward moves from position (x, y) def countOptions (board , x , y): count , n = 0 , len (board) for dx , dy in dir : nx , ny = x + dx , y + dy if 0 <= nx < n and 0 <= ny < n and board [nx][ny] == - 1 : count += 1 return count # Generate valid knight moves from (x, y), sorted by fewest onward moves def getSortedMoves (board , x , y): moveList , n = [], len (board) for i in range (8): nx , ny = x + dir [i][0], y + dir [i][1] if 0 <= nx < n and 0 <= ny < n and board [nx][ny] == - 1 : options = countOptions (board , nx , ny) moveList . append ([options , i]) moveList . sort () return moveList # Recursive function to solve the Knight's Tour def knightTourUtil (x , y , step , n , board): if step == n * n : return True moves = getSortedMoves (board , x , y) for move in moves : dirIdx = move [1] nx , ny = x + dir [dirIdx][0], y + dir [dirIdx][1] board [nx][ny] = step if knightTourUtil (nx , ny , step + 1 , n , board): return True # Backtrack board [nx][ny] = - 1 return False # Function to start Knight's Tour def knightTour (n): board = [[- 1] * n for _ in range (n)] board [0][0] = 0 if knightTourUtil (0 , 0 , 1 , n , board): return board return [[- 1]] if __name__ == '__main__' : n = 5 result = knightTour (n) for row in result : print (* row) C# using System ; using System.Collections.Generic ; class GFG { // Define 8 knight moves globally static int [,] dir = { { 2 , 1 }, { 1 , 2 }, { - 1 , 2 }, { - 2 , 1 }, { - 2 , - 1 }, { - 1 , - 2 }, { 1 , - 2 }, { 2 , - 1 } }; // Count the number of onward moves from position (x, y) static int countOptions (int [,] board , int x , int y) { int count = 0 , n = board . GetLength (0); for (int i = 0 ; i < 8 ; i ++) { int nx = x + dir [i , 0]; int ny = y + dir [i , 1]; if (nx >= 0 && ny >= 0 && nx < n && ny < n && board [nx , ny] == - 1) count ++ ; } return count ; } // Generate valid knight moves from (x, y), sorted by fewest onward moves static List < int [] > getSortedMoves (int [,] board , int x , int y) { List < int [] > moveList = new List < int [] > (); int n = board . GetLength (0); for (int i = 0 ; i < 8 ; i ++) { int nx = x + dir [i , 0]; int ny = y + dir [i , 1]; if (nx >= 0 && ny >= 0 && nx < n && ny < n && board [nx , ny] == - 1) { int options = countOptions (board , nx , ny); moveList . Add (new int []{ options , i }); } } moveList . Sort ((a , b) => a [0] != b [0] ? a [0]. CompareTo (b [0]) : a [1]. CompareTo (b [1])); return moveList ; } // Recursive function to solve the Knight's Tour static bool knightTourUtil (int x , int y , int step , int n , int [,] board) { if (step == n * n) return true ; List < int [] > moves = getSortedMoves (board , x , y); foreach (var move in moves) { int dirIdx = move [1]; int nx =

x + dir [dirIdx , 0]; int ny = y + dir [dirIdx , 1]; board [nx , ny] = step ; if (knightTourUtil (nx , ny , step + 1 , n , board)) return true ; // Backtrack board [nx , ny] = - 1 ; } return false ; } // Function to start Knight's Tour static List < List < int >> knightTour (int n) { int [,] board = new int [n , n]; List < List < int >> res = new List < List < int >> (); for (int i = 0 ; i < n ; i ++) for (int j = 0 ; j < n ; j ++) board [i , j] = - 1 ; board [0 , 0] = 0 ; if (knightTourUtil (0 , 0 , 1 , n , board)) { for (int i = 0 ; i < n ; i ++) { List < int > rowList = new List < int > (); for (int j = 0 ; j < n ; j ++) rowList . Add (board [i , j]); res . Add (rowList); } return res ; } res . Add (new List < int > { - 1 }); return res ; } public static void Main () { int n = 5 ; List < List < int >> result = knightTour (n); foreach (var row in result) { Console . WriteLine (string . Join (" " , row)); } } } JavaScript // Define 8 knight moves globally let dir = [[2 , 1], [1 , 2], [- 1 , 2], [- 2 , 1], [- 2 , - 1], [- 1 , - 2], [1 , - 2], [2 , - 1]]; // Count the number of onward moves from position (x, y) function countOptions (board , x , y) { let count = 0 , n = board . length ; for (let i = 0 ; i < 8 ; i ++) { let nx = x + dir [i][0]; let ny = y + dir [i][1]; if (nx >= 0 && ny >= 0 && nx < n && ny < n && board [nx][ny] === - 1) count ++ ; } return count ; } // Generate valid knight moves from (x, y), sorted by fewest onward moves function getSortedMoves (board , x , y) { let moveList = [], n = board . length ; for (let i = 0 ; i < 8 ; i ++) { let nx = x + dir [i][0]; let ny = y + dir [i][1]; if (nx >= 0 && ny >= 0 && nx < n && ny < n && board [nx][ny] === - 1) { let options = countOptions (board , nx , ny); moveList . push ([options , i]); } } moveList . sort ((a , b) => (a [0] - b [0]) || (a [1] - b [1])); return moveList ; } // Recursive function to solve the Knight's Tour function knightTourUtil (x , y , step , n , board) { if (step === n * n) return true ; let moves = getSortedMoves (board , x , y); for (let move of moves) { let dirIdx = move [1]; let nx = x + dir [dirIdx][0]; let ny = y + dir [dirIdx][1]; board [nx][ny] = step ; if (knightTourUtil (nx , ny , step + 1 , n , board)) return true ; // Backtrack board [nx][ny] = - 1 ; } return false ; } // Function to start Knight's Tour function knightTour (n) { let board = Array . from ({ length : n }, () => Array (n). fill (- 1)); board [0][0] = 0 ; if (knightTourUtil (0 , 0 , 1 , n , board)) return board ; return [[- 1]]; } // Driver Code let n = 5 ; let result = knightTour (n); for (let row of result) { console . log (row . join (" ")); } Output 0 21 10 15 6 11 16 7 20 9 24 1 22 5 14 17 12 3 8 19 2 23 18 13 4 Comment Article Tags: Article Tags: Backtracking Mathematical DSA Amazon chessboard-problems + 1 More