Sudoku Solver - GeeksforGeeks

Source: https://www.geeksforgeeks.org/sudoku-backtracking-7/

Sudoku Solver Last Updated : 27 Sep, 2025 Given an incomplete Sudoku in the form of matrix mat[][] of order 9*9, Complete the Sudoku. A sudoku solution must satisfy all of the following rules : Each of the digits 1-9 must occur exactly once in each row. Each of the digits 1-9 must occur exactly once in each column. Each of the digits 1-9 must occur exactly once in each of the 9, 3x3 sub-boxes of the grid. Note: Zeros in the mat[][] indicate blanks, which are to be filled with some number between 1 to 9. You can not replace the element in the cell which is not blank. Examples: Input: Output: Explanation: Each row, column and 3*3 box of the output matrix contains unique numbers. Try it on GfG Practice Table of Content [Approach 1] Using Backtracking [Approach 2] Using Bit Masking with Backtracking - O(9^(n*n)) Time and O(n) Space [Approach 1] Using Backtracking The idea to solve Sudoku is to use backtracking, where we recursively try to fill the empty cells with numbers from 1 to 9. For every unassigned cell, we place a number and then check whether it is valid to place that number in the given row, column, and 3×3 subgrid. If it is valid, we move to the next cell; if not, we backtrack and try another number. This process continues until all cells are filled or no solution exists. C++ #include <iostream> #include <vector> using namespace std ; // Function to check if it is safe to place num at mat[row][col] bool isSafe (vector < vector < int >> & mat , int row , int col , int num) { // Check if num exist in the row for (int x = 0 ; x <= 8 ; x ++) if (mat [row][x] == num) return false ; // Check if num exist in the col for (int x = 0 ; x <= 8 ; x ++) if (mat [x][col] == num) return false ; // Check if num exist in the 3x3 sub-matrix int startRow = row - (row % 3), startCol = col - (col % 3); for (int i = 0 ; i < 3 ; i ++) for (int j = 0 ; j < 3 ; j ++) if (mat [i + startRow][j + startCol] == num) return false ; return true ; } // Function to solve the Sudoku problem bool solveSudokuRec (vector < vector < int >> & mat , int row , int col) { int n = mat . size (); // base case: Reached nth column of last row if (row == n - 1 && col == n) return true ; // If last column of the row go to next row if (col == n) { row ++ ; col = 0 ; } // If cell is already occupied then move forward if (mat [row][col] != 0) return solveSudokuRec (mat , row , col + 1); for (int num = 1 ; num <= n ; num ++) { // If it is safe to place num at current position if (isSafe (mat , row , col , num)) { mat [row][col] = num ; if (solveSudokuRec (mat , row , col + 1)) return true ; mat [row][col] = 0 ; } } return false ; } void solveSudoku (vector < vector < int >> & mat) { solveSudokuRec (mat , 0 , 0); } int main () { vector < vector < int >> mat = { { 3 , 0 , 6 , 5 , 0 , 8 , 4 , 0 , 0 }, { 5 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0 }, { 0 , 8 , 7 , 0 , 0 , 0 , 0 , 3 , 1 }, { 0 , 0 , 3 , 0 , 1 , 0 , 0 , 8 , 0 }, { 9 , 0 , 0 , 8 , 6 , 3 , 0 , 0 , 5 }, { 0 , 5 , 0 , 0 , 9 , 0 , 6 , 0 , 0 }, { 1 , 3 , 0 , 0 , 0 , 0 , 2 , 5 , 0 }, { 0 , 0 , 0 , 0 , 0 , 0 , 0 , 7 , 4 }, { 0 , 0 , 5 , 2 , 0 , 6 , 3 , 0 , 0 }}; solveSudoku (mat); for (int i = 0 ; i < mat . size (); i ++) { for (int j = 0 ; j < mat . size (); j ++) cout << mat [i][j] << " " ; cout << endl ; } return 0 ; } Java import java.util.Arrays ; class GfG { // Function to check if it is safe to place num at mat[row][col] static boolean isSafe (int [][] mat , int row , int col , int num) { // Check if num exists in the row for (int x = 0 ; x < 9 ; x ++) if (mat [row][x] == num) return false ; // Check if num exists in the col for (int x = 0 ; x < 9 ; x ++) if (mat [x][col] == num) return false ; // Check if num exists in the 3x3 sub-matrix int startRow = row - (row % 3), startCol = col - (col % 3); for (int i = 0 ; i < 3 ; i ++) for (int j = 0 ; j < 3 ; j ++) if (mat [i + startRow][j + startCol] == num) return false ; return true ; } // Function to solve the Sudoku problem static boolean solveSudokuRec (int [][] mat , int row , int col) { // base case: Reached nth column of the last row if (row == 8 && col == 9) return true ; // If last column of the row go to the next row if (col == 9) { row ++ ; col = 0 ; } // If cell is already occupied then move forward if (mat [row][col] != 0) return solveSudokuRec (mat , row , col + 1); for (int num = 1 ; num <= 9 ; num ++) { // If it is safe to place num at current position if (isSafe (mat , row , col , num)) { mat [row][col] = num ; if (solveSudokuRec (mat , row , col + 1)) return true ; mat [row][col] = 0 ; } } return false ; } static void solveSudoku (int [][] mat) { solveSudokuRec (mat , 0 , 0); } public static void main (String [] args) { int [][] mat = { { 3 , 0 , 6 , 5 , 0 , 8 , 4 , 0 , 0 }, { 5 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0 }, { 0 , 8 , 7 , 0 , 0 , 0 , 0 , 3 , 1 }, { 0 , 0 , 3 , 0 , 1 , 0 , 0 , 8 , 0 }, { 9 , 0 , 0 , 8 , 6 , 3 , 0 , 0 , 5 }, { 0 , 5 , 0 , 0 , 9 , 0 , 6 , 0 , 0 }, { 1 , 3 , 0 , 0 , 0 , 0 , 2 , 5 , 0 }, { 0 , 0 , 0 , 0 , 0 , 0 , 0 , 7 , 4 }, { 0 , 0 , 5 , 2 , 0 , 6 , 3 , 0 , 0 } }; solveSudoku (mat); for (int i = 0 ; i < mat . length ; i ++) { for (int j = 0 ; j < mat [i] . length ; j ++) System . out . print (mat [i][j] + " "); System . out . println (); } } } Python # Function to check if it is safe to place num at mat[row][col] def isSafe (mat , row , col , num): # Check if num exists in the row for x in range (9): if mat [row][x] == num : return False # Check if num exists in the col for x in range (9): if mat [x][col] == num : return False # Check if num exists in the 3x3 sub-matrix startRow = row - (row % 3) startCol = col - (col % 3) for i in range (3): for j in range (3): if mat [i + startRow][j + startCol] == num : return False return True # Function to solve the Sudoku problem def solveSudokuRec (mat , row , col): # base case: Reached nth column of the last row if row == 8 and col == 9 : return True # If last column of the row go to the next row if col == 9 : row += 1 col = 0 # If cell is already occupied then move forward if mat [row][col] != 0 : return solveSudokuRec (mat , row , col + 1) for num in range (1 , 10): # If it is safe to place num at current position if isSafe (mat , row , col , num): mat [row][col] = num if solveSudokuRec (mat , row , col + 1): return True mat [row][col] = 0 return False def solveSudoku (mat): solveSudokuRec (mat , 0 , 0) if __name__ == "__main__" : mat = [[3 , 0 , 6 , 5 , 0 , 8 , 4 , 0 , 0], [5 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0], [0 , 8 , 7 , 0 , 0 , 0 , 0 , 3 , 1], [0 , 0 , 3 , 0 , 1 , 0 , 0 , 8 , 0], [9 , 0 , 0 , 8 , 6 , 3 , 0 , 0 , 5], [0 , 5 , 0 , 0 , 9 , 0 , 6 , 0 , 0], [1 , 3 , 0 , 0 , 0 , 0 , 2 , 5 , 0], [0 , 0 , 0 , 0 , 0 , 0 , 0 , 7 , 4], [0 , 0 , 5 , 2 , 0 , 6 , 3 , 0 , 0]] solveSudoku (mat) for row in mat : print (" " . join (map (str , row))) C# using System ; class GfG { // Function to check if it is safe to place num at mat[row][col] static bool isSafe (int [,] mat , int row , int col , int num) { // Check if num exists in the row for (int x = 0 ; x < 9 ; x ++) if (mat [row , x] == num) return false ; // Check if num exists in the col for (int x = 0 ; x < 9 ; x ++) if (mat [x , col] == num) return false ; // Check if num exists in the 3x3 sub-matrix int startRow = row - (row % 3), startCol = col - (col % 3); for (int i = 0 ; i < 3 ; i ++) for (int j = 0 ; j < 3 ; j ++) if (mat [i + startRow , j + startCol] == num) return false ; return true ; } // Function to solve the Sudoku problem static bool solveSudokuRec (int [,] mat , int row , int col) { // base case: Reached nth column of the last row if (row == 8 && col == 9) return true ; // If last column of the row go to the next row if (col == 9) { row ++ ; col = 0 ; } // If cell is already occupied then move forward if (mat [row , col] != 0) return solveSudokuRec (mat , row , col + 1); for (int num = 1 ; num <= 9 ; num ++) { // If it is safe to place num at current position if (isSafe (mat , row , col , num)) { mat [row , col] = num ; if (solveSudokuRec (mat , row , col + 1)) return true ; mat [row , col] = 0 ; } } return false ; } static void solveSudoku (int [,] mat) { solveSudokuRec (mat , 0 , 0); } public static void Main () { int [,] mat = { { 3 , 0 , 6 , 5 , 0 , 8 , 4 , 0 , 0 }, { 5 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0 }, { 0 , 8 , 7 , 0 , 0 , 0 , 0 , 3 , 1 }, { 0 , 0 , 3 , 0 , 1 , 0 , 0 , 8 , 0 }, { 9 , 0 , 0 , 8 , 6 , 3 , 0 , 0 , 5 }, { 0 , 5 , 0 , 0 , 9 , 0 , 6 , 0 , 0 }, { 1 , 3 , 0 , 0 , 0 , 0 , 2 , 5 , 0 }, { 0 , 0 , 0 , 0 , 0 , 0 , 0 , 7 , 4 }, { 0 , 0 , 5 , 2 , 0 , 6 , 3 , 0 , 0 } }; solveSudoku (mat); for (int i = 0 ; i < 9 ; i ++) { for (int j = 0 ; j < 9 ; j ++) Console . Write (mat [i , j] + " "); Console . WriteLine (); } } } JavaScript // Function to check if it is safe to place num at mat[row][col] function isSafe (mat , row , col , num) { // Check if num exists in the row for (let x = 0 ; x < 9 ; x ++) if (mat [row][x] === num) return false ; // Check if num exists in the col for (let x = 0 ; x < 9 ; x ++) if (mat [x][col] === num) return false ; // Check if num exists in the 3x3 sub-matrix const startRow = row - (row % 3), startCol = col - (col % 3); for (let i = 0 ; i < 3 ; i ++) for (let j = 0 ; j < 3 ; j ++) if (mat [i + startRow][j + startCol] === num) return false ; return true ; } // Function to solve the Sudoku problem function solveSudokuRec (mat , row , col) { // base case: Reached nth column of the last row if (row === 8 && col === 9) return true ; // If last column of the row go to the next row if (col === 9) { row ++ ; col = 0 ; } // If cell is already occupied then move forward if (mat [row][col] !== 0) return solveSudokuRec (mat , row , col + 1); for (let num = 1 ; num <= 9 ; num ++) { // If it is safe to place num at current position if (isSafe (mat , row , col , num)) { mat [row][col] = num ; if (solveSudokuRec (mat , row , col + 1)) return true ; mat [row][col] = 0 ; } } return false ; } function solveSudoku (mat) { solveSudokuRec (mat , 0 , 0); } // Driver Code const mat = [[3 , 0 , 6 , 5 , 0 , 8 , 4 , 0 , 0], [5 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0], [0 , 8 , 7 , 0 , 0 , 0 , 0 , 3 , 1], [0 , 0 , 3 , 0 , 1 , 0 , 0 , 8 , 0], [9 , 0 , 0 , 8 , 6 , 3 , 0 , 0 , 5], [0 , 5 , 0 , 0 , 9 , 0 , 6 , 0 , 0], [1 , 3 , 0 , 0 , 0 , 0 , 2 , 5 , 0], [0 , 0 , 0 , 0 , 0 , 0 , 0 , 7 , 4], [0 , 0 , 5 , 2 , 0 , 6 , 3 , 0 , 0]]; solveSudoku (mat); mat . forEach (row => console . log (row . join (" "))); Output 3 1 6 5 7 8 4 9 2 5 2 9 1 3 4 7 6 8 4 8 7 6 2 9 5 3 1 2 6 3 4 1 5 9 8 7 9 7 4 8 6 3 1 2 5 8 5 1 7 9 2 6 4 3 1 3 8 9 4 7 2 5 6 6 9 2 3 5 1 8 7 4 7 4 5 2 8 6 3 1 9 Time complexity: O(9 (n*n)), For every unassigned index, there are 9 possible options and for each index

Auxiliary Space: O(1) [Approach 2] Using Bit Masking with Backtracking - O(9 (n*n)) Time and O(n) Space We can solve this efficiently by using backtracking combined with bitmasking . The idea is simple: for every empty cell, we attempt to place numbers from 1 to 9 and move recursively to the next cell. The key optimization lies in avoiding repeated scanning of the row, column, and 3×3 box each time we place a number. Instead, we maintain three bitmask arrays—rows, cols, and boxes. In these arrays, each bit indicates whether a particular number has already been used in that row, column, or box. Before placing a number, we simply check the corresponding bits in O(1) time to determine if it is valid. If it is safe, we set the bits and continue with recursion. If the placement leads to a dead end, we backtrack by unsetting the bits and trying another option. C++ #include <iostream> #include <vector> using namespace std ; // Function to heck if it is safe to place num at mat[row][col] bool isSafe (vector < vector < int >> & mat , int i , int j , int num , vector < int > & row , vector < int > & col , vector < int > & box) { if ((row [i] & (1 << num)) || (col [j] & (1 << num)) || (box [i / 3 * 3 + j / 3] & (1 << num))) return false ; return true ; } bool sudokuSolverRec (vector < vector < int >> & mat , int i , int j , vector < int > & row , vector < int > & col , vector < int > & box) { int n = mat . size (); // base case: Reached nth column of last row if (i == n - 1 && j == n) return true ; // If reached last column of the row go to next row if (j == n) { i ++ ; j = 0 ; } // If cell is already occupied then move forward if (mat [i][j] != 0) return sudokuSolverRec (mat , i , j + 1 , row , col , box); for (int num = 1 ; num <= n ; num ++) { // If it is safe to place num at current position if (isSafe (mat , i , j , num , row , col , box)) { mat [i][j] = num ; // Update masks for the corresponding row, column and box row [i] |= (1 << num); col [j] |= (1 << num); box [i / 3 * 3 + j / 3] |= (1 << num); if (sudokuSolverRec (mat , i , j + 1 , row , col , box)) return true ; // Unmask the number num in the corresponding row, column and box masks mat [i][j] = 0 ; row [i] &= ~ (1 << num); col [j] &= ~ (1 << num); box [i / 3 * 3 + j / 3] &= ~ (1 << num); } } return false ; } void solveSudoku (vector < vector < int >> & mat) { int n = mat . size (); vector < int > row (n , 0), col (n , 0), box (n , 0); // Set the bits in bitmasks for values that are initital present for (int i = 0 ; i < n ; i ++) { for (int j = 0 ; j < n ; j ++) { if (mat [i][j] != 0) { row [i] |= (1 << mat [i][j]); col [j] |= (1 << mat [i][j]); box [(i / 3) * 3 + j / 3] |= (1 << mat [i][j]); } } } sudokuSolverRec (mat , 0 , 0 , row , col , box); } int main () { vector < vector < int >> mat = { { 3 , 0 , 6 , 5 , 0 , 8 , 4 , 0 , 0 }, { 5 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0 }, { 0 , 8 , 7 , 0 , 0 , 0 , 0 , 3 , 1 }, { 0 , 0 , 3 , 0 , 1 , 0 , 0 , 8 , 0 }, { 9 , 0 , 0 , 8 , 6 , 3 , 0 , 0 , 5 }, { 0 , 5 , 0 , 0 , 9 , 0 , 6 , 0 , 0 }, { 1 , 3 , 0 , 0 , 0 , 0 , 2 , 5 , 0 }, { 0 , 0 , 0 , 0 , 0 , 0 , 0 , 7 , 4 }, { 0 , 0 , 5 , 2 , 0 , 6 , 3 , 0 , 0 }}; solveSudoku (mat); for (int i = 0 ; i < mat . size (); i ++) { for (int j = 0 ; j < mat . size (); j ++) cout << mat [i][j] << " " ; cout << endl ; } return 0 ; } Java import java.util.Arrays ; class GFG { // Function to check if it is safe to place num at mat[row][col] static boolean isSafe (int [][] mat , int i , int j , int num , int [] row , int [] col , int [] box) { if ((row [i] & (1 << num)) != 0 || (col [j] & (1 << num)) != 0 || (box [i / 3 * 3 + j / 3] & (1 << num)) != 0) return false ; return true ; } static boolean sudokuSolverRec (int [][] mat , int i , int j , int [] row , int [] col , int [] box) { int n = mat . length ; // base case: Reached nth column of last row if (i == n - 1 && j == n) return true ; // If reached last column of the row go to next row if (j == n) { i ++ ; j = 0 ; } // If cell is already occupied then move forward if (mat [i][j] != 0) return sudokuSolverRec (mat , i , j + 1 , row , col , box); for (int num = 1 ; num <= n ; num ++) { // If it is safe to place num at current position if (isSafe (mat , i , j , num , row , col , box)) { mat [i][j] = num ; // Update masks for the corresponding row, column and box row [i] |= (1 << num); col [j] |= (1 << num); box [i / 3 * 3 + j / 3] |= (1 << num); if (sudokuSolverRec (mat , i , j + 1 , row , col , box)) return true ; // Unmask the number num in the corresponding row, column and box masks mat [i][j] = 0 ; row [i] &= ~ (1 << num); col [j] &= ~ (1 << num); box [i / 3 * 3 + j / 3] &= ~ (1 << num); } } return false ; } static void solveSudoku (int [][] mat) { int n = mat . length ; int [] row = new int [n] ; int [] col = new int [n] ; int [] box = new int [n] ; // Set the bits in bitmasks for values that are initially present for (int i = 0 ; i < n ; i ++) { for (int j = 0 ; j < n ; j ++) { if (mat [i][j] != 0) { row [i] |= (1 << mat [i][j]); col [j] |= (1 << mat [i][j]); box [(i / 3) * 3 + j / 3] |= (1 << mat [i][j]); } } } sudokuSolverRec (mat , 0 , 0 , row , col , box); } public static void main (String [] args) { int [][] mat = { { 3 , 0 , 6 , 5 , 0 , 8 , 4 , 0 , 0 }, { 5 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0 }, { 0 , 8 , 7 , 0 , 0 , 0 , 0 , 3 , 1 }, { 0 , 0 , 3 , 0 , 1 , 0 , 0 , 8 , 0 }, { 9 , 0 , 0 , 8 , 6 , 3 , 0 , 0 , 5 }, { 0 , 5 , 0 , 0 , 9 , 0 , 6 , 0 , 0 }, { 1 , 3 , 0 , 0 , 0 , 0 , 2 , 5 , 0 }, { 0 , 0 , 0 , 0 , 0 , 0 , 0 , 7 , 4 }, { 0 , 0 , 5 , 2 , 0 , 6 , 3 , 0 , 0 } }; solveSudoku (mat); for (int i = 0 ; i < mat . length ; i ++) { for (int j = 0 ; j < mat [i] . length ; j ++) System . out . print (mat [i][j] + " "); System . out . println (); } } } Python def isSafe (mat , i , j , num , row , col , box): if (row [i] & (1 << num)) or (col [j] & (1 << num)) or (box [i // 3 * 3 + j // 3] & (1 << num)): return False return True def sudokuSolverRec (mat , i , j , row , col , box): n = len (mat) # base case: Reached nth column of last row if i == n - 1 and j == n : return True # If reached last column of the row go to next row if j == n : i += 1 j = 0 # If cell is already occupied then move forward if mat [i][j] != 0 : return sudokuSolverRec (mat , i , j + 1 , row , col , box) for num in range (1 , n + 1): # If it is safe to place num at current position if isSafe (mat , i , j , num , row , col , box): mat [i][j] = num # Update masks for the corresponding row, column and box row [i] |= (1 << num) col [j] |= (1 << num) box [i // 3 * 3 + j // 3] |= (1 << num) if sudokuSolverRec (mat , i , j + 1 , row , col , box): return True # Unmask the number num in the corresponding row, column and box masks mat [i][j] = 0 row [i] &= ~ (1 << num) col [j] &= ~ (1 << num) box [i // 3 * 3 + j // 3] &= ~ (1 << num) return False def solveSudoku (mat): n = len (mat) row = [0] * n col = [0] * n box = [0] * n # Set the bits in bitmasks for values that are initially present for i in range (n): for j in range (n): if mat [i][j] != 0 : row [i] |= (1 << mat [i][j]) col [j] |= (1 << mat [i][j]) box [(i // 3) * 3 + j // 3] |= (1 << mat [i][j]) sudokuSolverRec (mat , 0 , 0 , row , col , box) if __name__ == "__main__" : mat = [[3 , 0 , 6 , 5 , 0 , 8 , 4 , 0 , 0], [5 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0], [0 , 8 , 7 , 0 , 0 , 0 , 0 , 3 , 1], [0 , 0 , 3 , 0 , 1 , 0 , 0 , 8 , 0], [9 , 0 , 0 , 8 , 6 , 3 , 0 , 0 , 5], [0 , 5 , 0 , 0 , 9 , 0 , 6 , 0 , 0], [1 , 3 , 0 , 0 , 0 , 0 , 2 , 5 , 0], [0 , 0 , 0 , 0 , 0 , 0 , 7 , 4], [0 , 0 , 5 , 2 , 0 , 6 , 3 , 0 , 0]] solveSudoku (mat) for row in mat : print (" " . join (map (str , row))) C# using System ; class GFG { // Function to check if it is safe to place num at mat[row, col] static bool isSafe (int [,] mat , int i , int j , int num , int [] row , int [] col , int [] box) { if ((row [i] & (1 << num)) != 0 || (col [j] & (1 << num)) != 0 || (box [i / 3 * 3 + j / 3] & (1 << num)) != 0) return false ; return true ; } static bool sudokuSolverRec (int [,] mat , int i , int j , int [] row , int [] col , int [] box) { int n = mat . GetLength (0); // base case: Reached nth column of last row if (i == n - 1 && j == n) return true ; // If reached last column of the row, go to next row if (j == n) { i ++ ; j = 0 ; } // If cell is already occupied, then move forward if (mat [i , j] != 0) return sudokuSolverRec (mat , i , j + 1 , row , col , box); for (int num = 1 ; num <= n ; num ++) { // If it is safe to place num at current position if (isSafe (mat , i , j , num , row , col , box)) { mat [i , j] = num ; // Update masks for the corresponding row, column, and box row [i] |= (1 << num); col [j] |= (1 << num); box [i / 3 * 3 + j / 3] |= (1 << num); if (sudokuSolverRec (mat , i , j + 1 , row , col , box)) return true ; // Unmask the number num in the corresponding row, column and box masks mat [i , j] = 0 ; row [i] &= ~ (1 << num); col [j] &= ~ (1 << num); box [i / 3 * 3 + j / 3] &= ~ (1 << num); } } return false ; } static void solveSudoku (int [,] mat) { int n = mat . GetLength (0); int [] row = new int [n]; int [] col = new int [n]; int [] box = new int [n]; // Set the bits in bitmasks for values that are initially present for (int i = 0 ; i < n ; i ++) { for (int j = 0 ; j < n ; j ++) { if (mat [i , j] != 0) { row [i] |= (1 << mat [i , j]); col [j] |= (1 << mat [i , j]); box [(i / 3) * 3 + j / 3] |= (1 << mat [i , j]); } } } sudokuSolverRec (mat , 0 , 0 , row , col , box); } public static void Main (string [] args) { int [,] mat = { { 3 , 0 , 6 , 5 , 0 , 8 , 4 , 0 , 0 }, { 5 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0 }, { 0 , 8 , 7 , 0 , 0 , 0 , 0 , 3 , 1 }, { 0 , 0 , 3 , 0 , 1 , 0 , 0 , 8 , 0 }, { 9 , 0 , 0 , 8 , 6 , 3 , 0 , 0 , 5 }, { 0 , 5 , 0 , 0 , 9 , 0 , 6 , 0 , 0 }, { 1 , 3 , 0 , 0 , 0 , 0 , 2 , 5 , 0 }, { 0 , 0 , 0 , 0 , 0 , 0 , 0 , 7 , 4 }, { 0 , 0 , 5 , 2 , 0 , 6 , 3 , 0 , 0 } }; solveSudoku (mat); for (int i = 0 ; i < mat . GetLength (0); i ++) { for (int j = 0 ; j < mat . GetLength (1); j ++) Console . Write (mat [i , j] + " "); Console . WriteLine (); } } } JavaScript // Function to check if it is safe to place num at mat[row][col] function isSafe (mat , i , j , num , row , col , box) { if ((row [i] & (1 << num)) !== 0 || (col [j] & (1 << num)) !== 0 || (box [Math . floor (i / 3) * 3 + Math . floor (j / 3)] & (1 << num)) !== 0) return false ; return true ; } function sudokuSolverRec (mat , i , j , row , col , box) { const n = mat . length ; // base case: Reached nth column of last row if (i === n - 1 && j === n) return true ; // If reached last column of the row, go to next row if (j === n) { i ++ ; j = 0 ; } // If cell is already occupied, then move forward if (mat [i][j] !== 0) return sudokuSolverRec (mat , i , j + 1 , row , col , box); for (let num = 1 ; num <= n ; num ++) { // If it is safe to place num at current position if (isSafe (mat , i , j , num , row , col , box)) { mat [i][j] = num ; // Update masks for the corresponding row, column, and box row [i] |= (1 << num); col [j] |= (1 << num); box [Math . floor (i / 3) * 3 + Math . floor (j / 3)] |= (1 << num); if (sudokuSolverRec (mat , i , j + 1 , row , col , box)) return true ; // Unmask the number num in the corresponding row, column and box masks mat [i][j] = 0 ; row [i] &= ~ (1 << num); col [j] &= ~ (1 << num); box [Math . floor (i / 3) * 3 + Math . floor (j / 3)] &= ~ (1 << num); } } return false ; } function solveSudoku (mat) { const n = mat . length ; const row = new Array (n). fill (0); const col = new Array (n). fill (0); const box = new Array (n). fill (0); // Set the bits in bitmasks for values that are initially present for (let i = 0 ; i < n ; i ++) { for (let j = 0 ; j < n ; j ++) { if (mat [i][j] !== 0) { row [i] |= (1 << mat [i][j]); col [j] |= (1 << mat [i][j]); box [Math . floor (i / 3) * 3 + Math . floor (j / 3)] |= (1 << mat [i][j]); } } } sudokuSolverRec (mat , 0 , 0 , row , col , box); } // Driver Code const mat = [[3 , 0 , 6 , 5 , 0 , 8 , 4 , 0 , 0], [5 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0], [0 , 8 , 7 , 0 , 0 , 0 , 0 , 3 , 1], [0 , 0 , 3 , 0 , 1 , 0 , 0 , 8 , 0], [9 , 0 , 0 , 8 , 6 , 3 , 0 , 0 , 5], [0 , 5 , 0 , 0 , 9 , 0 , 6 , 0 , 0], [1 , 3 , 0 , 0 , 0 , 0 , 2 , 5 , 0], [0 , 0 , 0 , 0 , 0 , 0 , 0 , 7 , 4], [0 , 0 , 5 , 2 , 0 , 6 , 3 , 0 , 0]]; solveSudoku (mat); for (let i = 0 ; i < mat . length ; i ++) { console . log (mat [i]. join (" ")); } Output 3 1 6 5 7 8 4 9 2 5 2 9 1 3 4 7 6 8 4 8 7 6 2 9 5 3 1 2 6 3 4 1 5 9 8 7 9 7 4 8 6 3 1 2 5 8 5 1 7 9 2 6 4 3 1 3 8 9 4 7 2 5 6 6 9 2 3 5 1 8 7 4 7 4 5 2 8 6 3 1 9 Comment Article Tags: Article Tags: Backtracking Matrix DSA Microsoft

Amazon Google Oracle Flipkart Directi Zoho MAQ Software MakeMyTrip Ola Cabs PayPal + 10 More