N Queen Problem - GeeksforGeeks

Source: https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/

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 N Queen Problem Last Updated : 27 Sep, 2025 Given an integer n , place n queens on an n × n chessboard such that no two queens attack each other. A queen can attack another queen if they are placed in the same row , the same column , or on the same diagonal . Find all possible distinct arrangements of the queens on the board that satisfy these conditions. The output should be an array of solutions, where each solution is represented as an array of integers of size n, and the i-th integer denotes the column position of the queen in the i-th row. If no solution exists, return an empty array. Examples: Input: n = 4 Output: [[2, 4, 1, 3], [3, 1, 4, 2]] Explanation: Below is Solution for 4 queen problem Input: n = 3 Output: [] Explanation: There are no possible solutions for n = 3 Table of Content [Naive Approach] - Using Backtracking - O(n!) Time and O(n^2) Space [Optimized Approach] Using Column and Diagonal Hashing [Naive Approach] - Using Backtracking The idea is to use backtracking to place queens on an n × n chessboard. We can proceed either row by row or column by column . For each row (or column), try placing a queen in every column (or row) and check if it is safe (i.e., no other queen in the same column, row, or diagonals). If safe, place the queen and move to the next row/column. If no valid position exists, backtrack to the previous step and try a different position. Continue until all queens are placed or all possibilities are explored. Below is a sample part of the recursive tree for the above approach (shown using column-by-column placement of queens). C++ #include <iostream> #include <vector> using namespace std ; // Function to check if it is safe to place int isSafe (vector < vector < int >>& mat , int row , int col) { int n = mat . size (); int i , j ; // Check this col on upper side for (i = 0 ; i < row ; i ++) if (mat [i][col]) return 0 ; // Check upper diagonal on left side for (i = row -1 , j = col -1 ; i >= 0 && j >= 0 ; i -- , j --) if (mat [i][j]) return 0 ; // Check upper diagonal on right side for (i = row -1 , j = col + 1 ; j < n && i >= 0 ; i -- , j ++) if (mat [i][j]) return 0 ; return 1 ; } // Recursive function to place queens void placeQueens (int row , vector < vector < int >>& mat , vector < vector < int >>& result) { int n = mat . size (); // base case: If all queens are placed if (row == n) { // store current solution vector < int > ans ; for (int i = 0 ; i < n ; i ++){ for (int j = 0 ; j < n ; j ++){ if (mat [i][j]){ ans . push_back (j + 1); } } } result . push_back (ans); return ; } // Consider the row and try placing // queen in all columns one by one for (int i = 0 ; i < n ; i ++){ // Check if the queen can be placed if (isSafe (mat , row , i)){ mat [row][i] = 1 ; placeQueens (row + 1 , mat , result); // backtrack mat [row][i] = 0 ; } } } // Function to find all solutions vector < vector < int >> nQueen (int n) { // Initialize the board vector < vector < int >> mat (n , vector < int > (n , 0)); vector < vector < int >> result ; // Place queens placeQueens (0 , mat , result); return result ; } int main () { int n = 4 ; vector < vector < int >> result = nQueen (n); for (auto & ans : result){ for (auto i : ans){ cout << i << " " ; } cout << endl ; } return 0 ; } Java import java.util.ArrayList ; class GFG { // Function to check if it is safe to place static int isSafe (int [][] mat , int row , int col) { int n = mat . length ; int i , j ; // Check this col on upper side for (i = 0 ; i < row ; i ++) if (mat [i][col] == 1) return 0 ; // Check upper diagonal on left side for (i = row - 1 , j = col - 1 ; i >= 0 && j >= 0 ; i -- , j --) if (mat [i][j] == 1) return 0 ; // Check upper diagonal on right side for (i = row - 1 , j = col + 1 ; j < n && i >= 0 ; i -- , j ++) if (mat [i][j] == 1) return 0 ; return 1 ; } // Recursive function to place queens static void placeQueens (int row , int [][] mat , ArrayList < ArrayList < Integer >> result) { int n = mat . length ; // base case: If all queens are placed if (row == n) { // store current solution ArrayList < Integer > ans = new ArrayList <> (); for (int i = 0 ; i < n ; i ++) { for (int j = 0 ; j < n ; j ++) { if (mat [i][j] == 1) { ans . add (j + 1); } } } result . add (ans); return ; } // Consider the row and try placing // queen in all columns one by one for (int i = 0 ; i < n ; i ++) { // Check if the queen can be placed if (isSafe (mat , row , i) == 1) { mat [row][i] = 1 ; placeQueens (row + 1 , mat , result); // backtrack mat [row][i] = 0 ; } } } // Function to find all solutions static ArrayList < ArrayList < Integer >> nQueen (int n) { // Initialize the board int [][] mat = new int [n][n] ; ArrayList < ArrayList < Integer >> result = new ArrayList <> (); // Place queens placeQueens (0 , mat , result); return result ; } public static void main (String [] args) { int n = 4 ; ArrayList < ArrayList < Integer >> result = nQueen (n); for (ArrayList < Integer > ans : result) { for (int i : ans) { System . out . print (i + " "); } System . out . println (); } } } Python def isSafe (mat , row , col): n = len (mat) # Check this col on upper side for i in range (row): if mat [i][col]: return 0 # Check upper diagonal on left side i , j = row - 1 , col - 1 while i >= 0 and j >= 0 : if mat [i][j]: return 0 i -= 1 j -= 1 # Check upper diagonal on right side i , j = row - 1 , col + 1 while i >= 0 and j < n : if mat [i][j]: return 0 i -= 1 j += 1 return 1 # Recursive function to place queens def placeQueens (row , mat , result): n = len (mat) # base case: If all queens are placed if row == n : # store current solution ans = [] for i in range (n): for j in range (n): if mat [i][j]: ans . append (j + 1) result . append (ans) return # Consider the row and try placing # queen in all columns one by one for i in range (n): # Check if the queen can be placed if isSafe (mat , row , i): mat [row][i] = 1 placeQueens (row + 1 , mat , result) # backtrack mat [row][i] = 0 # Function to find all solutions def nQueen (n): # Initialize the board mat = [[0] * n for _ in range (n)] result = [] # Place queens placeQueens (0 , mat , result) return result if __name__ == "__main__" : n = 4 result = nQueen (n) for ans in result : print (" " . join (map (str , ans))) C# using System ; using System.Collections.Generic ; class GFG { // Function to check if it is safe to place static int isSafe (int [,] mat , int row , int col) { int n = mat . GetLength (0); int i , j ; // Check this col on upper side for (i = 0 ; i < row ; i ++) if (mat [i , col] == 1) return 0 ; // Check upper diagonal on left side for (i = row - 1 , j = col - 1 ; i >= 0 && j >= 0 ; i -- , j --) if (mat [i , j] == 1) return 0 ; // Check upper diagonal on right side for (i = row - 1 , j = col + 1 ; j < n && i >= 0 ; i -- , j ++) if (mat [i , j] == 1) return 0 ; return 1 ; } // Recursive function to place queens static void placeQueens (int row , int [,] mat , List < List < int >> result) { int n = mat . GetLength (0); // base case: If all queens are placed if (row == n) { // store current solution List < int > ans = new List < int > (); for (int i = 0 ; i < n ; i ++) { for (int j = 0 ; j < n ; j ++) { if (mat [i , j] == 1) { ans . Add (j + 1); } } } result . Add (ans); return ; } // Consider the row and try placing // queen in all columns one by one for (int i = 0 ; i < n ; i ++) { // Check if the queen can be placed if (isSafe (mat , row , i) == 1) { mat [row , i] = 1 ; placeQueens (row + 1 , mat , result); // backtrack mat [row , i] = 0 ; } } } // Function to find all solutions static List < List < int >> nQueen (int n) { // Initialize the board int [,] mat = new int [n , n]; List < List < int >> result = new List < List < int >> (); // Place queens placeQueens (0 , mat , result); return result ; } public static void Main () { int n = 4 ; List < List < int >> result = nQueen (n); foreach (var ans in result) { foreach (var i in ans) { Console . Write (i + " "); } Console . WriteLine (); } } } JavaScript // Function to check if it is safe to place function isSafe (mat , row , col) { let n = mat . length ; let i , j ; // Check this col on upper side for (i = 0 ; i < row ; i ++) if (mat [i][col]) return 0 ; // Check upper diagonal on left side for (i = row - 1 , j = col - 1 ; i >= 0 && j >= 0 ; i -- , j --) if (mat [i][j]) return 0 ; // Check upper diagonal on right side for (i = row - 1 , j = col + 1 ; j < n && i >= 0 ; i -- , j ++) if (mat [i][j]) return 0 ; return 1 ; } // Recursive function to place queens function placeQueens (row , mat , result) { let n = mat . length ; // base case: If all queens are placed if (row === n) { // store current solution let ans = []; for (let i = 0 ; i < n ; i ++) { for (let j = 0 ; j < n ; j ++) { if (mat [i][j]) { ans . push (j + 1); } } } result . push (ans); return ; } // Consider the row and try placing // queen in all columns one by one for (let i = 0 ; i < n ; i ++) { // Check if the queen can be placed if (isSafe (mat , row , i)) { mat [row][i] = 1 ; placeQueens (row + 1 , mat , result); // backtrack mat [row][i] = 0 ; } } } // Function to find all solutions function nQueen (n) { // Initialize the board let mat = Array . from ({ length : n }, () => Array (n). fill (0)); let result = []; // Place queens placeQueens (0 , mat , result); return result ; } // Driver code let n = 4 ; let result = nQueen (n); for (let ans of result) { console . log (ans . join (' ')); } Output 2 4 1 3 3 1 4 2 Time Complexity: O(n!) For the first queen, we have n columns to choose from. Each subsequent queen has fewer valid positions because previous queens block their columns and diagonals, roughly reducing choices to n–2, n–4, and so on, giving an approximate O(n!) time. Auxiliary Space: O(n 2), We use an n × n board to track queen placements, which requires O(n^2) space, plus O(n) space for the recursion stack during backtracking. [Optimized Approach] Using Column and Diagonal Hashing Instead of checking every row and diagonal, use three arrays to track occupied columns and diagonals. A queen can be placed at a cell only if its column and both diagonals are free. This reduces the safe-check from O(n) to O(1). We use three arrays to efficiently check if a queen can be placed at (i, j) without conflicts: cols[] – Tracks if a queen is already placed in a column. cols[j] = 1 means column j is occupied. rightDiagonal[] – Tracks diagonals where i + j is constant. On a chessboard, all cells on the same top-left to bottom-right diagonal have the same sum of row and column indices (i + j). So we use rightDiagonal[i + j] to quickly check if that diagonal is occupied. leftDiagonal[] – Tracks diagonals where i - j is constant. All cells on the same top-right to bottom-left diagonal have the same difference (i - j). To avoid negative indices, we shift by n - 1: leftDiagonal[i - j + n - 1]. This lets us represent all diagonals with non-negative indices in the array. A queen can be placed at (i, j) only if all three arrays are 0. Mark them when placing a queen, and unmark when backtracking. C++ #include <iostream> #include <vector> using namespace std ; // Recursive function to place queens void placeQueens (int i , vector < int > & cols , vector < int > & leftDiagonal , vector < int > & rightDiagonal , vector < int > & cur , vector < vector < int >> & result) { int n = cols . size (); // base case: If all queens are placed if (i == n) { result . push_back (cur); return ; } // Consider the row and try placing // queen in all columns one by one for (int j = 0 ; j < n ; j ++){ // Check if the queen can be placed if (cols [j] || rightDiagonal [i + j] || leftDiagonal [i - j + n - 1]) continue ; // mark the cell occupied cols [j] = 1 ; rightDiagonal [i + j] = 1 ; leftDiagonal [i - j + n - 1] = 1 ; cur . push_back (j + 1); placeQueens (i + 1 , cols , leftDiagonal , rightDiagonal , cur , result); // remove the queen from current cell cur . pop_back (); cols [j] = 0 ; rightDiagonal [i + j] = 0 ; leftDiagonal [i - j + n - 1] = 0 ; } } // Function to find the solution // to the N-Queens problem vector < vector < int >> nQueen (int n) { // array to mark the occupied cells vector < int > cols (n , 0); vector < int > leftDiagonal (n * 2 , 0); vector < int > rightDiagonal (n * 2 , 0); vector < int > cur ; vector < vector < int >> result ; // Place queens placeQueens (0 , cols , leftDiagonal , rightDiagonal , cur , result); return result ; } int main () { int n = 4 ; vector < vector < int >> ans = nQueen (n); for (auto & a : ans){ for (auto i : a){ cout << i << " " ; } cout << endl ; } return 0 ; } Java import java.util.ArrayList ; class GFG { // Recursive function to place queens static void placeQueens (int i , int [] cols , int [] leftDiagonal , int [] rightDiagonal , ArrayList < Integer > cur , ArrayList < ArrayList < Integer >> result) { int n = cols . length ; // base case: If all queens are placed if (i == n) { result . add (new ArrayList <> (cur)); return ; } // Consider the row and try placing // queen in all columns one by one for (int j = 0 ; j < n ; j ++) { // Check if the queen can be placed if (cols [j] == 1 || rightDiagonal [i + j] == 1 || leftDiagonal [i - j + n - 1] == 1) continue ; // mark the cell occupied cols [j] = 1 ; rightDiagonal [i + j] = 1 ; leftDiagonal [i - j + n - 1] = 1 ; cur . add (j + 1); placeQueens (i + 1 , cols , leftDiagonal , rightDiagonal , cur , result); // remove the queen from current cell cur . remove (cur . size () - 1); cols [j] = 0 ; rightDiagonal [i + j] = 0 ; leftDiagonal [i - j + n - 1] = 0 ; } } // Function to find the solution // to the N-Queens problem static ArrayList < ArrayList < Integer >> nQueen (int n) { // array to mark the occupied cells int [] cols = new int [n] ; int [] leftDiagonal = new int [n * 2] ; int [] rightDiagonal = new int [n * 2] ; ArrayList < Integer > cur = new ArrayList <> (); ArrayList < ArrayList < Integer >> result = new ArrayList <> (); // Place queens placeQueens (0 , cols , leftDiagonal , rightDiagonal , cur , result); return result ; } public static void main (String [] args) { int n = 4 ; ArrayList < ArrayList < Integer >> ans = nQueen (n); for (ArrayList < Integer > a : ans) { for (int i : a) { System . out . print (i + " "); } System . out . println (); } } } Python # Recursive function to place queens def placeQueens (i , cols , leftDiagonal , rightDiagonal , cur , result): n = len (cols) # base case: If all queens are placed if i == n : result . append (cur [:]) return # Consider the row and try placing # queen in all columns one by one for j in range (n): # Check if the queen can be placed if cols [j] or rightDiagonal [i + j] or leftDiagonal [i - j + n - 1]: continue # mark the cell occupied cols [j] = 1 rightDiagonal [i + j] = 1 leftDiagonal [i - j + n - 1] = 1 cur . append (j + 1) placeQueens (i + 1 , cols , leftDiagonal , rightDiagonal , cur , result) # remove the queen from current cell cur . pop () cols [j] = 0 rightDiagonal [i + j] = 0 leftDiagonal [i - j + n - 1] = 0 # Function to find the solution # to the N-Queens problem def nQueen (n): # array to mark the occupied cells cols = [0] * n leftDiagonal = [0] * (2 * n) rightDiagonal = [0] * (2 * n) cur = [] result = [] # Place queens placeQueens (0 , cols , leftDiagonal , rightDiagonal , cur , result) return result if __name__ == "__main__" : n = 4 ans = nQueen (n) for a in ans : print (" " . join (map (str , a))) C# using System ; using System.Collections.Generic ; class GFG { // Recursive function to place queens static void placeQueens (int i , int [] cols , int [] leftDiagonal , int [] rightDiagonal , List < int > cur , List < List < int >> result) { int n = cols . Length ; // base case: If all queens are placed if (i == n) { result . Add (new List < int > (cur)); return ; } // Consider the row and try placing // queen in all columns one by one for (int j = 0 ; j < n ; j ++) { // Check if the queen can be placed if (cols [j] == 1 || rightDiagonal [i + j] == 1 || leftDiagonal [i - j + n - 1] == 1) continue ; // mark the cell occupied cols [j] = 1 ; rightDiagonal [i + j] = 1 ; leftDiagonal [i - j + n - 1] = 1 ; cur . Add (j + 1); placeQueens (i + 1 , cols , leftDiagonal , rightDiagonal , cur , result); // remove the queen from current cell cur . RemoveAt (cur . Count - 1); cols [j] = 0 ; rightDiagonal [i + j] = 0 ; leftDiagonal [i - j + n - 1] = 0 ; } } // Function to find the solution // to the N-Queens problem static List < List < int >> nQueen (int n) { // array to mark the occupied cells int [] cols = new int [n]; int [] leftDiagonal = new int [2 * n]; int [] rightDiagonal = new int [2 * n]; List < int > cur = new List < int > (); List < List < int >>

result = new List < List < int >> (); // Place queens placeQueens (0 , cols , leftDiagonal , rightDiagonal , cur , result); return result ; } static void Main () { int n = 4 ; List < List < int >> ans = nQueen (n); foreach (var a in ans) { Console . WriteLine (string . Join (" " , a)); } } } JavaScript // Recursive function to place queens function placeQueens (i , cols , leftDiagonal , rightDiagonal , cur , result) { const n = cols . length ; // base case: If all queens are placed if (i === n) { result . push ([... cur]); return ; } // Consider the row and try placing // queen in all columns one by one for (let j = 0 ; j < n ; j ++) { // Check if the queen can be placed if (cols [j] || rightDiagonal [i + j] || leftDiagonal [i - j + n - 1]) { continue ; } // mark the cell occupied cols [j] = 1 ; rightDiagonal [i + j] = 1 ; leftDiagonal [i - j + n - 1] = 1 ; cur . push (j + 1); placeQueens (i + 1 , cols , leftDiagonal , rightDiagonal , cur , result); // remove the queen from current cell cur . pop (); cols [j] = 0 ; rightDiagonal [i + j] = 0 ; leftDiagonal [i - j + n - 1] = 0 ; } } // Function to find the solution // to the N-Queens problem function nQueen (n) { // array to mark the occupied cells const cols = new Array (n). fill (0); const leftDiagonal = new Array (2 * n). fill (0); const rightDiagonal = new Array (2 * n). fill (0); const cur = []; const result = []; // Place queens placeQueens (0 , cols , leftDiagonal , rightDiagonal , cur , result); return result ; } // Driven Code const n = 4 ; const ans = nQueen (n); ans . forEach (a => { console . log (a . join (" ")); }); Output 2 4 1 3 3 1 4 2 Time Complexity: O(n!) – we try placing a queen in every row recursively, pruning invalid positions using the arrays. Auxiliary Space: O(n) Related Articles: Printing all solutions in N-Queen Problem Comment Article Tags: Article Tags: Backtracking DSA Amazon Twitter Accolite MAQ Software Visa Amdocs chessboard-problems + 5 More