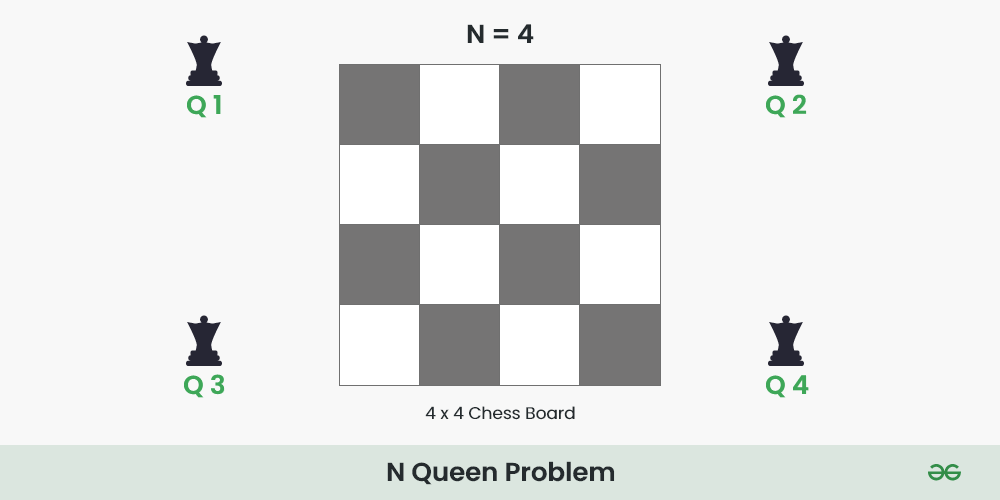
# **N Queen Problem**

We have discussed [**Knight’s tour**](https://www.geeksforgeeks.org/the-knights-tour-problem/) and [**Rat in a Maze**](https://www.geeksforgeeks.org/rat-in-a-maze/) problem earlier as examples of Backtracking problems. Let us discuss N Queen as another example problem that can be solved using backtracking.

## **What is N-Queen problem?**

****

The **N** Queen is the problem of placing **N** chess queens on an **N×N** chessboard so that no two queens attack each other.

For example, the following is a solution for the 4 Queen problem.



The expected output is in the form of a matrix that has ‘**Q**‘s for the blocks where queens are placed and the empty spaces are represented by **‘.’** . For example, the following is the output matrix for the above 4-Queen solution.

#### ***. Q . .***

#### ***. . . Q***

#### ***Q . . .***

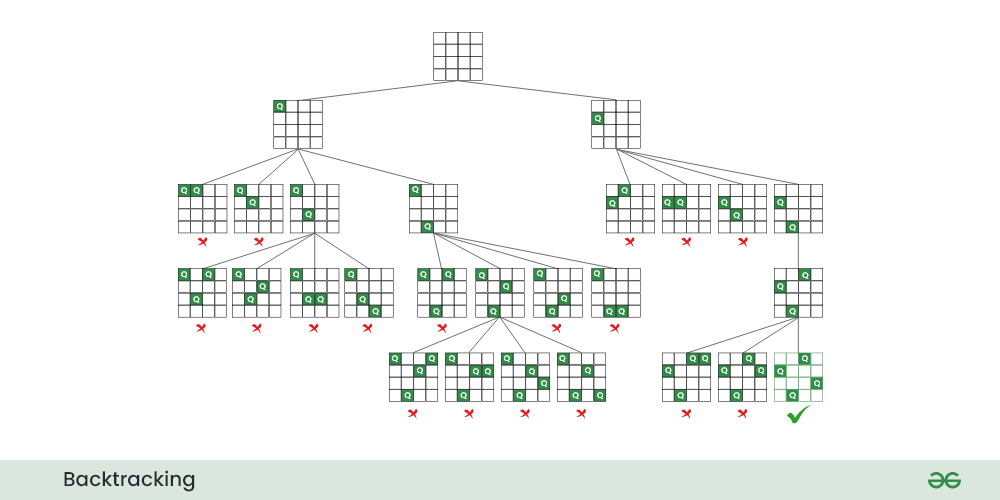
#### ***. . Q .***

[Recommended: Please solve it on “***PRACTICE*** ” first, before moving on to the solution.](https://www.geeksforgeeks.org/problems/n-queen-problem/0)

## **N Queen Problem using** [Backtracking](https://www.geeksforgeeks.org/introduction-to-backtracking-data-structure-and-algorithm-tutorials/)**:**

*The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return* ***false****.*

Below is the recursive tree of the above approach:



*Recursive tree for N Queen problem*

Follow the steps mentioned below to implement the idea:

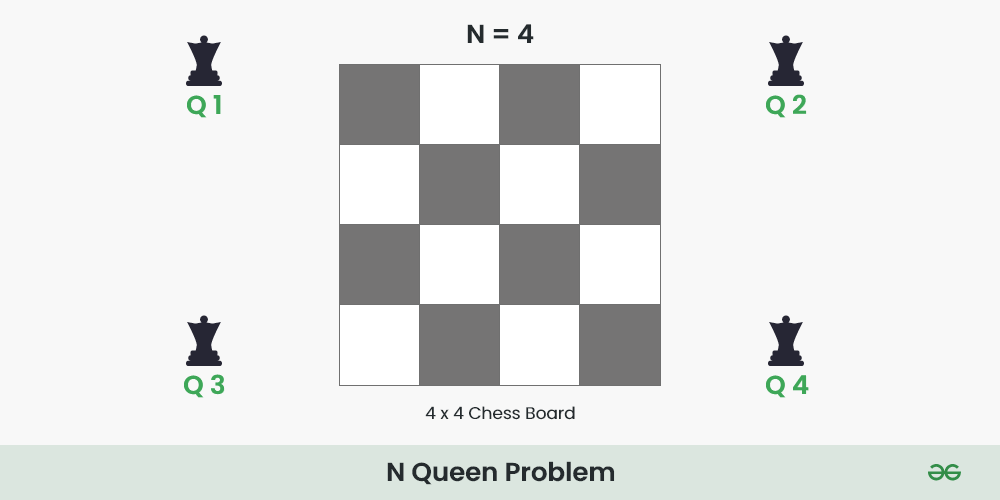
* Start in the leftmost column
* If all queens are placed return true
* Try all rows in the current column. Do the following for every row.
  + If the queen can be placed safely in this row
    - Then mark this **[row, column]** as part of the solution and recursively check if placing queen here leads to a solution.
    - If placing the queen in **[row, column]** leads to a solution then return **true**.
    - If placing queen doesn’t lead to a solution then unmark this **[row, column]** then backtrack and try other rows.
  + If all rows have been tried and valid solution is not found return **false** to trigger backtracking.

***For better visualisation of this backtracking approach, please refer*** [***4 Queen problem***](https://www.geeksforgeeks.org/4-queens-problem/)***.***

# **4 Queens Problem**

The **4 Queens** Problem consists in placing four queens on a **4 x 4** chessboard so that no two queens attack each other. That is, no two queens are allowed to be placed on the **same row**, the **same column** or the **same diagonal**.

We are going to look for the solution for n=4 on a 4 x 4 chessboard in this article.

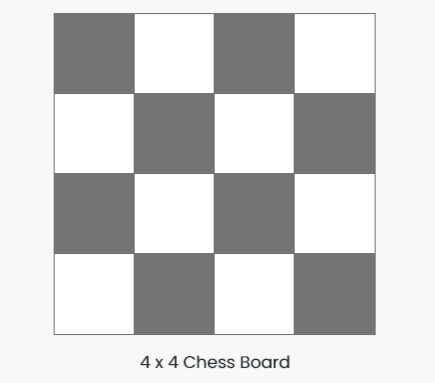


## **4 Queens Problem using** [Backtracking Algorithm](https://www.geeksforgeeks.org/introduction-to-backtracking-data-structure-and-algorithm-tutorials/)**:**

*Place each queen one by one in different rows, starting from the topmost row. While placing a queen in a row, check for clashes with already placed queens. For any column, if there is no clash then mark this row and column as part of the solution by placing the queen. In case, if no safe cell found due to clashes, then backtrack (i.e, undo the placement of recent queen) and return false.*

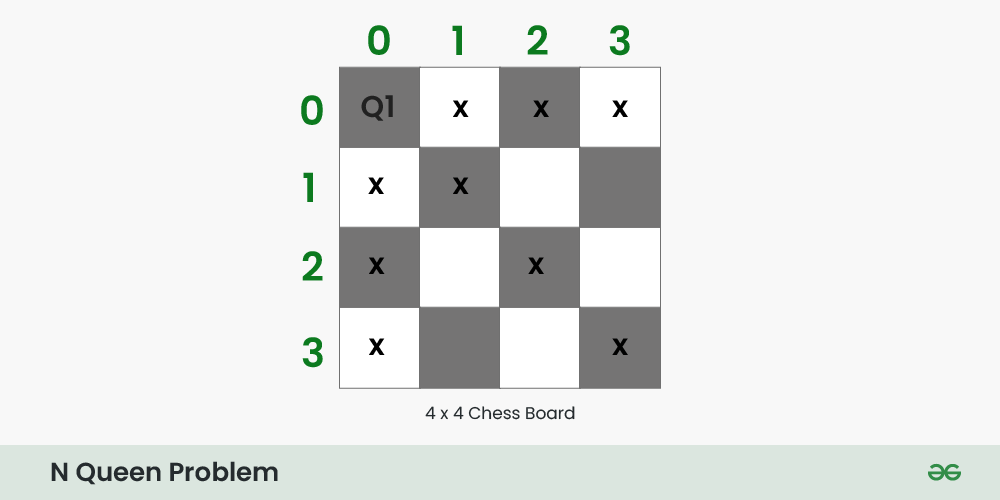
### **Illustration of 4 Queens Solution:**

**Step 0:** Initialize a 4×4 board.



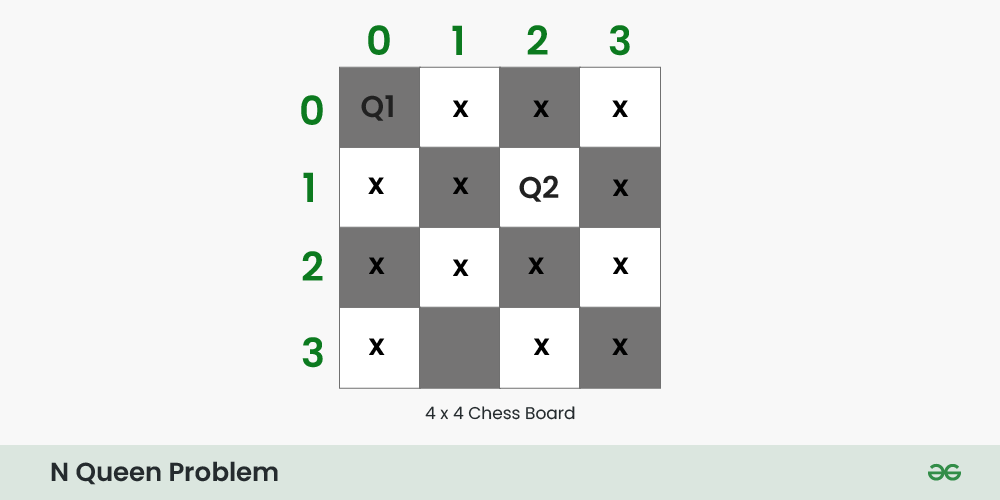
**Step 1:**

* Put our first Queen (**Q1**) in the **(0,0)** cell .
* ‘**x**‘ represents the cells which is not safe i.e. they are under attack by the Queen (**Q1**).
* After this move to the next row [ 0 -> 1 ].



**Step 2:**

* Put our next Queen (**Q2**) in the **(1,2)** cell .
* After this move to the next row [ 1 -> 2 ].

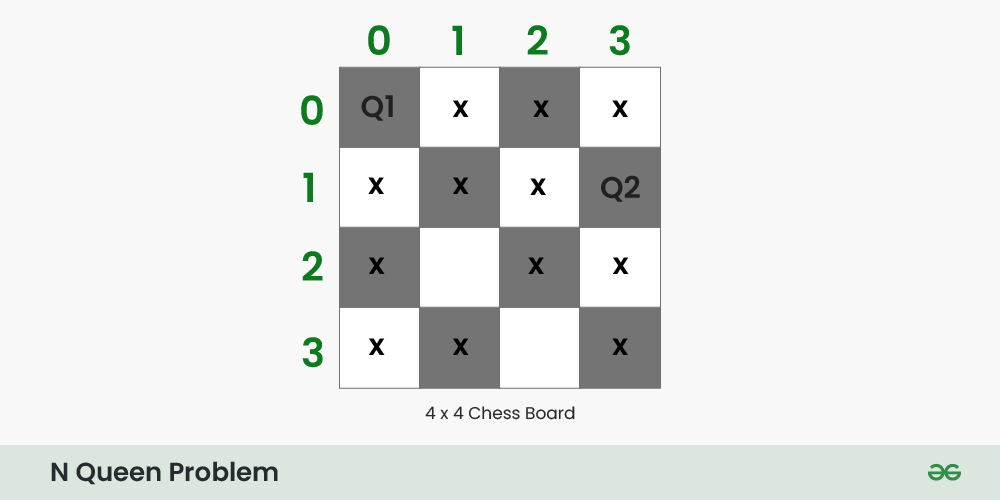


**Step 3:**

* At row 2 there is no cell which are safe to place Queen (**Q3**) .
* So, backtrack and remove queen **Q2** queen from cell ( 1, 2 ) .

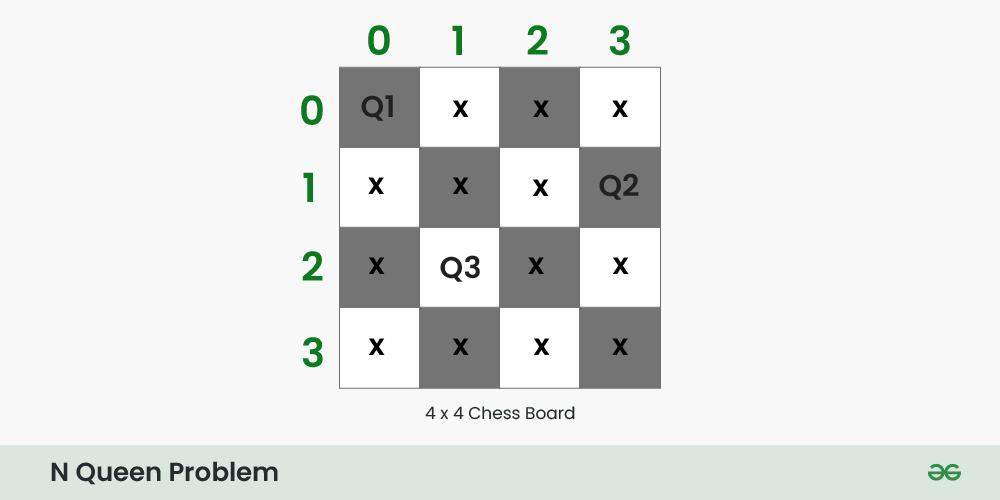
**Step 4:**

* There is still a safe cell in the row 1 i.e. cell ( 1, 3 ).
* Put Queen ( **Q2** ) at cell ( 1, 3).



**Step 5:**

* Put queen **( Q3 )** at cell ( 2, 1 ).

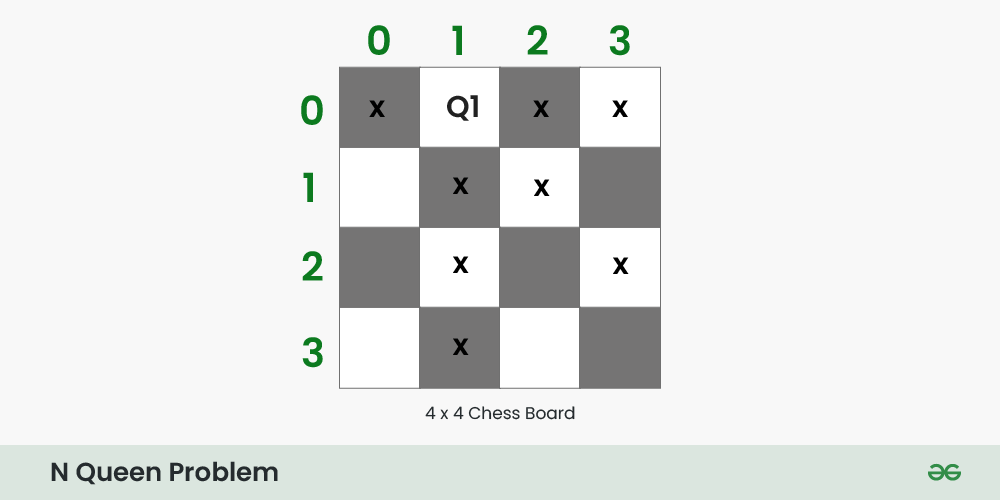


**Step 6:**

* There is no any cell to place Queen ( **Q4** ) at row 3.
* Backtrack and remove Queen ( **Q3 )** from row 2.
* Again there is no other safe cell in row 2, So backtrack again and remove queen ( **Q2** ) from row 1.
* Queen ( **Q1 )** will be remove from cell **(0,0)** and move to next safe cell i.e. **(0 , 1)**.

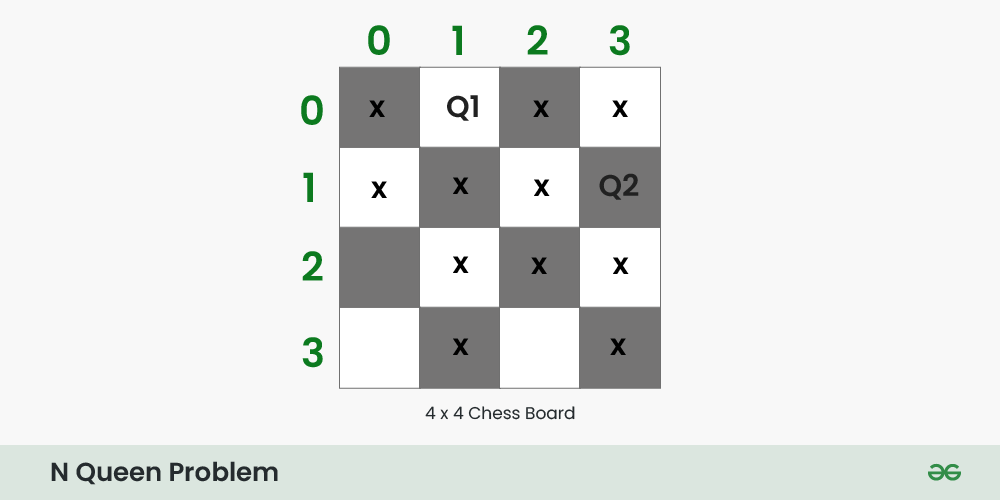
**Step 7:**

* Place Queen Q1 at cell (0 , 1), and move to next row.



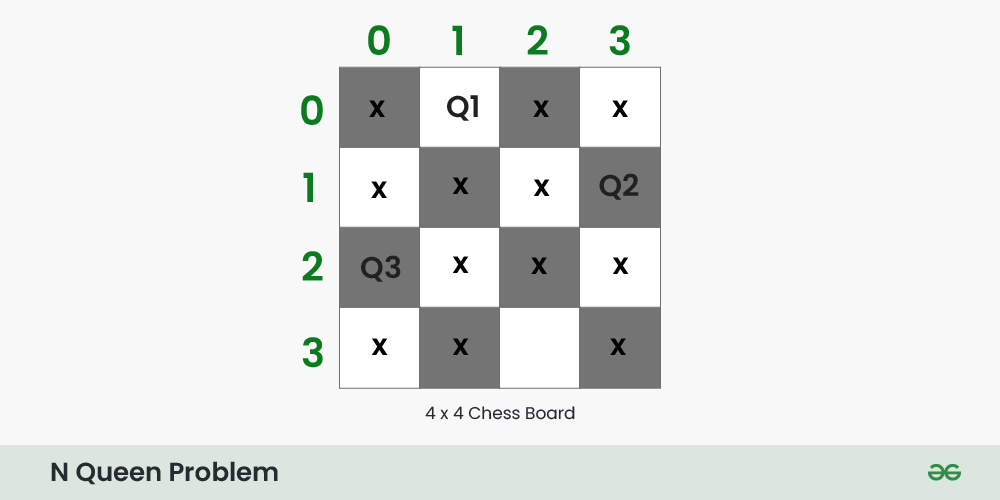
**Step 8:**

* Place Queen **Q2** at cell **(1 , 3)**, and move to next row.



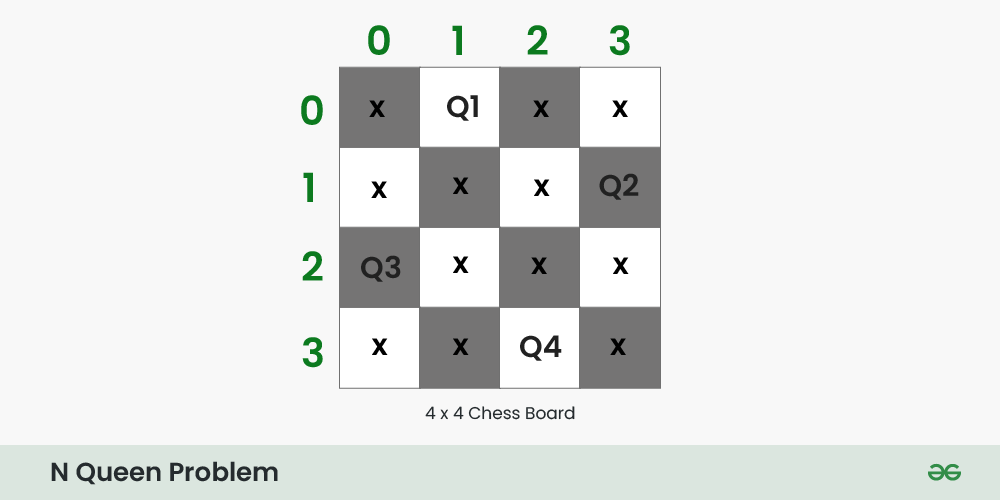
**Step 9:**

* Place Queen **Q3** at cell **(2 , 0)**, and move to next row.



**Step 10:**

* Place Queen **Q4** at cell **(3 , 2)**, and move to next row.
* This is one possible configuration of solution



Follow the steps below to implement the idea:

* Make a recursive function that takes the state of the board and the current row number as its parameter.
* Start in the topmost row.
* If all queens are placed, return **true**
* For every row.
  + Do the following for each column in current row.
    - If the queen can be placed safely in this column
      * Then mark this **[row, column]** as part of the solution and recursively check if placing queen here leads to a solution.
    - If placing the queen in **[row, column]** leads to a solution, return **true**.
    - If placing queen doesn’t lead to a solution then unmark this **[row, column]** and track back and try other columns.
* If all columns have been tried and nothing worked, return **false** to trigger backtracking.