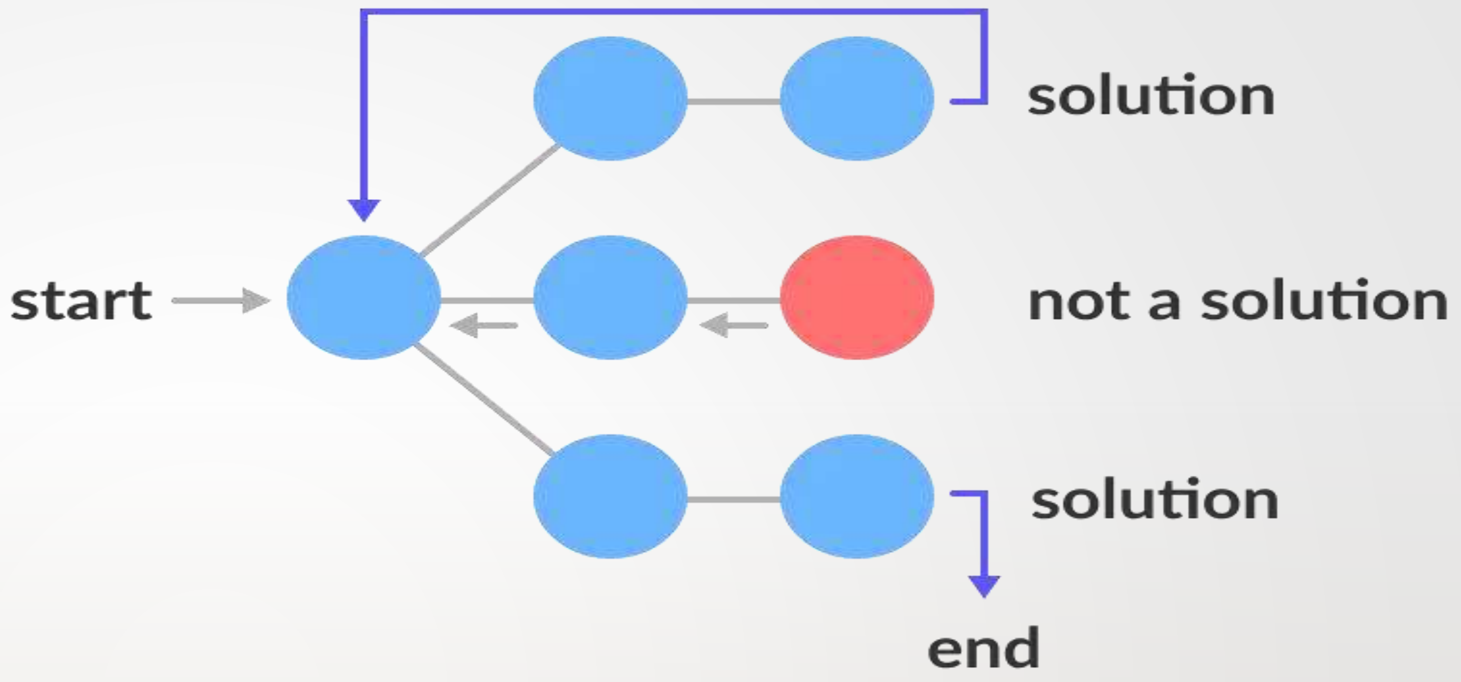
Backtracking Algorithm

- **A backtracking algorithm is a problem-solving algorithm that uses a brute force approach for finding the desired output.**
- The **Brute force approach tries out all the possible solutions** and chooses the desired/best solutions.
- The term backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach.
- **This approach is used to solve problems that have multiple solutions. If you want an optimal solution, you must go for dynamic programming.**

State Space Tree :

A space state tree is a tree representing **all the possible states (solution or nonsolution) of the problem** from the root as an initial state to the leaf as a terminal state.



Backtracking Algorithm

```
Backtrack(x)
    if x is not a solution
        return false
    if x is a new solution
        add to list of solutions
    backtrack(expand x)
```

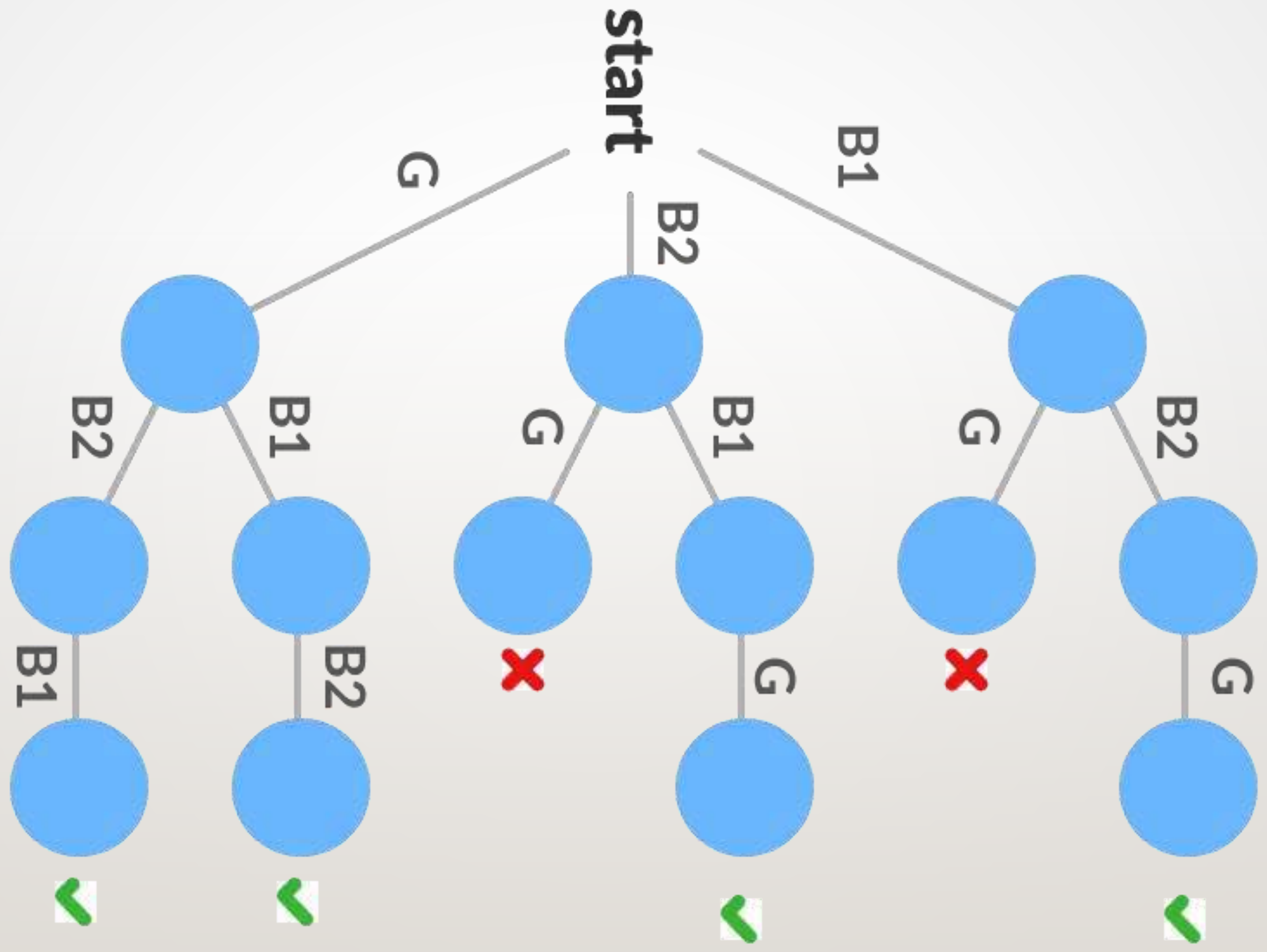
Example Backtracking Approach :

Problem: You want to find all the possible ways of **arranging 2 boys and 1 girl on 3 benches. Constraint: Girl should not be on the middle bench**.

Solution: There are a total of 3! = 6 possibilities. We will try all the possibilities and get the possible solutions. We recursively try all the possibilities.
All the possibilities are:

B1	B2	G	B2	G	B1
B1	G	B2	G	B1	B2
B2	B1	G	G	B2	B1

The following **state space tree** shows the possible solutions :



The terms related to the backtracking are:

Many problems can be solved by backtracking strategy, and that problems satisfy complex set of constraints, and these constraints are of two types:

- **Implicit constraint:** It is a rule in which how each element in a tuple is related.
- **Explicit constraint:** The rules that restrict each element to be chosen from the given set.

STATE SPACE TREE TERMINOLOGY

- **Problem state: Each node in the tree** organization defines a problem state.
- **State space tree:** If we represent **solution space in the form of a tree** then the tree is referred as the state space tree.
- **Solution Space: All tuples that satisfy the explicit constraints** define a possible solution space for a particular instance 'I' of the problem
- **Solution States:** These are those problem states S for which the path form the root to S define a tuple in the solution space. A solution state is **a node s for which each path from root node to node s together can represent a tuple in solution set.**
- **Answer States:** These **solution states** S for which the path from the root to S defines a tuple which is a member of the set of solution (i.e. **it satisfies the implicit constraints**) of the problem.
- **Live node**: A node which has been generated and all of whose **children have not yet been generated is live node**.
- **E-node:** The live nodes **whose children are currently being generated** is called E-node (node being expanded)
- **Dead node:** It is a generated node that is **either not to be expanded further or one for which all of its children has been generated.**
- **Bounding function:** It will be used to **kill live nodes without generating all their children**

Backtracking methodology Explicit Constraints

- Explicit constraints are rules that restrict each x_i takes values from a given set S_i.

- All solutions to the 4-queens problem can therefore be represented as 4-tuple (x_1, x_2, x_3, x_4), where x_i is the column on which queen Q_i is placed.

  Ex: $S_i = \{1, 2, 3, 4\}$ where $1 \le i \le 4$

- The explicit constraints depend on the particular instance I of the problem being solved.

  Ex: The placement of Q_2 depend upon the placement of Q_1

All tuples that satisfy the explicit constraints define a possible solution space for I. The solution space, therefore, consists of 4^4 (= 256) 4-tuples.

- All tuples that satisfy the explicit constraints define a possible solution space for I. Ex: $I_1 = \{x_2, x_4, x_1, x_3\}$

Backtracking methodology Implicit Constraints

- The implicit constraints are rules that determine which of the tuples in the solution space I satisfy the criterion function.
- Thus, implicit constraints describe the way in which the x_i must relate to each other.

Ex: (a) No two x_i's can be the same (i.e. all queens must be on different columns)

(b) No two queens on the same diagonal

(c) No two queens on the same row

The constraints (a) and (b) implies that all solutions are permutations of the 4-tuple (1, 2, 3, 4).

This realization reduces the size of the solution space from 4^4 to 4! tuples.

Difference between the Backtracking and Recursion

- **Recursion is a technique that calls the same function again and again until you reach the base case.**
- **Backtracking is an algorithm that finds all the possible solutions and selects the desired solution from the given set of solutions.**

- Backtracking is used when we have multiple solutions, and we require all those solutions.
- Backtracking name itself suggests that we are going back and coming forward; if it satisfies the condition, then return success, else we go back again. It is used to solve a problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criteria.

When to use a Backtracking algorithm?

When we have multiple choices, then we make the decisions from the available choices. In the following cases, we need to use the backtracking algorithm:

- A piece of sufficient information is not available to make the best choice, so we use the backtracking strategy to try out all the possible solutions.
- Each decision leads to a new set of choices. Then again, we backtrack to make new decisions. In this case, we need to use the backtracking strategy.

Backtracking Algorithm Applications:

- N-queens problem
- Sum of subset problem
- Graph coloring
- Hamiliton cycle

N-Queens Problem

- N - Queens problem is **to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.**

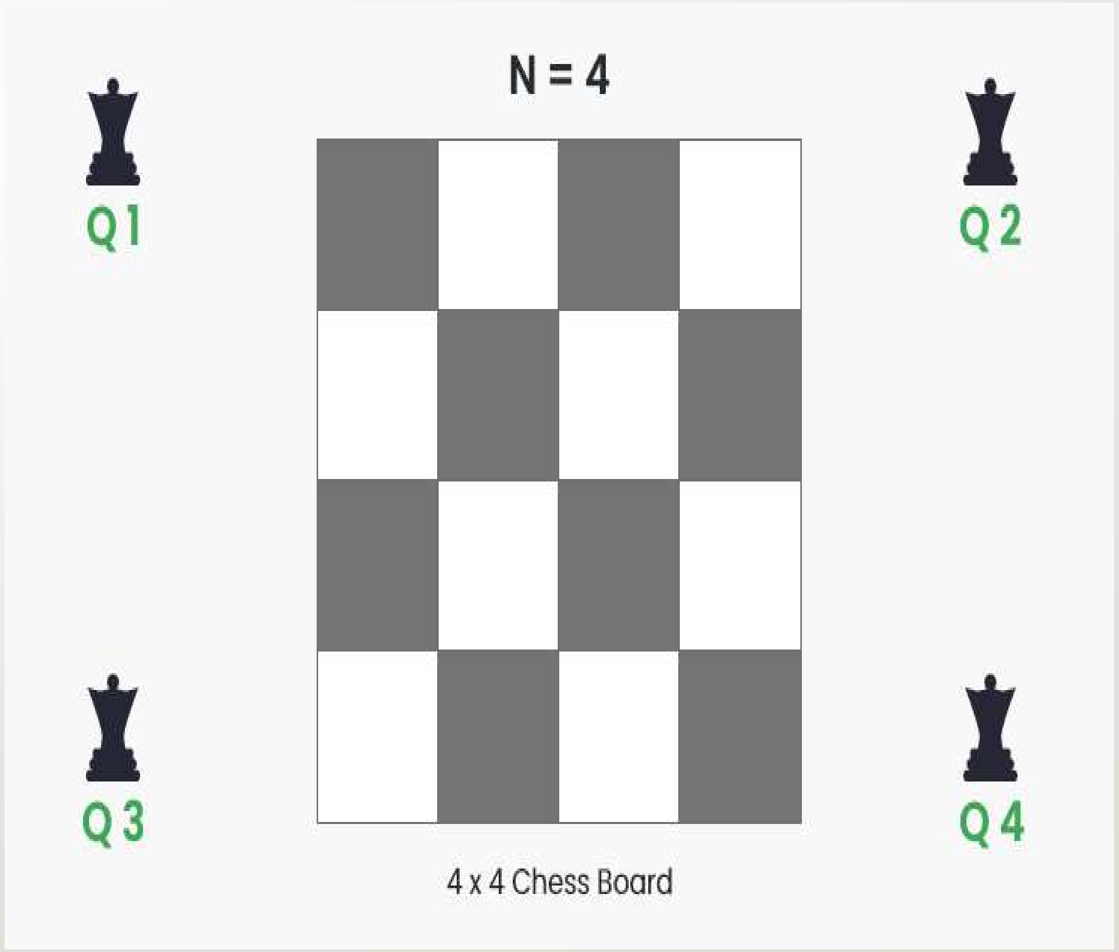
- It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3. So first we will consider the 4 queens problem and then generate it to n - queens problem.

- Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4.

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.**

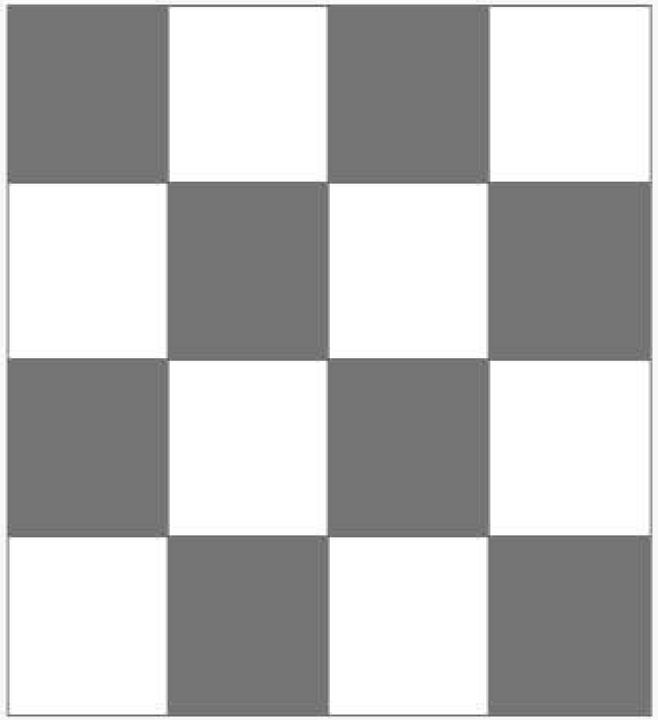
- We are going to look for the solution for n=4 on a 4 x 4 chessboard :



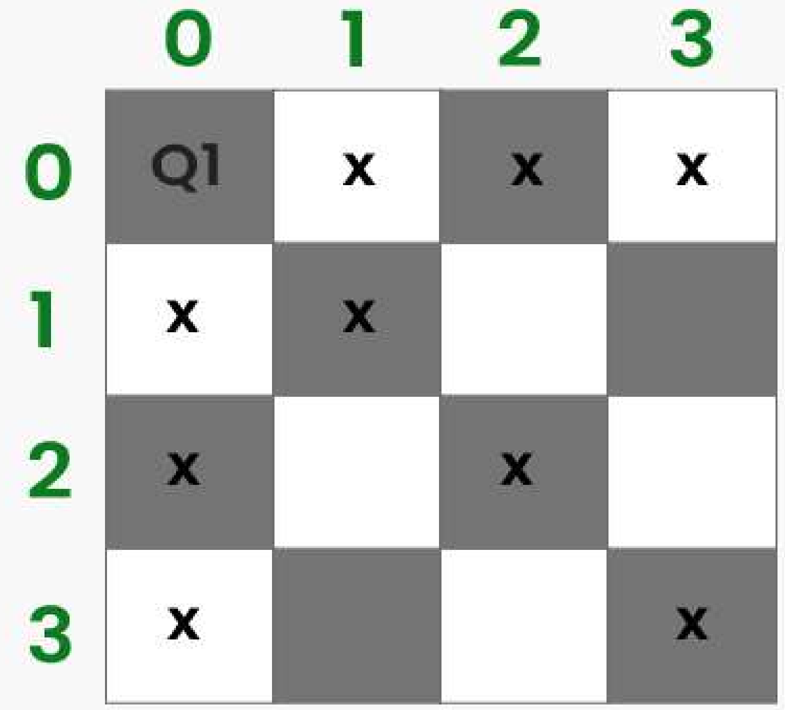
4 Queens Problem using Backtracking Algorithm:

- 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


4 x 4 Chess Board

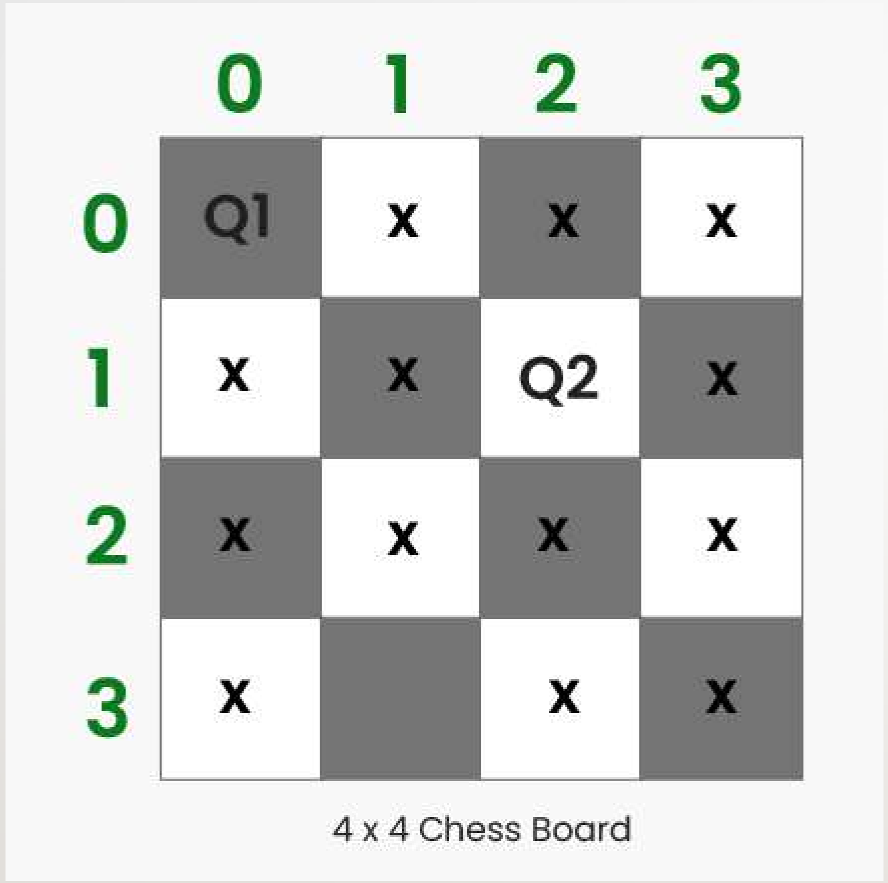
Step 0: Initialize a 4×4 board.



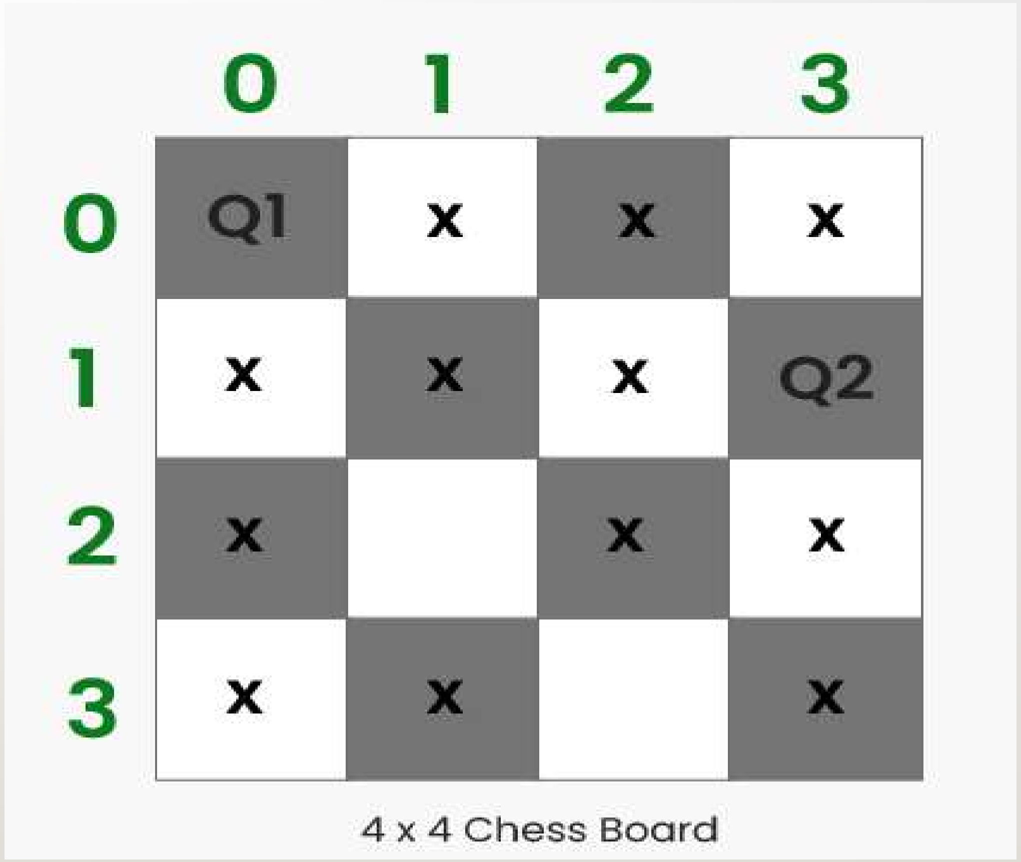
4 x 4 Chess Board

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 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).

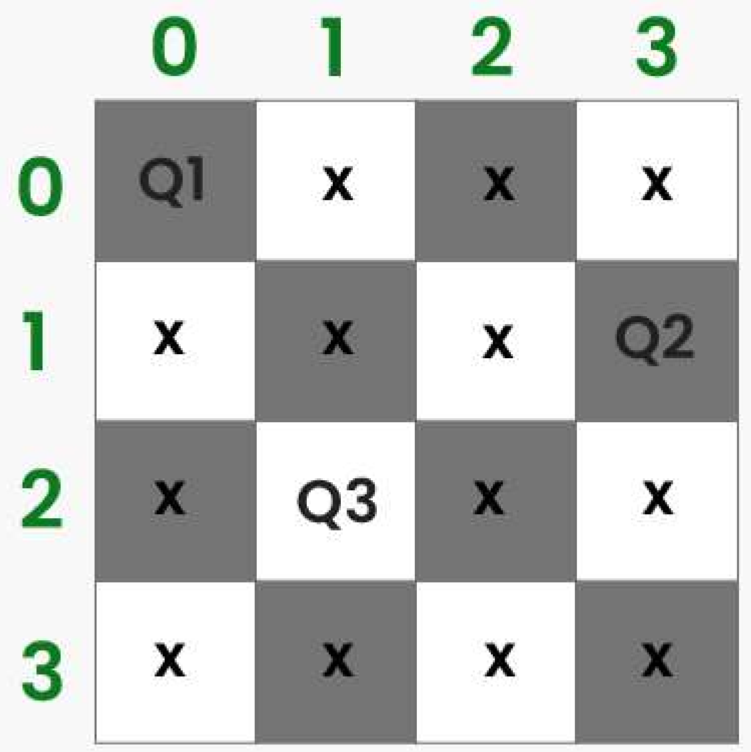


4 x 4 Chess Board



4 x 4 Chess Board

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

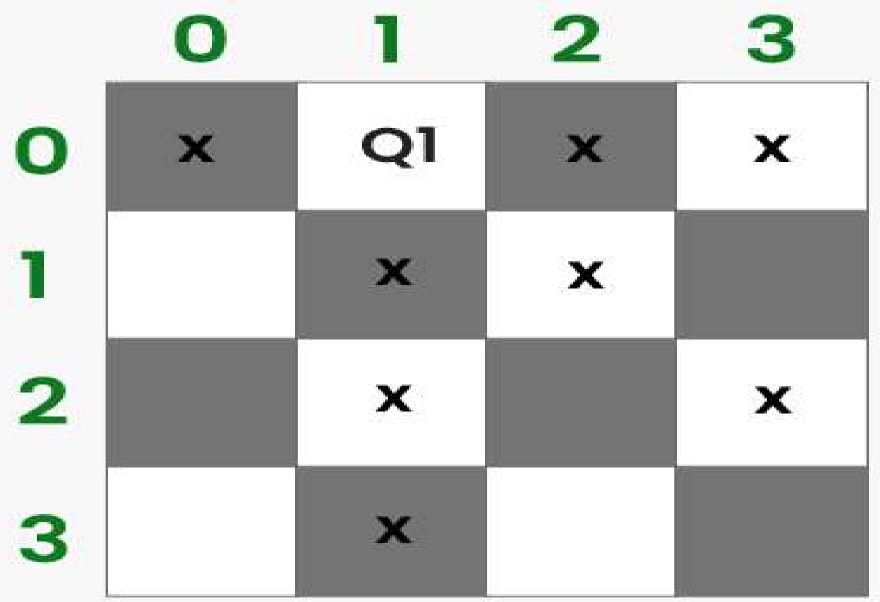


4 x 4 Chess Board

Step 5: Put queen (Q3) at cell (2, 1).

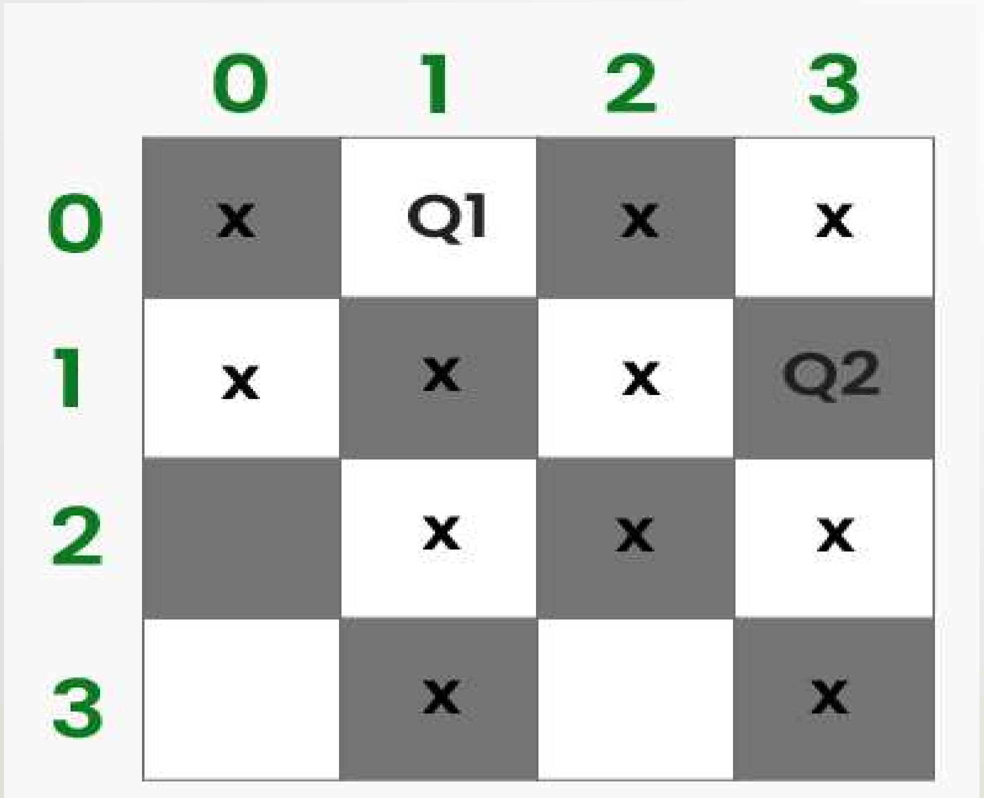
Step 6: There is no any cell to place Queen (Q4) at row 3. Backtrack and remove Queen (Q3) from row 2. Again there is no other safe cell in row 2, So backtrack again and remove queen (Q2) from row 1. Queen (Q1) will be remove from cell (0,0) and move to next safe cell i.e. (0 , 1).

Step 7: Place Queen Q1 at cell (0 , 1), and move to next row.



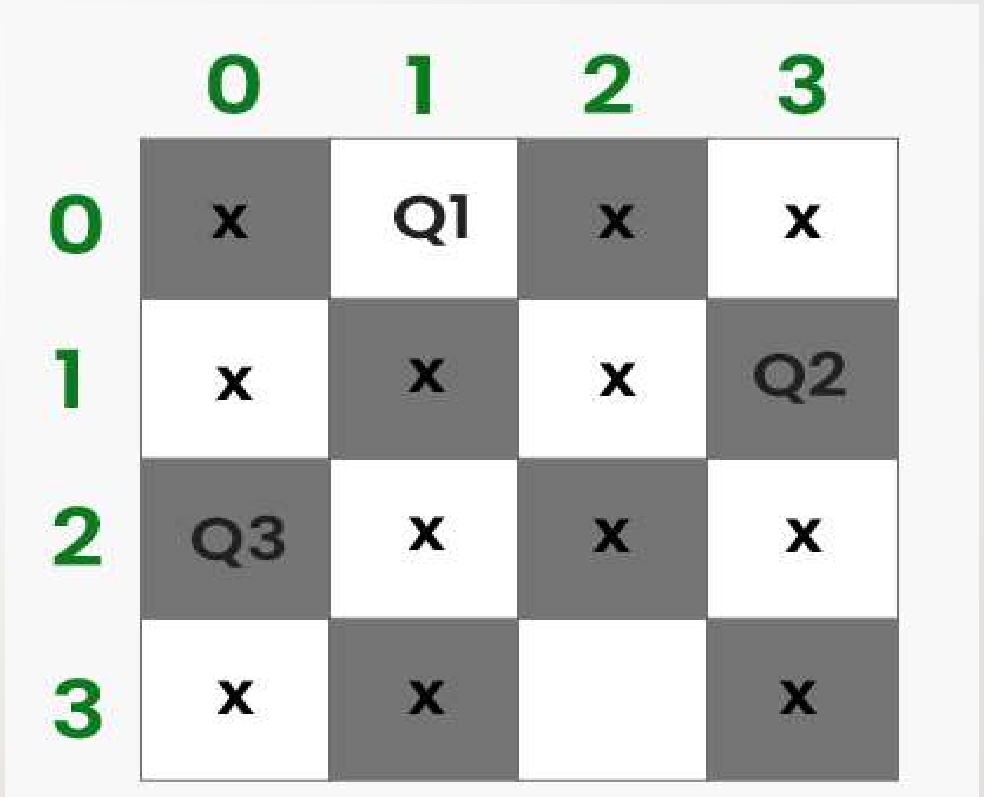
4 x 4 Chess Board

Step 8: Place Queen Q2 at cell (1 , 3), and move to next row.



	0	1	2	3
0	x	Q1	x	x
1	x	x	x	Q2
2		x	x	x
3		x		x

4 x 4 Chess Board

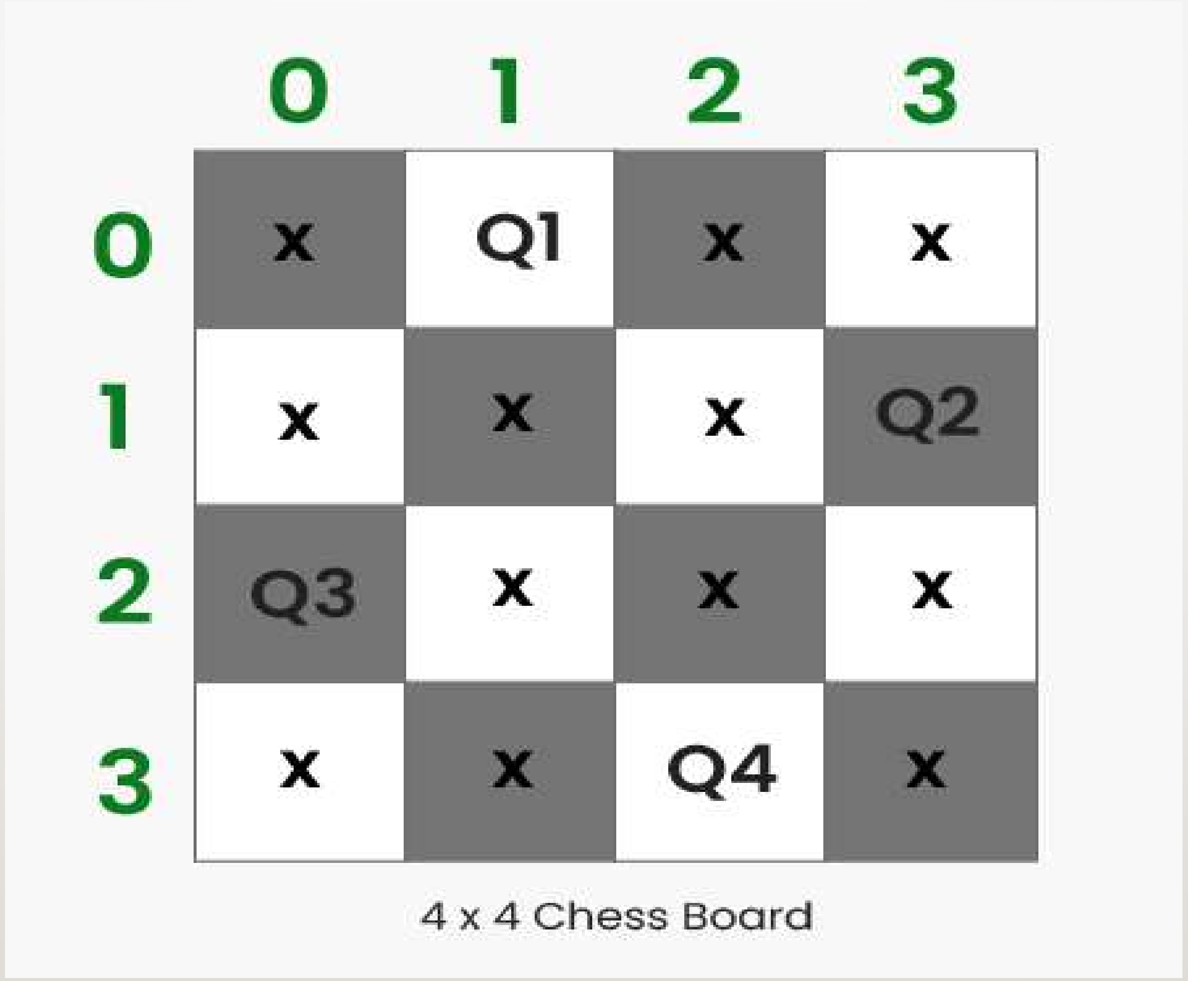


	0	1	2	3
0	x	Q1	x	x
1	x	x	x	Q2
2	Q3	x	x	x
3	x	x		x

4 x 4 Chess Board

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.



4 x 4 Chess Board

Q1	X	X	X
X	X		
X		X	
X			X

Step 1

Q1	X	X	X
X	X	Q2	X
X	X	X	X
X		X	X

Step 2

Q1	X	X	X
X	X	X	Q2
X		X	X
X	X		X

Step 3

Q1	X	X	X
X	X	X	Q2
X	Q3	X	X
X	X	X	X

Step 4

X	Q1	X	X
X	X	X	
	X		X
	X		

Step 5

X	Q1	X	X
X	X	X	Q2
	X	X	X
	X		X

Step 6

X	Q1	X	X
X	X	X	Q2
Q3	X	X	X
X	X		X

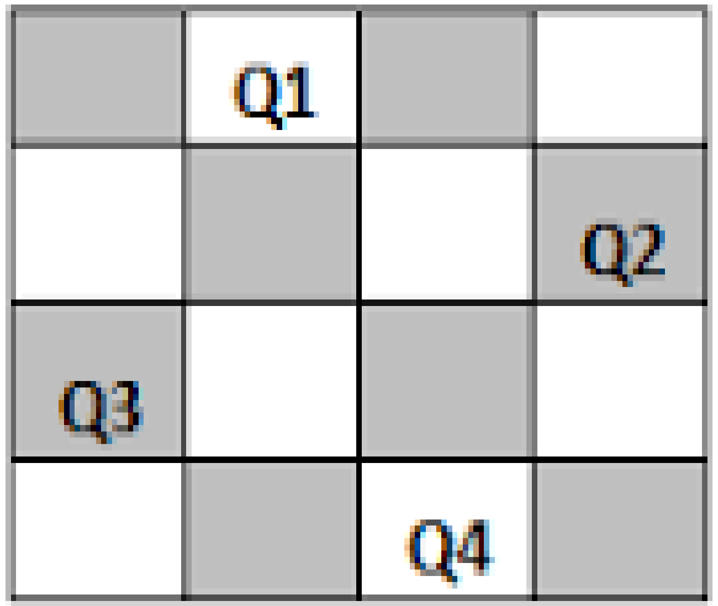
Step 7

X	Q1	X	X
X	X	X	Q2
Q3	X	X	X
X	X	Q4	X

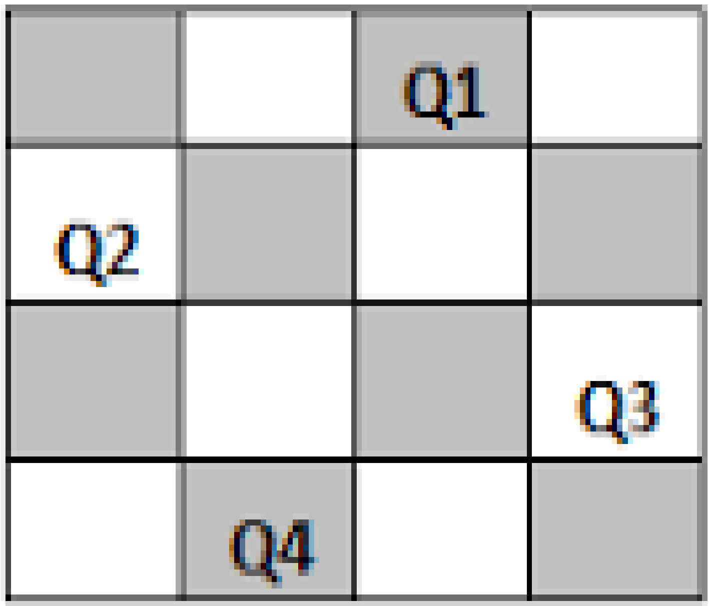
Step 8

	Q1		
			Q2
Q3			
		Q4	

4 Queens Problem Solution 1 - (2, 4, 1, 3)



Solution 1



Solution 2

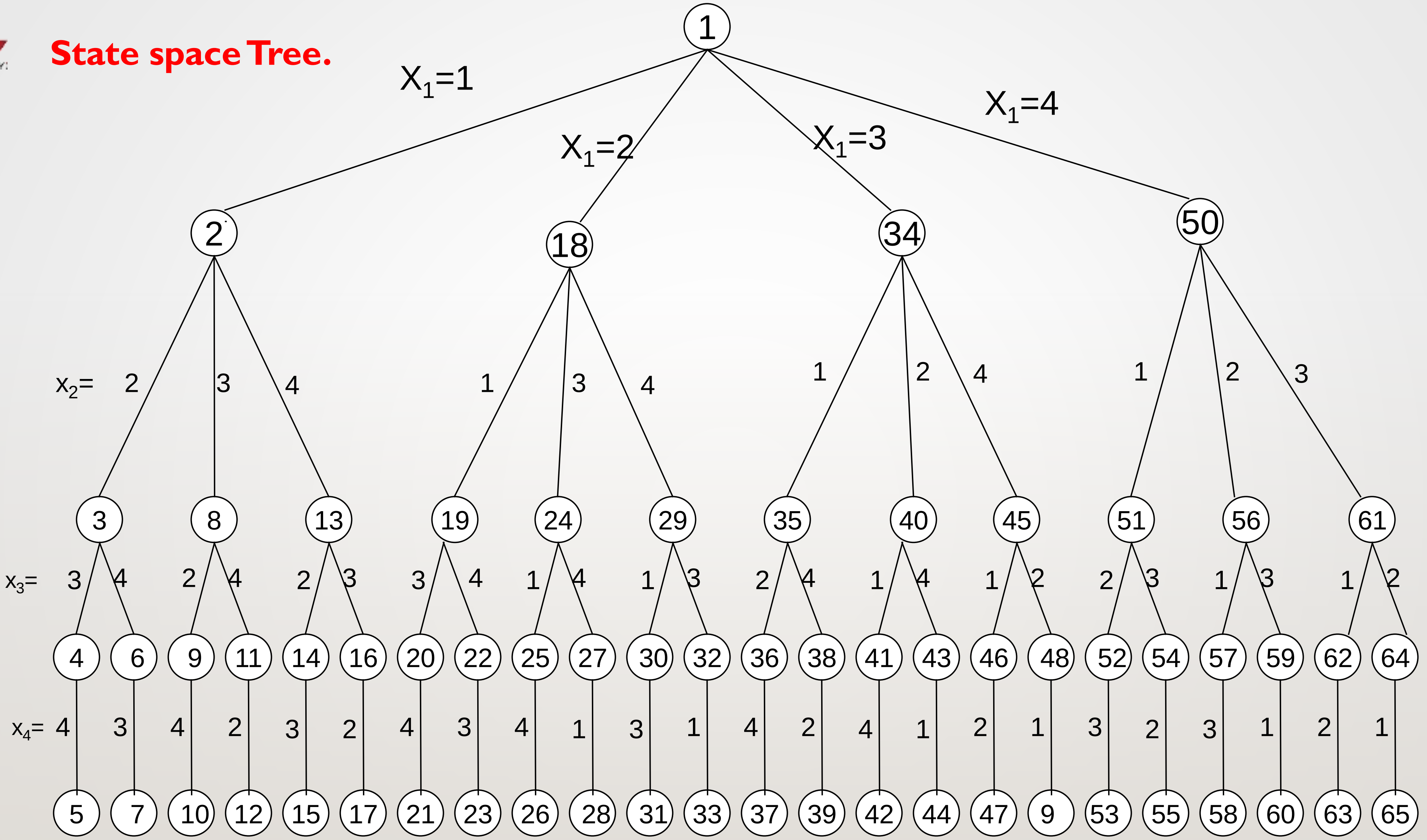
Follow the steps below to implement the idea:

- 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 :

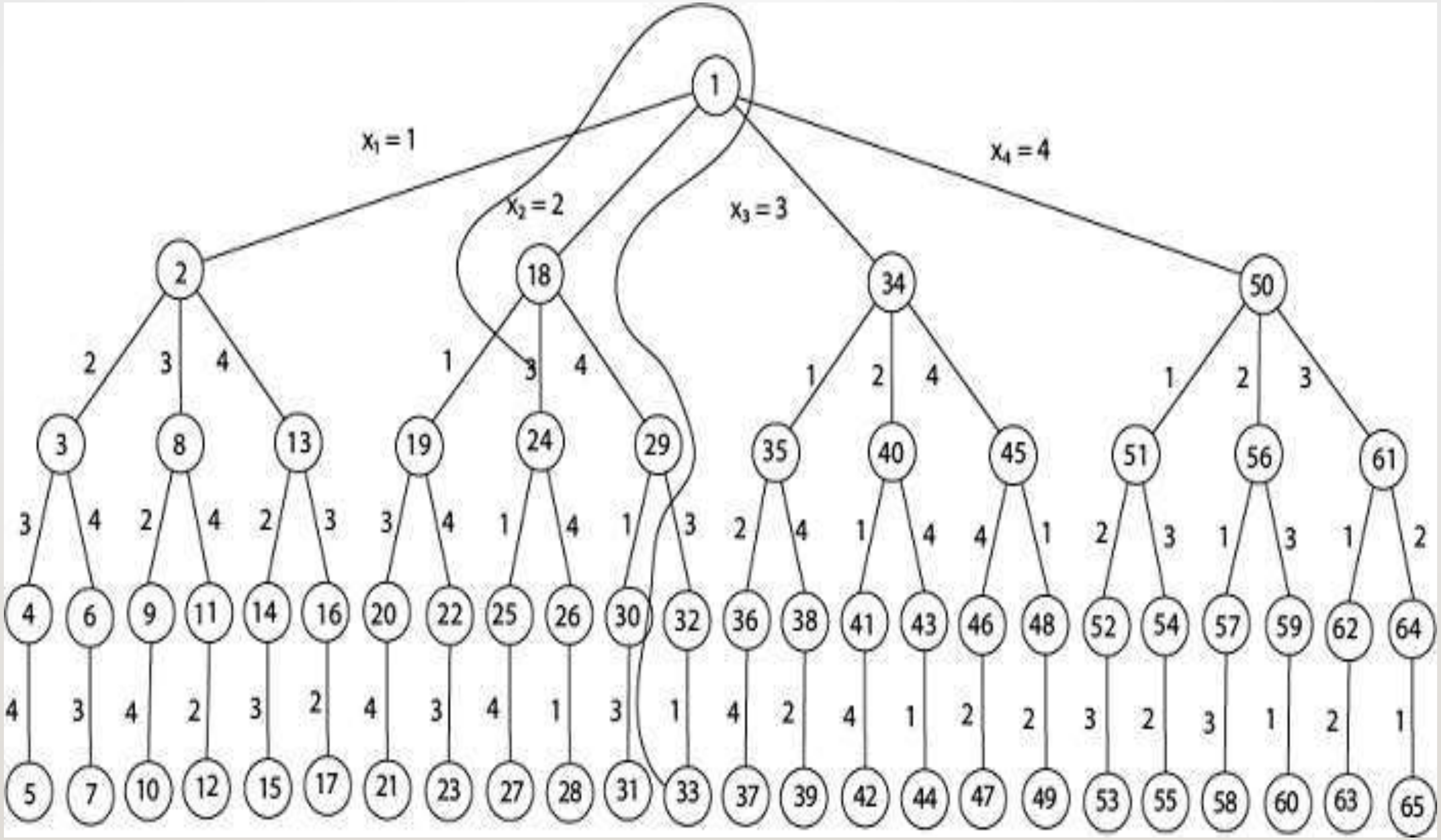
  Do the following for each column in current row.

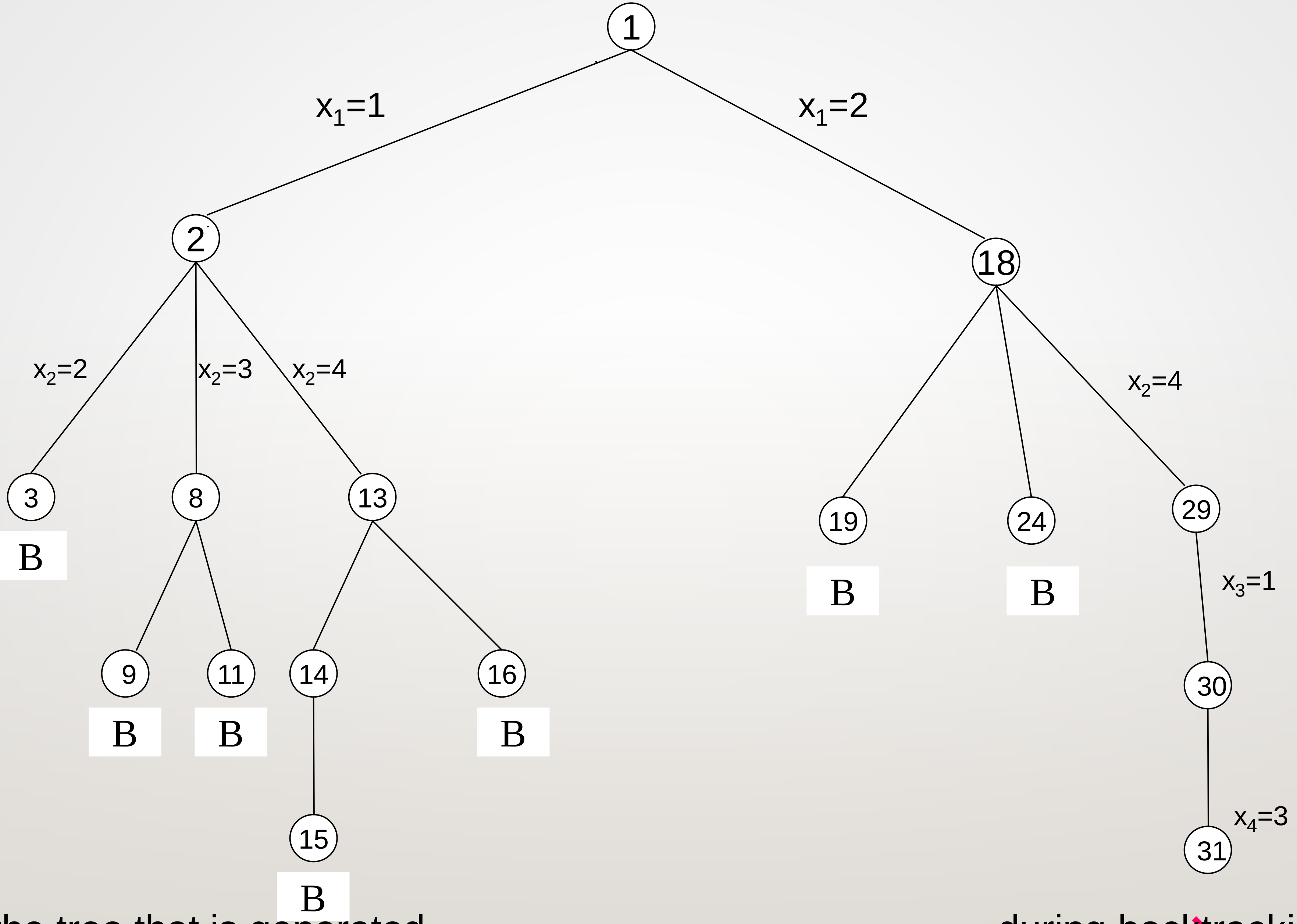
  - Then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
  - If placing the queen in [row, column] leads to a solution, return true.
  - If placing queen doesn't lead to a solution then unmark this [row, column] and track back and try other columns.

State space Tree.

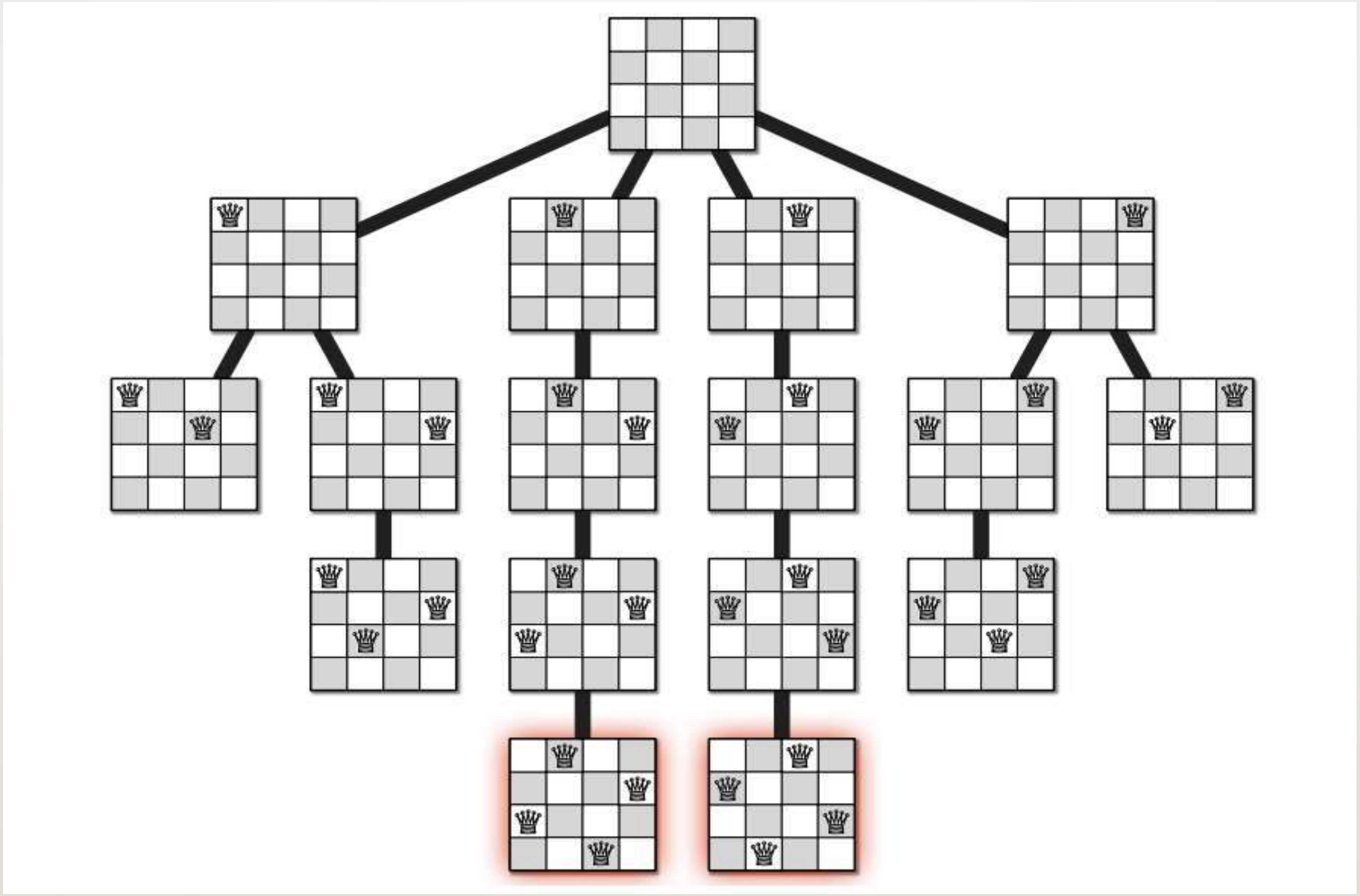


Tree organization of the 4-queens solution space. Nodes are numbered as in **depth first search.**

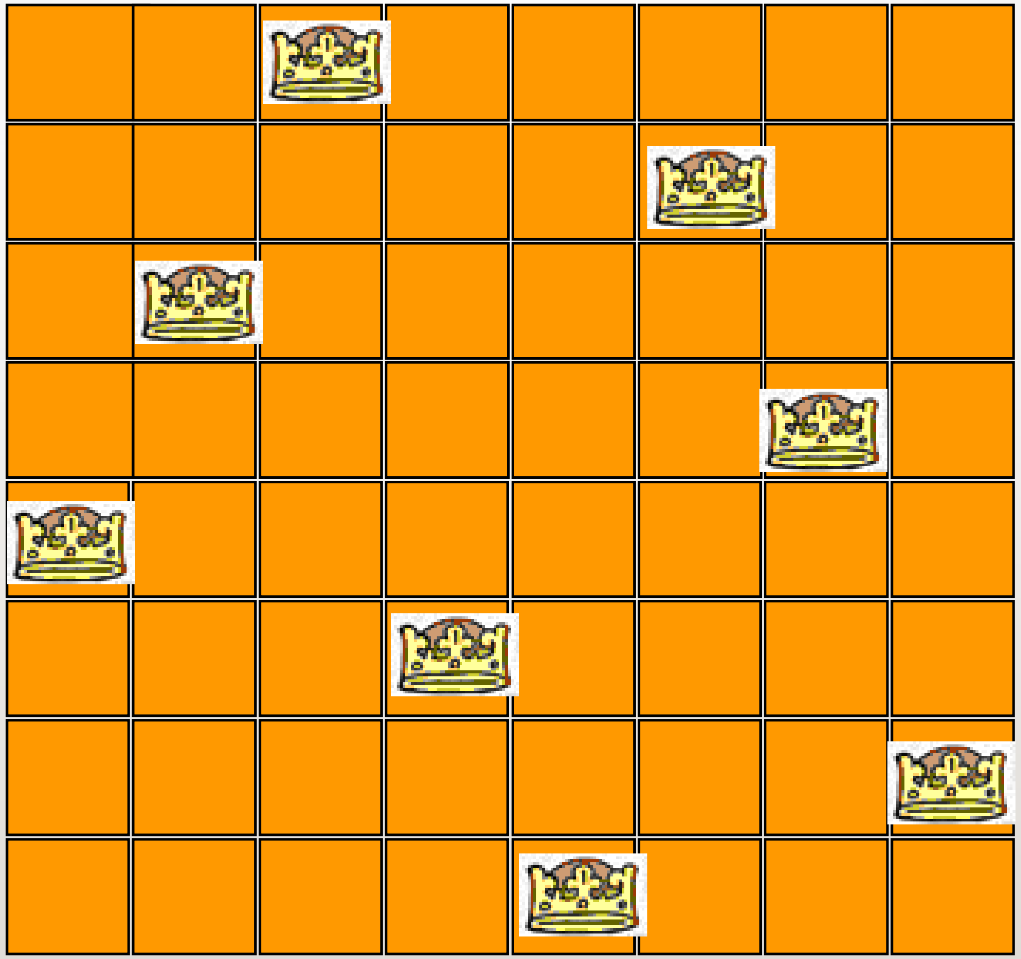

$x_1 = 1$
$x_2 = 2$
$x_3 = 3$
$x_4 = 4$
1 2 18 34 50
3 8 13 19 24 29 35 40 45 51 56 61
4 6 9 11 14 16 20 22 25 26 30 32 36 38 41 43 46 48 52 54 57 59 62 64
5 7 10 12 15 17 21 23 27 28 31 33 37 39 42 44 47 49 53 55 58 60 63 65

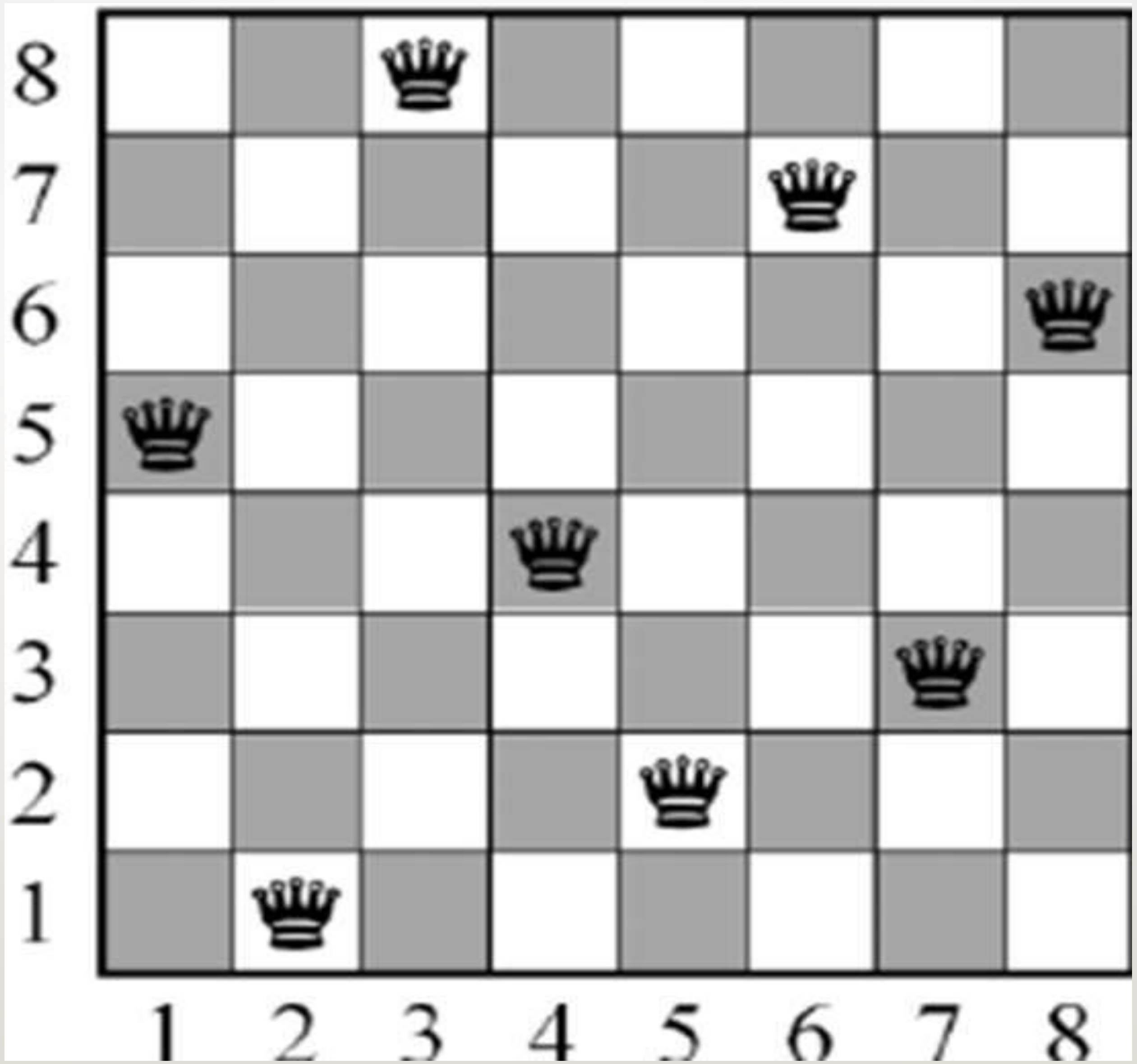


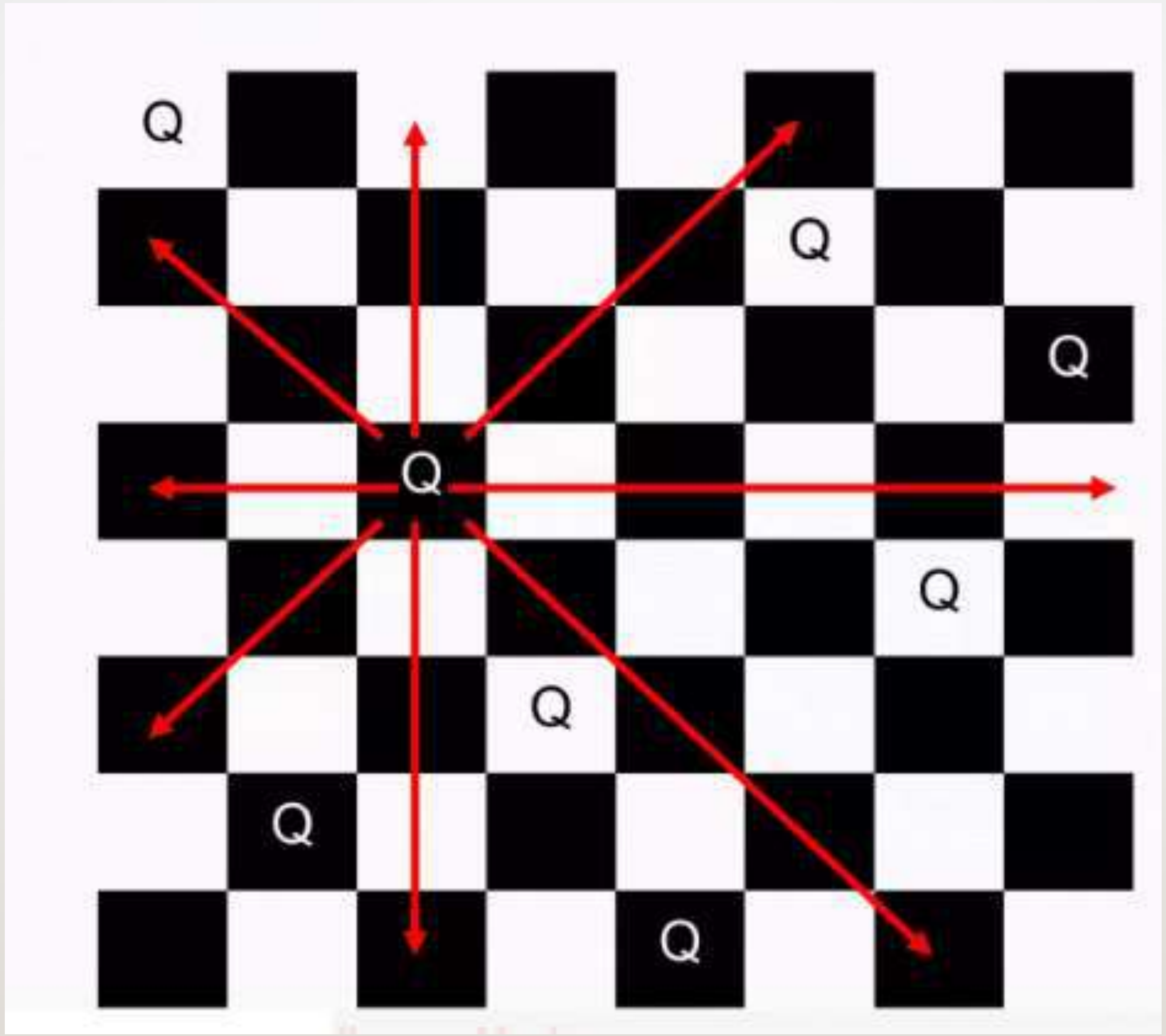
Portion of the tree that is generated during backtracking(n=4).

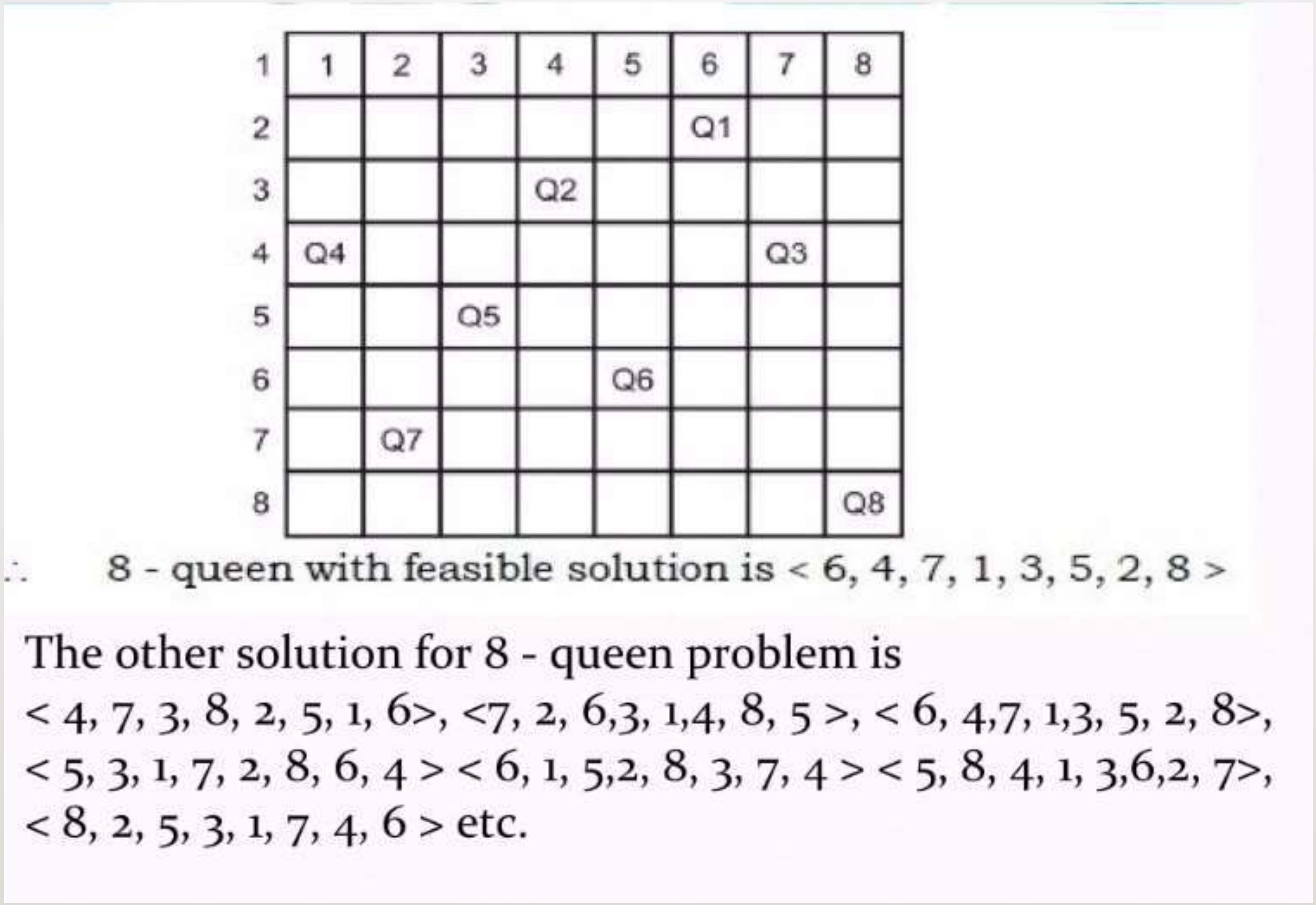


8-QUEENS PROBLEM




8
7
6
5
4
3
2
1
1
2
3
4
5
6
7
8


Q
Q
Q
Q
Q
Q
Q
Q



1	1	2	3	4	5	6	7	8
2						Q1		
3				Q2				
4	Q4						Q3	
5			Q5					
6					Q6			
7		Q7						
8								Q8

∴ 8 - queen with feasible solution is < 6, 4, 7, 1, 3, 5, 2, 8 >

The other solution for 8 - queen problem is
< 4, 7, 3, 8, 2, 5, 1, 6>, <7, 2, 6,3, 1,4, 8, 5 >, < 6, 4,7, 1,3, 5, 2, 8>,
< 5, 3, 1, 7, 2, 8, 6, 4 > < 6, 1, 5,2, 8, 3, 7, 4 > < 5, 8, 4, 1, 3,6,2, 7>,
< 8, 2, 5, 3, 1, 7, 4, 6 > etc.

n– queens : Defining the problem:-

- The problem is to place n queens on an n x n chessboard so that no two queens "attack" i.e.no two queens on the same row, column, or diagonal.

- Assume rows and columns of chessboard are numbered 1 through n. Queens also be numbered 1 through n.

- **Condition to ensure - No two queens on same Row**: Each queen must be on a different row ,hence queen i is to be placed on row i. Therefore, all solutions to the n-queens problem can be represented as n-tuples ($x_1, x_2, \ldots .x_n$), where x_i is the column on which queen i is placed.

- **Condition to ensure - No two queens on same Column**:

- Select a distinct value for each x_i

Condition to ensure - No two queens on same Diagonal

If two queens are placed at positions (i, j) and (k, l). They are on the same diagonal then following conditions hold good.

1) Every element on the same diagonal that runs from the upper left to the lower right has the same row – column value.

i – j = k – l -----(1) // (1,1) (2,2)

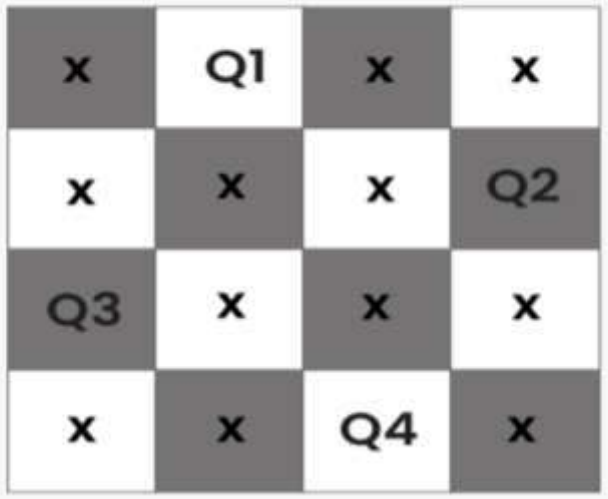
2) Similarly, every element on the same diagonal that goes from the upper right to the lower left has the same row + column value.

i + j = k + l -----(2) // (1,2) (2,1)

- First equation implies : j – l = i – k

  Second equation implies : j – l = k – i

- Therefore, two queens lie on the same diagonal iff **|j – l | = |i – k|**



4 x 4 Chess Board

```
Algorithm place( k,  i )
// It returns true if a queen can be placed in kth row and ith
// column . Otherwise it returns false.  x[] is a goal array whose
// first( k-1) values have been set. Abs(r) returns the absolute value of
   r.
   {
        for j = 1 to k-1 do
        {          // Two in the same column  or in the same diagonal
             if (  ( x [ j ] = i )   or    ( abs( x[ j ] - i ) = abs( j - k ) ))  then
                 return false;
        }
    return true ;
   }
```

```
Algorithm Nqueen(k, n)
// Using backtracking,  this procedure prints all possible placements
// of n queens on an n×n chessboard so that they are non-attacking.
{
        for i = 1 to n do        // check place of column for queen k
        {
                if place( k, i ) then
                {
                        x[ k ] = i;
                        if( k = n ) then write ( x[1:n] );
                        else  Nqueen( k+1, n);
                }
        }

}
```