|  |  |
| --- | --- |
| **AssignmentNo.** | **5** |
| **Title** | To Implement backtracking algorithm for N Queen Problem. |
| **PROBLEM STATEMENT/DEFINITION** | Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen‘s matrix. |
| **Objectives** | Understand and implement backtracking algorithm for N Queen Problem. |
| **Software packages and hardware apparatus used** | Technology : Java/Python  Ubuntu /Linux  Latest Version of 64 bit Operating Systems, Open Source Fedora-GHz. 8 G.B.  RAM, 500 G.B. HDD, 15"Color Monitor, Keyboard, Mouse |
| **References** | 1. <https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/> |
|  |  |

**Concepts Related Theory**

**What is Backtracking?**

**Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.**

**What is Backtracking Algorithm?**

**Backtracking is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).**

**Types of Backtracking Algorithm**

**There are three types of problems in backtracking –**

**Decision Problem – In this, we search for a feasible solution.**

**Optimization Problem – In this, we search for the best solution.**

**Enumeration Problem – In this, we find all feasible solutions.**

**When can be Backtracking Algorithm used?**

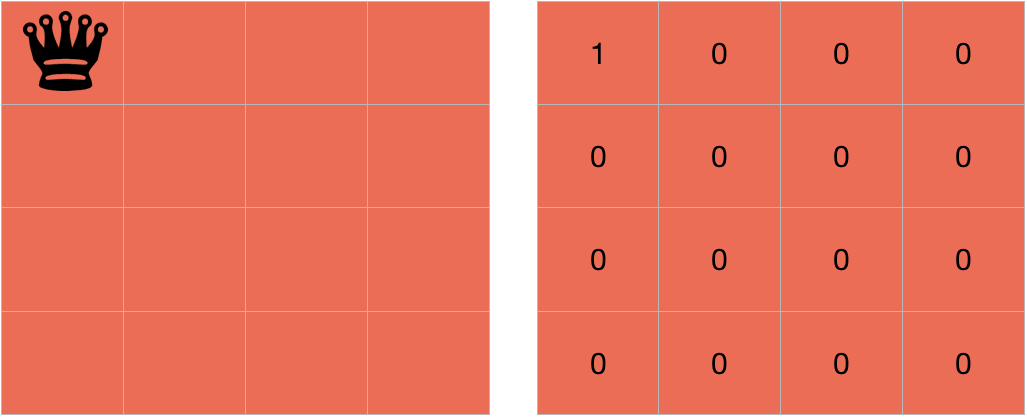
**For example, consider the Sudoku solving Problem, we try filling digits one by one. Whenever we find that current digit cannot lead to a solution, we remove it (backtrack) and try next digit. This is better than naive approach (generating all possible combinations of digits and then trying every combination one by one) as it drops a set of permutations whenever it backtracks.**

**Steps:**

Let's first write a function to check if a place is safe to put a queen or not.

We need to check if a cell (i, j) is under attack or not. For that, we will pass these two in our function along with the chessboard and its size - IS-ATTACK(i, j, board, N).

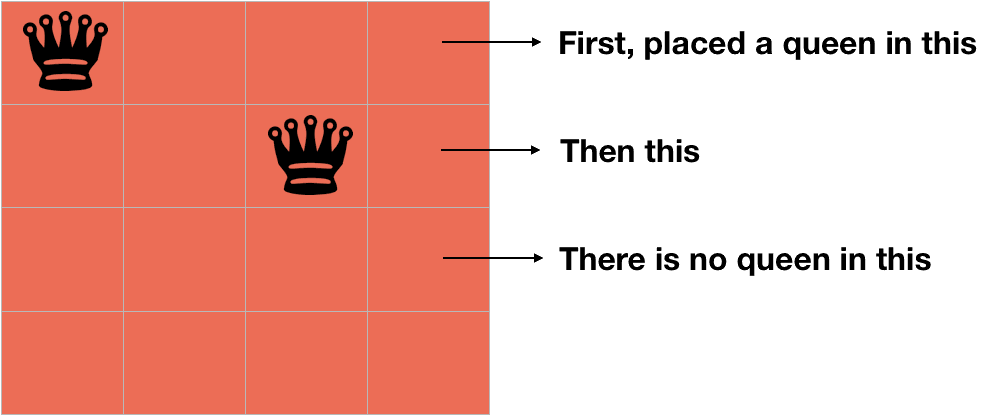
If there is a queen in a cell of the chessboard, then its value will be 1, otherwise, 0.



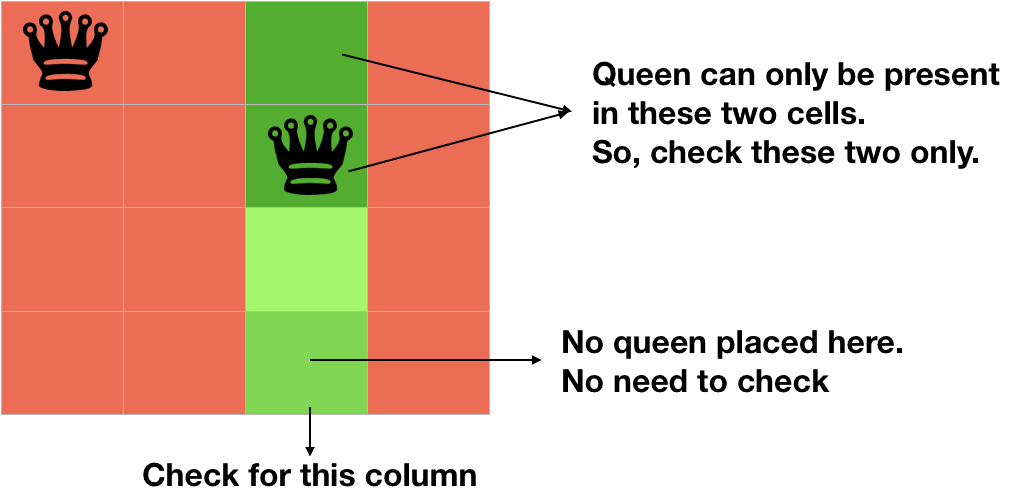
The cell (i,j) will be under attack in three condition - if there is any other queen in row i, if there is any other queen in the column j or if there is any queen in the diagonals.



We are already proceeding row-wise, so we know that all the rows above the current row(i) are filled but not the current row and thus, there is no need to check for row i.



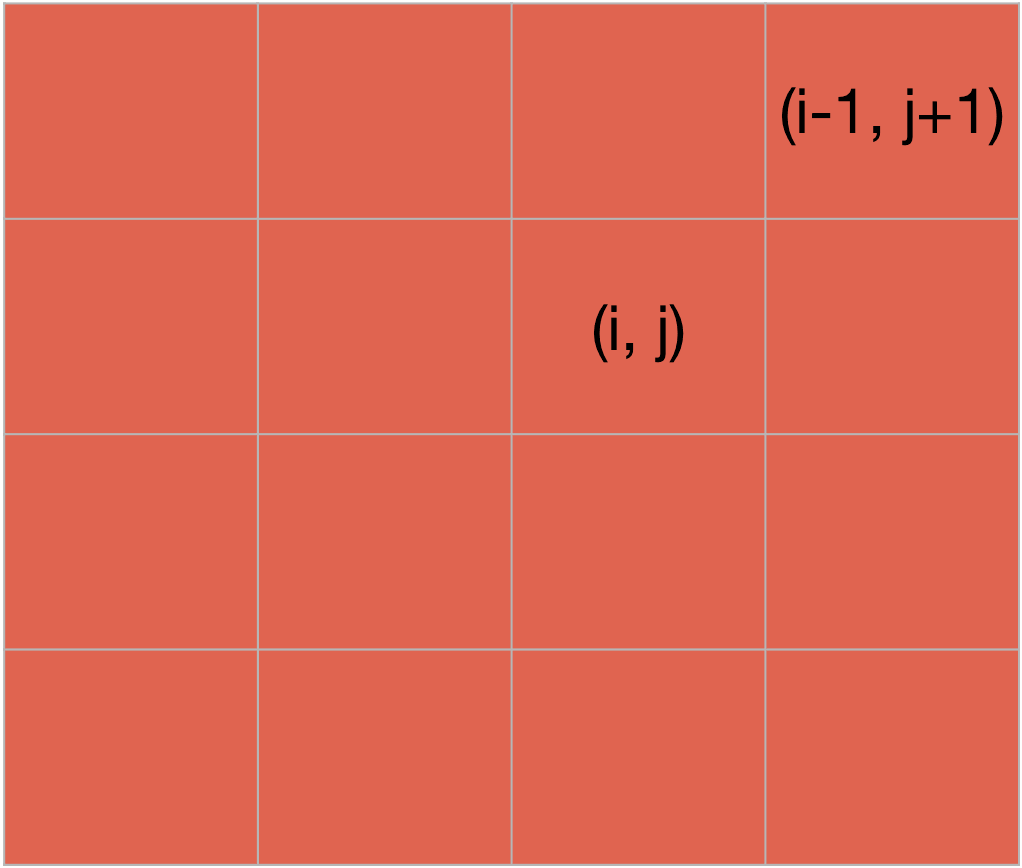
We can check for the column j by changing k from 1 to i-1 in board[k][j] because only the rows from 1 to i-1 are filled.



for k in 1 to i-1  
  if board[k][j]==1  
    return TRUE

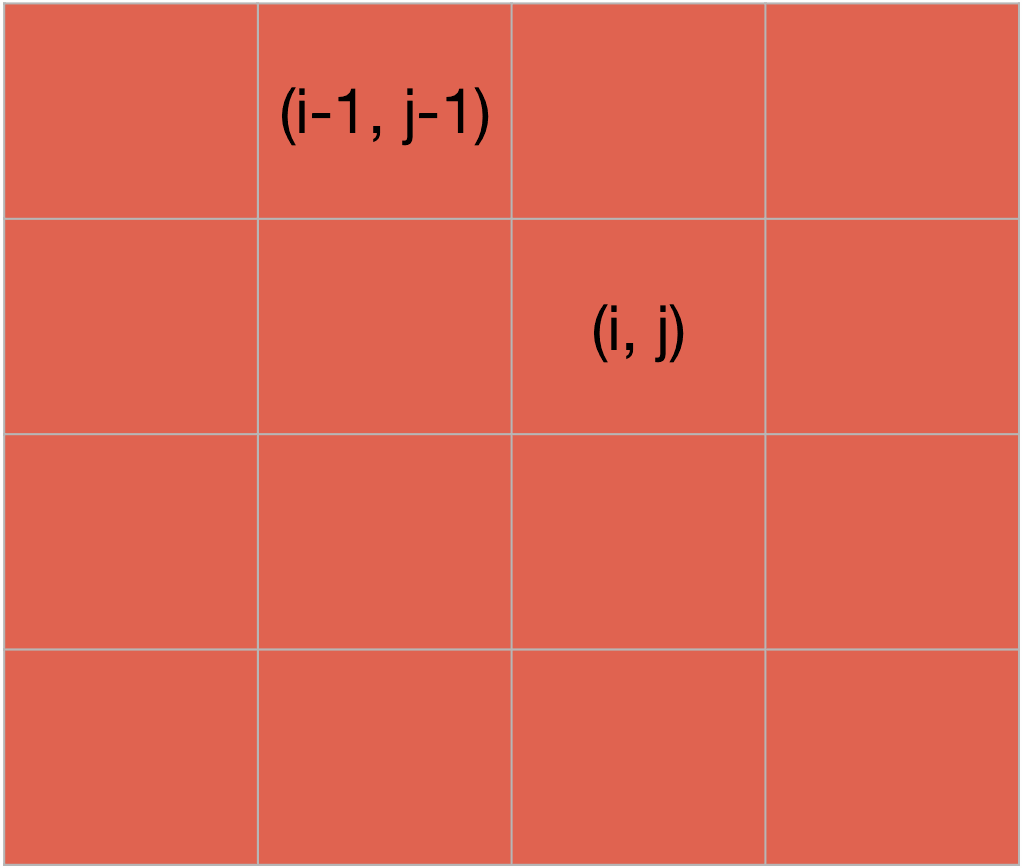
Now, we need to check for the diagonal. We know that all the rows below the row i are empty, so we need to check only for the diagonal elements which above the row i.

If we are on the cell (i, j), then decreasing the value of i and increasing the value of j will make us traverse over the diagonal on the right side, above the row i.



k = i-1  
l = j+1  
while k>=1 and l<=N  
  if board[k][l] == 1  
    return TRUE  
  k=k-1  
  l=l+1

Also if we reduce both the values of i and j of cell (i, j) by 1, we will traverse over the left diagonal, above the row i.



k = i-1  
l = j-1  
while k>=1 and l>=1  
  if board[k][l] == 1  
    return TRUE  
  k=k-1  
  l=l-1

At last, we will return false as it will be return true is not returned by the above statements and the cell (i,j) is safe

**Algorithm:**

N-QUEEN(row, n, N, board)

if n==0

return TRUE

for j in 1 to N

if !IS-ATTACK(row, j, board, N)

board[row][j] = 1

if N-QUEEN(row+1, n-1, N, board)

return TRUE

board[row][j] = 0 //backtracking, changing current decision

return FALSE

**Pseudocode:**

IS-ATTACK(i, j, board, N)

// checking in the column j

for k in 1 to i-1

if board[k][j]==1

return TRUE

// checking upper right diagonal

k = i-1

l = j+1

while k>=1 and l<=N

if board[k][l] == 1

return TRUE

k=k+1

l=l+1

// checking upper left diagonal

k = i-1

l = j-1

while k>=1 and l>=1

if board[k][l] == 1

return TRUE

k=k-1

l=l-1

return FALSE

## Conclusion:

Time Complexity: O(N!)   
Auxiliary Space: O(N)