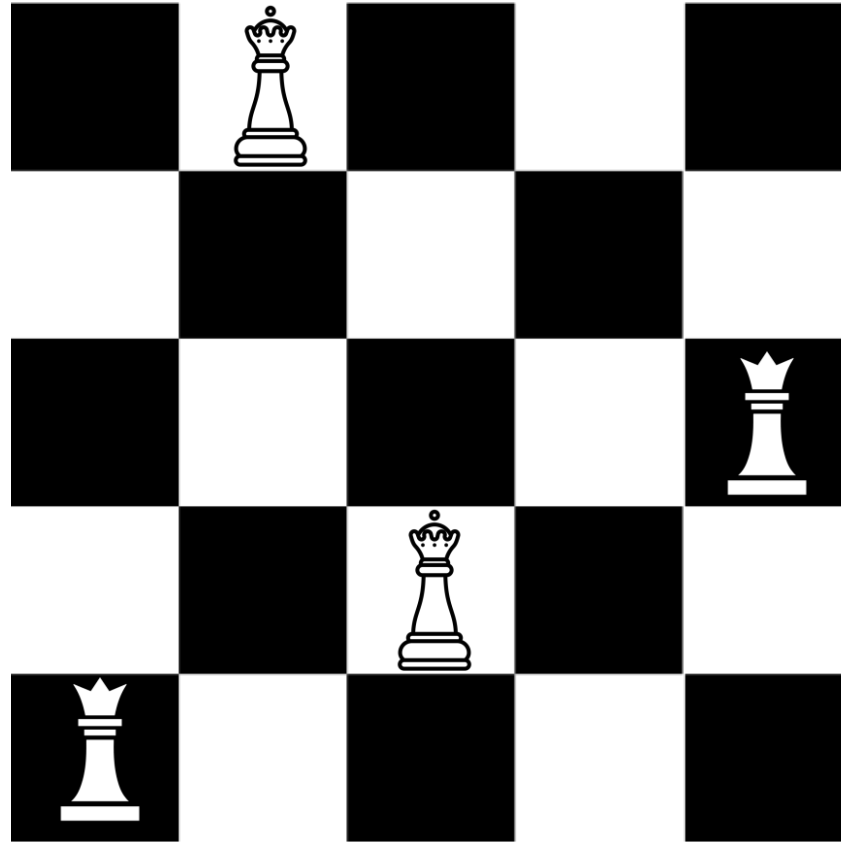


# Artificial Intelligence

**Solution of N-Queen**

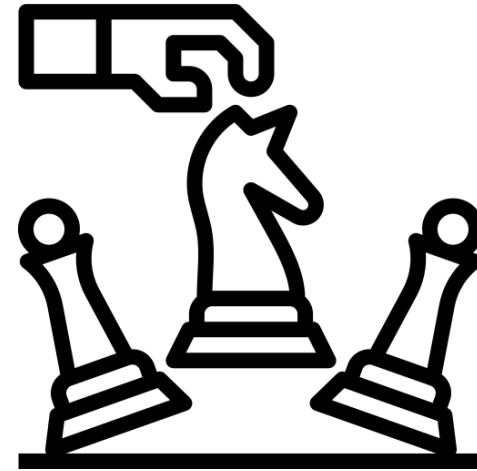
**Problem Using**

**Backtracking Algorithm**



# Team Members

- Amr Muhammad Gaber
- Muhammad Ahmed Abdel-Aziz
- Muhammad Ehab Muhammad Khalil



Group: 3

# N-Queen Problem

The N Queen is the problem of placing N chess queens on an  $N \times N$  chessboard so that no two queens attack each other

# Backtracking

Backtracking is an algorithmic paradigm that tries different solutions until finds a solution that “works”. Problems that are typically solved using the backtracking technique have the following property in common. These problems can only be solved by trying every possible configuration and each configuration is tried only once. A Naive solution for these problems is to try all configurations and output a configuration that follows given problem constraints. Backtracking works incrementally and is an optimization over the Naive solution where all possible configurations are generated and tried

## **Constraint Satisfaction problems**


Constraint satisfaction problems (CSPs) are mathematical questions defined as a set of objects whose state must satisfy several constraints or limitations.

Now no need to go all over again to illustrate the problem with boring details we are going to show you a simple example of N-Queen problem ,that would be :

(The 4-Queen problem ) we are going to solve this problem step by step to clarify the concept of N Queen problem and Backtracking

We start with empty 4x4 board that has 4 columns and 4 rows each from index 0 to 3 as follow:

Step 0 :  $c=0$   $r=0$  now we check the cell of board  $[0][0]$  if it safe or not board  $[0][0]$  =safe we put the first Queen then increment column by 1.

	0	1	2	3
0				
1				
2				
3				

Board[r][c]



Step 1.0 :  $c=0$   $r=0$  check ,board [0][1] != safe so we increment row by 1.

Step 1.1 :  $c=1$   $r= 1$  check ,board [1][1] != safe so we increment row by 1.

Step 1.2 :  $c=1$   $r= 2$  check ,board [2][1] = safe it 's safe

**so, we will place Queen.**

Now because we have already placed a Queen in this column,  
we increment column by 1

	0	1	2	3
0		Step1.0 X		
1		Step1.1 X		
2				
3				

Board[r][c]



Step 2.0 : c=2 r=0 check ,board [0][2] != safe so we increment row by 1.

Step 2.1: c=2 r=1 check ,board [1][2] != safe so we increment row by 1.



Step 2.2: c=2 r=1 check ,board [2][2] != safe so we increment row by 1.

Step 2.3: c=2 r=1 check ,board [3][2] != safe

We reached last row of column 2 and there is no safe position.

We have to backtrack

Note: we have to remove last placed Queen the one at position board [2][1] new row = 2+1 = 3 column = 1

	0	1	2	3
0		Step1.0 X	Step2.0 X	
1		Step1.1 X	Step2.1 X	
2			Step2.2 X	
3			Step2.3 X	

Board[r][c]

Step 1.3 :  $c=1$   $r=3$  check ,board [3][1] = safe

**so, we will place Queen,**





we increment column by 1.

Step 2.0 :  $c=2$   $r=0$  check ,board [0][2] != safe so we increment row by 1.

Step 2.1 :  $c=2$   $r=1$  check ,board [1][2] = safe

**so, we will place Queen,**

we increment column by 1.

	0	1	2	3
0		Step1.0 X	Step2.0 X	
1		Step1.1 X		
2				
3				

Board[r][c]

Step 3.0 : c=3 r=0 check ,board [0][3] != safe so we increment row by 1.

Step 3.1: c=3 r=1 check ,board [1][3] != safe so we increment row by 1.

Step 3.2: c=3 r=2 check ,board [2][3] != safe so we increment row by 1.





Step 3.3: c=3 r=3 check ,board [3][3] != safe

We reached last row of last column and there is no safe position.

We have to backtrack Note: we have to remove last placed Queen the one at position board [1][2]

new row = 1+1 = 2

column = 2

	0	1	2	3
0		Step1.0 X	Step2.0 X	Step3.0 X
1		Step1.1 X		Step3.1 X
2				Step3.2 X
3				Step3.3 X
Board[r][c]				

Step 2.2:  $c=2$   $r=2$  check ,board  $[2][2] \neq$  safe so we increment row by 1.

Step 2.3:  $c=2$   $r=3$  check ,board  $[3][2] \neq$  safe

We reached last row of last column and there is no safe position.

We have to backtrack.

Note: we have to remove last placed Queen the one at position board  $[3][1]$

Again, no more valid positions in this column we have to backtrack

Note: we have to remove last placed Queen the one at position board  $[0][0]$

new row =  $1+0 = 1$  column = 0

Step 0.1 :  $c=0$   $r=1$  check ,board  $[1][0] =$  safe,

**so, we will place Queen,**

So, we increment column by 1.

	0	1	2	3
0				
1	♔			
2				
3				

Board[r][c]

Step 1.0 :  $c=1$   $r=0$  check ,board  $[0][1] \neq$  safe so we increment row by 1.

Step 1.1 :  $c=1$   $r=1$  check ,board  $[1][1] \neq$  safe so we increment row by 1.

Step 1.2:  $c=1$   $r=2$  check ,board  $[2][1] \neq$  safe so we increment row by 1.

Step 1.3:  $c=1$   $r=3$  check ,board  $[3][1] =$  safe

**so, we will place Queen,**

so, we increment column by 1.




	0	1	2	3
0		Step1.0 X		
1	♔	Step1.1 X		
2		Step1.1 X		
3		♔		

Board[r][c]

Step 2.0 : c=2 r=0 check ,board [0][2] = safe

**so, we will place Queen,**

So, we increment column by 1

	0	1	2	3
0		Step1.0 X		
1		Step1.1 X		
2		Step1.1 X		
3				

Board[r][c]

Step 3.0 :  $c=3$   $r=0$  check ,board  $[0][3] \neq$  safe so we increment row by 1.

Step 3.1 :  $c=3$   $r=1$  check ,board  $[1][3] \neq$  safe so we increment row by 1.

Step 3.2 :  $c=3$   $r=2$  check ,board  $[2][3] =$  safe Place Queen.

Finally, all 4 Queens are placed at the board, so we reached or solution.

	0	1	2	3
0		Step1.0 X	♔	
1	♔	Step1.1 X		
2		Step1.1 X		♔
3		♔		

Board[r][c]

Now when we compare this result with our output there are exactly the same so that proves that our algorithm worked successfully

0	0	1	0
1	0	0	0
0	0	0	1
0	1	0	0

Our output

	0	1	2	3
0			♔	
1	♔			
2				♔
3		♔		

Board[r][c]

One of the expected  
solutions

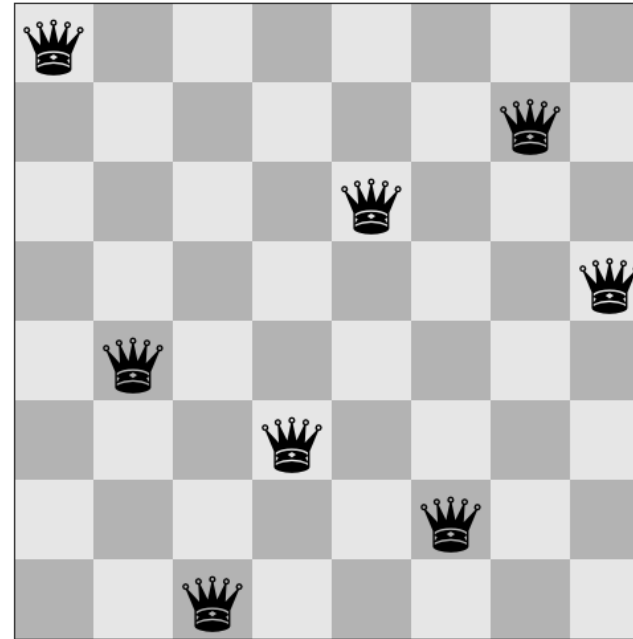


Now let's talk about the 8 Queen problem : 8 Queens problem has so many arrangements to check so we will only compare our output with one of the known solutions of the problem to see if our algorithm works or not

Now when we compare this result with our output there are exactly the same so that proves that our algorithm worked successfully

1	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0
0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0

Our output



One of the expected solutions

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.

And the next slides represent the basic functions of our code.

```
def printSolution(board):  
    for i in range(N):  
        for j in range(N):  
            print(board[i][j], end=" ")  
        print()
```

## Print Solution function

```
def isSafe(board, row, col):  
  
    for i in range(col):  
        if board[row][i] == 1:  
            return False  
  
    for i, j in zip(range(row, -1, -1),  
                    range(col, -1, -1)):  
        if board[i][j] == 1:  
            return False  
  
    for i, j in zip(range(row, N, 1),  
                    range(col, -1, -1)):  
        if board[i][j] == 1:  
            return False  
  
    return True
```

## isSafe Function

```
def solveNUtil(board, col):  
    if col >= N:  
        return True  
  
    for i in range(N):  
        if isSafe(board, i, col):  
            board[i][col] = 1  
            if solveNUtil(board, col + 1) == True:  
                return True  
            board[i][col] = 0  
  
    return False
```

## solveNUtil Function

```
def solveNQ():  
    board = [[0]*N for _ in range(N)]  
  
    if solveNQUtil(board, 0) == False:  
        print("Solution does not exist")  
        return False  
  
    printSolution(board)  
    return True
```

## solveNQ Function