

## PART A

### Experiment No.06

**A.1 Aim:** To Implement 8 queen puzzle

**A.2 Prerequisite:** Discrete structure

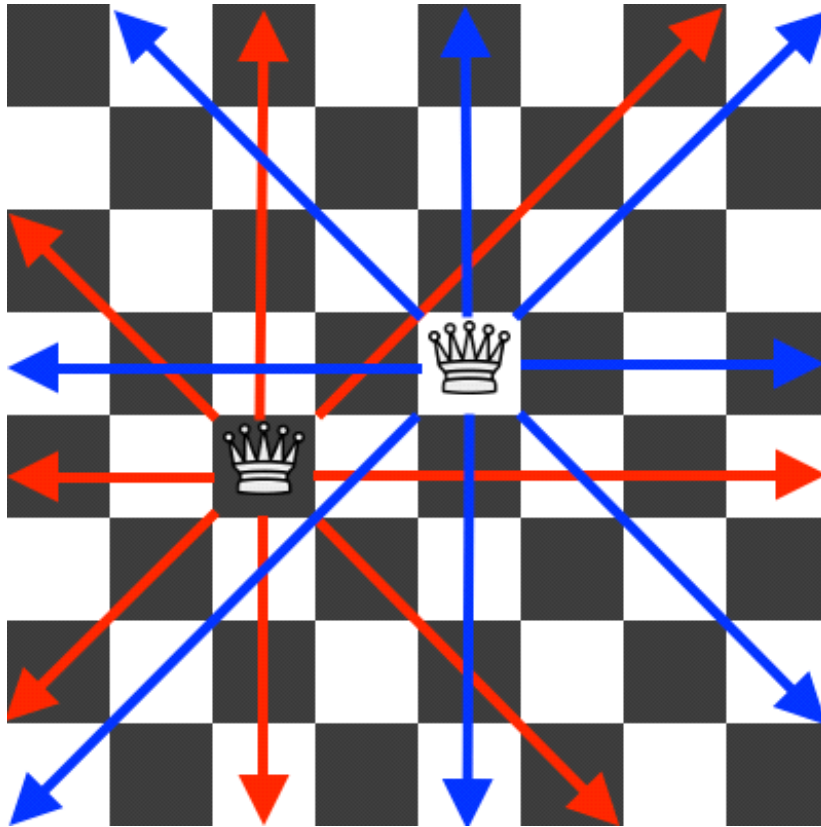
**A.3 Outcome:**

**After successful completion of this experiment students will be able to**

- Apply ideas of, problem solving by searching in AI
- **Tools Required:** Python

**A.4 Theory:**

The [eight](https://en.wikipedia.org/wiki/Eight_queens_puzzle) [HYPERLINK "https://en.wikipedia.org/wiki/Eight\\_queens\\_puzzle" HYPERLINK](https://en.wikipedia.org/wiki/Eight_queens_puzzle)  
["https://en.wikipedia.org/wiki/Eight\\_queens\\_puzzle"](https://en.wikipedia.org/wiki/Eight_queens_puzzle) [HYPERLINK](https://en.wikipedia.org/wiki/Eight_queens_puzzle)  
["https://en.wikipedia.org/wiki/Eight\\_queens\\_puzzle"](https://en.wikipedia.org/wiki/Eight_queens_puzzle)queens [HYPERLINK](https://en.wikipedia.org/wiki/Eight_queens_puzzle)  
["https://en.wikipedia.org/wiki/Eight\\_queens\\_puzzle"](https://en.wikipedia.org/wiki/Eight_queens_puzzle) [HYPERLINK](https://en.wikipedia.org/wiki/Eight_queens_puzzle)  
["https://en.wikipedia.org/wiki/Eight\\_queens\\_puzzle"](https://en.wikipedia.org/wiki/Eight_queens_puzzle) [HYPERLINK](https://en.wikipedia.org/wiki/Eight_queens_puzzle)  
["https://en.wikipedia.org/wiki/Eight\\_queens\\_puzzle"](https://en.wikipedia.org/wiki/Eight_queens_puzzle) puzzle, or the eight queens problem, asks how to place eight queens on a chessboard without attacking each other. If you never played chess before, a queen can move in any direction (horizontally, vertically and diagonally) any number of places. In the next figure, you can see two queens with their attack patterns:



At the end of the article we present a Python 3 solution to the eight queens puzzle.

We can generate a solution to the problem by scanning each row of the board and placing one queen per column, while checking at every step, that no two queens are in the line of attack of the other. A brute force approach to the problem will be to generate all possible combinations of the eight queens on the chessboard and reject the invalid states. How many combinations of 8 queens on a 64 cells chessboard are possible ?

The [combinations formula](#) is

$$C(n,k) = \frac{n!}{k!(n-k)!}$$

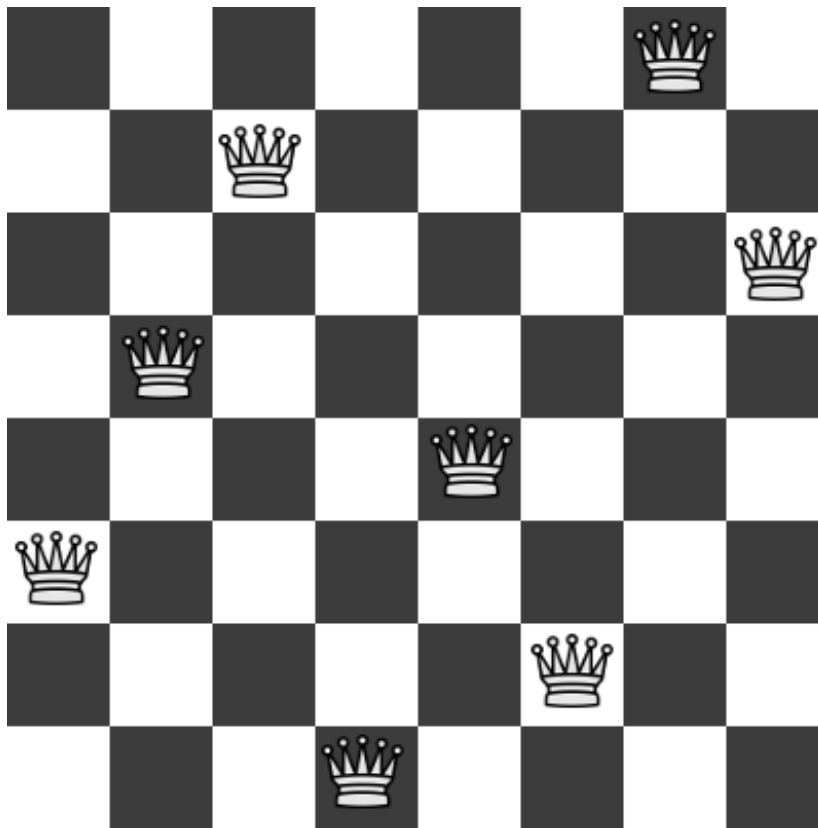
which, for our particular case is:

$$C(64,8) = \frac{64!}{8!(64-8)!} = 4,426,165,368$$

Clearly, the brute force approach is not practical!

We can further reduce the number of potential solutions if we observe that a valid solution can have only one queen per row, which means that we can represent the board as an array of eight elements, where each entry represents the column position of the queen from a particular row.

Take as an example the next solution of the problem:



The queens positions on the above board, can be represented as the occupied positions of a two dimensional 8x8 array: [0, 6], [1, 2], [2, 7], [3, 1], [4, 4], [5, 0], [6, 5], [7, 3]. Or, as described above, we can use a one dimensional 8 elements array: [6, 2, 7, 1, 4, 0, 5, 3].

If we look closely at the example solution [6, 2, 7, 1, 4, 0, 5, 3], we note that a potential solution to the eight queens puzzle can be constructed by generating all possible permutations of an array of eight numbers, [0, 1, 2, 3, 4, 5, 6, 7], and rejecting the invalid states (the ones in which any two queens can attack each other). The number of all permutations of  $n$  unique objects is  $n!$ , which for our particular case is:

$$n! = 8! = 40,320$$

which is more reasonable than the previous 4,426,165,368 situations to analyze for the brute force approach.

A slightly more efficient solution to the puzzle uses a recursive approach: assume that we've already generated all possible ways to place  $k$  queens on the first  $k$  rows. In order to generate the valid positions for the  $k+1$  queen we place a queen on all columns of row  $k+1$  and we reject the invalid states. We do the above steps until all eight queens are placed on the board. This approach will generate all 92 distinct solutions for the eight queens puzzle.

## Algorithm

### isValid(board, row, col)

**Input:** *The chess board, row and the column of the board.*

**Output** – True when placing a queen in row and place position is a valid or not.

Begin

```
    if there is a queen at the left of current col, then
        return false
    if there is a queen at the left upper diagonal, then
        return false
    if there is a queen at the left lower diagonal, then
        return false;
    return true //otherwise it is valid place
```

End

### solveNQueen(board, col)

**Input** – The chess board, the col where the queen is trying to be placed.

**Output** – The position matrix where queens are placed.

Begin

```
    if all columns are filled, then
        return true
    for each row of the board, do
        if isValid(board, i, col), then
            set queen at place (i, col) in the board
            if solveNQueen(board, col+1) = true, then
                return true
            otherwise remove queen from place (i, col) from board.
    done
    return false
```

End

## PART B

(PART B : TO BE COMPLETED BY STUDENTS)

Roll. No.B30	Name: Bhatt Pranjal Deepak
Class: TE B COMPS	Batch:B2
Date of Experiment: 3/3/25	Date of Submission: 3/3/25
Grade:	

### **B.1 Software Code written by student:**

```
def solveNQueens(board, col, N):
    # If all queens are placed, return True
    if col == N:
        printSolution(board, N)
        return True

    # Try placing this queen in all rows one by one
    for i in range(N):
        if isSafe(board, i, col, N):
            board[i][col] = 1 # Place the queen

            # Recur to place the rest of the queens
            if solveNQueens(board, col + 1, N):
                return True

        # If placing queen in board[i][col] doesn't lead to a solution,
backtrack
        board[i][col] = 0

    return False # If the queen cannot be placed in any row in this
column

def isSafe(board, row, col, N):
    # Check this row on the left side
    for x in range(col):
        if board[row][x] == 1:
            return False

    # Check upper diagonal on the left side
    for x, y in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[x][y] == 1:
            return False

    # Check lower diagonal on the left side
    for x, y in zip(range(row, N, 1), range(col, -1, -1)):
        if board[x][y] == 1:
            return False

    return True
```

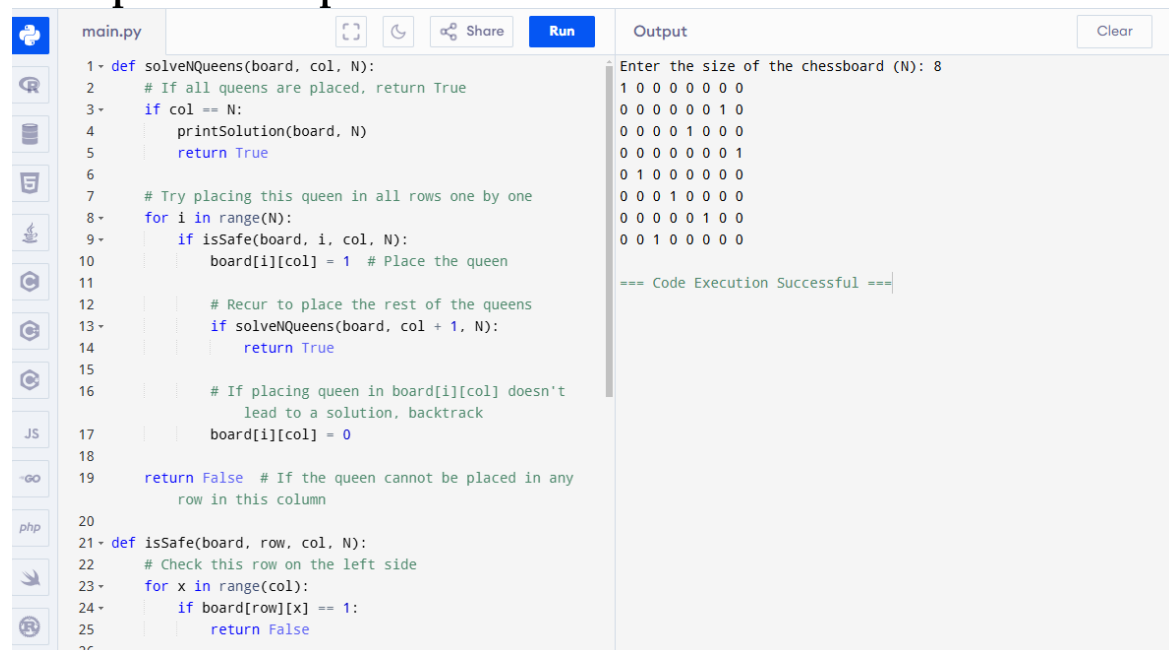
```
def printSolution(board, N):
    # A function to print the solution
    for i in range(N):
        for j in range(N):
            print(board[i][j], end=" ")
        print()

# Get the size of the chessboard from the user
N = int(input("Enter the size of the chessboard (N): "))

# Initialize the board with 0s (empty squares)
board = [[0 for x in range(N)] for y in range(N)]

# Start solving the problem from the first column
if not solveNQueens(board, 0, N):
    print("No solution found")
```

## B.2 Input and Output:



The screenshot shows a code editor with a file named 'main.py'. The code defines a recursive function 'solveNQueens' and a helper function 'isSafe'. The 'solveNQueens' function iterates through rows, placing a queen in each column and checking for conflicts. The 'isSafe' function checks if a queen can be placed in a given row and column without conflicting with previously placed queens. The code prompts the user to enter the size of the chessboard (N), which is 8. The output displays the 8x8 chessboard with the solution, showing the positions of the eight queens. The output is as follows:

```
Enter the size of the chessboard (N): 8
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

=== Code Execution Successful ===
```

## B.3 Observations and learning:

The eight queens puzzle is a classic problem in computer science and artificial intelligence, demonstrating problem-solving by searching and constraint satisfaction. The brute force approach to generate all possible combinations is

impractical due to the vast number of possible states (4,426,165,368). Instead, more efficient methods like backtracking and recursive approaches can be used to find solutions. By reducing the problem space and generating permutations of an array of eight numbers, the number of potential solutions is manageable. The recursive approach, in particular, ensures that only valid states are considered at each step, leading to all 92 distinct solutions.

## **B.4 Conclusion:**

By performing this experiment we have learned to implement the 8 queen puzzle problem and the problem highlights the importance of intelligent search techniques and the power of combinatorial optimization in solving complex puzzles.

## **B.5 Question of Curiosity**

1) How to formulate a problem?

Formulating the eight queens puzzle problem involves defining the constraints and goals clearly. The objective is to place eight queens on an 8x8 chessboard such that no two queens can attack each other. This means that no two queens can share the same row, column, or diagonal. The problem can be approached by scanning each row and placing one queen per column while checking at each step that no two queens are in the line of attack of another. If a position is valid, the queen is placed; otherwise, another position is tried. The process is repeated until a valid configuration is found.

2) State space representation of the problem?

The state space for the eight queens puzzle can be represented as an 8x8 chessboard where each state represents a particular arrangement of queens. However, a more efficient representation is to use a one-dimensional array of size eight, where each element of the array represents the column position of the queen in the corresponding row. For example, an array [6, 2, 7, 1, 4, 0, 5, 3] represents a valid arrangement where each number corresponds to the column position of a queen in rows 0 through 7. This reduces the problem space significantly, as each queen must be in a different row and column, eliminating many invalid states.

