
🟦 AI Practical 4 – N-Queen Problem (Backtracking)

N = 5 Queens

🟣 1. THEORY (Explain to external in simple words)

✔ What is the N-Queen Problem?

The N-Queen problem is:

"Place N queens on an N x N chessboard such that
no two queens attack each other."

Queens attack in:

- Same row
- Same column
- Diagonal (left & right)

So we must place them carefully.

✔ What is Backtracking?

Backtracking is a trial and error technique where:

1. You place a queen.
2. If it becomes invalid, you go back (backtrack).
3. Try next position.
4. Continue until solution is found.

It is used in:

- Sudoku solver
- Maze solving
- Crossword puzzle
- N-Queen

✔ Why N-Queen is an AI problem?

Because it uses:

- Search
- Constraint satisfaction
- Backtracking (AI problem-solving method)

🟢 2. FULL CODE (Working for 5×5 board)

```
class NQueenProblem:
  def __init__(self):
    self.N = 5


  def printSolution(self, board):
    for i in range(self.N):
      for j in range(self.N):
        print(" " + str(board[i][j]) + " ", end="")
      print()


  def isSafe(self, board, row, col):
    # Check left side of the current row
    for i in range(col):
      if board[row][i] == 1:
        return False


    # Check upper diagonal on left side
    i, j = row, col
    while i >= 0 and j >= 0:
      if board[i][j] == 1:
        return False
      i -= 1
```

```
            j -= 1

        # Check lower diagonal on left side
        i, j = row, col
        while i < self.N and j >= 0:
            if board[i][j] == 1:
                return False
            i += 1
            j -= 1

        return True

    def solveNQUtil(self, board, col):
        # Base case: If all queens are placed
        if col >= self.N:
            return True

        for i in range(self.N):
            if self.isSafe(board, i, col):
                board[i][col] = 1

                if self.solveNQUtil(board, col + 1):
                    return True

                # Backtrack
                board[i][col] = 0

        return False
    def solveNQ(self):
        board = [[0 for _ in range(self.N)] for _ in range(self.N)]

        if not self.solveNQUtil(board, 0):
            print("Solution does not exist")
            return False

        self.printSolution(board)
        return True


# Driver Code
nQueen = NQueenProblem()
nQueen.solveNQ()
```

🔎 Line-by-line detailed explanation

(Explain this to your teacher exactly — say each short sentence aloud if you wish.)

class NQueenProblem:

1. class — keyword that defines a new blueprint (a template for objects).
2. NQueenProblem — the name of the class; we put all functions and data inside this template.
3. : — signals the start of the class body (indented block below).

Short: We are creating a blueprint named NQueenProblem.

---

def __init__(self):

1. def — starts a function (method) definition.

2. __init__ — the constructor method; it runs automatically when we create an object from the class.
3. self — a reference to the current object (every method must accept it).
4. : — start of constructor body.

Short: This sets up initial data when we create a NQueenProblem object.

---

self.N = 5

1. self.N — an attribute (stored with the object) named N.
2. = — assign operator.
3. 5 — number five.

Points to say:

- 1. N specifies the board size: here a 5×5 board.
- 2. We will place 5 queens on this board.
- 3. Change 5 to another number to solve other N-Queen problems.

---

def printSolution(self, board):

1. Defines a method to display the board.
2. board parameter is the 2D list representing the chessboard.

Short: This method prints the board row by row.

---

for i in range(self.N):

1. Loop variable i goes from 0 to N-1.
2. Iterates over each row.

Short: For every row in the board.

---

for j in range(self.N):

1. Nested loop variable j goes from 0 to N-1.
2. Iterates over each column in the current row.

Short: For every column in that row.

---

print(" " + str(board[i][j]) + " ", end="")

1. board[i][j] — the element at row i, column j. It's 1 if a queen is there; 0 otherwise.
2. str(...) — converts number to string for printing.
3. " " + ... + " " — adds spaces for nicer formatting.
4. end="" — prevents print from moving to a new line after printing the cell (keeps printing on same row).

Say: This prints one cell and keeps printing cells on the same line.

---

print()

1. A plain print() with no arguments prints a newline.
   Short: After finishing a row, move to the next line.

---

def isSafe(self, board, row, col):

1. Defines a method that checks whether placing a queen at (row, col) is safe.
2. Returns True if no other queen attacks this position, otherwise False.

Short: It checks left row and two left diagonals (because we place queens column-by-column from left to right).

Section: # Check left side of the current row

```
for i in range(col):
  if board[row][i] == 1:
    return False
```

1. range(col) loops through columns left of col only (0 .. col-1).
2. board[row][i] == 1 checks if there is a queen in the same row to the left.

3. return False — immediately say the spot is unsafe.

Points:

-
    1. We don’t check the right side because we haven’t placed queens there yet.
-
    2. If any queen found in same row to left, we cannot place here.

Section: # Check upper diagonal on left side

```
i, j = row, col
while i >= 0 and j >= 0:
  if board[i][j] == 1:
    return False
  i -= 1
  j -= 1
```

1. i, j = row, col copies starting coordinates.
2. while i >= 0 and j >= 0: keeps moving up-left while inside board.
3. board[i][j] == 1 checks each cell on that diagonal for existing queen.
4. i -= 1; j -= 1 moves one step up and left.

Say: If any queen is found on upper-left diagonal, it's unsafe.

Section: # Check lower diagonal on left side

```
i, j = row, col
while i < self.N and j >= 0:
  if board[i][j] == 1:
    return False
  i += 1
  j -= 1
```

1. Reset i, j to start.

2. while i < self.N and j >= 0: moves down-left while inside board.
3. i += 1 moves down, j -= 1 moves left.

Say: If queen found on lower-left diagonal, return False.

---

return True

Short: If none of the checks found a queen, the position (row, col) is safe.

---

def solveNQUtil(self, board, col):

1. This method attempts to place queens column by column using recursion and backtracking.
2. col is the current column number we are trying to fill.

Say: This is the core backtracking routine.

---

if col >= self.N:

```
if col >= self.N:
    return True
```

1. If col equals N (i.e., we've placed a queen in all columns 0..N-1), then all queens are successfully placed.
2. return True tells caller: solution found.

Say: Base case — success when we pass last column.

---

for i in range(self.N):

1. Loop through every row in the current column.
2. Try placing a queen at (i, col).

Say: We attempt each row for the current column.

---

if self.isSafe(board, i, col):

1. Calls isSafe to ensure placing at (i, col) is allowed.

Say: Only place queen here if it's safe.

---

board[i][col] = 1

1. Place a queen at this cell by setting cell value to 1.

Say: Place a queen tentatively (we may undo this later).

---

if self.solveNQUtil(board, col + 1):

1. Recursively try to place queens in the next column.
2. If recursive call returns True, then the whole configuration leads to a solution.

Say: If placing this queen leads to a full solution, bubble up success.

---

board[i][col] = 0

1. This is the backtrack step — undo the placement.
2. Happens when further recursive calls fail to find a valid arrangement.

Say: Remove queen and try next row.

---

return False

Short: If no row in this column works, return False to tell previous column we failed here — this triggers backtracking.

---

def solveNQ(self):

1. Top-level method that prepares the board and kicks off the recursion.

---

board = [[0 for _ in range(self.N)] for _ in range(self.N)]

1. Constructs a 2D list (list of lists) filled with zeros.
2. Outer list has N rows; inner list has N zeros (columns).

Say: This creates an empty N×N chessboard with no queens.

---

if not self.solveNQUtil(board, 0):

1. Calls the recursive solver starting at column 0.
2. If it returns False, then no solution exists.

---

print("Solution does not exist")

1. Shows message if solver failed.

---

self.printSolution(board)

1. If solver succeeded, print the final board layout.

---

return True / return False

1. True indicates success to caller; False indicates failure.

Driver block

nQueen = NQueenProblem()

nQueen.solveNQ()

1. nQueen = NQueenProblem() — create an object named nQueen.
2. nQueen.solveNQ() — run the solver.

Say: Create the solver object and start solving.

📚 Expanded Viva Questions & Answers — many points per answer

Below are 30+ viva questions you may be asked, each with numbered bullet points as answers (easy to recite).

---

1. What is the N-Queen problem?

   1. Place N queens on an N×N board.
   2. No two queens may attack each other.
   3. Queens attack horizontally, vertically and diagonally.

---

2. What technique is used in this program?

1. Backtracking — a trial-and-error recursive method.
2. We place a queen, recurse, and undo (backtrack) if it fails.
3. It's a depth-first search of possible configurations.

---

3. Why do we check only the left side in isSafe()?

1. We place queens column-by-column from left to right.
2. All queens already placed are on the left of current column.
3. No need to check right side (no queens placed there yet) — saves time.

---

4. What directions does isSafe() check?

1. Left side of the same row.
2. Upper-left diagonal.
3. Lower-left diagonal.

---

5. Why do we not check same column in isSafe()?

1. Because we place only one queen per column by design.
2. When placing in column col, column col has no queen yet.

---

6. What is backtracking and where is it in the code?

1. Backtracking = try, fail, undo, try next choice.
2. Located where board[i][col] = 1 then later board[i][col] = 0.
3. solveNQUtil() handles the recursion/backtrack logic.

---

7. What is the role of solveNQUtil()?

1. Tries to place queens in columns from col to N-1.
2. Recursively ensures remaining columns can be filled.
3. Returns True on success, False to trigger backtracking.

---

8. What is the structure of board?

    1. A 2-dimensional list (list of lists).
    2. board[i][j] = 1 means queen at row i, column j.
    3. 0 means empty cell.

---

9. Explain the base case if col >= self.N: return True

    1. When col == N, all columns 0..N-1 are filled.
    2. Means all queens placed safely.
    3. Return True to indicate success.

---

10. Why loops use range(self.N)?

    1. To iterate through all rows (0 to N-1).
    2. Try placing queen in every row of the current column.

---

11. Complexity — how expensive is this algorithm?

    1. Worst-case time complexity is approximately O(N!) (very large).
    2. Backtracking prunes many possibilities; actual time is much less for many boards.
    3. Space complexity O(N^2) for board plus recursion depth O(N).

---

12. How would you change the board size to 8?

    1. Set self.N = 8 in __init__.
    2. The algorithm will then try to place 8 queens on 8×8 board.

---

13. Why do we copy i, j = row, col before diagonals?

    1. To preserve original row and col values while iterating.
    2. We decrement/increment i and j to move along diagonal.

---

14. Could we use sets to speed up isSafe()?

1. Yes — we can store columns and diagonals used in sets.
2. This reduces isSafe from O(N) to O(1) checks, speeding up the algorithm.
3. Implementation needs mapping of diagonals (row-col and row+col keys).

---

15. Why print 1 and 0 instead of Q and .?

1. 1 is simple numeric representation (queen present).
2. Teachers often accept both; you can change print to show Q for readability.

---

16. What happens if solution does not exist?

1. solveNQUtil returns False.
2. solveNQ() prints "Solution does not exist" and returns False.

---

17. Why do we return True immediately when recursive call succeeds?

1. Because once one valid arrangement is found, no further search is needed.
2. This code returns the first valid solution (not all possible solutions).

---

18. How to modify code to print all possible solutions?

1. Remove return True inside the if block.
2. Instead of returning, store or print the board each time a solution (col >= N) is reached.
3. Continue searching after backtracking to find more solutions.

---

19. How many solutions for N=5?

1. There are 10 distinct solutions for 5 queens (if counting rotations/reflections separately).
2. If you want to verify, modify code to count and print all.

(If you are unsure in viva, say: “There are multiple solutions; we can modify the code to count them.”)

---

20. What is recursion and why used here?

    1. Recursion is a function calling itself.
    2. It simplifies exploring choices column-by-column.
    3. Each recursive call handles the next column as a new subproblem.

---

21. Why is board[i][col] = 0 necessary?

    1. It undoes the placement — the essence of backtracking.
    2. Without it, later tries would see old queen and fail incorrectly.

---

22. Explain the flow when a placement fails deep in recursion

    1. Try place at (r,c), set 1 and call recursion for c+1.
    2. If deeper call returns False, we undo board[r][c] = 0.
    3. Then try next row in same column.

---

23. What are diagonals checked (i-=1,j-=1 and i+=1,j-=1)?

    1. i-=1,j-=1 — move upper-left along diagonal.
    2. i+=1,j-=1 — move lower-left along diagonal.
    3. Both check diagonals to the left of current column.

---

24. Could we check columns to the left? Why not?

    1. We place exactly one queen per column, so a column left of current could contain a queen but we check row and diagonals to detect attacks.
    2. Column check (same column) is unnecessary because we never insert more than one queen per column in this algorithm.

---

25. If teacher asks why self is needed?

    1. self refers to the current object instance.
    2. We use self.N, self.printSolution etc. to access object's data and methods.
    3. Methods inside a class must accept self as first parameter.

---

26. If teacher asks to explain board = [[0 for _ in range(self.N)] for _ in range(self.N)]

   1. This is nested list comprehension creating N lists each with N zeros.
   2. for _ in range(self.N) repeats N times; _ is a throwaway loop variable.
   3. Result: a 2D list or matrix filled with zeros.

---

27. If asked how to represent board differently

   1. Use a single list col_positions where col_positions[c] = r means queen in column c at row r.
   2. This reduces memory and simplifies diagonal checks with row-col and row+col.

---

28. What is the advantage of column-by-column placement?

   1. It limits the search space naturally.
   2. Ensures no two queens are in the same column.
   3. Simplifies isSafe() checks to only left side.

---

29. If teacher asks: what is the first queen position placed by your program?

   1. It starts trying from column 0 and row 0 upwards.
   2. The final solution depends on backtracking and the first successful rows chosen by the algorithm.

---

30. How to make the output prettier (show Q and .)?

   1. Replace str(board[i][j]) with 'Q' if board[i][j] == 1 else '.'.
   2. Update print line accordingly.

✅ Quick oral script you can memorize (30–45 seconds)

"This program defines a NQueenProblem class to place 5 queens on a 5×5 board. We create an empty board, then try to place queens column by column using solveNQUtil. For each column we try every row; before placing we call isSafe to ensure no queen on the left row or left diagonals attacks this position. We place a queen (set cell to 1) and recursively solve next column. If recursion fails, we remove the queen (board[i][col]=0) and try the next row — this is backtracking. When col >= N we have placed all queens and print the board.”
