Assignment-3(N-Queen Problem)

You are tasked with solving the N-Queens problem using a backtracking algorithm. The N-Queens problem is to place N chess queens on an N×N chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal. Your goal is to implement the algorithm, find all possible solutions for a given N, and analyze its time complexity.

1. Implement a backtracking algorithm in Python to solve the N-Queens problem.
2. Apply your algorithm to find all possible solutions for a given N (e.g., N = 4 or N = 8). Ensure that you generate all unique solutions.
3. Present the solutions as chessboard representations, indicating the placement of queens (e.g., using 'Q' for queens and '.' for empty squares).
4. Explain the backtracking approach, including how you generate and validate solutions and how you handle conflicts between queens.
5. Analyze the time complexity of your algorithm and discuss its efficiency for larger values of N.

Answers:-

2. Apply your algorithm to find all possible solutions for a given N (e.g., N = 4 or N = 8). Ensure that you generate all unique solutions.

for 4-queen:-

```
N = 4
solutions = solve_n_queens(N)

# Print solutions as chessboard representations
for i, solution in enumerate(solutions):
    print(f"Solution {i + 1}:")
    for row in solution:
        print(row)
    print()
```

for 8-queen:-

```
N = 8
solutions = solve_n_queens(N)

# Print solutions as chessboard representations
for i, solution in enumerate(solutions):
    print(f"Solution {i + 1}:")
    for row in solution:
        print(row)
    print()
```

3. Present the solutions as chessboard representations, indicating the placement of queens (e.g., using 'Q' for queens and '.' for empty squares).

Algorithm:-

Initialize an empty N×N chessboard represented as a 2D array of '.' (empty squares).

Define a helper function is_safe(row, col) to check if it's safe to place a queen in a given cell by examining the row, upper-left diagonal, and lower-left diagonal for any existing queens.

Implement a recursive backtracking function backtrack(col) that explores all possible placements of queens column by column. When a valid placement is found, the queen is placed, and the function recursively moves on to the next column.

If a solution is found (all columns are filled), it is added to the list of solutions.

The backtracking process starts with the first column (col = 0), and all possible configurations are explored.

Finally, the code returns a list of solutions, where each solution is represented as a chessboard with 'Q' indicating queen placements and '.' for empty squares.

```
def solve_n_queens(n):
    def is_safe(board, row, col):
```

```
    # Check if there is a queen in the same column
    for i in range(row):
        if board[i][col] == 'Q':
            return False

    # Check upper-left diagonal
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 'Q':
            return False

    # Check upper-right diagonal
    for i, j in zip(range(row, -1, -1), range(col, n)):
        if board[i][j] == 'Q':
            return False

    return True

def solve(row):
    if row == n:
        solutions.append(["".join(row) for row in board])
        return

    for col in range(n):
        if is_safe(board, row, col):
            board[row][col] = 'Q'
            solve(row + 1)
            board[row][col] = '.'

board = [['.' for _ in range(n)] for _ in range(n)]
solutions = []
solve(0)
```

```
    return solutions

def print_solutions_as_chessboards(solutions):
    for i, solution in enumerate(solutions):
        print(f"Solution {i + 1}:")
        for row in solution:
            print(row)
        print()

# Example usage with N = 4
N = 4
solutions = solve_n_queens(N)

# Print solutions as chessboard representations
print_solutions_as_chessboards(solutions)
```

Output:-

```
.Q..
...Q
Q...
..Q.

..Q.
Q...
...Q
.Q..
```

4. Explain the backtracking approach, including solution generation, validation, and conflict resolution.

Solution Generation:

The goal is to find all possible arrangements of queens on an N×N chessboard, such that no two queens can attack each other.

We use a recursive approach to generate solutions incrementally. We start by placing a queen in the first row, then move to the second row, and so on until we reach the Nth row.

At each row, we try placing the queen in each column (i.e., we iterate through columns). If a placement is valid, we move on to the next row and continue the process recursively.

Validation (is_safe function):

At each step, we need to check if placing a queen in a particular row and column is safe, meaning it doesn't violate the N-Queens constraints.

To validate a placement, we consider three types of conflicts: column conflicts, diagonal conflicts (both upper-left to lower-right and upper-right to lower-left diagonals), and row conflicts (which are automatically avoided since we place only one queen per row).

If none of these conflicts are present, we consider the placement safe and proceed with the next row. Otherwise, we backtrack (undo the placement) and try the next column.

Conflict Resolution:

When a conflict is detected, we backtrack to the previous row and try the next column in that row.

The backtracking mechanism is crucial in exploring different possibilities efficiently. It allows us to "undo" choices that lead to conflicts and explore alternative paths.

If we reach the Nth row without conflicts, we have found a valid solution, and we add it to the list of solutions.

Termination and All Solutions:

The process continues until we have explored all possible placements for the queens in all rows.

Each valid arrangement encountered during the process is considered a solution.

After exploring all possibilities, the algorithm terminates, and we have a list of all possible solutions.

5. Analyze the time complexity of your algorithm and discuss its performance for larger N values.Time Complexity Analysis:

In the worst case, the algorithm explores all possible placements of queens on the N×N chessboard.

At each row, there are N choices for the column in which to place a queen.

Therefore, the total number of recursive calls (function invocations) is roughly N^N, which represents the branching factor raised to the depth of the search tree.

In terms of time complexity, this gives us an upper bound of O(N^N).

Performance for Larger N Values:

As N increases, the time complexity grows exponentially.

For small values of N (e.g., $N \leq 15$), the algorithm can find all solutions quickly and efficiently.

However, as N becomes larger (e.g., $N > 15$), the time required to find all solutions increases dramatically.

Beyond a certain point, the algorithm may become impractical to use for large N values due to its exponential time complexity.

Optimizations:

Various optimizations can be applied to improve the algorithm's performance. One common optimization is to reduce the number of redundant checks. For example, you can optimize the is_safe function by precomputing and storing information about threatened rows, columns, and diagonals.

Another optimization is to use bitwise operations to check conflicts more efficiently, especially for smaller N values.

Parallelization:

When dealing with very large N values, parallelization can be used to distribute the workload across multiple processors or threads.

Parallel algorithms can significantly speed up the search for solutions by exploring different branches of the search tree concurrently.

Heuristic Algorithms:

For extremely large N values, heuristic algorithms such as genetic algorithms or simulated annealing might be more suitable.

These algorithms do not guarantee finding all solutions but can quickly find good solutions in a reasonable time frame.

.