Batch: C2 Roll No.: 16010122257

Experiment No. 8

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

Title: Implementation of N-Queen Problem using Backtracking Algorithm

Objective: To learn the Backtracking strategy of problem solving for 8-Queens problem

CO to be achieved:

Sr. No	Objective
CO 1	Compare and demonstrate the efficiency of algorithms using asymptotic complexity notations.
CO 2	Analyze and solve problems for divide and conquer strategy, greedy method, dynamic programming approach and backtracking and branch & bound policies.

Books/ Journals/ Websites referred:

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran," Fundamentals of computer algorithm", University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein," Introduction to algortihtms",2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://www.math.utah.edu/~alfeld/queens/queens.html**
4. **http://www-isl.ece.arizona.edu/ece175/assignments275/assignment4a/Solving%208%20queen%20problem.pdf**
5. **http://www.slideshare.net/Tech_MX/8-queens-problem-using-back-tracking**
6. **http://www.mathcs.emory.edu/~cheung/Courses/170.2010/Syllabus/Backtracking/8queens.html**
7. **http://www.geeksforgeeks.org/backtracking-set-3-n-queen-problem/**
8. **http://www.hbmeyer.de/backtrack/achtdamen/eight.htm**

Pre Lab/ Prior Concepts:
Data structures, Concepts of algorithm analysis

Historical Profile:
The **N-Queens puzzle** is the problem of placing N queens on an N×N chessboard so that no two queens attack each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

New Concepts to be learned:
Application of algorithmic design strategy to any problem, Backtracking method of problem-solving Vs other methods of problem solving, 8- Queens problem and its applications.

Algorithm N Queens Problem: -

```
void NQueens(int k, int n)
// Using backtracking, this procedure prints all possible placements of n queens on an n X n
chessboard so that they are nonattacking.
{       for (int i=1; i<=n; i++)
    {
            if (Place(k, i))
             {
               x[k] = i;
               if (k==n)
                       for (int j=1;j<=n;j++)          Print  x[j] ;
               else NQueens(k+1, n);
             }
        }
}

Boolean Place(int k, int i)
// Returns true if a queen can be placed in kth row and ith column.  Otherwise it returns false.
// x[] is a global array whose first (k-1) values have been set. abs(r) returns absolute value of r.
{
for (int j=1; j < k; j++)

        if ((x[j] == i)  // Two in the same column

      || (abs(x[j]-i) == abs(j-k)))                   // or in the same diagonal

          return(false);

return(true);

 }
```

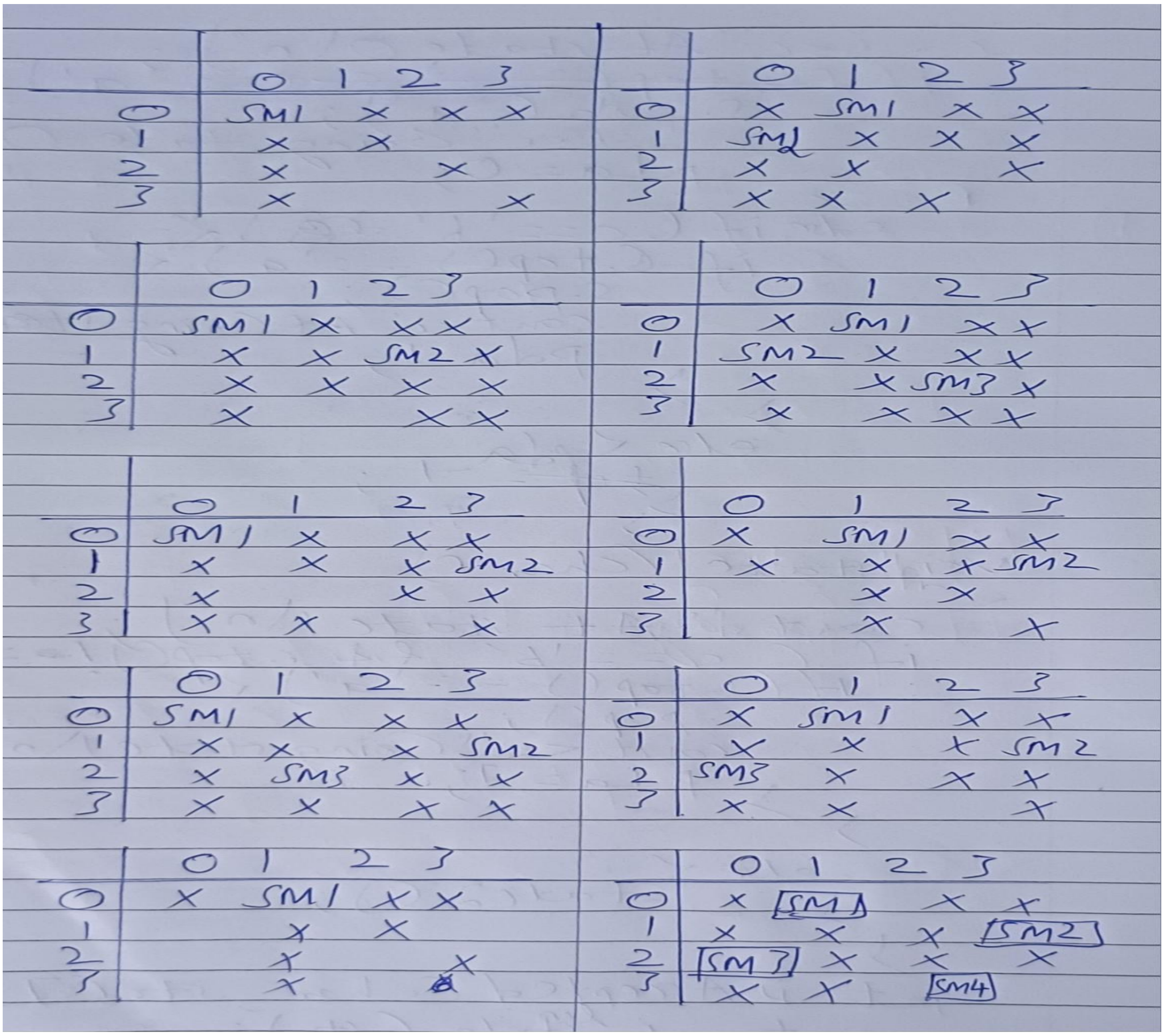
Example 8-Queens Problem:

The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other i.e. no two queens share the same row, column, or diagonal.

Solution Using Backtracking Approach:

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.

<u>State Space tree for N-Queens (Solution):</u>



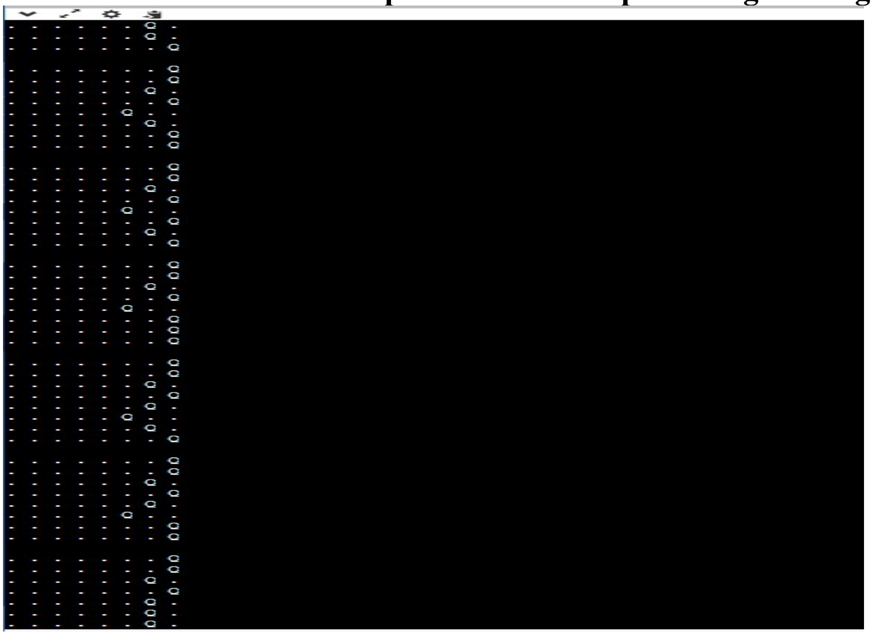
Implementation (Code):

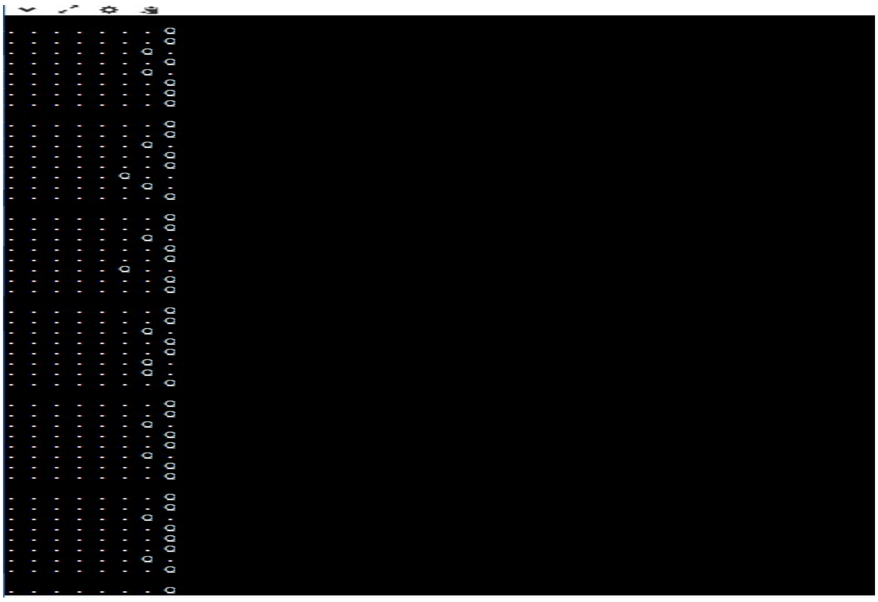
main.cpp

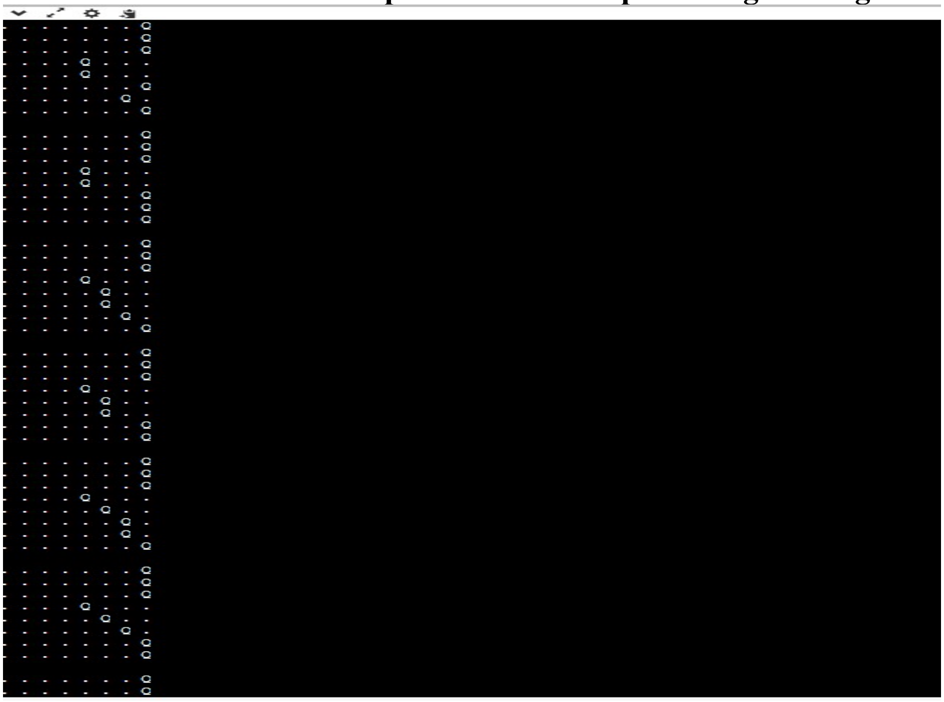
```cpp
#include <iostream>
#include <vector>
using namespace std;

#define N 8


void printSolution(const vector<int>& board) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            cout << (board[i] == j ? "Q " : ". ");
        cout << endl;
    }
    cout << endl;
}

bool isSafe(vector<int>& board, int row, int col) {
    for (int i = 0; i < col; i++)
        if (board[i] == board[row] || abs(board[i] - board[row]) == abs(i - col))
            return false;
    return true;
}

void solveNQUtil(vector<int>& board, int col, vector<vector<int>>& solutions) {

    if (col >= N) {
        solutions.push_back(board);
        return;
    }


    for (int i = 0; i < N; i++) {
```

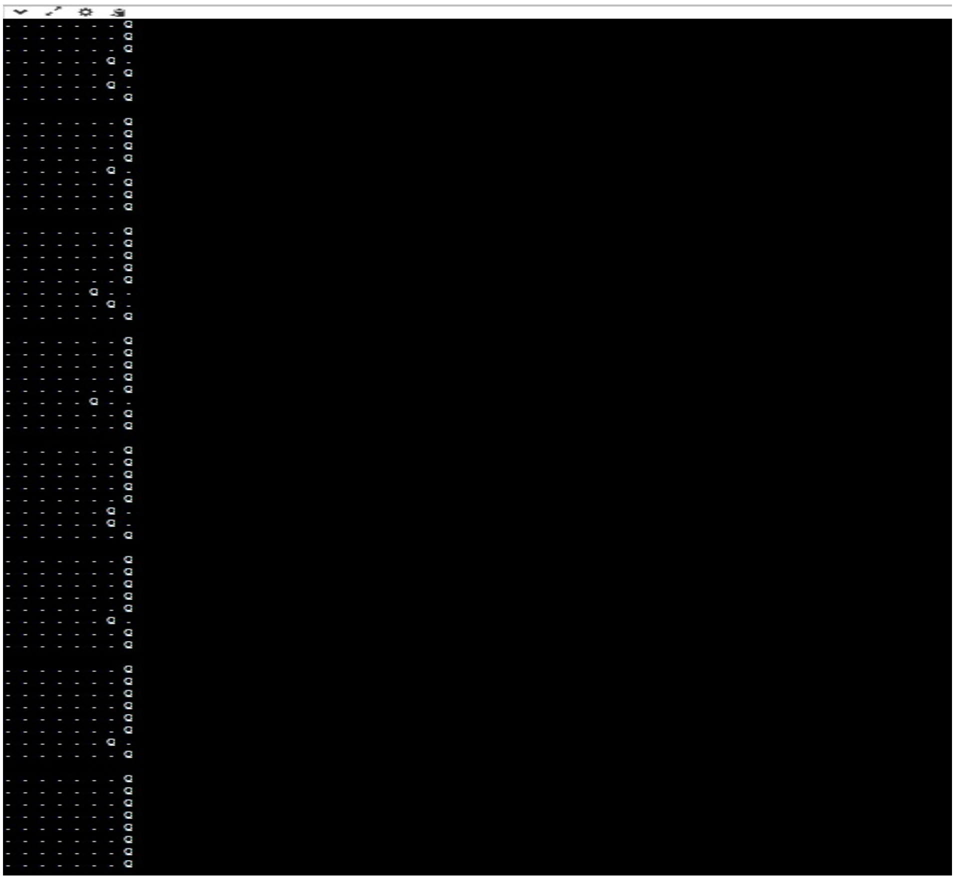
```cpp
        if (isSafe(board, i, col)) {
            board[col] = i;


            solveNQUtil(board, col + 1, solutions);


            board[col] = -1;
        }
    }
}


vector<vector<int>> solveNQ() {
    vector<vector<int>> solutions;
    vector<int> board(N, -1);


    solveNQUtil(board, 0, solutions);

    return solutions;
}


int main() {
    vector<vector<int>> solutions = solveNQ();
    for (const auto& solution : solutions) {
        printSolution(solution);
    }
    return 0;
}
```

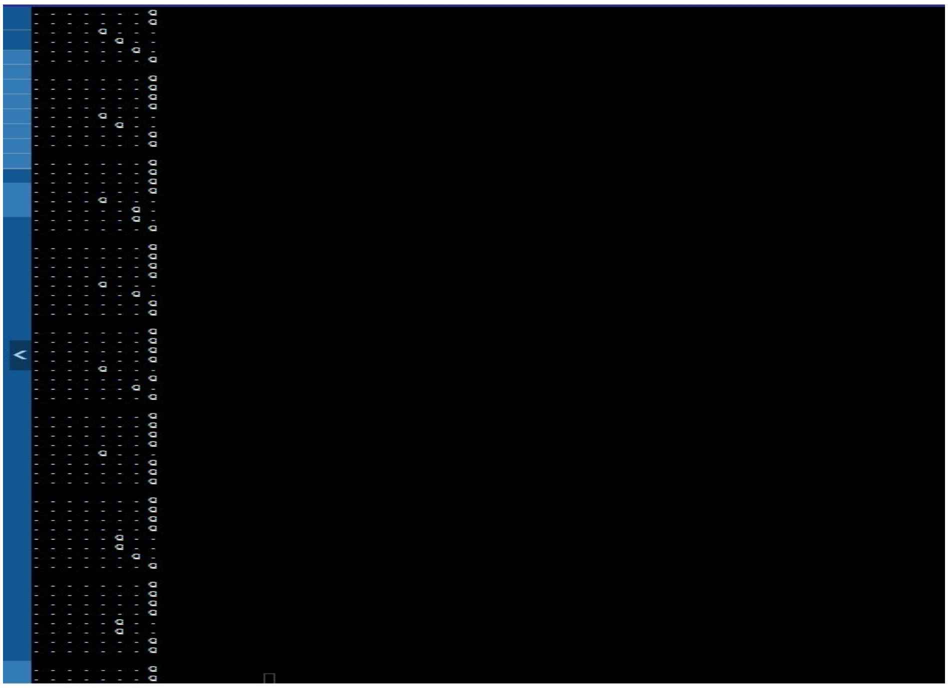
OUTPUT SCREENSHOTS:

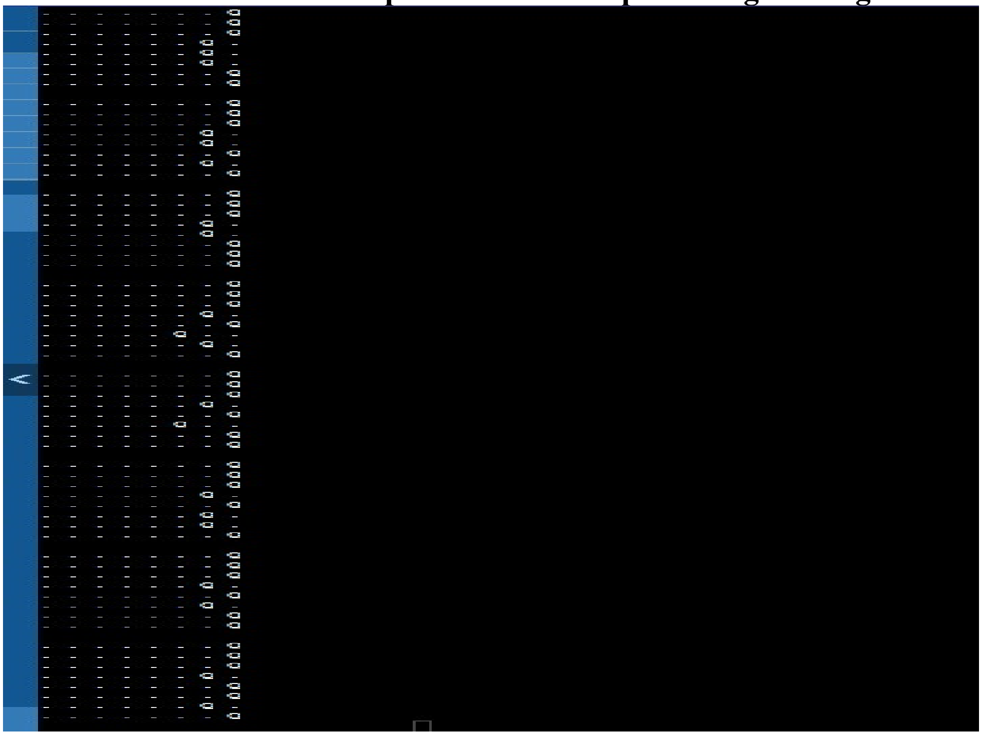


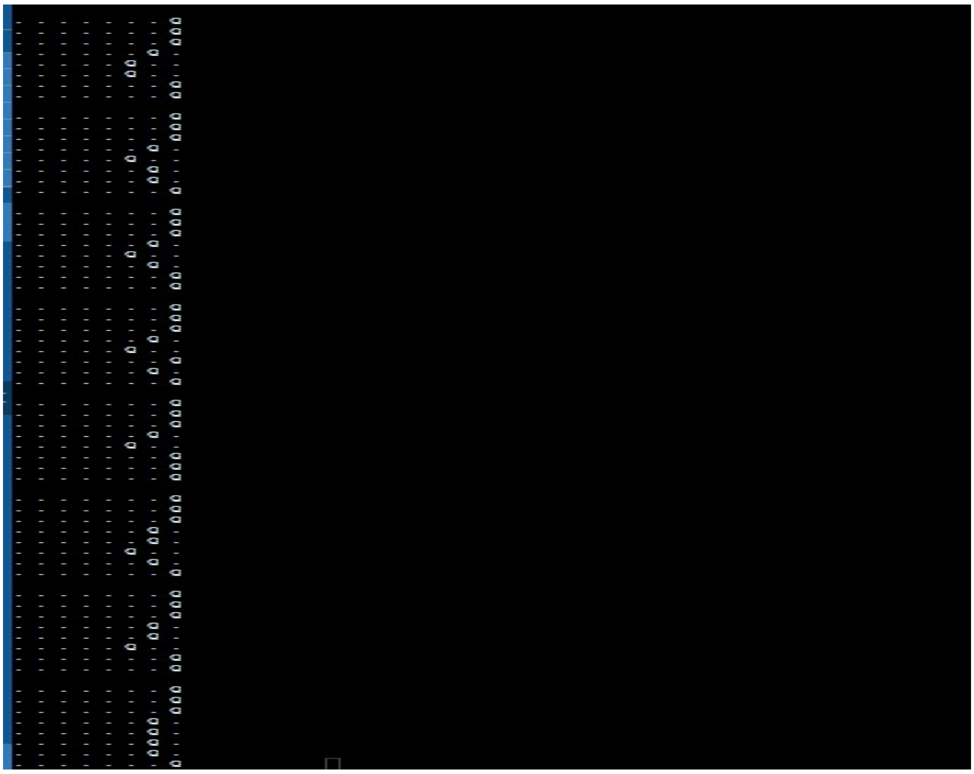


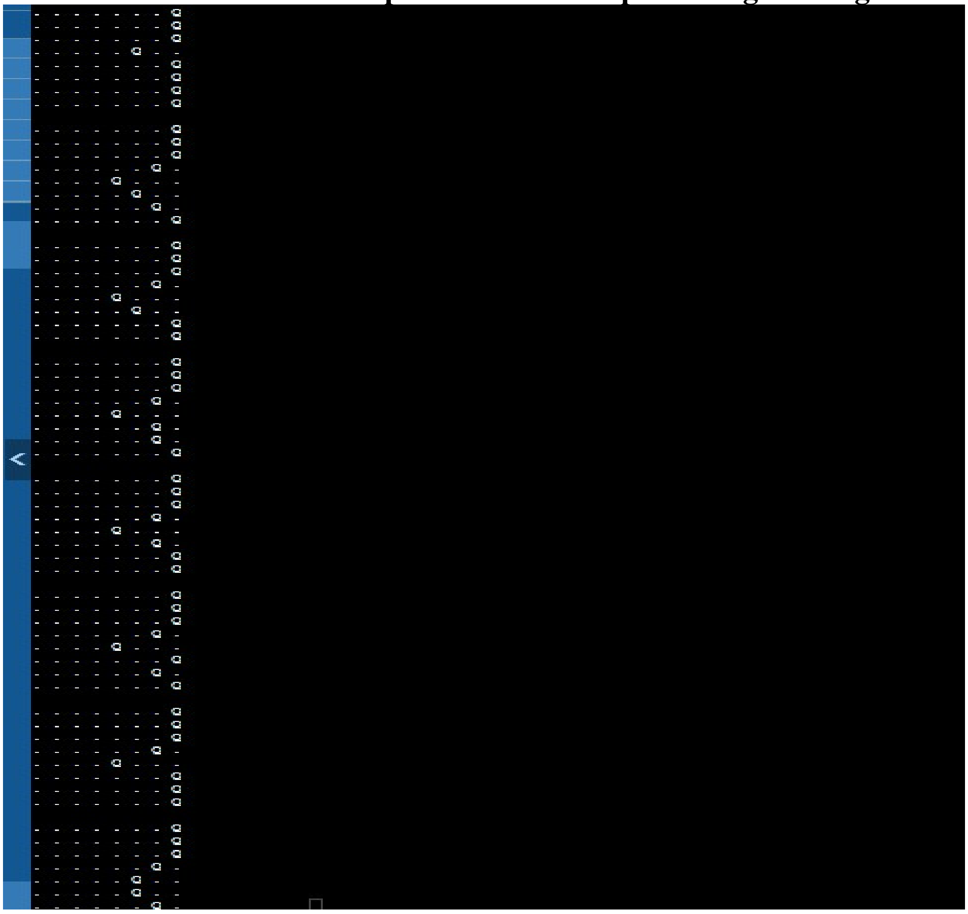


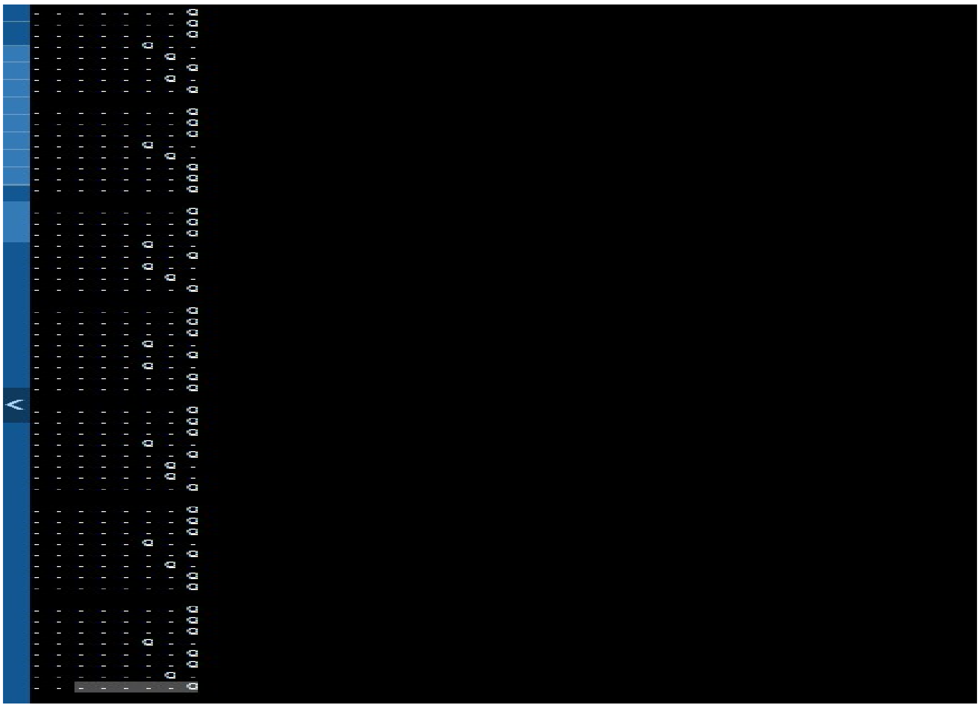


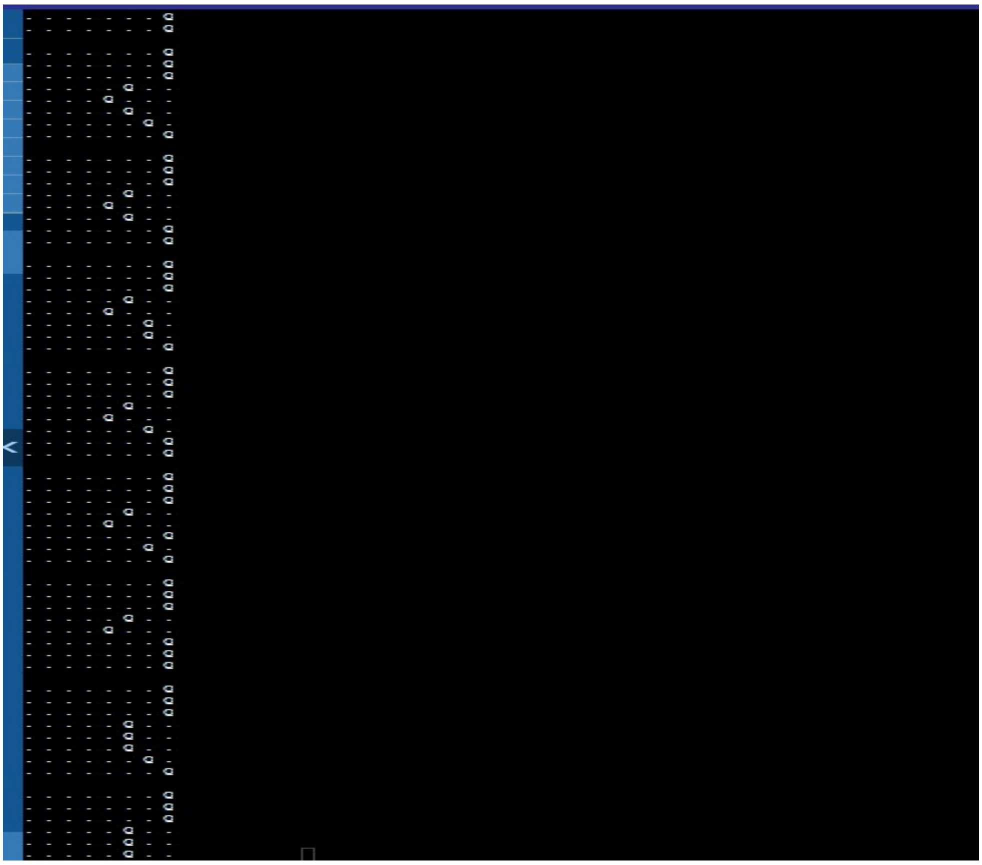


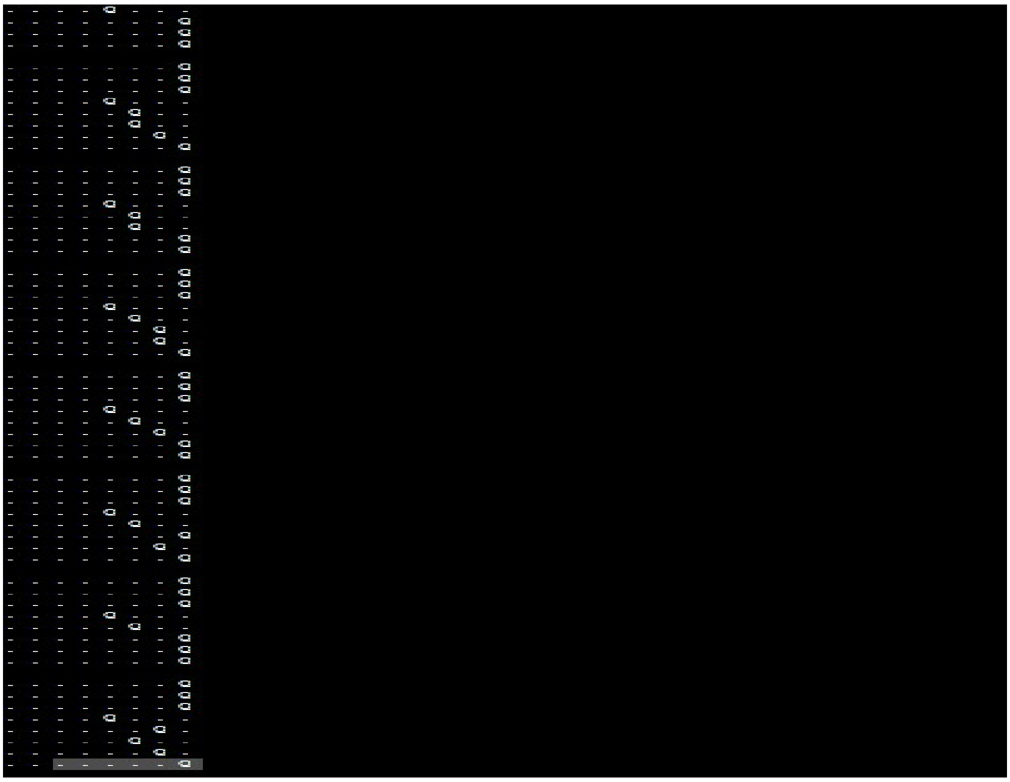


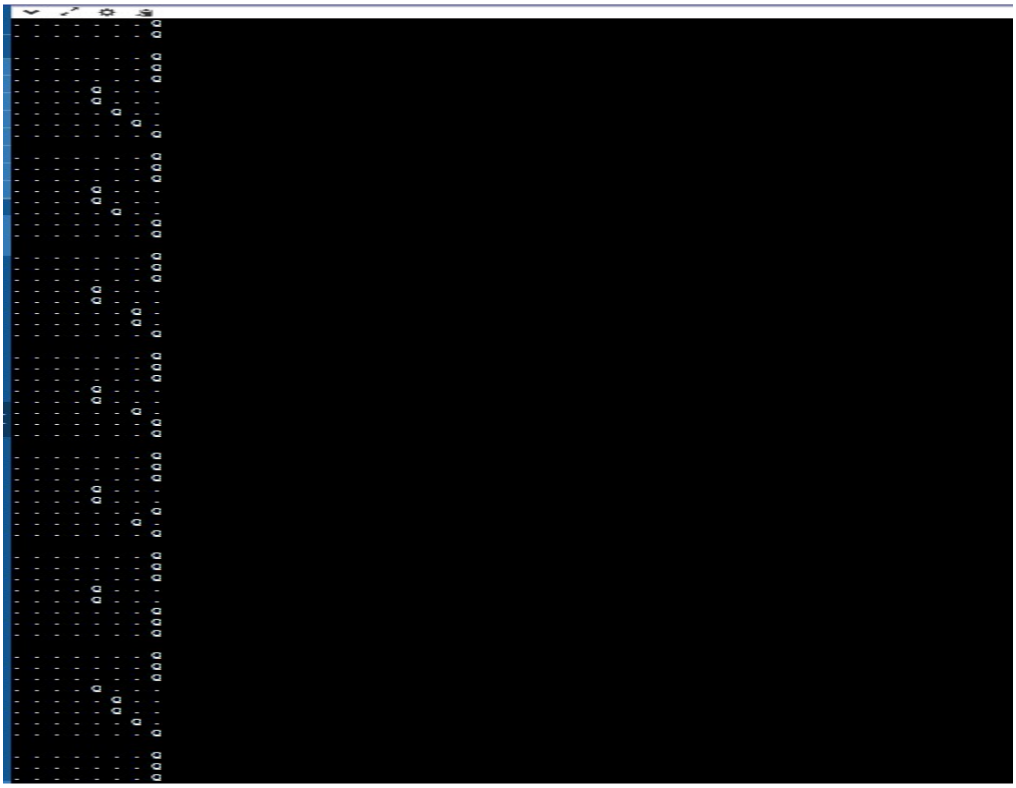


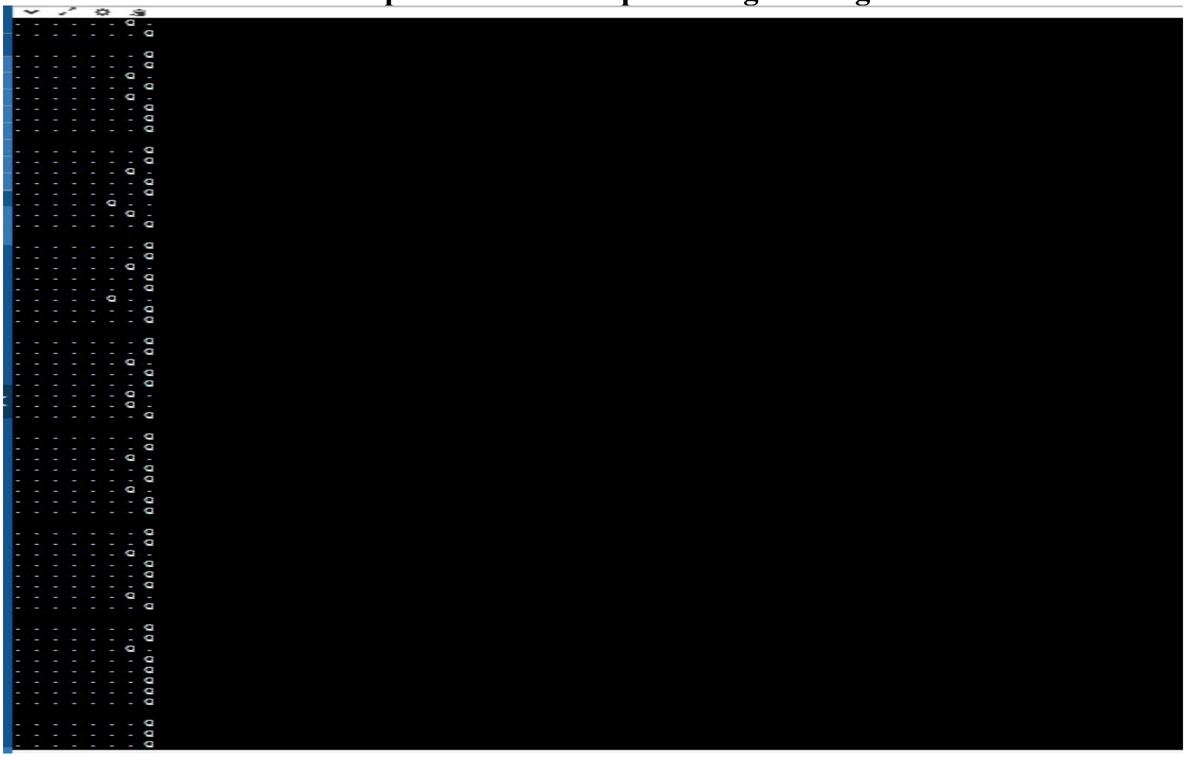












<u>Algorithm:</u>
The outlined algorithm effectively tackles the N-Queens problem using backtracking. Here's a rephrased summary:

- Start with an empty board of size N x N.
- Utilize a recursive function named "solve" to explore possible queen placements, tracking all arrangements.
- Traverse row by row for each column, checking if a cell is safe for queen placement using the "isSafe()" function.
- If a cell is deemed safe, mark it with a queen ('Q') on the board and recursively call "solve" for the next column.
- When all columns have been explored (i.e., column equals the board length), return.
- The "isSafe()" function ensures no conflicting queens in rows, columns, or diagonals.
- Analyzing the backtracking solution reveals a time complexity of O(N * N!) due to the N iterations per cell and an auxiliary space complexity of O(N^2) for the chessboard.

Analysis of Backtracking solution:

As we got to know in the algorithm for each cell, to check if the queen can be placed there or not, we are iterating for N times. So the recurrence relation comes out to be:
T(N) = N * T(N-1) + N.
T(N-1) = N * T(N-2) + N0

T(1) = 1
This totals T(N) = N* N!. therefore, the time complexity comes out to be O(N * N!).
And as we have used an extra board of characters of N x N Size, The space complexity comes out to be O(N *N).
Auxiliary Space: O(N^2) for the
chessboard.

CONCLUSION:

In summary, the algorithm effectively solves the N-Queens problem using backtracking, providing a clear and concise approach to understanding its implementation and complexity.