EAST WEST UNIVERSITY

Project Report

Project Title:

"N-Queen Problem"

Course Code: CSE246

Course Title: Algorithms

Semester: Summer 23

Section: 02

Group No: 10

Submitted By:

NO.	Name	ID
01.	Jubaer Ahmed	2021-2-60-139
02.	Mobashhira Jahan	2020-1-60-282

Submitted To:

Md. Mohsin Uddin

Senior Lecturer

Department of Computer Science and Engineering

Submission Date: 30 August, 2023.

1. Problem Statement:

The **'N-Queen'** problem involves placing N chess queens on an N×N chessboard in such a way that no two queens attack each other, i.e., they do not share the same row, column, or diagonal. In this project, the goal is to solve the N-queen problem that shows all possible placements of N queens. The project aims to provide a solution to the N-queen problem that can be applied to various scenarios involving chessboards of different sizes. The solution should be applicable to various real-world scenarios requiring optimal arrangements or assignments under specific constraints.

2. Algorithm Discussion and simulation through examples:

Algorithm Discussion:

The algorithm used to solve the N-Queens problem is a recursive backtracking algorithm. It systematically searches for a solution by incrementally placing queens on the chessboard, then backtracking and trying alternative placements if a conflict arises. The algorithm focuses on placing one queen per column and checks if the current placement is safe before moving on to the next column.

Step by Step Flow:

1. Initialize an N x N chessboard with all 0's, representing empty cells.

2. Start with the first column (col = 0).

3. For each row in the current column, perform the following steps:

   a. Check if it is safe to place a queen in the current cell (row, col) using the **`isSafe()`** function.

   b. If it is safe, place the queen by setting the cell value to 1.

   c. Move on to the next col (col + 1) and repeat steps (3a to 3c) recursively.

   d. If all columns are filled with queens, add the current board configuration to the list of solutions.

   e. Backtrack by removing the queen from the current cell (row, col) by setting the cell value back to 0 and trying the next column in the same row.

4. Return the list of solutions.

Examples:

Let's consider solving the 4-Queens problem (N = 4).

1. Initialize a 4 x 4 chessboard with all 0's:

	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

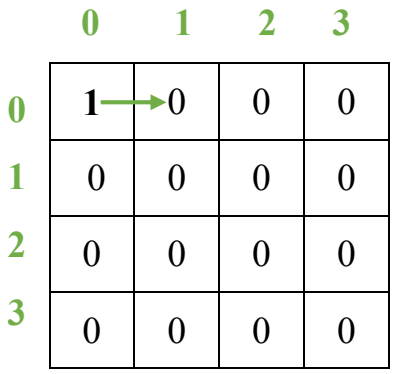
2. Start with the first column (col = 0).

3. Place the first queen in the first row of the first column (row 0, col 0).

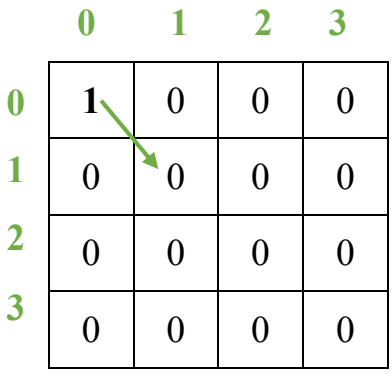
4.

	0	1	2	3
0	**1**	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

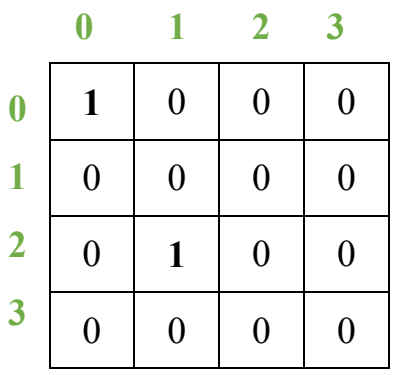
Next, we move to the next column as defined in the function recursively while saving the location of previously placed queen, col + 1.



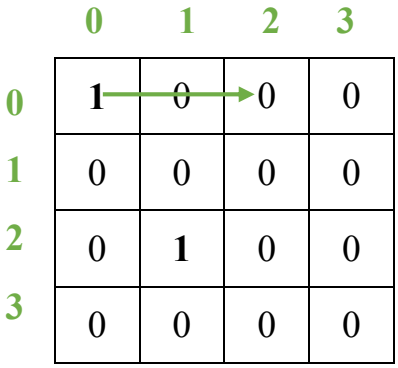
The queen is not safe because of being in same row of previous placed queen (it may attack each other). So, we move to the next row.



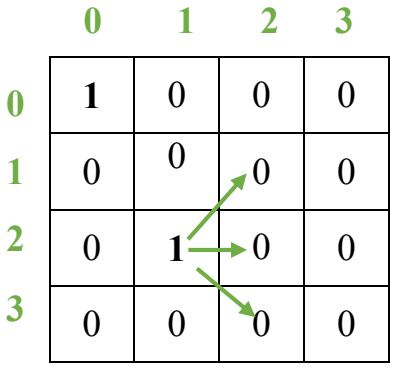
This place also not safe because of being diagonal to previous placed queen (it may attack each other). We keep moving until we get a safe position.



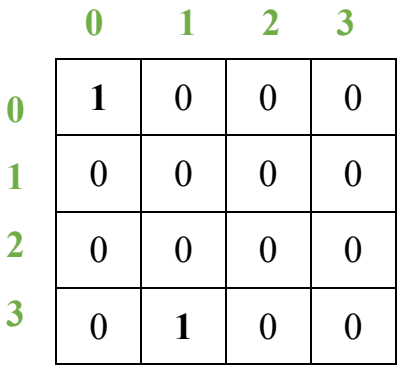
This position is safe. We move to column 2 from the first queen the same way and follow the same procedure.



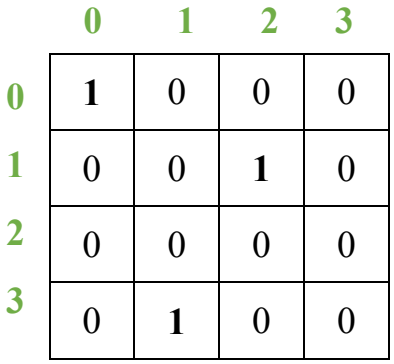
This position is not safe. Move to next row.



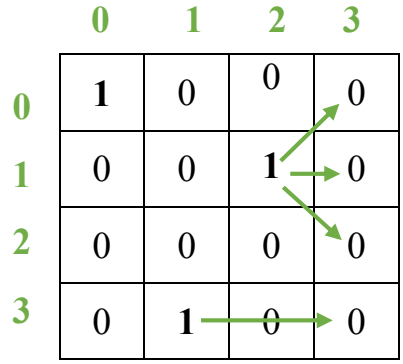
Since there is no safe position in column 2, we back track and move to column 1 to the placed queen.



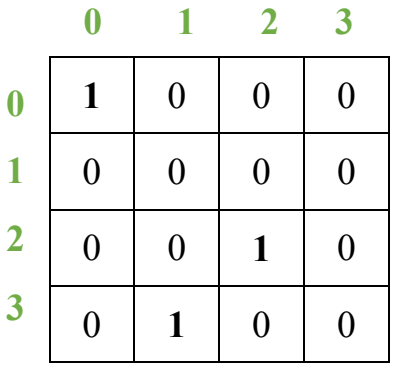
Now we find the next safe position in column 1 and place the queen. Then move to the next column and repeat



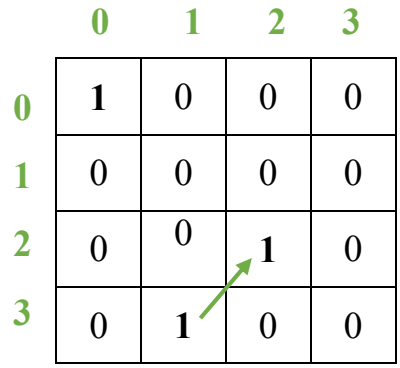
Safe position in column 2, move to next column, col + 1.



No safe place. So, we back track in column 2. Then move to the next row.



Placement of the queen in row 2.



Not a safe position for the queen of column 2, so backtrack.

	0	1	2	3
0	**1**	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	**1**	0	0

Since there is no safe position left here, we backtrack to the previous column, col 0.

	0	1	2	3
0	**1**	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

Since we can't find any possible solution for setting up queen in (row 0, col 0) position. We move to the next row in col 0 and repeat the same process.

	0	1	2	3
0	0	0	0	0
1	**1**	0	0	0
2	0	0	0	0
3	0	0	0	0

	0	1	2	3
0	0	0	0	0
1	**1**	0	0	0
2	0	0	0	0
3	0	**1**	0	0

	0	1	2	3
0	0	0	**1**	0
1	**1**	0	0	0
2	0	0	0	0
3	0	**1**	0	0

	0	1	2	3
0	0	0	**1**	0
1	**1**	0	0	0
2	0	0	0	**1**
3	0	**1**	0	0

Since, all of the columns have a queen placed, we have our solution.

5. Since we've tried all possible row placements in the first column, we have found all solutions for the 4-Queens problem:

Solution 1:

	0	1	2	3
0	0	0	**1**	0
1	**1**	0	0	0
2	0	0	0	**1**
3	0	**1**	0	0

Solution 2:

	0	1	2	3
0	0	0	**1**	0
1	**1**	0	0	0
2	0	0	0	**1**
3	0	**1**	0	0

These two solutions represent all possible ways to place 4 queens on a 4 x 4 chessboard such that no two queens threaten each other.

2. Complexity Analysis

Time complexity:

First of all, we will analysis the time complexity of each function in the code, then we will consider the total time complexity regarding N-Queen problem:

The time complexity of each loop in the '**isSafe**' function:

***First loop** (Check if there is a queen in the same row):

The loop iterates from 0 to col-1. In the worst case, when col = N-1, the loop runs N-1 times. Therefore, the time complexity of this loop is **O(N).**

***Second loop** (Check for queens attacking diagonally on the upper left side):

In this loop, both i and j are decremented in each iteration. The loop runs until either i or j becomes negative. The maximum number of iterations is the minimum of row and col. In the worst case, when row = N-1 and col = N-1, the loop runs N-1 times. Thus, the time complexity of this loop is also **O(N).**

***Third loop** (Check for queens attacking diagonally on the lower left side):

In this loop, i is incremented, and j is decremented in each iteration. The loop runs until either i reaches N-1 or j becomes negative. The maximum number of iterations is the minimum of (N-1 - row) and col. In the worst case, when row = 0 and col = N-1, the loop runs N-1 times. Consequently, the time complexity of this loop is **O(N).**

As all the loops run sequentially, the overall time complexity of the isSafe function is the sum of the time complexities of each loop, which is **O(N) + O(N) + O(N) = O(N).**

The time complexity of each loop in the **'solveNQueens'** function:

The function uses a for loop that iterates over each row in the current column (from 0 to N-1). This loop has a time complexity of **O(N).**

Inside the for loop, the '**isSafe'** function is called, which has a time complexity of **O(N).** However, since we call '**isSafe**' once for each iteration of the loop, the combined time complexity of the loop and the **'isSafe'** function inside the '**solveNQueens**' function is **O(N^2).**

But this **O(N^2)** complexity is not the overall complexity of the **'solveNQueens'** function. The overall complexity also includes the recursive calls made by the function.

The time complexity of the recursive calls depends on the depth of the recursion tree, the branching factor, and the pruning performed by backtracking. In the worst case, the algorithm would have to check all possible permutations of placing queens on the board, which is **O(N!)** because of there are N choices for the first queen, N-2 choices for the second queen (since it cannot be placed in the same row or diagonal as the first queen), N-4 choices for the third queen (since it cannot be placed in the same row or diagonal as the first two queens), and so on. Therefore, the total number of possible solutions is **N x (N-2) x (N-4) x ... x 1**, which is equal to **N!** in factorial notation. However, due to the backtracking and pruning, the actual number of possibilities explored is significantly less than **N!.**

Taking the recursive calls into account, the overall time complexity of the **'solveNQueens'** function is **O(N!).** This complexity includes the **O(N^2)** factor from the combination of the loop and the **'isSafe'** function inside the **'solveNQueens'** function.

The time complexity of each loop in the **'main'** function:

Calling the 'nQueens' function: This step involves solving the N-Queens problem using the backtracking algorithm. As we got before, the overall time complexity of this step is **O(N!).**

Printing the solutions: The loops used to print the solutions have the following complexities:

The outer loop iterates through each solution in the solutions vector. Let's denote the number of solutions by **S**. So, the complexity of the outer loop is **O(S).**

The two inner loops iterate through each row and column of the current solution. Since there are N rows and N columns, the combined complexity of these two inner loops is **O(N^2).**

The combined complexity of printing the solutions is **O(S * N^2).**

Now, combining the complexities of the two main components in the main function:

Calling the nQueens function: O(N!)

Printing the solutions: O(S * N^2)

The overall complexity of the main function will be the combination of these two components. The dominant term here is the complexity of the **'nQueens'** function, which is **O(N!).** The complexity of printing the solutions **(O(S * N^2))** is generally smaller compared to the complexity of finding the solutions. Thus, the overall time complexity of the main function can be approximated as **O(N!).** However, the important thing is, this is an upper bound, and in practice, the actual runtime is often much less than this bounds due to backtracking and pruning.

Therefore, the time complexity of N-Queen problem is O(N!).

Memory Complexity:

The overall memory complexity of the given **N-Queens** code consists of three main components:

Board: The chessboard is represented as an N x N matrix. The memory complexity for the board is **O(N^2).**

Solutions: The number of valid solutions for the N-Queens problem is much lower than the O(N!) upper bound. Each solution is an N x N matrix. The memory complexity for storing solutions is **O(S * N^2),** where S is the number of solutions.

Function Call Stack: In the worst case, the depth of the recursion tree can be N. The memory complexity for the function call stack is **O(N).**

Considering all the factors, the overall memory complexity of the given N-Queens code is **O(S * N^2 + N^2 + N).** In practice, the (**S * N^2**) term will be the dominant factor in most cases, so the memory complexity can be approximated as **O(S * N^2),** where **S** is the number of solutions.

3. Implementation with proper commenting:

```
#include <bits/stdc++.h>
using namespace std;

bool isSafe(vector<vector<int>>& board, int row, int col, int N) {
    for (int i = 0; i <= col-1; i++) {
        if (board[row][i] == 1) {
            return false;
        }
    }
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1) {
            return false;
        }
    }

    for (int i = row, j = col; i <= N-1 && j >= 0; i++, j--) {
        if (board[i][j] == 1) {
            return false;
        }
    }
    return true;
}
```

Check if a queen can be placed safely at the given

Check if there is a queen in the same row

Check for queens attacking diagonally on the upper left side

Check for queens attacking diagonally on the lower left side

If no queens are attacking, return true

The most important part solveNQueens()

```
void solveNQueens(vector<vector<int>>& board, int col, int
N, vector<vector<vector<int>>>& solutions) {

    if (col == N) { //Base case
        solutions.push_back(board);
        return;
    }
    //Recursive function, try placing a queen in each row of
the current column
    for (int i = 0; i <= N-1; i++) {
        if (isSafe(board, i, col, N)) {
            board[i][col] = 1;
            solveNQueens(board, col + 1, N, solutions);
            board[i][col] = 0;
        }
    }
}
```

Recursive function to solve the N-Queens problem using backtracking

If all queens are placed successfully, add the solution to the list

Iterating over each row in column

Place the queen at the corresponding position if it is safe, by updating the matrix with 1

Check if there is a possible arrangement for the current position of then queen in the column

If no possible arrangement is found then backtrack and remove the queen and try placing the queen in another row

```
vector<vector<vector<int>>> nQueens(int N) {

    vector<vector<int>> board(N, vector<int>(N, 0));
    vector<vector<vector<int>>> solutions;

    solveNQueens(board, 0, N, solutions);

    return solutions;
}
```

Function to find all possible placements of N queens on an N x N chessboard

Initialize the chessboard with all 0's

Vector to store all solutions

```
int main() {
    int N = 8;
    vector<vector<vector<int>>> solutions = nQueens(N);
    for (int i = 0; i < solutions.size(); i++) {
        cout << "Solution " << i + 1 << ":\n";
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
                cout << solutions[i][j][k] << " ";
            }
            cout << "\n";
        }
        cout << "\n";
    }

    return 0;     //the program has executed successfully
}
```

Iterate through each solution in the solutions vector

Print the solution number, starting from 1

Iterate through each row of the current solution

Iterate through each column of the current row

Print the value at the current cell in the solution

After printing all the values in a row, print a newline character to move to the next line

After printing all the rows of the current solution, print another newline character to separate it from the next solution

4. Applications:

There are some real-life uses of this N-queen's problem, which may not cover all the aspects but it might work generally to establish the same goal as this problem. Even though designing games and sites hold broad usage of this algorithm, but we found some relatable applications of this in arrangement and scheduling genre. Some application scenarios are elaborated below:

****Course scheduling in universities:**

The N-Queens problem can be adapted to model the scheduling of courses in a university in such a way that course conflicts are minimized, allowing students to attend their desired courses without overlap.

In this scenario, consider each course as a "queen" and each time slot as a "row" on the chessboard. The N-Queens problem can be modeled as an N x N grid, where N is the number of courses to be scheduled. The goal is to assign each course (queen) to a time slot (row) such that no two courses with a significant overlap in students (threatening condition in the N-Queens analogy) are scheduled at the same time (same column, or diagonals).

The N-Queens problem can help identify a suitable course schedule that respects the constraints of student course selections and optimizes the course schedule to minimize conflicts. Solving the adapted N-Queens problem helps in creating a more efficient and flexible course schedule for students.

****Library bookshelf arrangement:**

The N-Queens problem can be adapted to model the arrangement of books on library bookshelves in a way that maximizes the accessibility and discoverability of different genres or topics while minimizing the chance of conflicts between books of similar interest placed in close proximity.

In this scenario, consider each bookshelf as a "queen" and each possible arrangement of genres or topics as a "row" on the chessboard. The N-Queens problem can be modeled as an N x N grid, where N is the number of bookshelves to be arranged. The goal is to assign each bookshelf (queen) a genre or topic (row) such that no two bookshelves with a significant overlap in content or target audience (threatening condition in the N-Queens analogy) are placed in close proximity (same column, or diagonals).

The N-Queens problem can help identify a suitable bookshelf arrangement that respects the constraints of reader interests and genre conflicts, optimizing the library experience for all patrons. Solving the adapted N-Queens problem helps in creating a more efficient and diverse library layout that promotes exploration and discoverability of various genres and topics.

****Task scheduling in parallel computing:**

In parallel computing systems, multiple tasks need to be executed simultaneously by a set of processors or computing units. The N-Queens problem can be adapted to model the assignment of tasks to processors in such a way that no two tasks with shared resources or data dependencies are executed by the same processor at the same time.

In this scenario, consider each task as a "queen" and each processor as a "row" on the chessboard. The N-Queens problem can be modeled as an N x N grid, where N is the number of tasks to be executed. The goal is to assign each task (queen) to a processor (row) such that no two tasks sharing resources or having data dependencies (threatening condition in the N-Queens analogy) are assigned to the same processor at the same time (column).

The N-Queens problem can help identify a suitable assignment of tasks to processors that respects the data dependencies and resource sharing constraints. Solving the adapted N-Queens problem helps in minimizing conflicts and improving the efficiency of parallel computing systems.

5. Discussion:

In conclusion, the N-Queens problem is a challenging puzzle that involves placing N chess queens on an N×N chessboard in such a way that no two queens attack each other. In this project, a recursive backtracking algorithm was used to solve the problem and it provides a solution that can be applied to various scenarios involving chessboards of different sizes. The project aimed to create an efficient and flexible solution that respects specific constraints in real-world scenarios requiring optimal arrangements or assignments. The applications of the N-Queens problem were explored in three scenarios, namely course scheduling in universities, library bookshelf arrangement and task scheduling in parallel computing where the problem can be adapted to model the scheduling of courses and the arrangement of books on library bookshelves, respectively. Solving the adapted N-Queens problem helps in creating a more efficient and diverse layout that promotes exploration and discoverability of various genres and topics.

END