**KIET Group of Institutions, Ghaziabad**



***Computer Science and Information Technology***

**Design and Analysis of Algorithm Problem-based learning**

**Project**

**on**

# N-Queen Backtracking Visualizer

**Odd Semester (2022-23)**

## Submitted To: Mr. Praveer Dubey Submitted By: Devesh Chandra Branch/Sec : CSIT 5A

**ClassRoll No : 2000290110061**

**Index**

1. Acknowledgement
2. Introduction
   * Problem Statement
   * Motivation
3. Algorithm and Analysis
4. Implementation
   * Flowchart
   * Program
   * Output
5. Applications
6. Conclusion or Future Scope

# ACKNOWLEDGEMENT

I’ve got this golden opportunity to express my kind gratitude and sincere thanks to **Mr. Abhinav Juneja,** Head of Department of **Computer Science and Information Technology** and to our mentor **Mr. Praveer Dubey** for the time and efforts they provided throughout the year. Their useful advice and suggestions were helpful to me during the project’s completion. I’m also indebted to each and every person responsible for the making up of this project directly or indirectly.

I must also acknowledge or deep debt of gratitude each one of my colleague who led this project come out in the way it is. It’s my hard work and untiring sincere efforts and mutual cooperation to bring out the project work. Last but not the least, I would like to thank my parents for their sound counselling and cheerful support. They have always inspired us and kept our spirit up.

**Devesh Chandra CSIT 5A 2000290110061**

# INTRODUCTION

## PROBLEM STATEMENT :

The N-Queens problem is to place N-Queens on an NxN chessboard such that no queens attack each other.

Backtracking is a prime algorithmic technique that uses depth first search to explore the solution space which is most naturally formulated in terms of recursive programs. The backtracking algorithm, in general, checks all possible configurations and tests whether the required result is obtained or not.

This project visualizes the recursive solution algorithm. Here i'm using graphical user interface to view various solutions to the N-Queens problem.

## MOTIVATION:

The N queens puzzle is the problem of placing eight 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. The eight queens puzzle is an example of themore general n queens problem of placing n non-attacking queens on an `n×n` chessboard, for which solutions exist for all natural numbers `n` with the exception of `n=2` and `n=3`.

The problem of finding all solutions to the N-queens problem can be quite computationally expensive, as there are 4,426,165,368 possible arrangements of eight queens on an 8×8 board, but only 92 solutions. It is possible to use shortcuts that reduce computational requirements or rules of thumb that avoids brute-force computational techniques. For example, by applying a simple rule that chooses one queen from each column, it is possible to reduce the number of possibilities to 16,777,216 (that is, 88) possible combinations. Generating permutations further reduces the possibilities to just 40,320 (that is, 8!), which can then be checked for diagonal attacks.

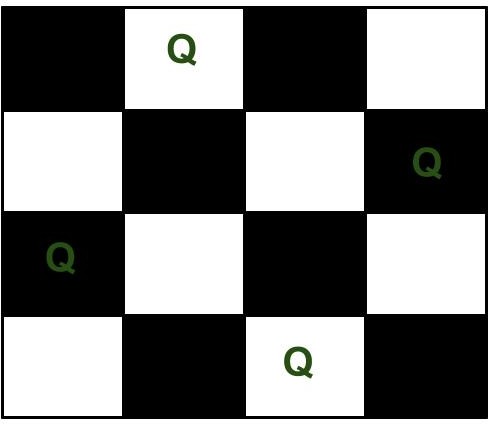
Finding all solutions to the eight queens puzzle is a good example of a simple but nontrivial problem. For this reason, it is often used as an example problem for various programming techniques, including nontraditional approaches such as constraint programming, logic programming or genetic algorithms. Most often, it is used as an example of a problem that can be solved with a recursive algorithm, by phrasing the *n* queens problem inductively in terms of adding a single queen to any solution to the problem of placing *n*−1 queens on an *n*×*n* chessboard. The induction bottoms out with the solution to the 'problem' of placing 0 queens on the chessboard, which is the empty chessboard.

Here we have emphasized and demonstrated the effectiveness of the procedure of the Backtracking approach through visual representation.

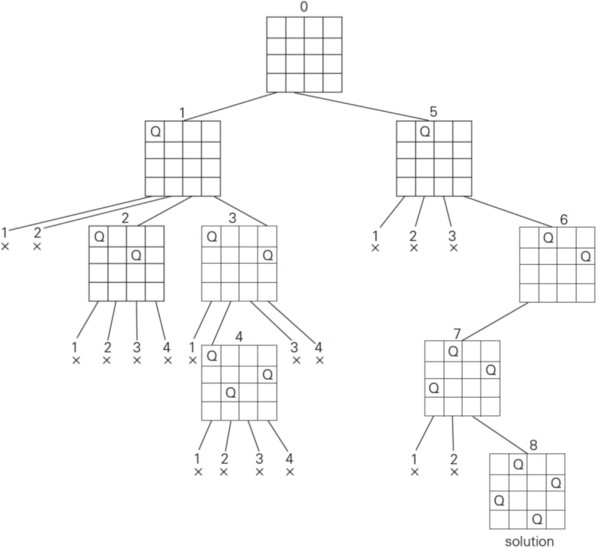
# Algorithm(with Analysis)

The N queens puzzle is the problem of placing eight 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. The eight queens puzzle is an example of themore general n queens problem of placing n non-attacking queens on an `n×n` chessboard, for which solutions exist for all natural numbers `n` with the exception of `n=2` and `n=3`.

For example, following is a solution for 4 Queen problem.



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example following is the output matrix for above 4 queen solution.



|  |  |  |
| --- | --- | --- |
| { 0, | 0, | 0, 1} |
| { 0, | 0, | 0, 1} |
| { 1, | 0, | 0, 0} |
| { 0, | 0, | 1, 0} |

## Naive Algorithm

Generate all possible configurations of queens on board and print a configuration that satisfies the given constraints.

while there are untried configurations

{

generate the next configuration

if queens don't attack in this configuration then

{

print this configuration;

}

}

## Backtracking Algorithm

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.

1. Start in the leftmost column
2. If all queens are placed return true
3. Try all rows in the current column. Do following for every tried row.
   1. If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
   2. If placing queen in [row, column] leads to a solution then return true.
   3. If placing queen doesn't lead to a solution then umark this [row, column] (Backtrack) and go to step (a) to try other rows.
4. If all rows have been tried and nothing worked, return false to trigger backtracking.

## Bitwise Solution

Bitwise algorithm basically approaches the problem like this:

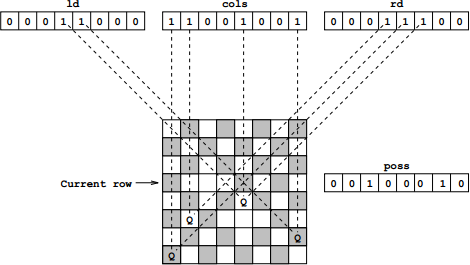
* + Queens can attack diagonally, vertically, or horizontally. As a result, there can only be one queen in each row, one in each column, and at most one on each diagonal.
  + Since we know there can only one queen per row, we will start at the first row, place a queen, then move to the second row, place a second queen, and so on until either a) we reach a valid solution or b) we reach a dead end (ie. we can't place a queen such that it is "safe" from the other queens).
  + Since we are only placing one queen per row, we don't need to worry about horizontal attacks, since no queen will ever be on the same row as another queen.
  + That means we only need to check three things before placing a queen on a certain square: 1) The square's column doesn't have any other queens on it, 2) the square's left diagonal doesn't have any other queens on it, and 3) the square's right diagonal doesn't have any other queens on it.
  + If we ever reach a point where there is nowhere safe to place a queen, we can give up on our current attempt and immediately test out the next possibility.

First let's talk about the recursive function. You'll notice that it accepts 3 parameters: leftDiagonal, column, and rightDiagonal. Each of these is technically an integer, but the algorithm takes advantage of the fact that an integer is represented by a sequence of bits. So, think of each of these parameters as a sequence of N bits.

Each bit in each of the parameters represents whether the corresponding location on the current row is "available".

For example:

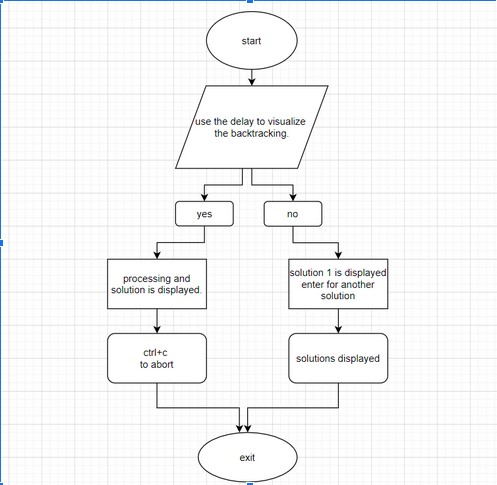
* + For N=4, column having a value of 0010 would mean that the 3rd column is already occupied by a queen.
  + For N=8, ld having a value of 00011000 at row 5 would mean that the top-left-to- bottom-right diagonals that pass through columns 4 and 5 of that row are already occupied by queens.

Below is a visual aid for leftDiagonal, column, and rightDiagonal.

# IMPLEMENTATION

Various techniques like brute force, permutations,greedy methods which can be used to solve n queens problem but among all backtracking with recursion is very efficient. Since backtracking with recursion is useful to solve the problem where the iteration is more prominent in finding the solutions, it is used to solve n queens problem . Graphical visualization of n-queens problem using backtracking with recursion is successful to the extent of finding and displaying all the distinct solutions to the single digit values of n.

## FLOWCHART:



**PROGRAM:**

//

#define COLOR\_BGW "\x1b[47m" #define COLOR\_BGB "\x1b[100m" #define COLOR\_BLK "\x1b[30m" #define COLOR\_RESET "\x1b[0m" #include <stdio.h>

#include <stdlib.h> #include <string.h> #include <time.h>

typedef enum {false,true} bool; void continuee() {

printf("\n Press [Enter] for another solution.");

while (getchar() !='\n');

}

void wait(unsigned int secs) { // Wait x seconds unsigned int retTime = time(0) + secs; // Get finishing time. while (time(0) < retTime); // Loop until it arrives.

}

int color(int i, int j) { return (i+j)%2;

}

int countSLT() { static int k = 0; return k++;

}

void printBoard(int N, int board[N][N], int delay) { system("clear");

int i,j;

if (delay) printf("\n N-QUEENS (delay=1s)\n\n"); else printf("\n N-QUEENS\n\n");

for (i=0; i<N; i++) { printf(" %d ", i+1); for (j=0; j<N; j++)

if (!color(i,j)) {

if (board[i][j])

printf(COLOR\_BGW COLOR\_BLK "\u265B" COLOR\_RESET);

else

printf(COLOR\_BGW " " COLOR\_RESET);

}

else {

if (board[i][j])

printf(COLOR\_BGB COLOR\_BLK "\u265B" COLOR\_RESET);

else

printf(COLOR\_BGB " " COLOR\_RESET);

}

printf("\n");

}

printf(" ");

for (i=0; i<N; i++) { printf(" %c ", i+97);

}

if (delay) printf("\n\n Processing...\n (ctrl+c to abort)\n"); else printf("\n");

}

bool valida(int N,int board[N][N], int row, int col) { int i,j;

for (i=0; i<col; i++) if (board[row][i])

return false;

for (i=row,j=col; i>=0 && j>=0; i--,j--) if (board[i][j])

return false;

for (i=row, j=col; j>=0 && i<N; i++,j--) if (board[i][j])

return false;

return true;

}

/\*bool solve(int board[N][N], int col) { int i;

if (col>=N) return true; for (i=0; i<N; i++) {

if (valida(board,i,col)) { board[i][col] = 1;

if (solve(board,col+1)) return true;

board[i][col] = 0;

}

}

return false;

} \*/

bool solve(int N, int board[N][N], int col, int delay) { int i;

bool solution = false;

if (col == N) { printBoard(N,board,delay);

printf("\n Solution: %d", countSLT()); continuee();

return true;

}

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

if (valida(N,board,i,col)) { board[i][col] = 1; printBoard(N,board,delay); if (delay) wait(1);

solution= solve(N,board,col+1,delay) || solution; board[i][col] = 0;

printBoard(N,board,delay); if (delay) wait(1);

}

}

return solution;

}

void callNQ() { system("clear"); int N = 50;

int board[N][N]; bool delay = false; char ask;

printf("\n Enter N for the number of Queens, \n Use the delay to visualize the backtracking? [y/n] :\n ");

scanf("%d %c",&N, &ask);

// printf("\n Use the delay to visualize the backtracking? [y/n]");

//ask = getchar();

// scanf("%c",&ask);

if (ask == 'y') delay = true; memset(board,0,sizeof(board)); if (!solve(N,board,0,delay)) {

printf("There is no solution\n"); return;

}

return;

}

int main(void) { callNQ(); return 0;

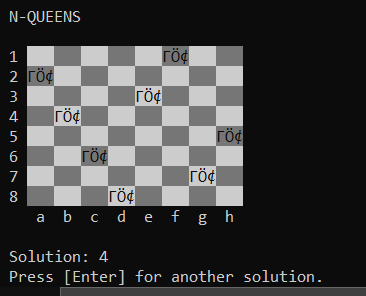
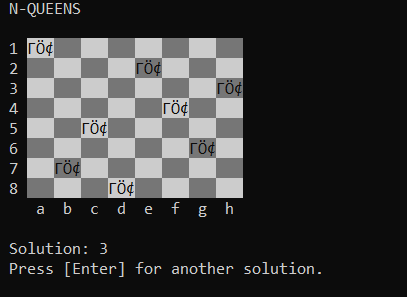
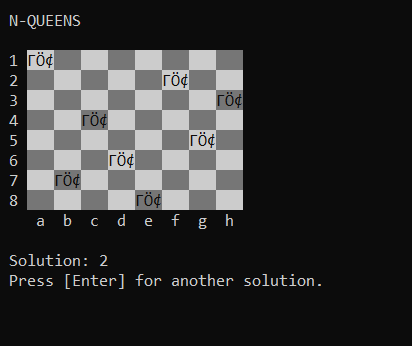
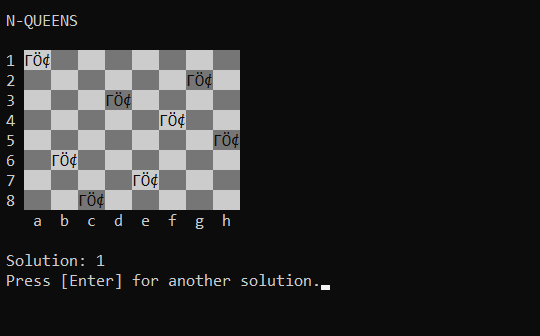
}

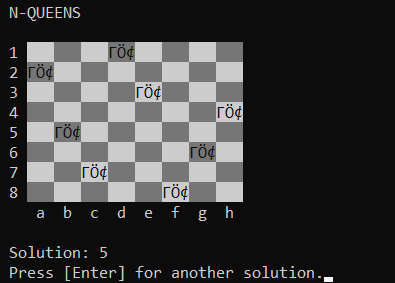
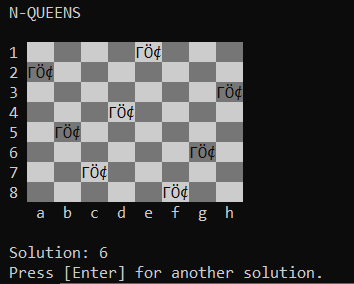
### OUTPUT:

Enter N for the number of Queens: 8

Use the delay to visualize the backtracking? [y/n] : n

( A total of 92 disinct solution exitsts when N is 8, few solutions are shown as below )





# APPLICATIONS OF N-QUEEN

There are Variety of N-Queen Applications that is deal with the daily life and real world problems. Some of these are given below:

1. VLSI Testing.
2. Traffic control.
3. Deadlock Prevention.
4. Image Processing.
5. Motion Estimation.

# CONCLUSION AND FUTURE SCOPE

N Queen problem represents a class of constraint problem.It belongs to a set of NP Hard problems. It is applicable in many areas of science and engineering.

The backtracking solver will find the solution for us. But as the N increases it becomes slower. If N=25, it would take 322.89 seconds to find the solution and when N=26, it would take forever! So an approach is needed which could find the solution pretty much quicker. Therefore some other optimized approach can be developed to solve it more efficiently.

The efficiency of the traditional backtracking algorithm may be improved by the use of a hybrid approach taking advantage of sets to reduce the number of trials and error attempts. Time taken to solve the n-queen problem in the backtracking is more than that of the Tuned hybrid technique. Space taken to solve the n-queen problem in the backtracking is more than that of the Tuned hybrid technique. Complexity Analysis can be improved using different algorithms and that approach will be applied on the one of the applications of the N-Queen Problem to obtain the fast and better solution. Complexity Analysis can be based in the time, space, convergence-rate and conflict-minimization.

Genetic algorithm is a well-known optimization technique. The problem has to be represented in genetic form to solve it using genetic algorithm. Genetic Algorithms are a family of algorithms whose purpose is to solve problems more efficiently than usual standard algorithms by using natural science metaphors with parts of the algorithm being strongly inspired by natural evolutionary behaviour; such as the concept

of **mutation**, **crossover** and **natural selection**. The Backtracking solver could only solve the problem at maximum N=25 which took about 322.89 seconds to find the answer. The GA solver found the solution for N=25 in just 1.83 seconds.

### Table: 1 Time Complexity based on the number of Queens (n).

