//Name : Vidula Magdum

//Roll : 31249

//Group B : 1

//Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and

//Backtracking for n-queens problem or a graph coloring problem

// Constraint satisfaction means solving a problem under certain constraints or rules.

// With backtracking

// The search space is can be very large

// It is an exhaustive search

// Worst case complexity is exponential

// Branch and bound technique

// Limits the search space

// Through an estimate of the

// Upper bound or

// Lower bound

// In backtracking solution we backtrack when we hit a dead end. In Branch and Bound

// solution, after building a partial solution, we figure out that there is no point

// going any deeper as we are going to hit a dead end.

//Nqueen with backtracking

#include<iostream>

#include<vector>

using namespace std;

void printBoard(vector<vector<int>>&board)

{

cout<<"Solution matrix :"<<endl;

for(int i=0;i<board.size();i++)

{

for(int j=0;j<board.size();j++)

{

cout<<board[i][j]<<"\t\t";

}

cout<<endl;

}

}

bool isSafe(vector<vector<int>>&board,int row,int col)

{

int i, j;

// Check this row on left side

for (i = 0; i < col; i++)

if (board[row][i])

return false;

// Check this row on right side

for (i = col+1; i < board.size(); i++)

if (board[row][i])

return false;

//Check upper diagonal on left side

for (i = row-1, j = col-1; i >= 0 && j >= 0; i--, j--)

if (board[i][j])

return false;

//Check upper diagonal on right side

for (i = row-1, j = col+1; i >= 0 && j < board.size(); i--, j++)

if (board[i][j])

return false;

//Check lower diagonal on left side

for (i = row+1, j = col-1; j >= 0 && i < board.size(); i++, j--)

if (board[i][j])

return false;

//Check lower diagonal on right side

for (i = row+1, j = col+1; j < board.size() && i < board.size(); i++, j++)

if (board[i][j])

return false;

//Check upper portion of current column

for (i = row-1; i >=0; i--)

if (board[i][col])

return false;

//Check lower portion of current column

for (i = row+1; i < board.size(); i++)

if (board[i][col])

return false;

return true;

}

bool recurNqueen(vector<vector<int>>&board,int col,int N,int fixedrow,int fixedcol,int&count)

{

//placing a queen in a fixed row and column

board[fixedrow][fixedcol]=1;

// base case: If all queens are placed hen return true

if (col >= N)

return true;

//Placing the queen in column col and trying every row to find the safe place

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

{

//Check if queen is safe at position board[i][col]

if (isSafe(board, i, col))

{

board[i][col] = 1;

//recursive call to place other queens

if (recurNqueen(board, col + 1,N,fixedrow,fixedcol,count))

{

count++;

printBoard(board);

}

//above if condition failed, i.e the traced path isnt able to find safe place for

// some queen, therefore bactrack

board[i][col] = 0; // ! BACKTRACK

board[fixedrow][fixedcol]=1;

}

}

return false;

}

void voidNQueen(int N,int fixedrow,int fixedcol)

{

int count=0; //tells how many solutions were obtained for the given

//matrix size and fixed queen position

//create a board of size N X N

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

recurNqueen(board, 0, N,fixedrow,fixedcol,count);

if(count == 0)

cout<<"No solution for "<<N<<" Queens \n\n\n";

return;

}

int main()

{

int opt;

while(true)

{

cout<<"1.Solve N Queens Problem using Backtracking\n2.Exit\n\n";

cout<<"Select an option : ";

cin>>opt;

if(opt == 1)

{

int N,fixedrow,fixedcol;

cout<<"Enter the dimension of the board :";

cin>>N;

cout<<"Now enter valid position to preplace a queen ";

cin>>fixedrow>>fixedcol;

voidNQueen(N,fixedrow,fixedcol);

}

else if(opt == 2)

{

break;

}

}

}