CSE 402: Artificial Intelligence Sessional
Assignment 4: N-Queens Problem

Introduction. In this assignment, you will solve the famous N-Queens problem. You will model the problem first as a constraint satisfaction problem (CSP). You will then write a backtrack algorithm that will solve the CSP. In addition, you will also use minimum remaining (MRV) value heuristic for variable ordering and forward checking. You will need to print a valid solution first followed by all valid solutions of the problem.

Problem definition. An N queens problem asks you to place N queens in an $N \times N$ chessboard so that no two queens attack each other. The following figure shows a valid solution of N-Queens problem for $N = 8$ where Q indicates a placement of a queen.

	1	2	3	4	5	6	7	8
A			Q					
B					Q			
C		Q						
D								Q
E	Q							
F							Q	
G				Q				
H						Q		

Figure 1: A solution to the 8-Queens problem

N-Queens as a CSP. First, we will model the problem as a CSP.

We will assume N variables $X_1, X_2, \dots, X_N$ where

- Each variable X_i represents the column # of the queen in the i −th row.
- Each variable X_i can have one of the eight values: $1, 2, \dots, 8$.
- A variable $X_i = j$ implies that in the i-th row a queen is placed in $j - th$ column

Being a CSP, we will have the following three constraints for the N-queens problem:

- $X_i \neq X_j \ for \ 1 \leq i < j \leq 8$ (vertical attack constraint)
- $X_i \neq X_j + (j - i) \ for \ 1 \leq i < j \leq 8$ (diagonal attack constraint 1)
- $X_i \neq X_j - (j - i) \ for \ 1 \leq i < j \leq 8$ (diagonal attack constraint 2)

For the solution in Figure 1, $(X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8) = (3, 5, 2, 8, 1, 7, 4, 6)$. You should check whether all constraints are satisfied.

A simple backtrack algorithm. A simple backtrack algorithm for N-Queens problem is given below:

```
Algorithm $PlaceNQueens(X, N, i)$
if $i = N + 1$
    print solution $X[1..N]$
    return $true$
for $j \leftarrow 1$ $to$ $N$
    $X[i] \leftarrow j$ //place a queen for row i in column j
    $flag \leftarrow CheckConstraints(X, N, i)$
    if $flag = false$ continue //try next action in current step
    $succ \leftarrow PlaceNQueens(X, N, i + 1)$ //perform remaining steps recursively
    if $succ = true$ return $true$
return $false$
```

```
Algorithm $CheckConstraints(A, n, r)$
for $i \leftarrow 1$ $to$ $r - 1$
    if ( $A[i] = A[r]$ $or$ $A[r] - (r - i) = A[i]$ $or$ $A[r] + (r - i) = A[i]$ )
        return $false$
return $true$;
```

The above algorithm successfully prints one ***valid*** solution of the n Queens problem. In case, we need all solutions we can modify the above algorithm as follows:

```
Algorithm $PlaceNQueens(X, N, i)$ //places queens row by row incrementally
If $i = N + 1$
    print solution $X[1..N]$
    return $true$
for $j \leftarrow 1$ $to$ $N$
    $X[i] \leftarrow j$ //place a queen for row i in column j
    $flag \leftarrow CheckConstraints(X, i, j)$
    if $flag = false$ continue //try next position for queen at row i
    $succ \leftarrow PlaceNQueens(X, N, i + 1)$//perform remaining steps recursively
return $false$
```

Running time of the simple backtrack algorithm. As a recursive algorithm, we may try to write the running time equation of the above $PlaceNQueens$ backtrack algorithm as follows: Let, r be the number of rows remaining for the current call. Then the ***time for the current recursive call*** can be written as $T(r) = T(r - 1) * n + f(n)$ wehre $f(n)$ is the time required by the $CheckConstraints$ function. If we assume that $f(n) = O(1)$, then the solution of the above recurrence would be $T(r) = O(n^r)$. Since, for the first call $r = n$, the total run-time will be $T(n) = O(n^n)$ which is very huge! Even remember that, we assumed $f(n)$ to be $O(1)$, actually which is not true!!! Actually $f(n) = O(n)$, however, this would result in $T(n) = O(n^n)$.

CSE 402: Artificial Intelligence Sessional

Your task 1: Implement the simple backtrack algorithm to solve the N-Queens problem. For input/output see the input/output section below. (50 Points)

Using forward checking. We can make the simple backtracking algorithm efficient by using Forward Checking. Whenever we assign a value to a variable $X[i]$, i.e., select a column for queen at row i, we can mark all cells in other rows that are at an attacking position with $X[i]$. Consider figure 2 where we chose column 3 to place a queen at row 1. Then we mark all cells as X in other rows that are at attacking position with $X[1]$. These marked cells will not be considered later for placing queens. This idea is called forward checking. As part of task 2, you will need to implement forward checking.

	1	2	3	4	5	6	7	8
A			Q					
B		X	X	X				
C	X		X		X			
D			X			X		
E			X				X	
F			X					X
G			X					
H			X					

Figure 2: Use of forward checking to mark cells that will not be used for later assignments.

Using MRV heuristic for variable ordering. In the simple backtrack version, queens are placed in the rows in numerical order of row numbers. That means first a queen is placed at row 1, then a queen is placed at row 2, and so on. A more efficient strategy could be to select the rows for placing queens based on some heuristic value. One idea is to use MRV heuristic. According to this heuristic, the row which have maximum number of X's (minimum number of valid columns) is selected for placing a queen. In this strategy, early failure occurs for invalid assignments and therefore saves running time of the algorithm. For example, in Figure 2, after placing the first queen at row 1, either row 2 or row 3 can be selected as the next row for placing a queen. This is because both rows have minimum number of valid columns (=5) remaining. Suppose we select row 3 and place a queen at a valid cell 4. Figure 3 shows the valid and invalid cells after forwarding checking marked invalid cells as X. From figure 3 we see that the next row will either be row 5 or 6 (both rows have 3 valid cells remaining for queen placement). As part of task 2, you will need to implement MRV heuristic.

	1	2	3	4	5	6	7	8
A			Q					
B		X	X	X	X			
C	X	X	X	Q	X	X	X	X
D			X	X	X	X		
E		X	X	X		X	X	
F	X		X	X			X	X
G			X	X				X
H			X	X				

Figure 3: Invalid cells marked by Forward checking after Queen is placed at C4.

Pseudocode with FC and MRV. The following is the pseudo-code for N-queens problem that includes the FC and MRV checking.

```
Algorithm $PlaceNQueensFCandMRV(X,N)$ //places queens row by row incrementally
If  $NoMoreRows$
    print solution $X[1..N]$
    return $true$
$i \leftarrow SelectRowMRV$ //select a row to place queen using MRV heuristic
for $j$ $in$ $\{valid\ cells\ remaining\ at\ row\ i\}$
    $X[i] \leftarrow j$ //place a queen for row i in column j
    $flag \leftarrow ForwardCheck(X,i,j)$
    if $flag = false$ continue //try next position for queen at row i
    $succ \leftarrow PlaceNQueens(X,N)$//perform remaining steps recursively
return $false$
```

Note that, in the above pseudo-code, $ForwardCheck$ function should return false whenever it observes that there is no valid cell in some row.

Your task 2: Implement the backtrack algorithm with FC and MRV to solve N-Queens problem. For input/output see the input/output section below. (FC=25 points, MRV=25 points).

Language. C++/Java. No other language is allowed and your code will not be evaluated. ***All Your codes should be in a single cpp/java file.***

Input/output. Take the value of N as input. Then first print a valid solution as N integer numbers separated by space in a line where i-th number corresponds to the queen position (column no.) of row i.

Then print an integer number M where M is the total number of valid solutions possible. Then print all valid solutions in M lines.

Sample input:

8

Sample output:

3 5 2 8 1 7 4 6

(Then print number of valid solutions and all valid solutions one by one in a line)