Vishwakarma Institute of Technology
Department of Computer Science Engineering (CS)

Name	Vaishnavi Vijay Arthamwar
Division	CS-A
Batch	B3
Roll No.	16
PRN No.	12220242
Department	Computer
Subject	Artificial Intelligence
Assignment No.	4

Assingment No. 4

Title: Implementation of CSP Problem

Theory :

Constraint satisfaction problems (CSPs) are a type of problem that can be solved by finding values for a set of variables that satisfy a given set of constraints. CSPs are widely used in artificial intelligence, as they can model many real-world problems such as scheduling, planning, map coloring, sudoku, and crossword puzzles.

Now,we will see how to formulate CSP problems using three basic components: variables, domains, and constraints. We will also see an example of how to solve a CSP problem using a simple algorithm.

Variables

Variables are the entities that need to be assigned values in order to solve the problem. For example, in a map coloring problem, the variables are the regions or countries that need to be colored. In a sudoku problem, the variables are the cells that need to be filled with numbers.

Domains

Domains are the sets of possible values that each variable can take. For example, in a map coloring problem, the domain of each variable is the set of colors that can be used to color the map. In a sudoku problem, the domain of each variable is the set of numbers from 1 to 9.

Constraints

Constraints are the rules that restrict the possible assignments of values to variables. Constraints can be unary, binary, or higher-order, depending on how many variables they involve. For example, in a map coloring problem, a binary constraint is that two adjacent regions cannot have the same color. In a sudoku problem, a higher-order constraint is that each row, column, and 3x3 box must contain all the numbers from 1 to 9.

Formulating a CSP Problem

To formulate a CSP problem, we need to specify the following three components:

- X: A finite set of variables, such as {X1, X2, …, Xn}.
- D: A set of domains, one for each variable, such as {D1, D2, …, Dn}.
- C: A set of constraints, each of which is a pair <scope, relation>, where scope is a tuple of variables that participate in the constraint and relation is a set of valid combinations of values for those variables.

For example, let us formulate the map coloring problem for the map of Australia, as shown below:

The CSP formulation for this problem is:

- X: {WA, NT, SA, Q, NSW, V, T}, where each variable represents a state or territory of Australia.
- D: {red, green, blue}, where each variable has the same domain of three colors.
- C: {< (WA, NT), WA!= NT >, < (WA, SA), WA!= SA >, < (NT, SA), NT!= SA >, < (NT, Q), NT!= Q >, < (SA, Q), SA!= Q >, < (SA, NSW), SA!= NSW >, < (SA, V), SA!= V >, < (Q, NSW), Q!= NSW >, < (NSW, V), NSW!= V >}, where each constraint is a binary constraint that states that two adjacent regions must have different colors.

Water Jug Problem:

Water Jug Problem is also known as Water Pouring Puzzles, measuring puzzles and decanting problems. These belong to a class of puzzles, in which there are a finite and specific number of water jugs having predefined integral capacities, in terms of gallons or litres.

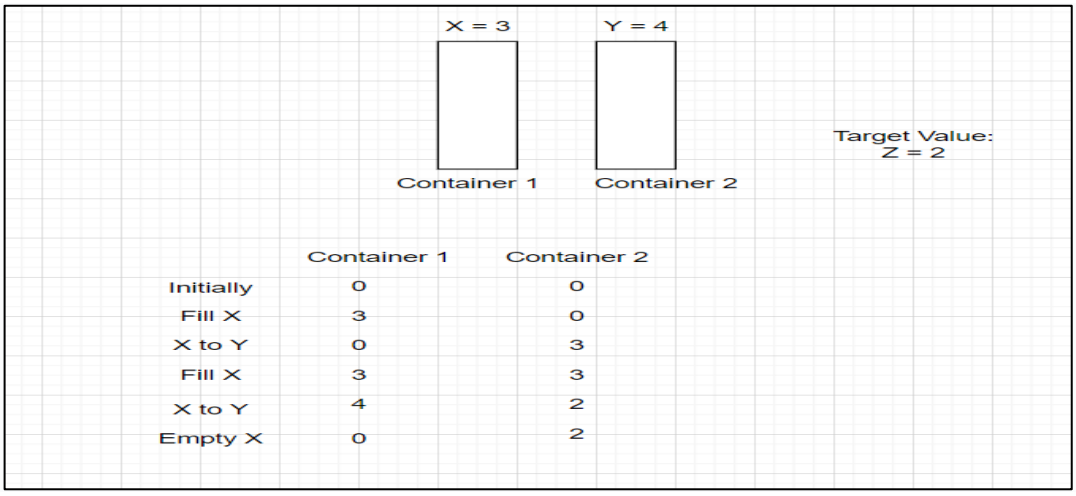
The prime challenge in the water jug problem is that these water jugs do not have any calibrations for measuring the intermediate water levels. In order to solve these puzzles, you are given a required measurement that you have to achieve by transferring waters from one jug to another, you can iterate multiple times until the final goal is reached, but in order to get the best results, the final solution should have a minimum cost and a minimum number of transfers.

Problem Statement:

You are given two jugs, one of m litre and another of n litre capacity. Both the jugs are initially empty. The jugs do not have any intermediate markings and labelling for measuring smaller quantities. You can be asked to measure d litres of water, such that d is less than n.

The operations you can perform are:

- Fill any of the jugs fully with water.
- Empty any of the jugs.
- Pour water from one jug into another till the other jug is completely full or the first jug itself is empty.



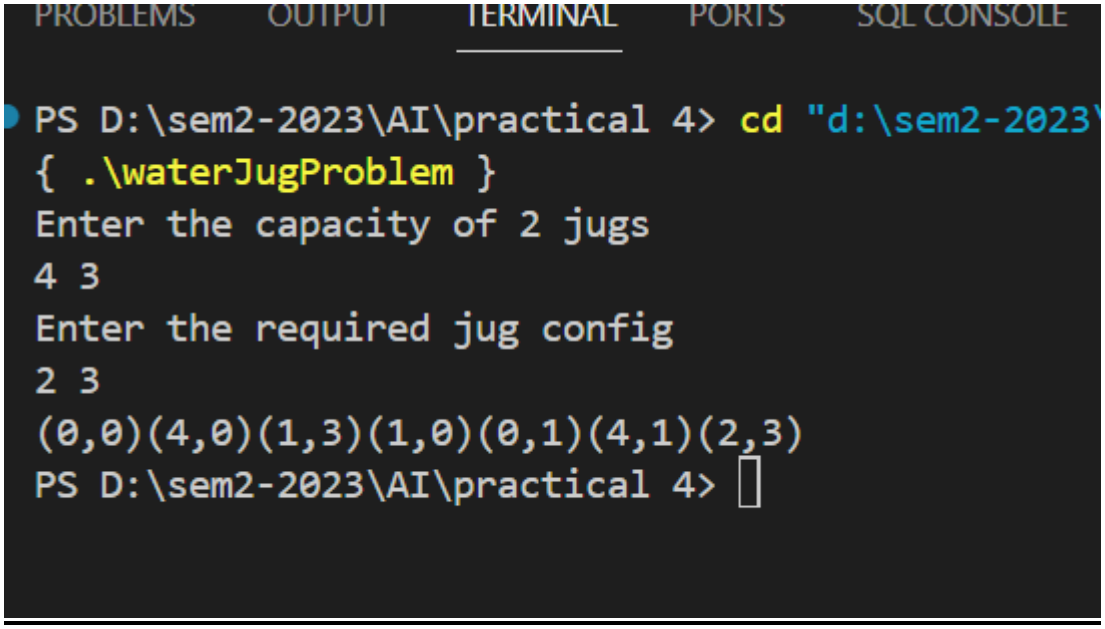
Code: Water Jug problem using Breadth-First Search and Queue. All the intermediary operations are written using if-else-if conditions.

```
#include <bits/stdc++.h>
using namespace std;
class nodes{
    public:
        pair<int,int> p;
        int first;
        int second;
        string s;
};
string makestring(int a,int b){
    std::stringstream out1;
    std::stringstream out2;
    string t1,t2,str;
    out1 << a;
    t1 = out1.str();
    out2 << b;
    t2 = out2.str();
    str = "("+t1+","+t2+")";
    return str;
}
int main()
{
    int counter = 0;
    ios::sync_with_stdio(false);
    //pair<int,int> cap,ini,final;
    nodes cap,ini,final;
    ini.p.first=0,ini.p.second=0;
    ini.s = makestring(ini.p.first,ini.p.second);
    //Input initial values
    cout<<"Enter the capacity of 2 jugs\n";
    cin>>cap.p.first>>cap.p.second;
    //input final values
    cout<<"Enter the required jug config\n";
    cin>>final.p.first>>final.p.second;
    //Using BFS to find the answer
    queue<nodes> q;
    q.push(ini);
    nodes jug;
    while(!q.empty()){
        //Base case
        jug = q.front();
        if(jug.p.first == final.p.first){// && jug.p.second ==
final.p.second){
            cout<<jug.s<<endl;
            // counter++;
```

```
            // if(counter==5)
            return 0;
        }
        nodes temp = jug;
        //Fill 1st Jug
        if(jug.p.first<cap.p.first){
            temp.p = make_pair(cap.p.first,jug.p.second);
            temp.s = jug.s + makestring(temp.p.first,temp.p.second);
            q.push(temp);
        }
        //Fill 2nd Jug
        if(jug.p.second<cap.p.second){
            temp.p = make_pair(jug.p.first,cap.p.second);
            temp.s = jug.s + makestring(temp.p.first,temp.p.second);
            q.push(temp);
        }
        //Empty 1st Jug
        if(jug.p.first>0){
            temp.p = make_pair(0,jug.p.second);
            temp.s = jug.s + makestring(temp.p.first,temp.p.second);
            q.push(temp);
        }
        //Empty 2nd Jug
        if(jug.p.second>0){
            temp.p = make_pair(jug.p.first,0);
            temp.s = jug.s + makestring(temp.p.first,temp.p.second);
            q.push(temp);
        }
        //Pour from 1st jug to 2nd until its full
        if(jug.p.first>0 && (jug.p.first+jug.p.second)>=cap.p.second){
            temp.p = make_pair((jug.p.first-(cap.p.second-
jug.p.second)),cap.p.second);
            temp.s = jug.s + makestring(temp.p.first,temp.p.second);
            q.push(temp);
        }
        //Pour from 2nd jug to 1st until its full
        if(jug.p.second>0 && (jug.p.first+jug.p.second)>=cap.p.first){
            temp.p = make_pair(cap.p.first,(jug.p.second-(cap.p.first-
jug.p.first)));
            temp.s = jug.s + makestring(temp.p.first,temp.p.second);
            q.push(temp);
        }
        //Pour all water from 1st to 2nd
        if(jug.p.first>0 && (jug.p.first+jug.p.second)<=cap.p.second){
            temp.p = make_pair(0,jug.p.first+jug.p.second);
            temp.s = jug.s + makestring(temp.p.first,temp.p.second);
            q.push(temp);
        }
```

```
        //Pour from 2nd jug to 1st until its full
        if(jug.p.second>0 && (jug.p.first+jug.p.second)<=cap.p.first){
            temp.p = make_pair(jug.p.first+jug.p.second,0);
            temp.s = jug.s + makestring(temp.p.first,temp.p.second);
            q.push(temp);
        }
        q.pop();
    }
    return 0;
}
```

Output:



N-Queens Problem:

Given an N×N chessboard, the task is to place N queens on the board such that no two queens threaten each other. Specifically, no two queens should share the same row, column, or diagonal.

Formulation of CSP for N-Queens Problem: Let's formalize the CSP for the N-Queens problem:

1. **Variables:**
    - For the N-Queens problem, each variable represents a column on the chessboard. The value assigned to each variable represents the row where a queen is placed in that column.
    - So, the variables are X1, X2, ..., XN, where Xi represents the row of the queen in the ith column.
2. **Domains:**
    - The domain D of each variable Xi consists of the set of rows where a queen can be placed in the ith column. In an N×N chessboard, this domain is {1, 2, ..., N}.
3. **Constraints:**
    - Constraints ensure that no two queens threaten each other. Specifically, we need to enforce that no two queens share the same row, column, or diagonal.
    - The constraints can be defined as follows:
        - No two queens can share the same row: $\forall i \neq j, Xi \neq Xj$
        - No two queens can share the same column: $\forall i \neq j, |Xi - Xj| \neq |i - j|$
        - No two queens can share the same diagonal: $\forall i \neq j, |Xi - Xj| \neq |i - j|$

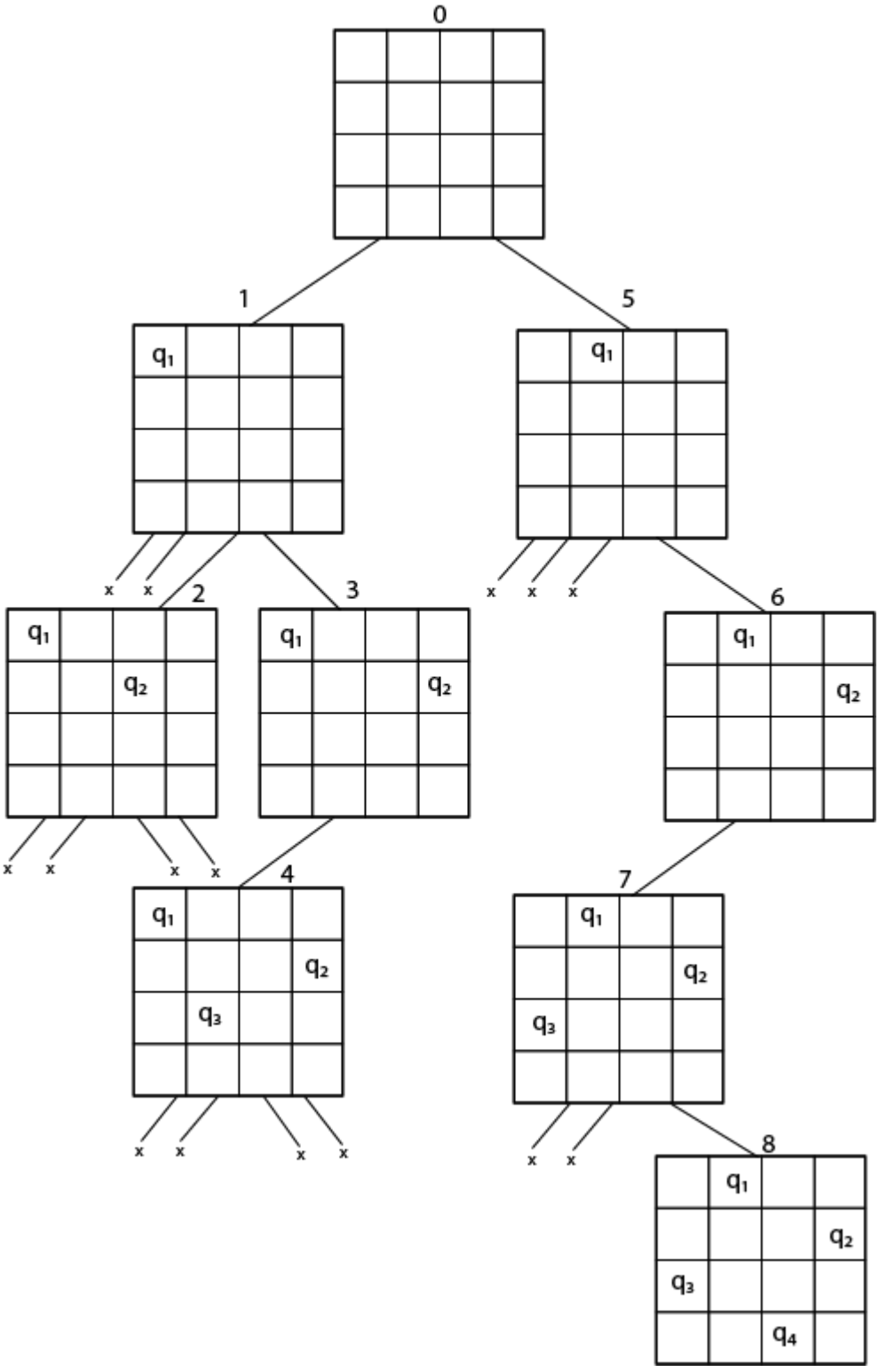
With this formulation, solving the CSP involves finding an assignment of rows to the variables such that no two queens threaten each other. If a solution is found, it corresponds to a valid placement of N queens on the chessboard. If no solution is found, it indicates that it's not possible to place N queens on the board without threatening each other.

Solving N-Queens Problem using Backtracking: The N-Queens problem can be efficiently solved using backtracking. Here's how:

1. Start with an empty board.
2. Place queens column by column:
    - For each column, try placing a queen in each row.
    - If placing a queen in a particular row doesn't violate any constraints, recursively try placing queens in subsequent columns.
    - If a valid placement of queens is found for all columns, a solution is found. If not, backtrack.

3. Backtrack:
    - If placing a queen in a particular row leads to a violation of constraints in subsequent columns, backtrack to the previous column and try the next row.

This process continues until all queens are placed on the board without threatening each other, or until all possible placements have been explored without finding a solution.

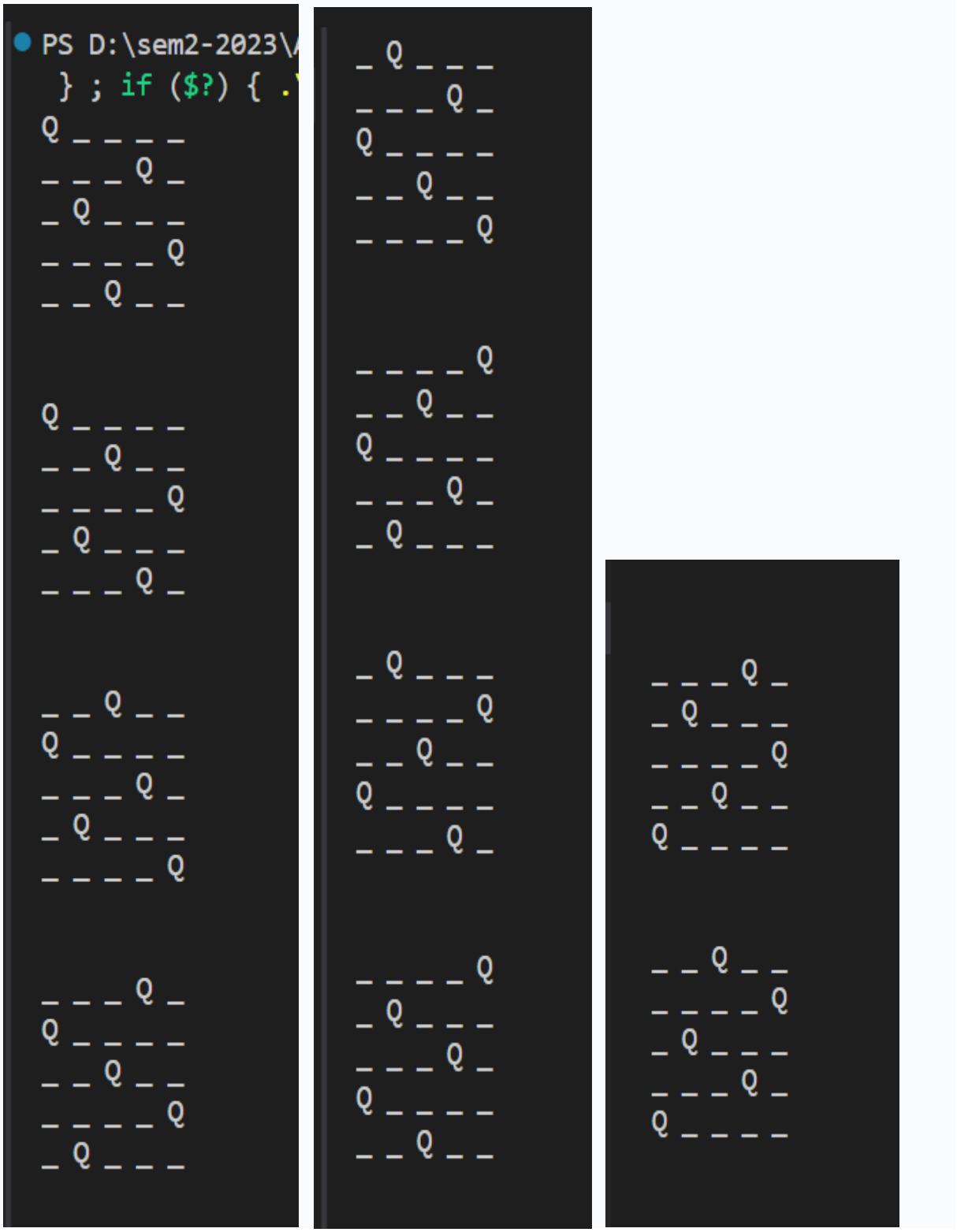


Code:

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

void printSolution(vector<vector<char>> board, int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl;
    cout << endl;
}

bool isSafe(int row, int col, vector<vector<char>> board, int n)
{
    int i = row;
    int j = col;
    while (j >= 0)
    {
        if (board[i][j] == 'Q')
        {
            return false;
        }
        j--;
    }
    i = row;
    j = col;
    while (i >= 0 && j >= 0)
    {
        if (board[i][j] == 'Q')
        {
            return false;
        }
        i--;
```

```
        j--;
    }
    i = row;
    j = col;
    while (i < n && j >= 0)
    {
        if (board[i][j] == 'Q')
        {
            return false;
        }
        i++;
        j--;
    }
    return true;
}
void solve(vector<vector<char>> board, int col, int n)
{
    // base case
    if (col >= n)
    {
        printSolution(board, n);
        return;
    }
    for (int row = 0; row < n; row++)
    {
        if (isSafe(row, col, board, n))
        {
            board[row][col] = 'Q';
            solve(board, col + 1, n);
            board[row][col] = '_';
        }
    }
}

int main()
{
    int n = 5;
    vector<vector<char>> board(n, vector<char>(n, '_'));
    int col = 0;
    solve(board, col, n);
    return 0;
}
```

Output:


PS D:\sem2-2023\
} ; if ($?) { .
Q _ _ _ _
_ _ _ Q _
_ Q _ _ _
_ _ _ _ Q
_ _ Q _ _
Q _ _ _ _
_ _ Q _ _
_ _ _ _ Q
_ Q _ _ _
_ _ _ Q _
_ _ Q _ _
Q _ _ _ _
_ _ _ Q _
_ Q _ _ _
_ _ _ _ Q
_ _ _ Q _
Q _ _ _ _
_ _ Q _ _
_ _ _ _ Q
_ Q _ _ _
_ Q _ _ _
_ _ _ Q _
Q _ _ _ _
_ _ Q _ _
_ _ _ _ Q
_ Q _ _ _
_ _ _ Q _
Q _ _ _ _
_ _ Q _ _
_ _ _ Q _
_ Q _ _ _
_ _ _ _ Q
_ _ Q _ _
Q _ _ _ _
_ _ _ Q _
_ Q _ _ _
_ _ _ _ Q
_ _ Q _ _
_ _ _ Q _
_ Q _ _ _
Q _ _ _ _
_ _ Q _ _
_ _ _ _ Q
_ Q _ _ _
_ _ _ Q _
Q _ _ _ _