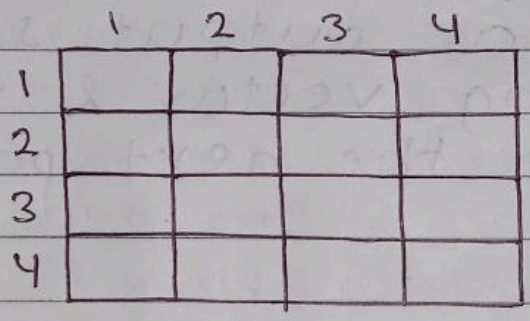
(class 3) 24-03-2023

Backtracking

Date.....

Ques N-Queen Problem.

N-Queen problem is to place n Queens in such a manner on n x n chessboar -d that no Queen attack each other by being in the same row, same column or same diagonal.
We've given 4x4 chessboard and 4 Queens:-

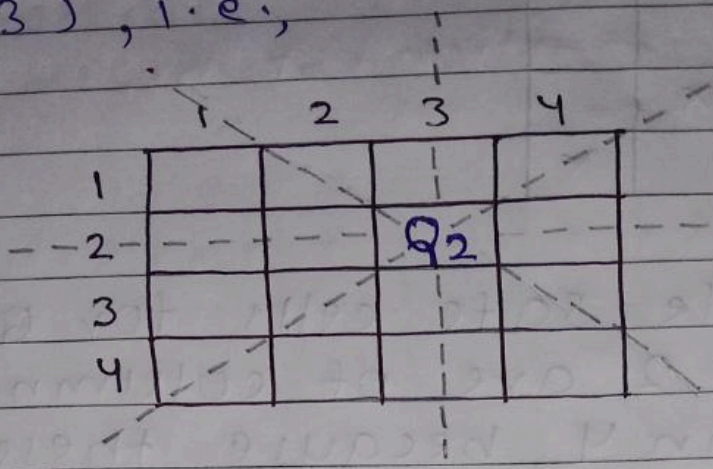


4x4 chess Board

Since we have to place 4 Queens such as Q_1, Q_2, Q_3, Q_4 on the chess board such that no two Queens attack each other.
In such condition each Queen must be placed on a different row, i.e., we put 'Q_i' on row 'i'.

Date.....

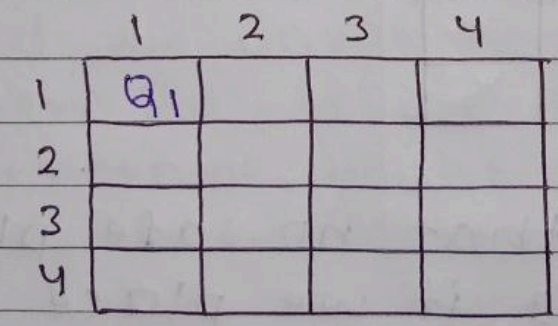
Let suppose we place first Queen at (2, 3), i.e.,



The position of Q_2 is (2, 3). It means, we can't place the next Queen in the same row & same column.

* The working algorithm to solve the 4-Queen problem :-

1.) create a 4 x 4 chessboard.
2.) Place the Queen Q_1 at the left most position which means row1 and column 1.



Mark the cells of the chessboard with cross marks that are under attack from Q_1.

	1	2	3	4
1	Q_1	X	X	X
2	X	X		
3	X		X	
4	X			X

3.) The possible safe cells for Queen Q_2 at row 2 are of column 3 & column 4, because these cells do not come under the attack from Queen Q_1.
So we place Q_2 at first possible safe cells which is row 2 & column 3.
And again make the cell of the board with cross marks that are under attack.

	1	2	3	4
1	Q_1	X	X	X
2	X	X	Q_2	X
3	X	X	X	X
4	X		X	X

4.) Now we see that no safe place is left for Q_3 if we place Q_2 at (2, 3)
So, make position (2, 3) false & backtrack.

Date.....

5.) Now, we place Q_2 at second possible safe cell which is row 2 & column 4.

	1	2	3	4
1	Q_1	x	x	x
2	x	x	x	Q_2
3	x		x	x
4	x	x		x

6.) The only remaining cell for Q_3 is row 3 & column 2.
So we place Q_3 at row 3 & column 2.

	1	2	3	4
1	Q_1	x	x	x
2	x	x	x	Q_2
3	x	Q_3	x	x
4	x	x	x	x

7.) Now, we don't have safe place for Queen Q_4, if we place Q_3 at (3, 2). Therefore, make position (3, 2) false & backtrack.
we can't move Q_2 at other positi -on, also we can't move Q_3 also at some other position.

Date.....

8.) This time we backtrack to the first Queen Q_1.
change the position of Q_1.

	1	2	3	4
1	×	Q_1	×	×
2	×	×	×	
3		×		×
4		×		

9.) The safe place for Q_2 is (2, 4)

	1	2	3	4
1	×	Q_1	×	×
2	×	×	×	Q_2
3		×		×
4		×		

10.) Place Q_3 at (3, 1)

	1	2	3	4
1	×	Q_1	×	×
2	×	×	×	Q_2
3	Q_3	×	×	×
4	×	×		

11.) Place Q_4 at (4, 3)

Date.....

	1	2	3	4
1	x	Q_1	x	x
2	x	x	x	Q_2
3	Q_3	x	x	x
4	x	x	Q_4	x

12.) Now, here we got the solution for 4 Queen problem because all 4 Queens are placed exactly in each row / column individual.

* Backtracking Algorithm for N-Queen problem :-

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 position, if we find a row which there is no clash, we mark the current row & column as a part of the solution. If we do not find such a row due to clashes, then we mark backtrack & return false.

code:-

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

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

// this function let us know that
placing the queen at current cell
is safe or not.
// by checking the previous left row,
upper left diagonal & lower left
diagonal
bool issafe(int row, int column,
  vector<vector<char>> &board, int n)
{
    //checking the left row
    int i=row;
    int j=column;
```

```
while (j >= 0)
{
    if (board[i][j] == 'Q')
    {
        return false;
    }
    j--;
}

// checking the upper left diagonal
i = row;
j = column;
while (i >= 0 and j >= 0)
{
    if (board [i][j] == 'Q')
    {
        return false;
    }
    i--;
    j--;
}

// checking the lower left diagonal
i = row;
j = column;
while (i < n and j >= 0)
{
    if (board[i][j] == 'Q')
    {
        return false;
    }
    i++;
```

Teacher's Sign .....

```
    }
    j--;
}

// if false is not returned till
now means the cell is safe so
we return true
return true;
}

void nQueenSolve(vector<vector
    <char>> &board, int column,
        int n)
{
    // base case --> we stop when we
    find 1 solution, i.e., when we
    place our Queen in all columns
    if(column >= n)
    {
        printSolution(board, n);
        return;
    }

    // now we try 1 Queen in all rows
    of current column & rest recurs
    -ion will handle.
    for(int row = 0; row < n; row++)
    {
        // at every iteration, first we
        check that placing Queen on
        current cell[i] is safe or not
        if(isSafe(row, column, board,
                n))
```

```
        {
            board[row][column] = 'Q';
            // we place 1 Queen above, now
            recursion will place rest
            nQueenSolve(board, column
                                + 1, n);
            // backtracking step
            board[row][column] = '-';
        }
    }
}

int main()
{
    int n;
    cout << "enter no. of queen; (n)
        you want to place in nxn
            matrix : ";
    cin >> n;
    if(n == 0 || n == 1 || n == 2 || n == 3)
    {
        cout << " solution not exist";
        return 0;
    }
    vector<vector<char>> board(n,
        vector<char>(n, '-'));
    int column = 0;
    cout << " Possible solutions are: ";
    nQueenSolve(board, column, n);
    return 0;
}
```

Teacher's Sign

Date.....

Algorithm:-

i) Start in the leftmost column.
ii) If all Queens are placed return true.
iii) Try all rows in the current column DO the following for every tried row :-
   a) If the Queen can be placed safely in this row, then mark current [row][column] as part of the solution & recursively check if placing Queen here leads to a solution.
   b) If placing the Queen in [row] [column] leads to a solution then return true.
   c) If placing the Queen doesn't lead to a solution then unmark the current [row][column] (Backtracking) & go to step a to try other rows.
iv) If all rows have been tried & nothing worked, return false to trigger Backtracking.