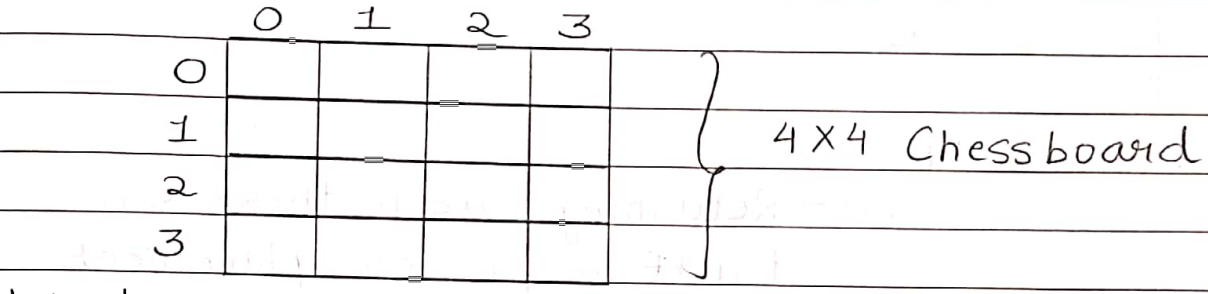
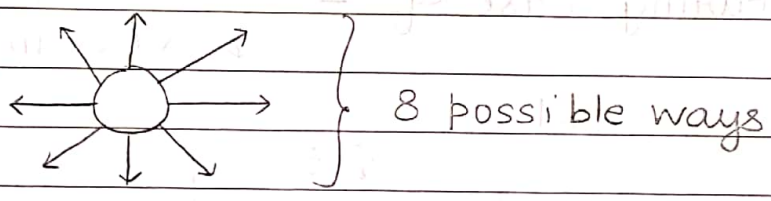
24/03/2023

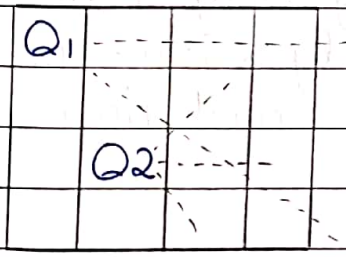
N-Queen Problem



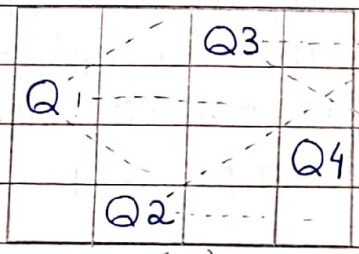
We have to place queens such that no queen can attack the other queen.
Queen can move in any direction.



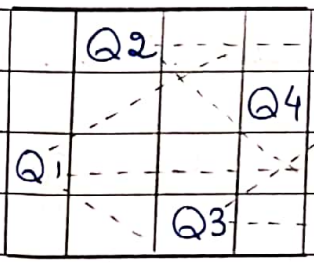
* In 2×2 chessboard, there is no arrangement of queens.
* In 3×3 chessboard, there is no possible arrangement of queens.
* Let's explore 4×4 checkboard



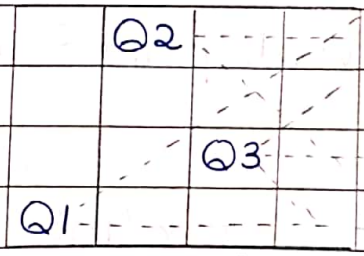
(1)



(2)



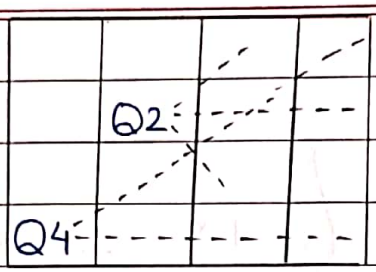
(3)



(4)

The dotted line indicates queen can't be placed there.

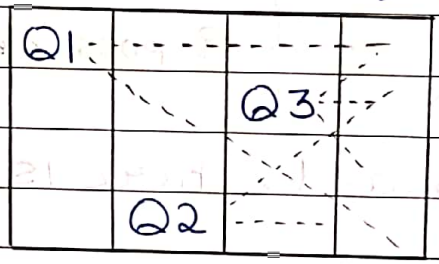
Q3) (can't be placed)



(4) → Returning case i.e there is a fault in previous placement

Hence 2 & 3 are the only possible solutions.

Note → Returing case of 1 (fault in previous placement)
↙ Q4 (can't be placed)



Hence 2 possible arrangements of queens are possible.

The above approach is having very high time complexity & we already know this as backtracking solutions have bad time complexity.

Code

```
void printSolution (vector <vector <int>> &board,
                    int n) {
    // 2 nested for loops to print 2d array
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
```

```
    cout << endl << endl;
}

bool isSafe (int row, int col, vector<vector
                <int>> & board, int n) {
// Can we place queen or not?
// Check 3 direxn → upper left, bottom left & left
int i = row;
int j = col;
//left row checking
while (j >= 0) {
        if (board[i][j] == 1) //Queen found
                return false;
        j--;
}
// bottom left
i = row;
j = col;
while (i<n && j >= 0) {
        if (board[i][j] == 1) //Queen found
                return false;
        i++;
        j--;
}
// Upper left
i = row;
j = col;
while (i >= 0 && j >= 0) {
        if (board[i][j] == 1) //Queen found
                return false;
        i--;
        j--;
}
```

```
return true;
}

void solve (vector<vector<int>> &board, int
            col, int n){
// Base case → Placed queens in all columns.
 if (col >= n){
        printSolution (board, n);
        return;
}
// Solve one case
// Place queen in every row & check for safety
for (int row = 0; row < n; row++){
      if (isSafe (row, col, board, n)){
            // Safe → Place queen
            board[row][col] = 1;
            // Recursive call for next column
            solve (board, col+1, n);
            // Backtracking → Recreate original state
            board[row][col] = 0;
      }
  }
}
```

1 → Queen present

0 → Queen absent

Optimization

Only can be done in isSafe as we can use the hashmaps to reduce the time complexity from O(n) to O(1).

In hashmaps, insertion & retrieval can be done in O(1) time.

Map → Stores values in the form of key-value pair
(string, int)

key → value
love → 98
babbar → 94

unordered_map <string, int> m;
↳ underscore

m["love"] = 98
m["babbar"] = 36
} Adding entry in map

* Left-row

Here in the N-Queens problem we will be creating a map of int, bool

unordered_map <int (row), bool> m;

* Bottom-left diagonal

	0	1	2	3
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6

→ row + col

unordered_map <row + col, bool> m;
Here pattern of row + col is used.

* Upper-left diagonal

(n-1) + col - row

	0	1	2	3
0	3	4	5	6
1	2	3	4	5
2	1	2	3	4
3	0	1	2	3

unordered_map <n-1+col-row, bool> m;
Here pattern of (n-1) + col - row is used.

Code

1) Create 3 unordered maps

```
unordered_map <int, bool> rowCheck;
unordered_map <int, bool> upperLeftDiagonalCheck;
unordered_map <int, bool> lowerLeftDiagonalCheck;
```

2) Modify isSafe function body.

```
left row ← if (rowCheck[row] == true)
                return false;
upper left diagonal ← if ( upperLeftDiagonalCheck[n-1+col-row]
                                == true)
                return false;
lower left diagonal ← if (lowerLeftDiagonalCheck[row+col] == true)
                return false;
    return true;
}
```

3) In solve function, when we have checked that it is safe to place queen, then

```
//Place Queen
board[row][col] = 1;
//Map modification
rowCheck[row] = true;
upperLeftDiagonalCheck[ n-1+col-row] = true;
lowerLeftDiagonalCheck[row+col] = true;
// Recursive Call
solve(board, col+1, n);
```

```
//Backtracking → Recreate original state
board[row][col] = 0;
rowCheck[row] = false;
upperLeftDiagonalCheck[n-1+col-row] = false;
lowerLeftDiagonalCheck[row+col] = false;
```