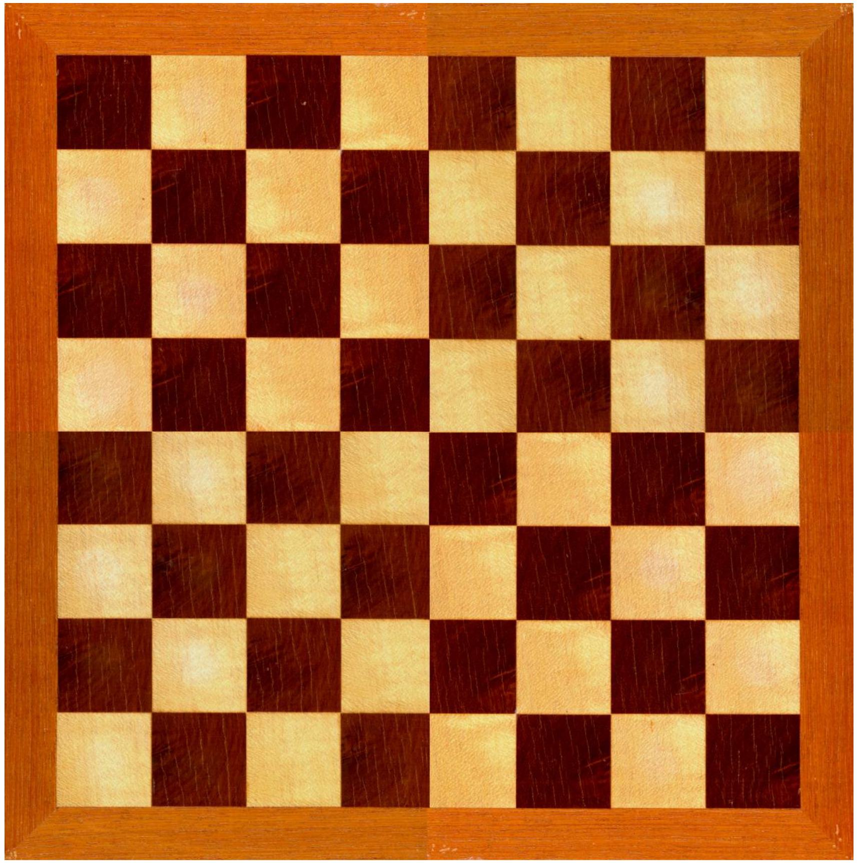
8-Queens Puzzle

Basic Rules

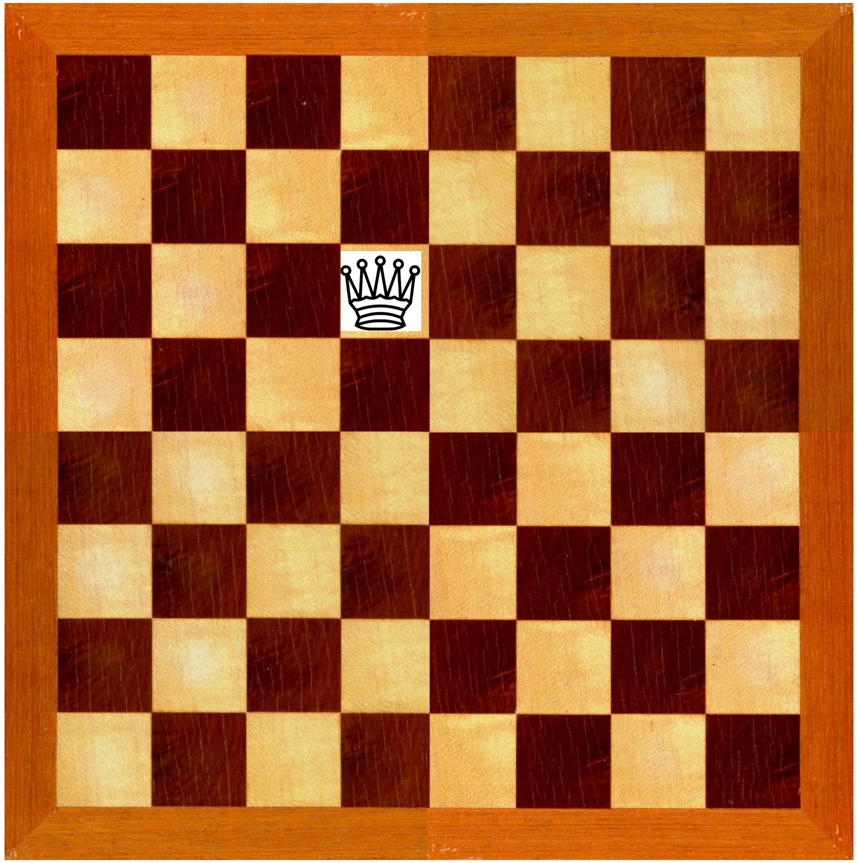
The board: a matrix of size N X N.

In standard chess: N = 8.



Basic Rules

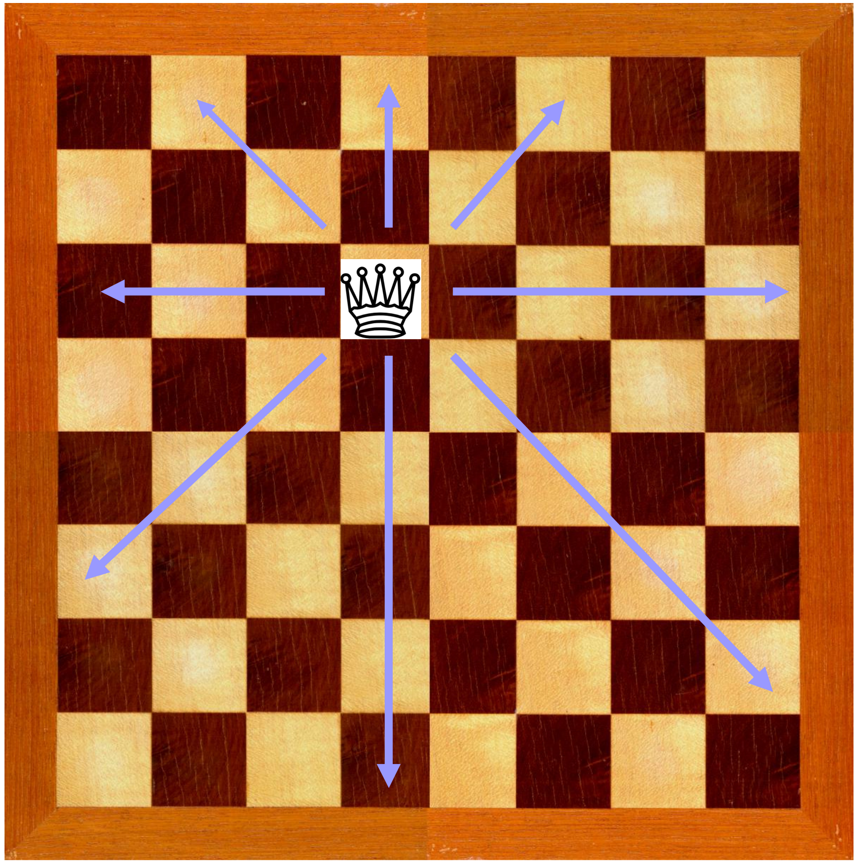
♕ The queen - moves horizontally, vertically, or diagonally.



Basic Rules

The queen - moves horizontally, vertically, or diagonally.

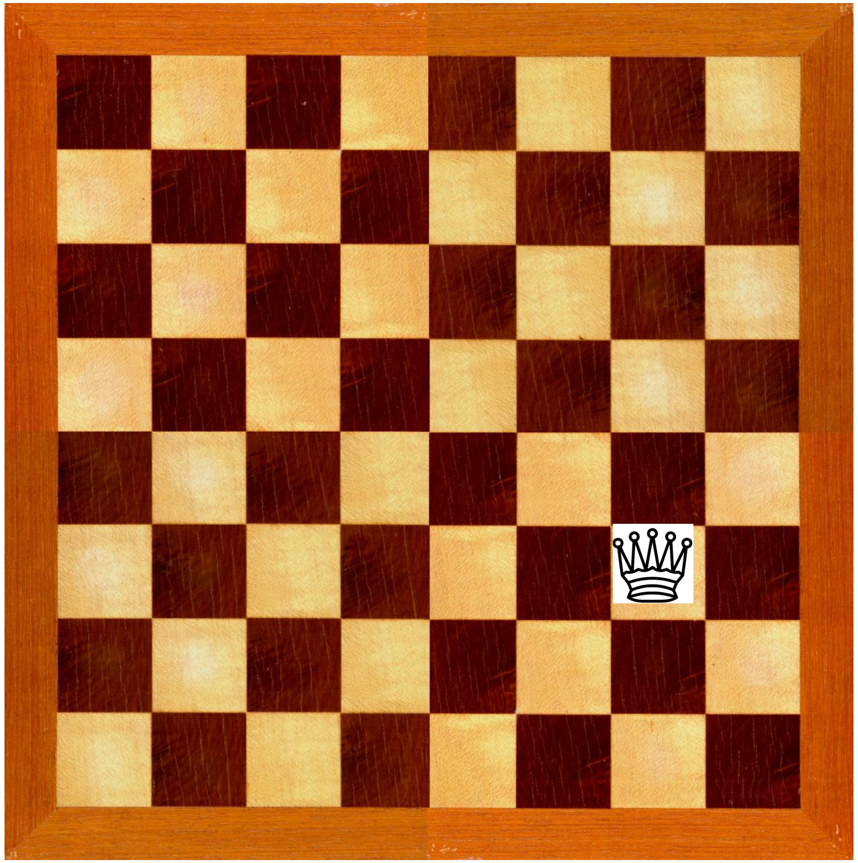
Can attack any piece on its way.



Basic Rules

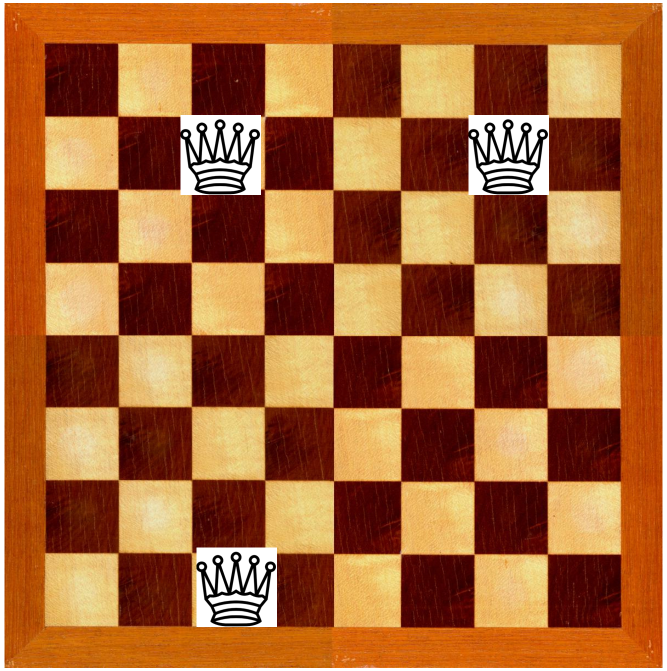
♕ The queen - moves horizontally, vertically, or diagonally.

Can attack any piece on its way.



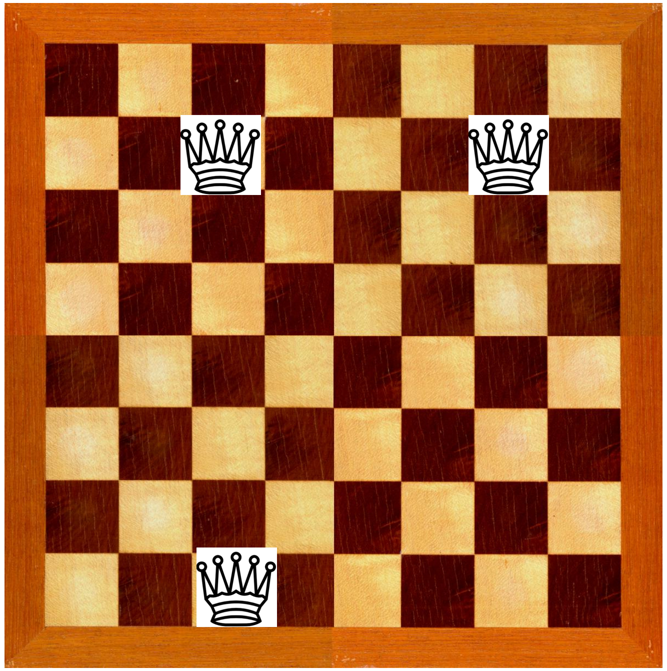
Basic Rules

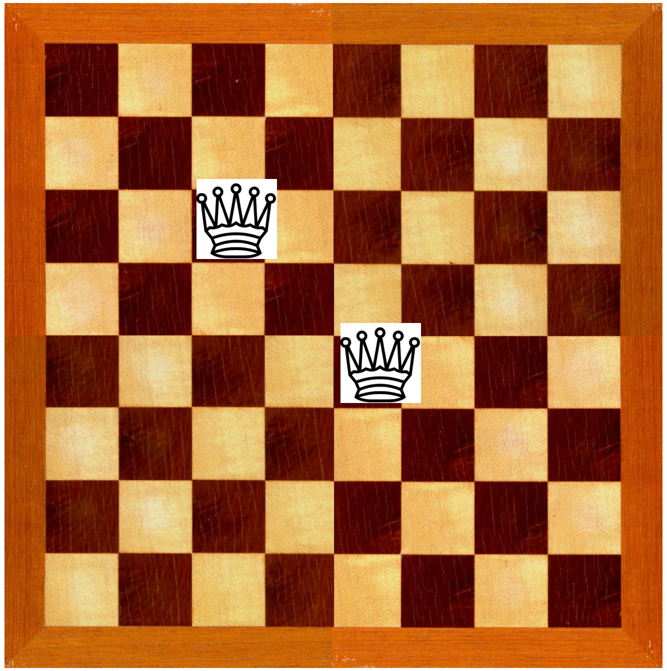
Two queens **threaten** each other if they are on the same vertical, horizontal, or diagonal line.



Basic Rules

Two queens **threaten** each other if they are on the same vertical, horizontal, or diagonal line.

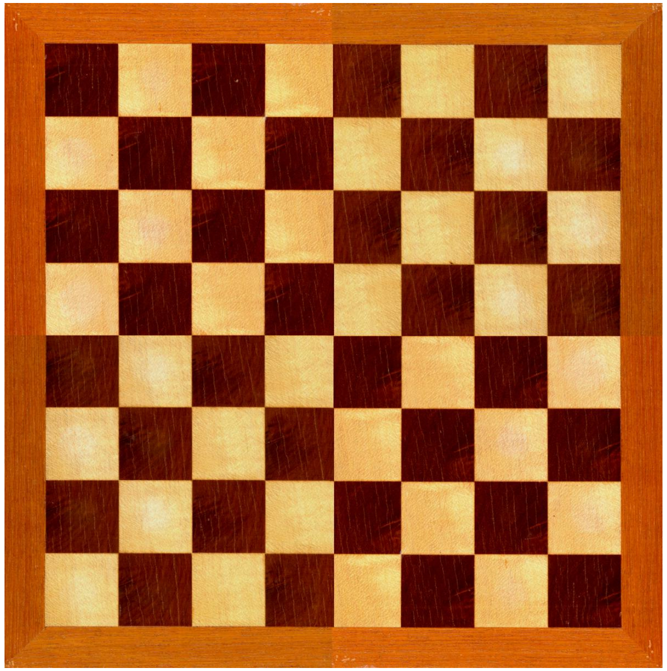




8-Queens puzzle

Place 8 queens on the board such that

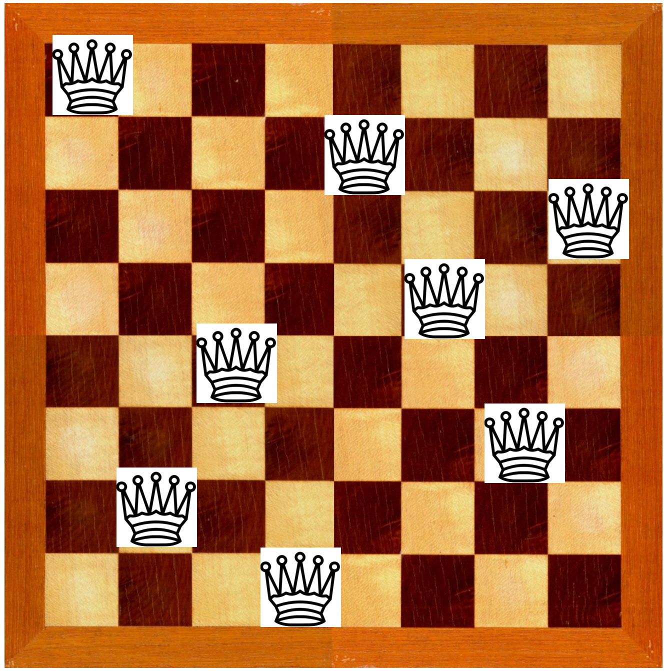
no two queens are threatening each other.



8-Queens puzzle

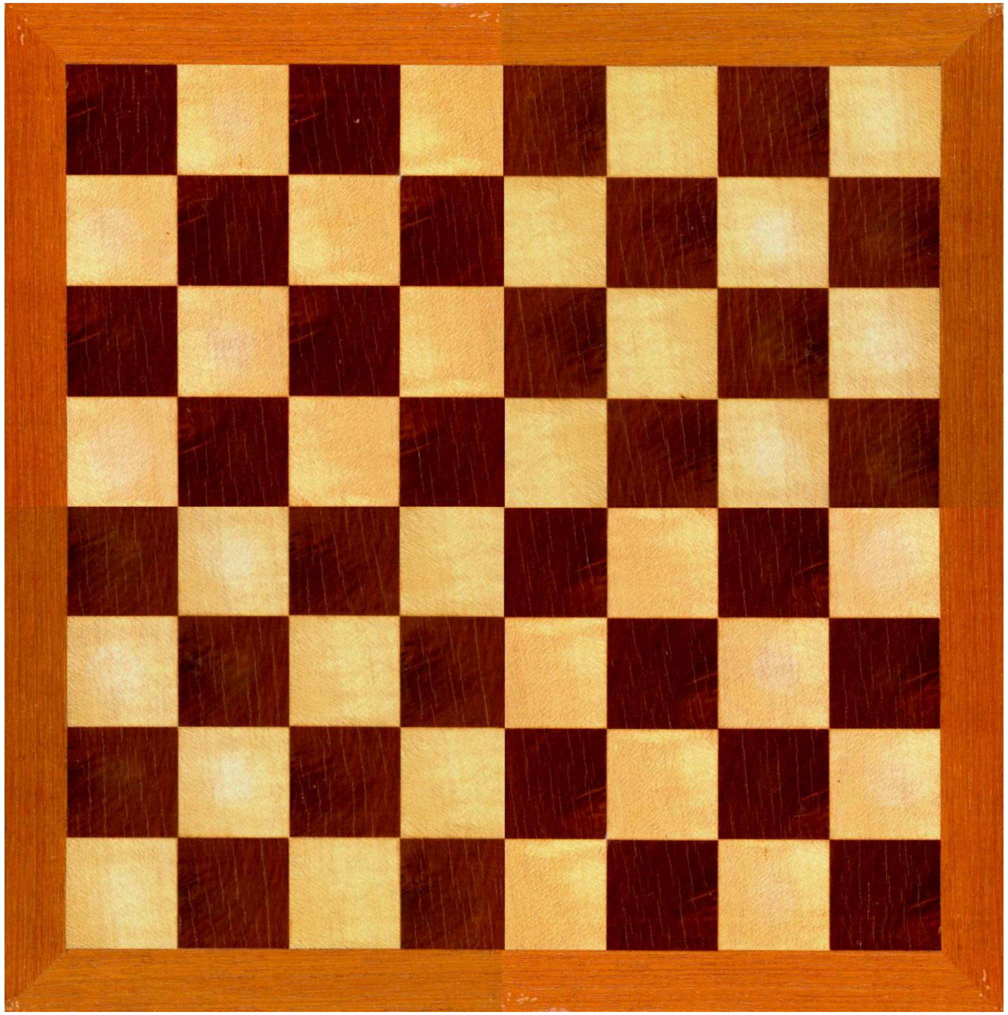
Place 8 queens on the board such that

no two queens are threatening each other.

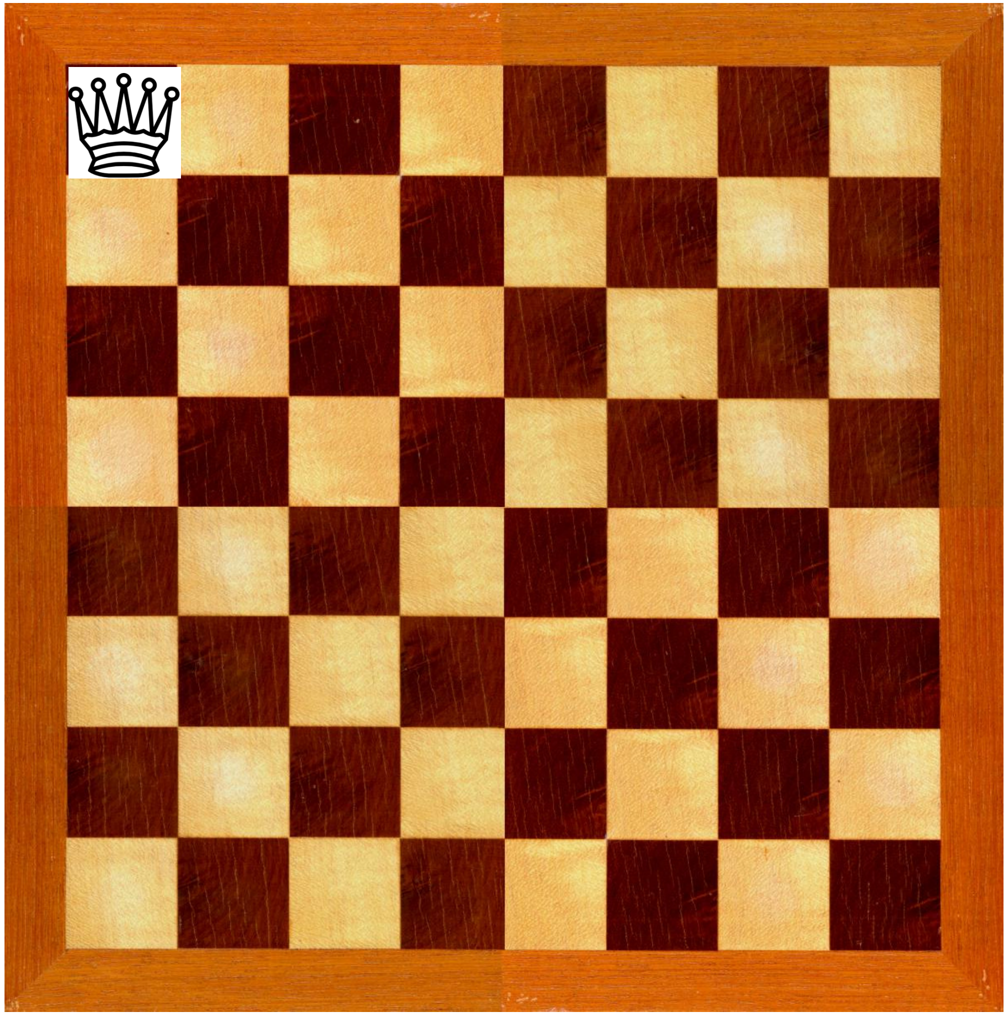


Recursive (non-OOP) solution

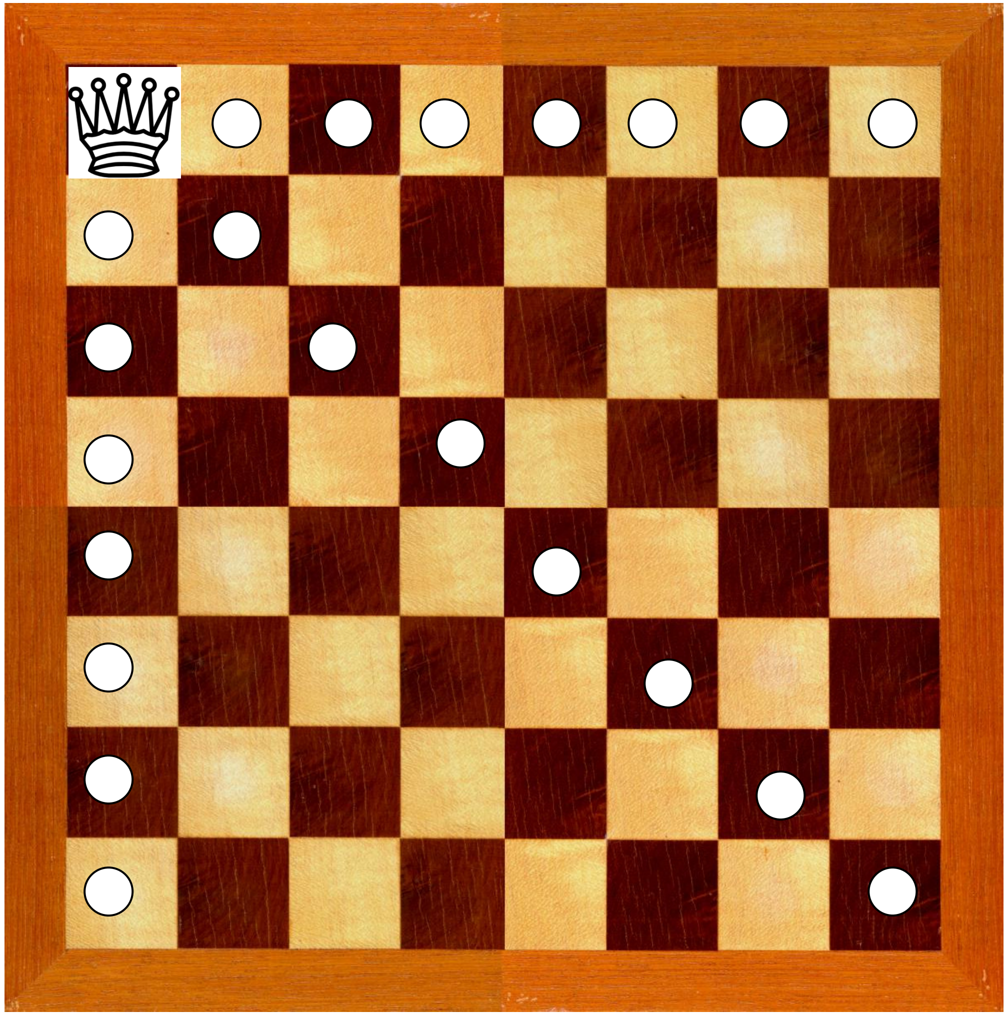
Place a queen at a non-threatened cell.



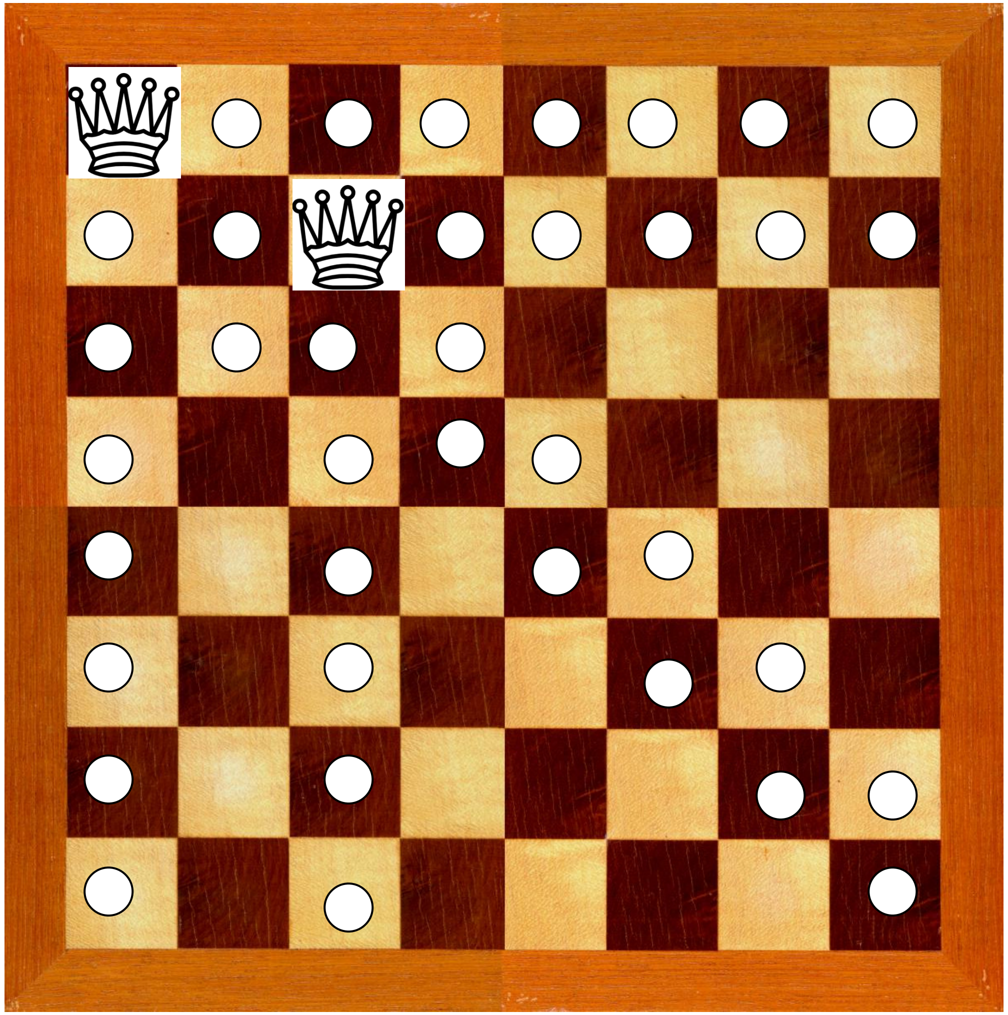
Place a queen at a non-threatened cell.



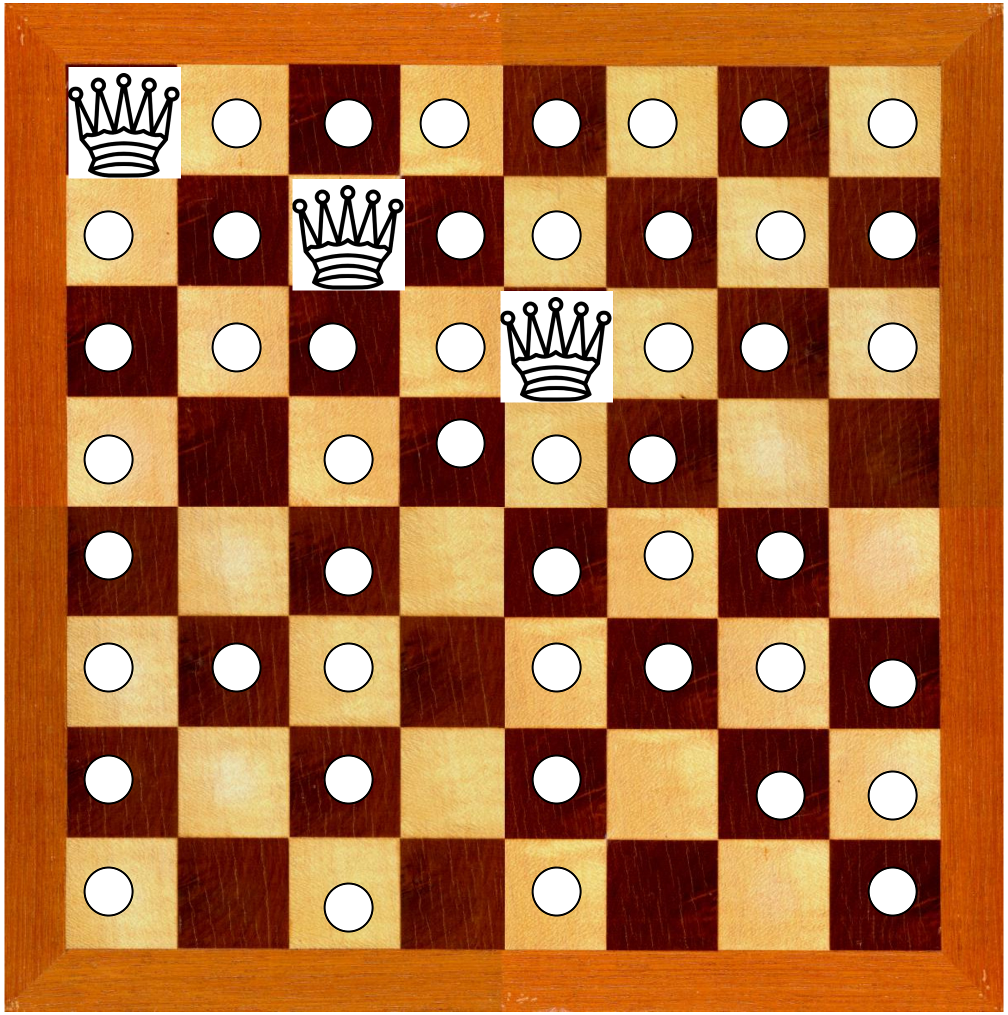
Place a queen at a non-threatened cell.



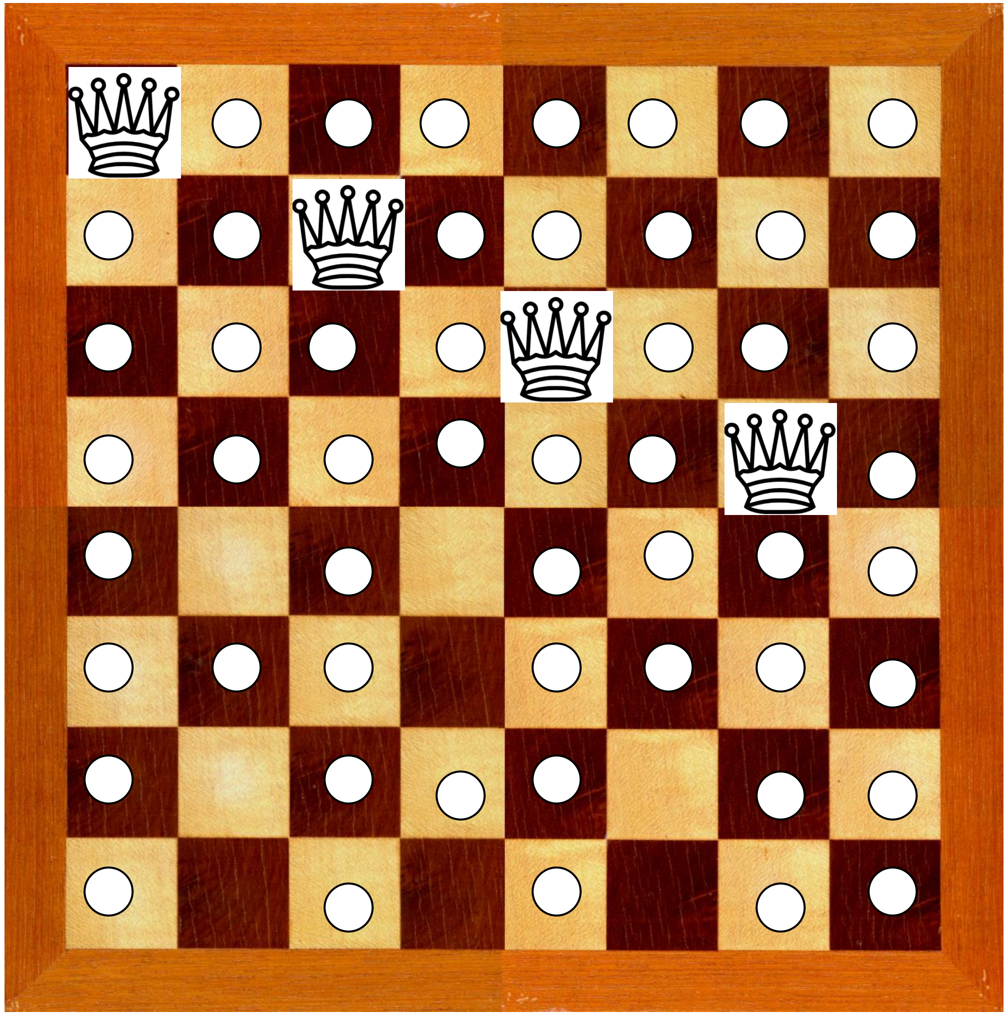
Place a queen at a non-threatened cell.



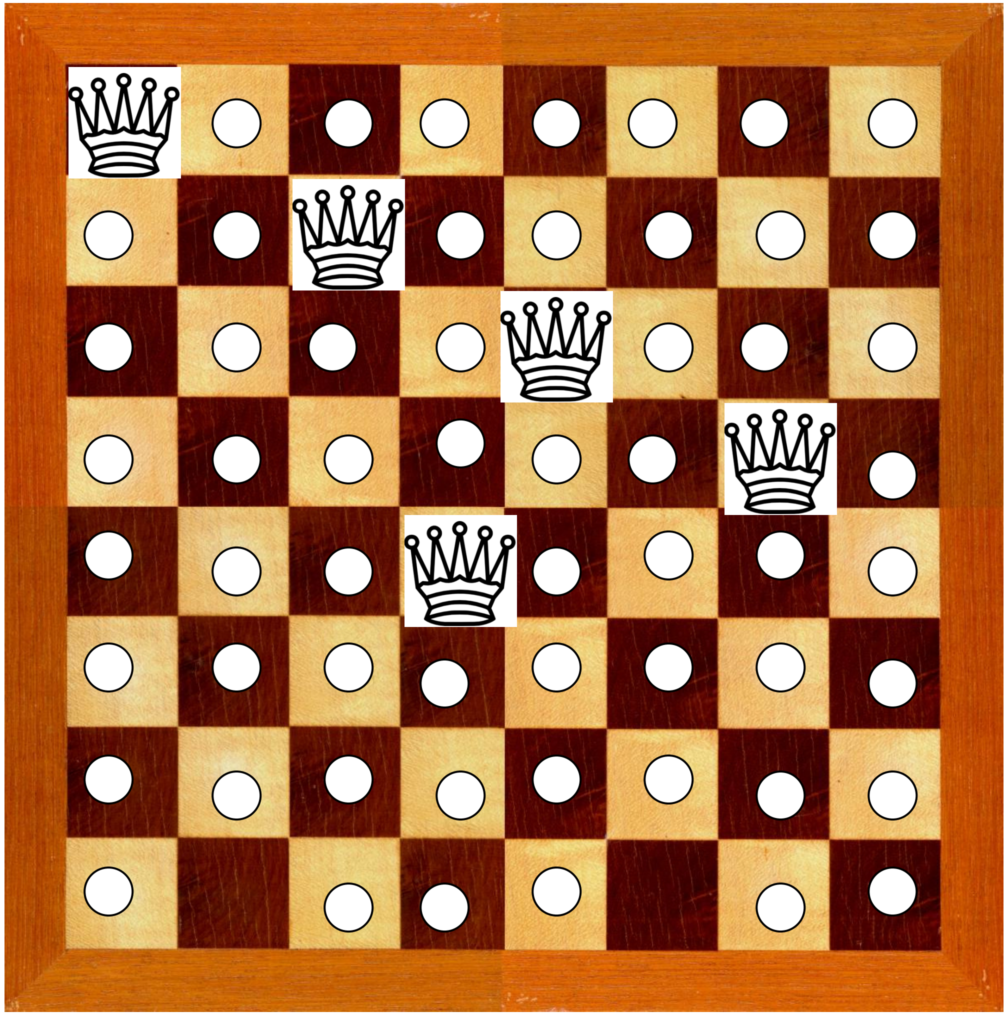
Place a queen at a non-threatened cell.



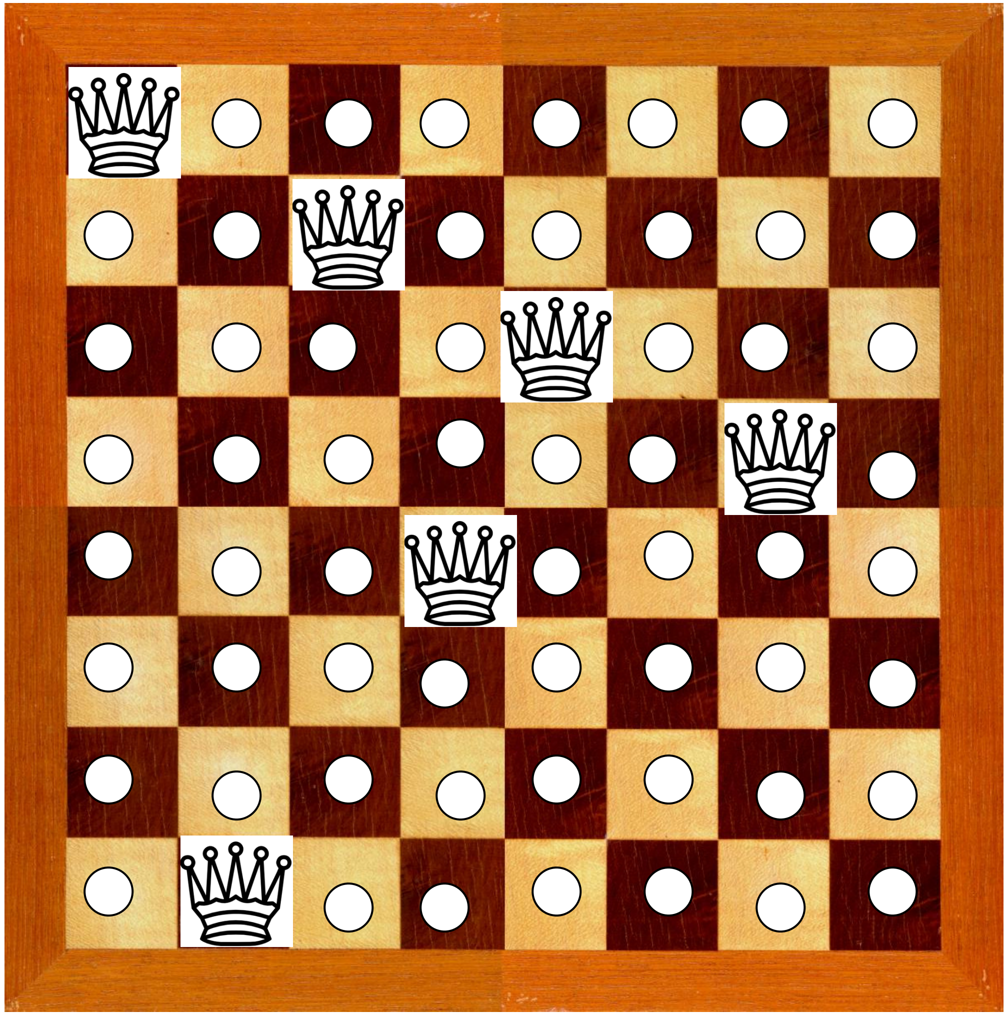
Place a queen at a non-threatened cell.



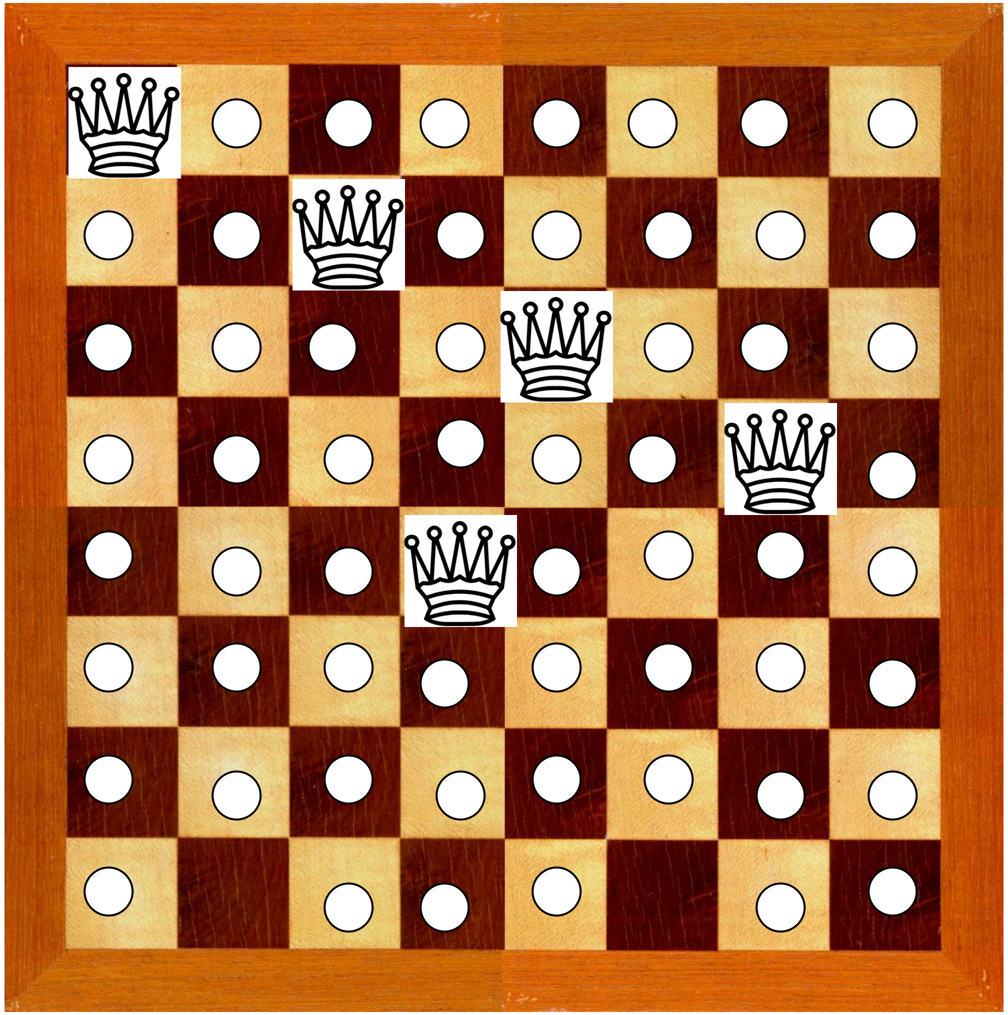
Place a queen at a non-threatened cell.



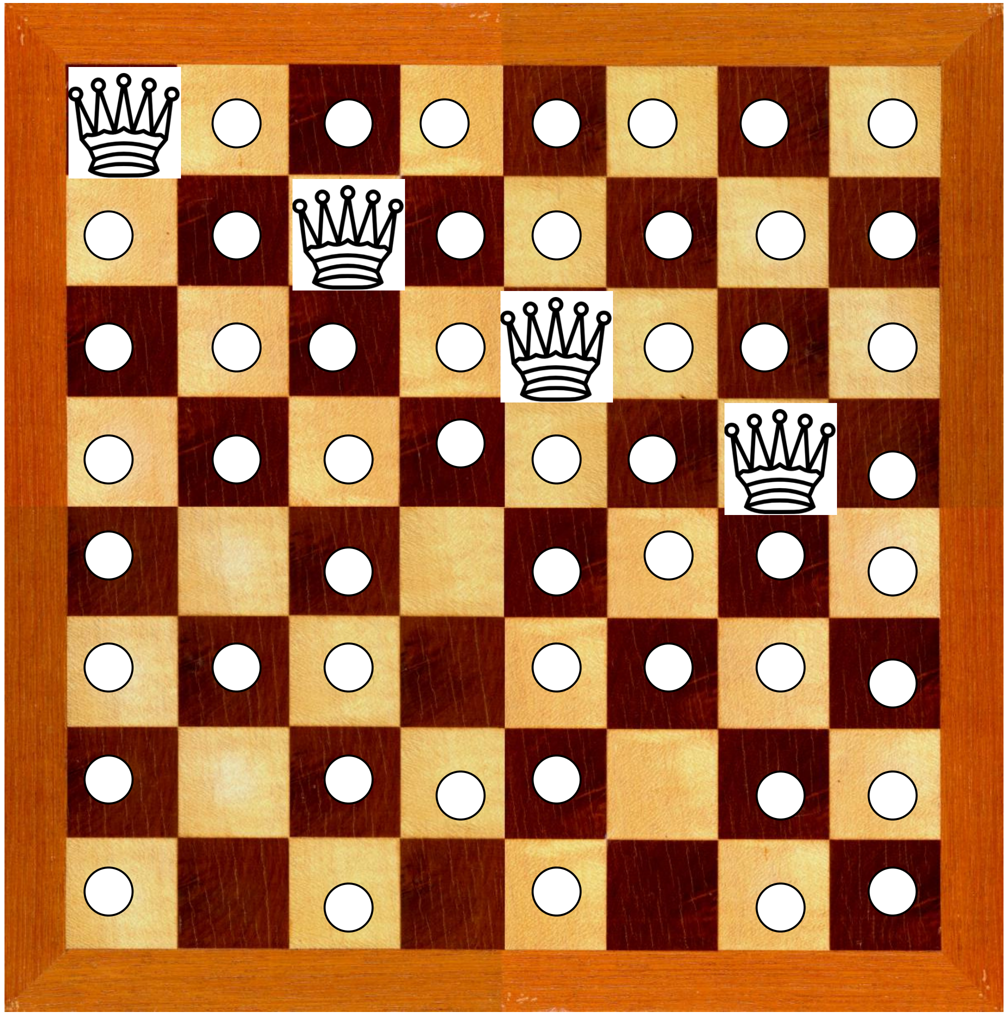
Place a queen at a non-threatened cell.



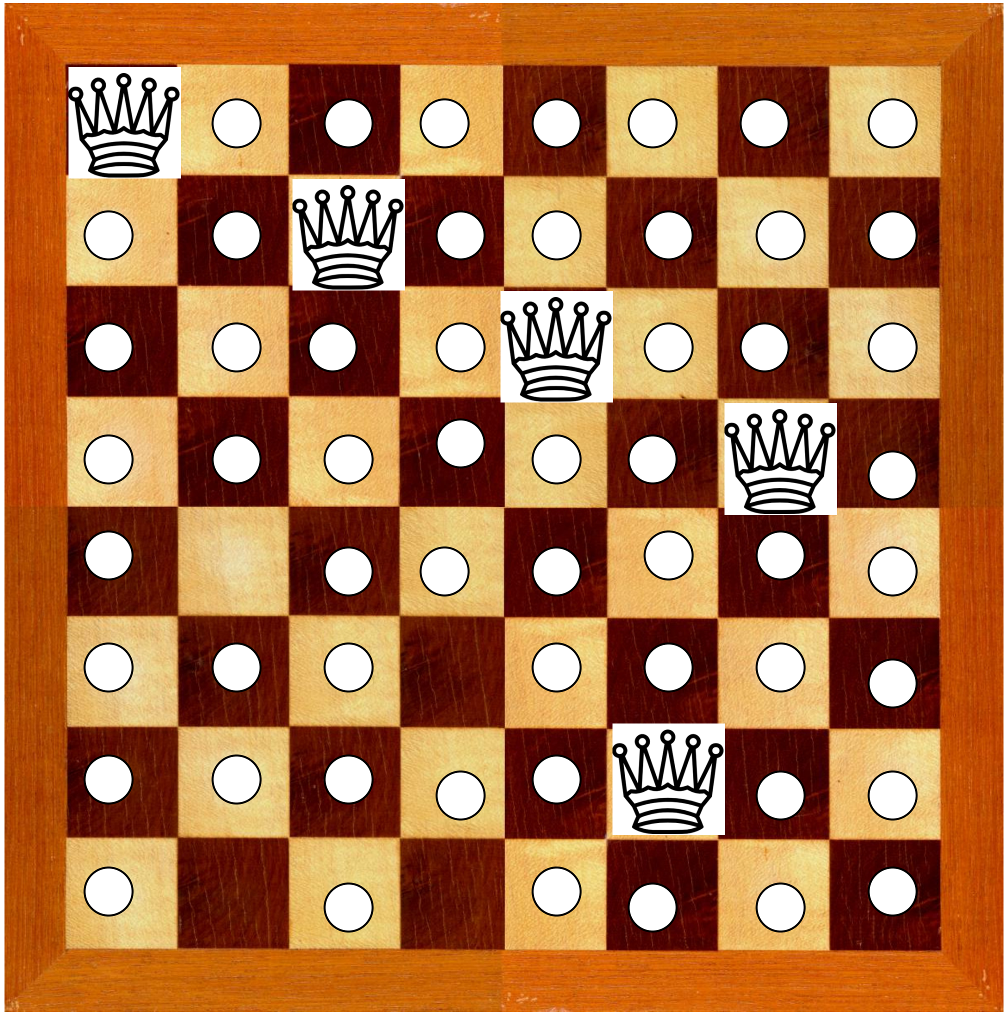
Backtrack.



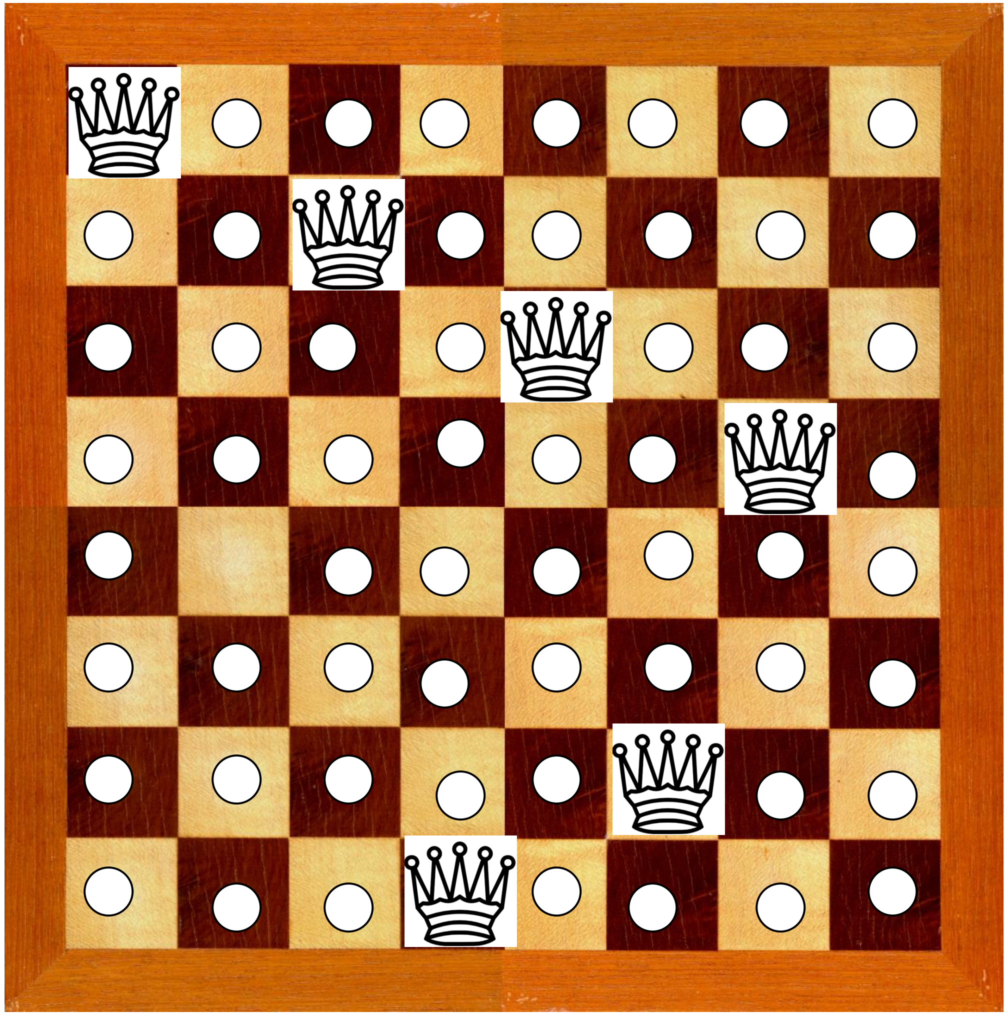
Backtrack.



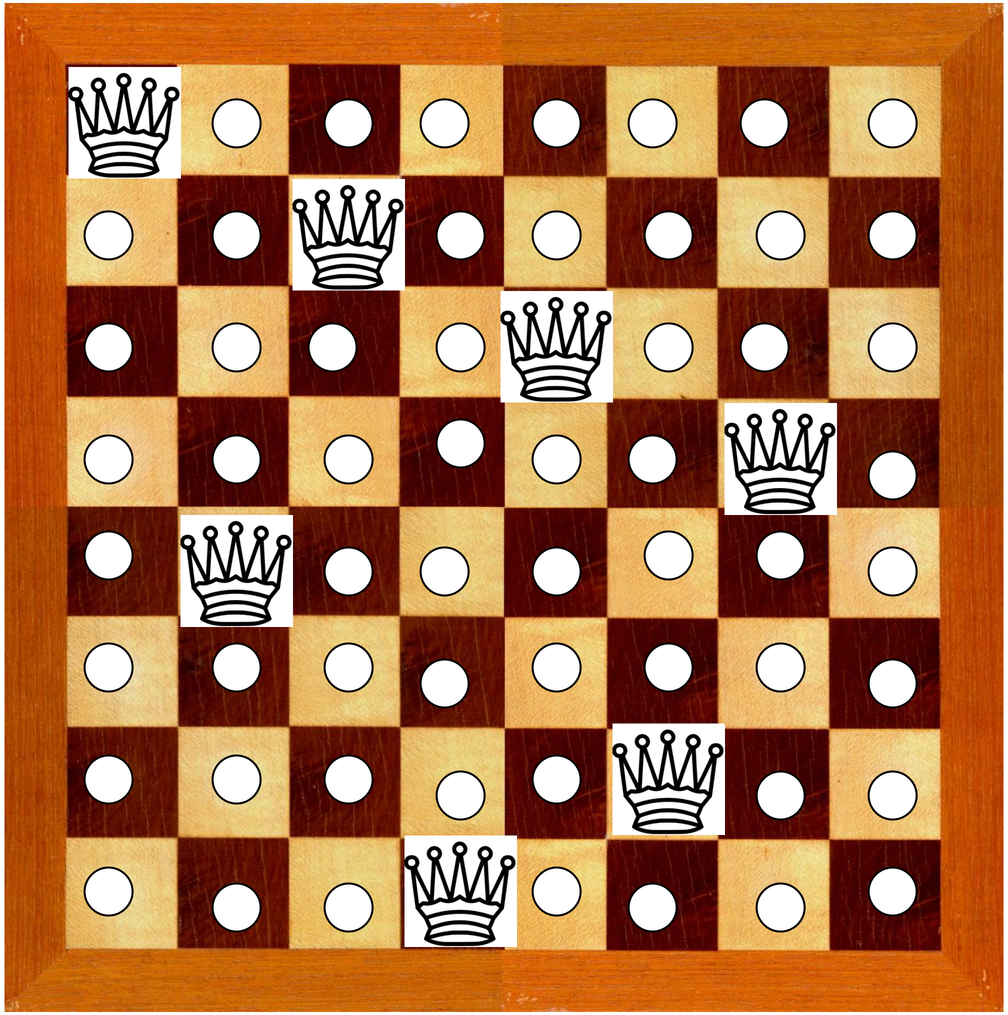
Place a queen at a non-threatened cell.



Place a queen at a non-threatened cell.



Place a queen at a non-threatened cell.



Backtrack…

```
public static void main(String args[]){
          final int N = 8;
          int [][] board = new int[N][N];
         init(board);
         solve(board,N);
 }
public static boolean solve(int [][] board, int cnt){
        if (cnt == 0){
            print(board);
            return true;
        }
        boolean solved = false;
        if (cnt > 0 && !solved){
             for (int i = 0; i < board.length && !solved; i++)
                  for (int j =0; j < board[i].length && !solved; j++)
                       if (board[i][j] == FREE){          //FREE – a constant, equals 0
                              int [][] newBoard = cloneBoard(board);
                              newBoard[i][j] = QUEEN;          // QUEEN - a constant, equals 1
                              threaten(newBoard,i,j);
                              solved = solve(newBoard, cnt - 1);
                      }
       }
      return solved;
}
```

```
public static void threaten(int [][] board, int i, int j){
for (int x = 0; x < board[i].length; x++){
    if (board[i][x] == FREE)
    board[i][x] = THREAT; // const. eq. 2
 }
for (int y = 0; y < board.length; y++ ){
    if (board[y][j] == FREE)
    board[y][j] = THREAT;
 }
int ltx,lty, rtx,rty, lbx,lby, rbx, rby;
ltx = rtx = lbx = rbx = i;
lty = rty = lby = rby = j;
for (int z = 0; z < board.length; z++){
    if (board[ltx][lty] == FREE)
       board[ltx][lty] = THREAT;
    if (board[rtx][rty] == FREE)
       board[rtx][rty] = THREAT;
    if (board[lbx][lby] == FREE)
      board[lbx][lby] = THREAT;
    if (board[rbx][rby] == FREE)
      board[rbx][rby] = THREAT;
```

```
public static void threaten(int [][] board, int i, int j){
for (int x = 0; x < board[i].length; x++){
    if (board[i][x] == FREE)
    board[i][x] = THREAT; // const. eq. 2
 }
for (int y = 0; y < board.length; y++ ){
    if (board[y][j] == FREE)
    board[y][j] = THREAT;
 }
int ltx,lty, rtx,rty, lbx,lby, rbx, rby;
ltx = rtx = lbx = rbx = i;
lty = rty = lby = rby = j;
for (int z = 0; z < board.length; z++){
    if (board[ltx][lty] == FREE)
       board[ltx][lty] = THREAT;
    if (board[rtx][rty] == FREE)
       board[rtx][rty] = THREAT;
    if (board[lbx][lby] == FREE)
      board[lbx][lby] = THREAT;
    if (board[rbx][rby] == FREE)
      board[rbx][rby] = THREAT;
```

```
     if (ltx >0 && lty >0){
         ltx--;  lty--;
     }
     if (rbx < board.length - 1 && rby <
board.length - 1 ){
        rbx++; rby++;
     }
     if (rtx < board.length -1 && rty > 0){
        rtx++; rty--;
     }
     if (lbx > 0 && lby < board.length - 1){
         lbx--; lby++;
     }

   } //end of for

} // end of function threaten
```

OOP solution

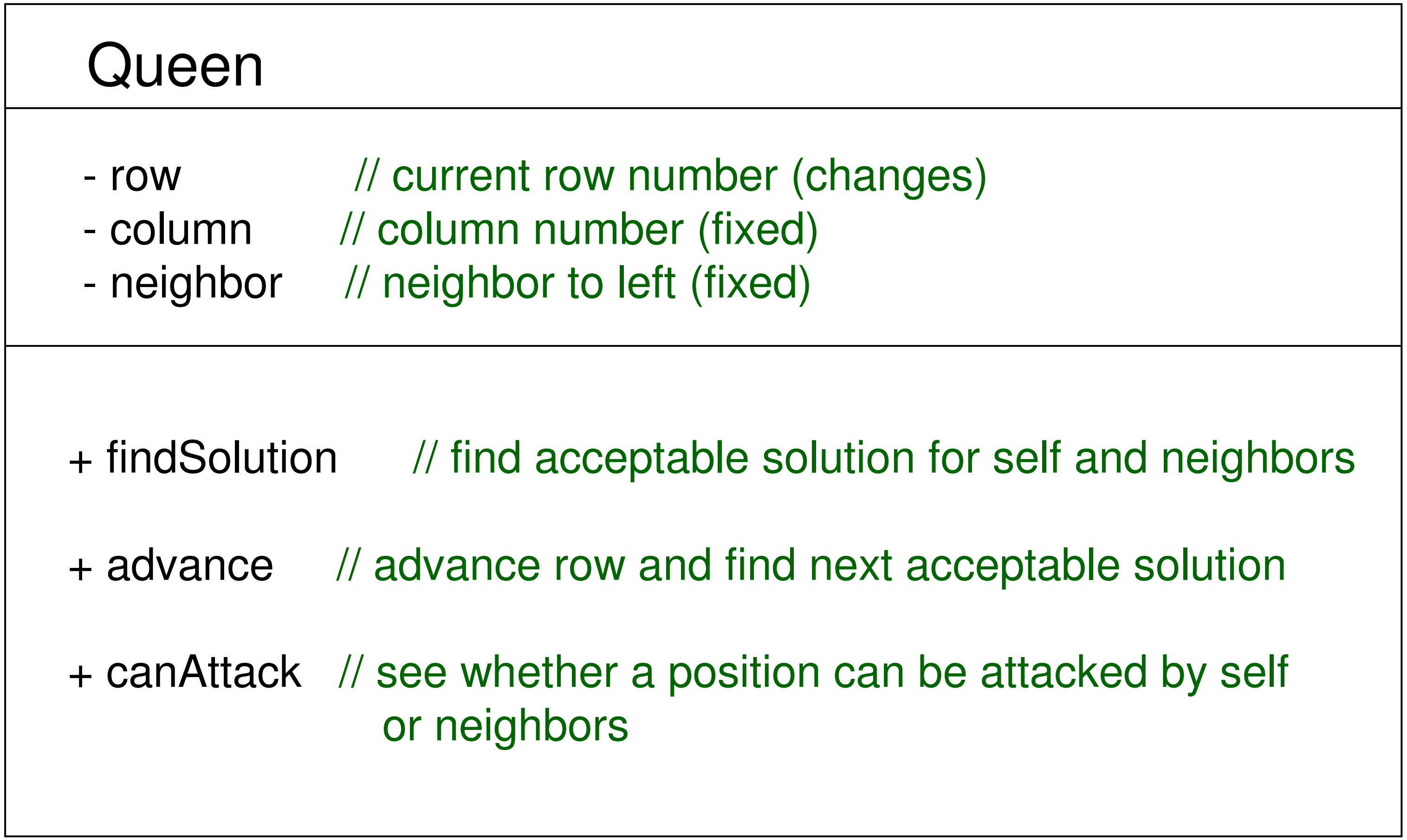
Main ideas

- **Each queen is an autonomous agent!**
- **Each queen has its own fixed column.**
- **Queens are added to the board from left to right.**
- **A queen tries to find a safe position in its column.**
- **If no safe position is found,**

  then the queen asks its neighbors to advance to the next legal position.

  (In which no two neighbors threaten each other.)

Queen class diagram



Queen

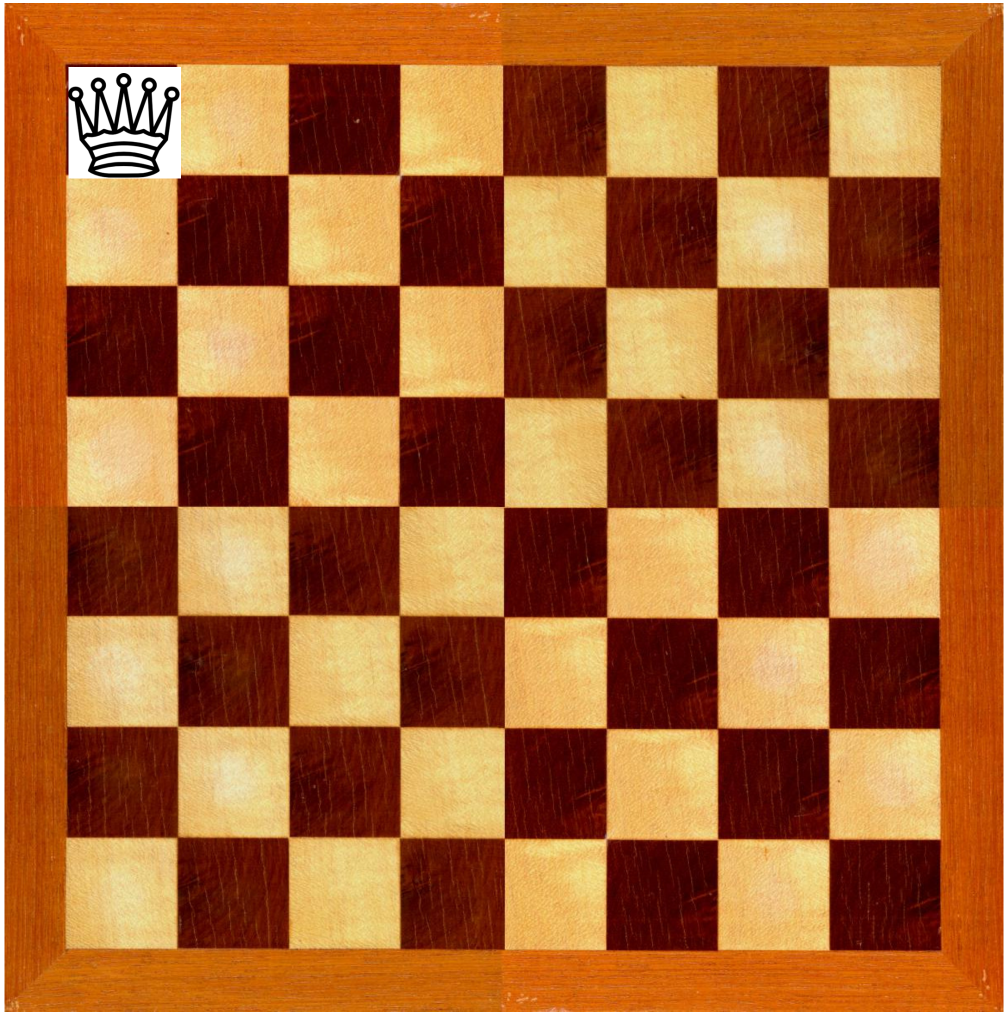
- row // current row number (changes)
- column // column number (fixed)
- neighbor // neighbor to left (fixed)

+ findSolution // find acceptable solution for self and neighbors

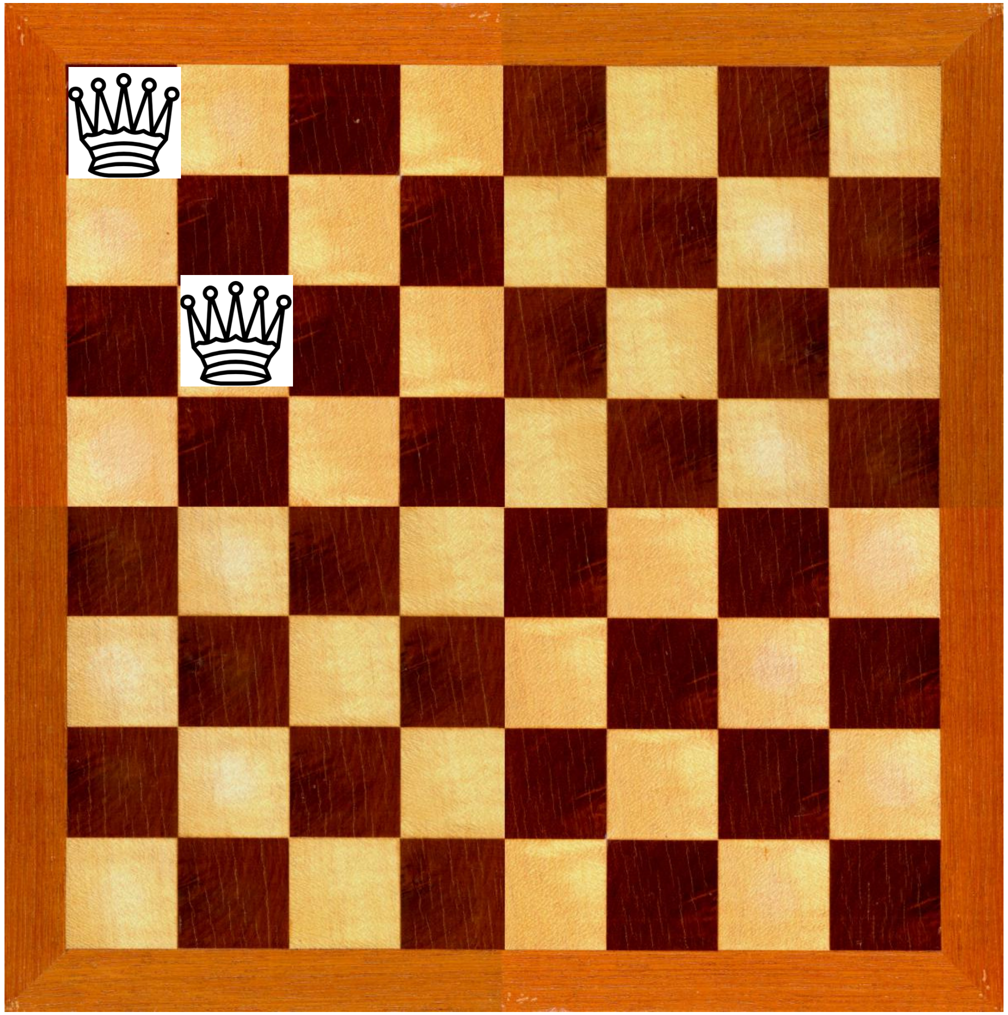
+ advance // advance row and find next acceptable solution

+ canAttack // see whether a position can be attacked by self or neighbors

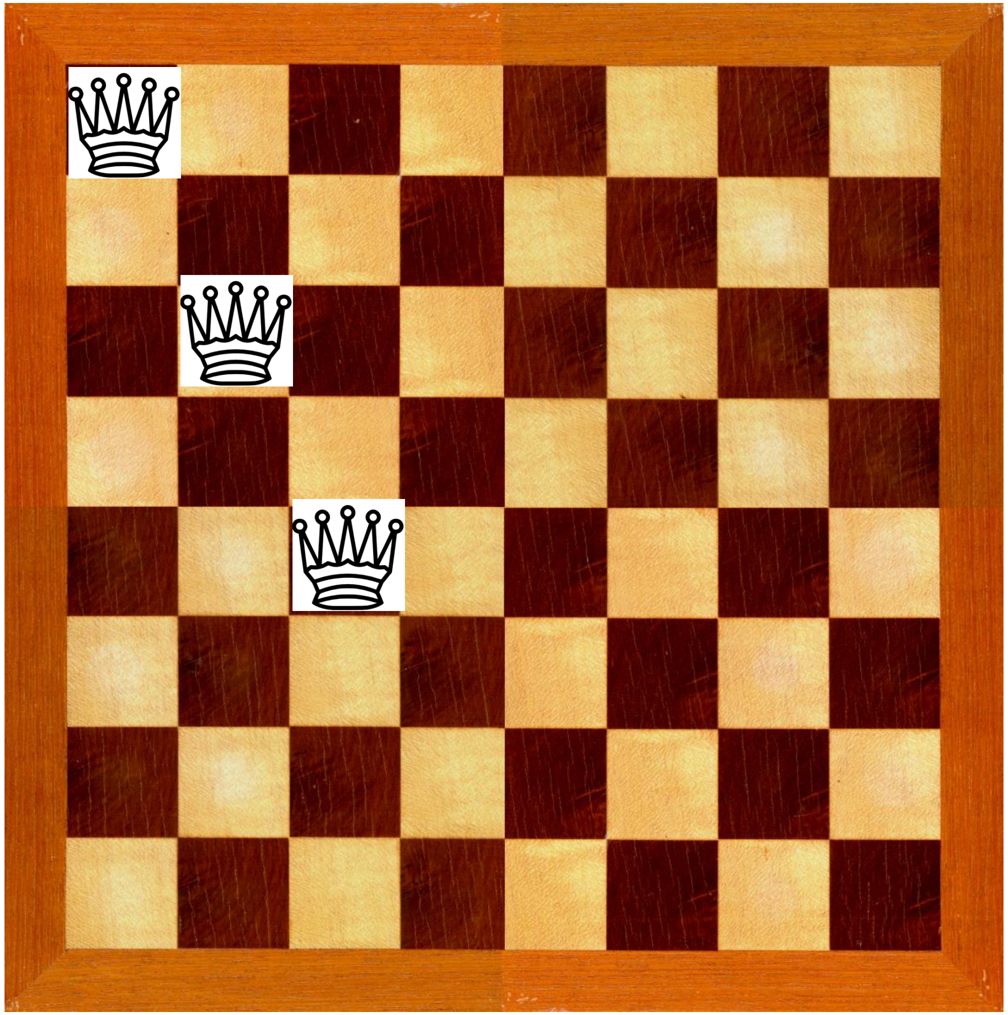
A queen places itself at a safe position in its column.



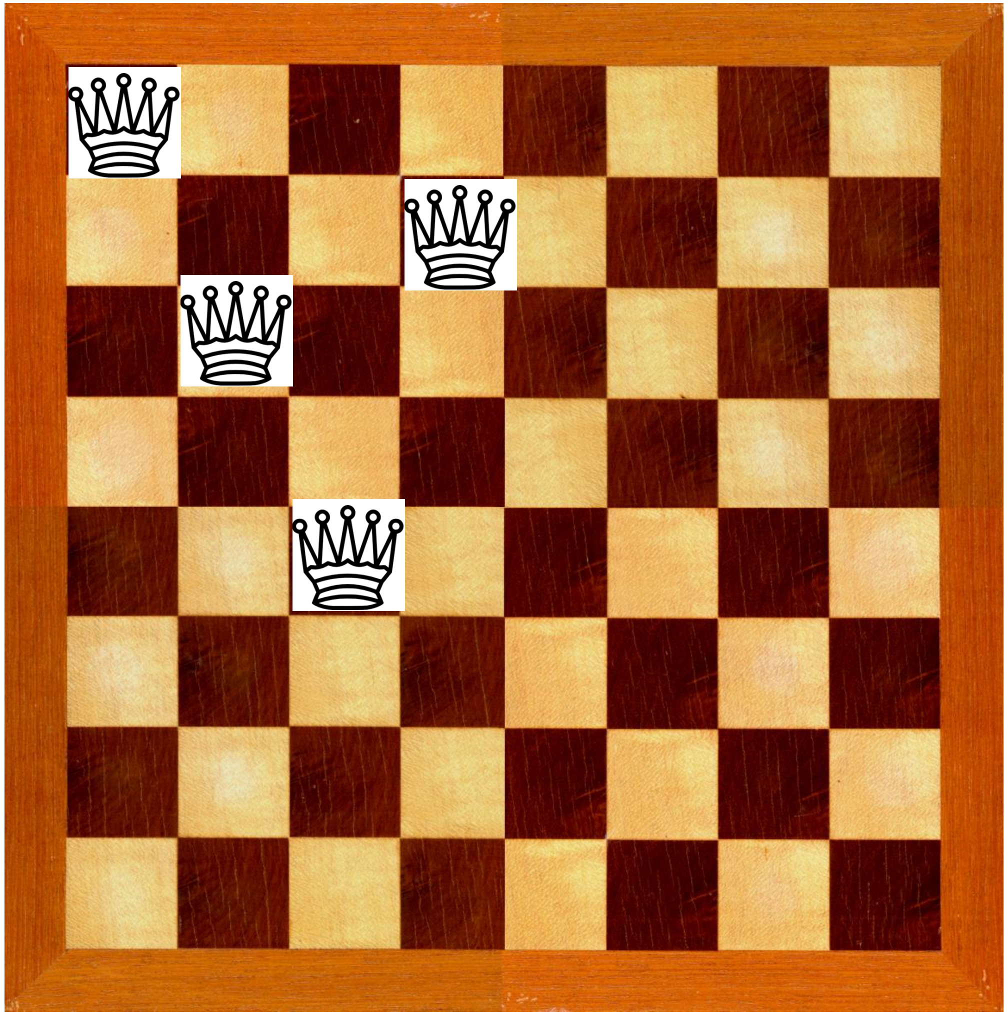
A queen places itself at a safe position in its column.



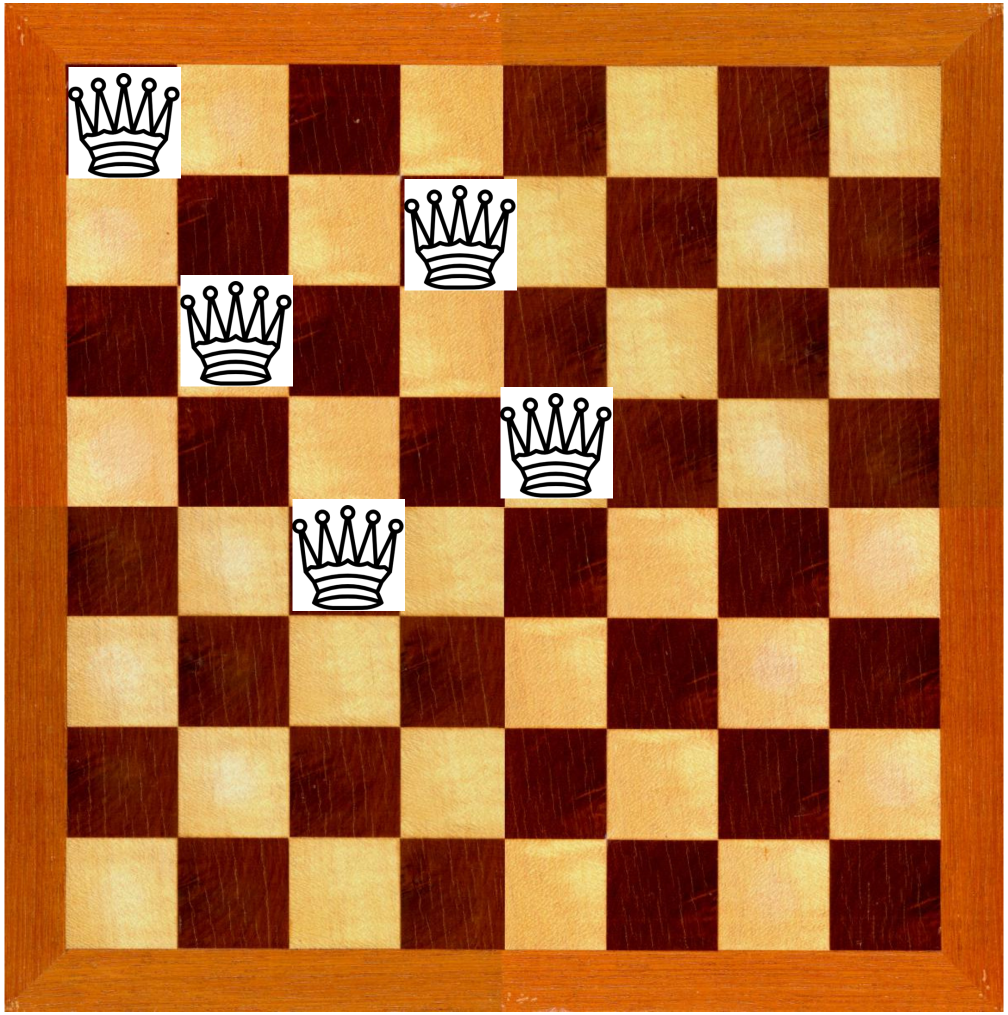
A queen places itself at a safe position in its column.



A queen places itself at a safe position in its column.

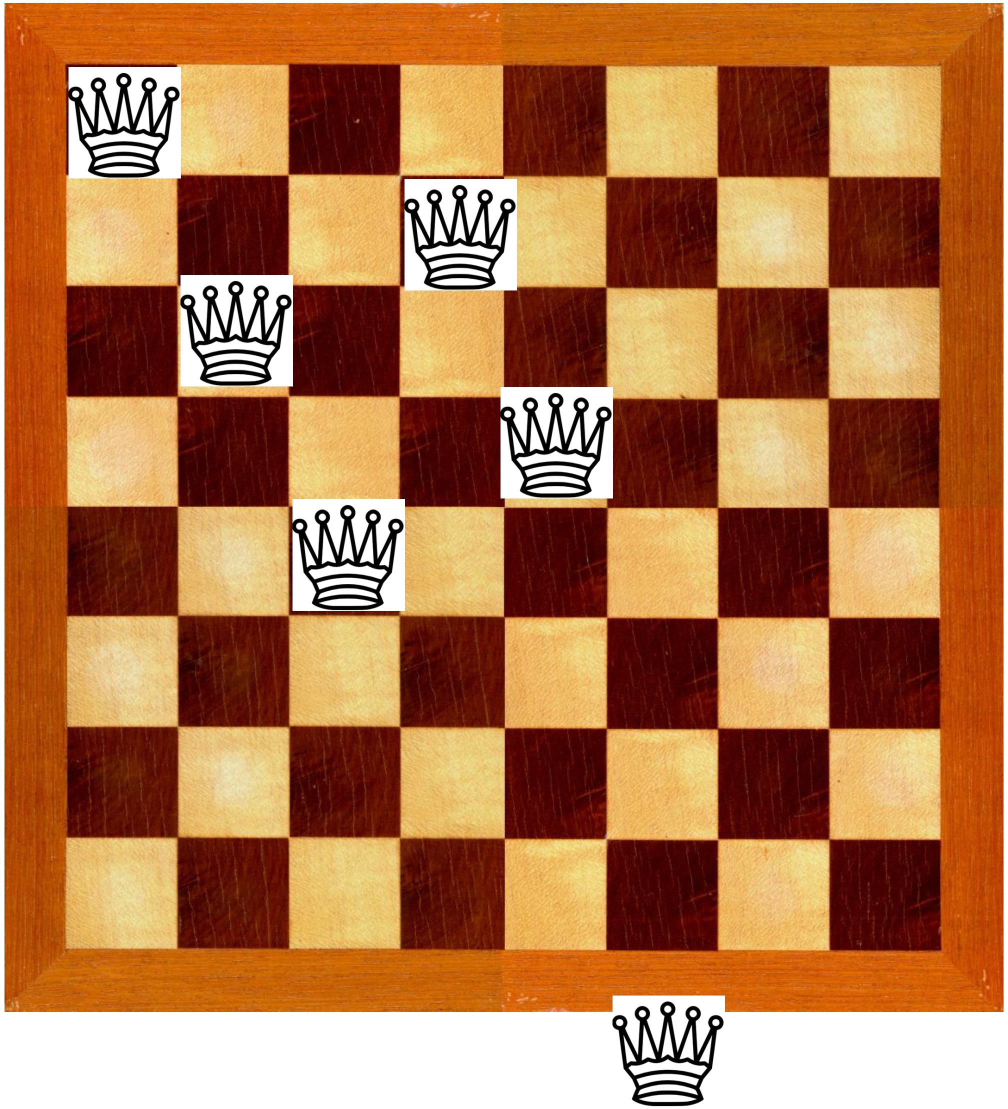


A queen places itself at a safe position in its column.



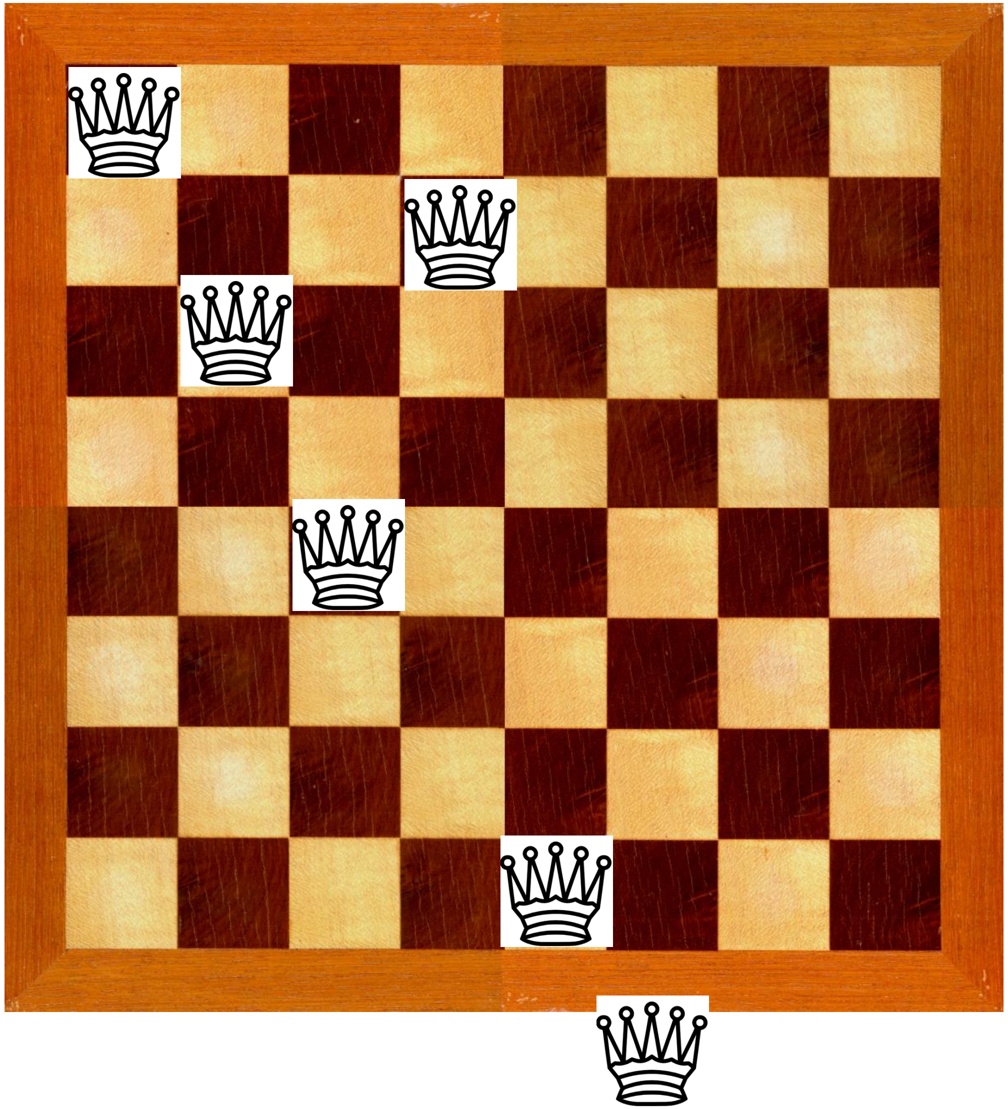
No safe position for queen 6 at column 6.

Asks neighbor (queen 5) to change position.



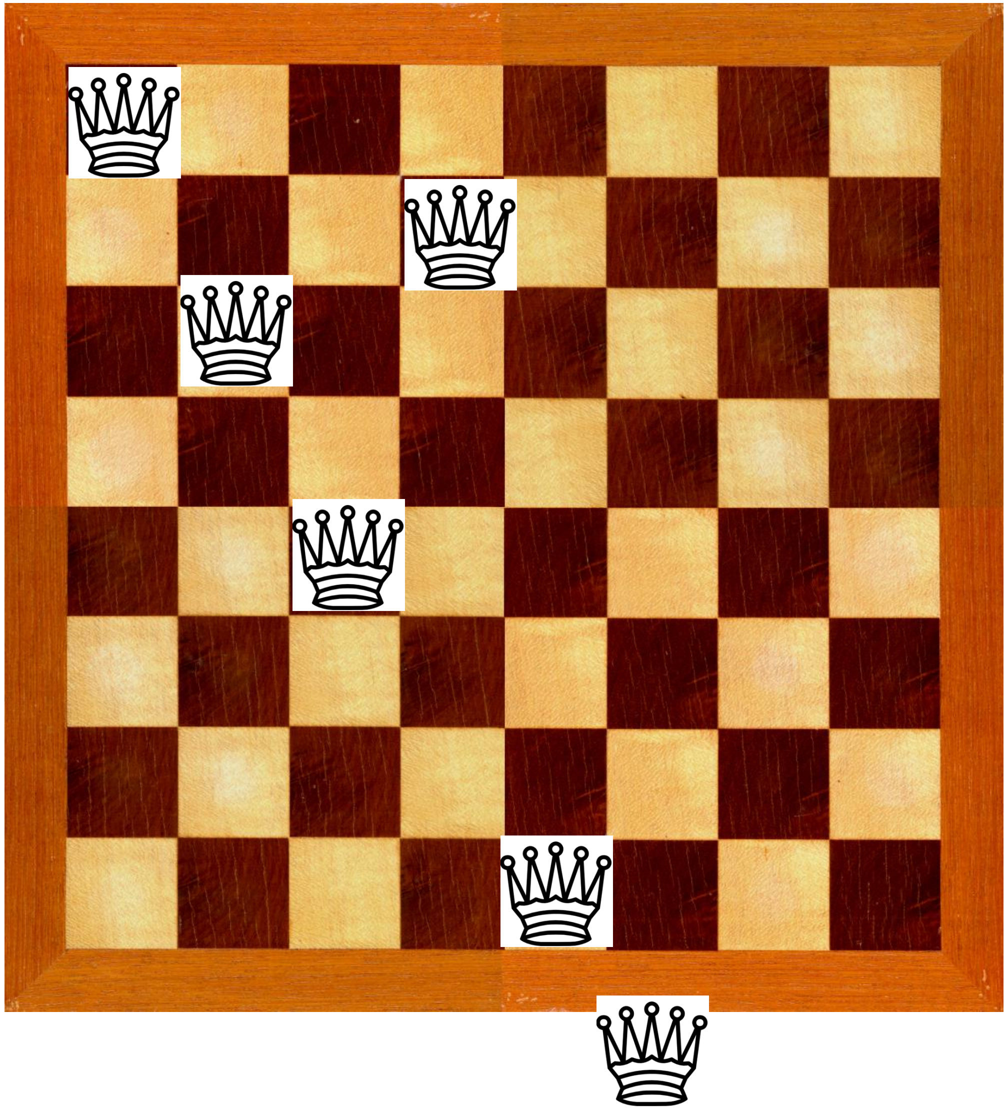
No safe position for queen 6 at column 6.

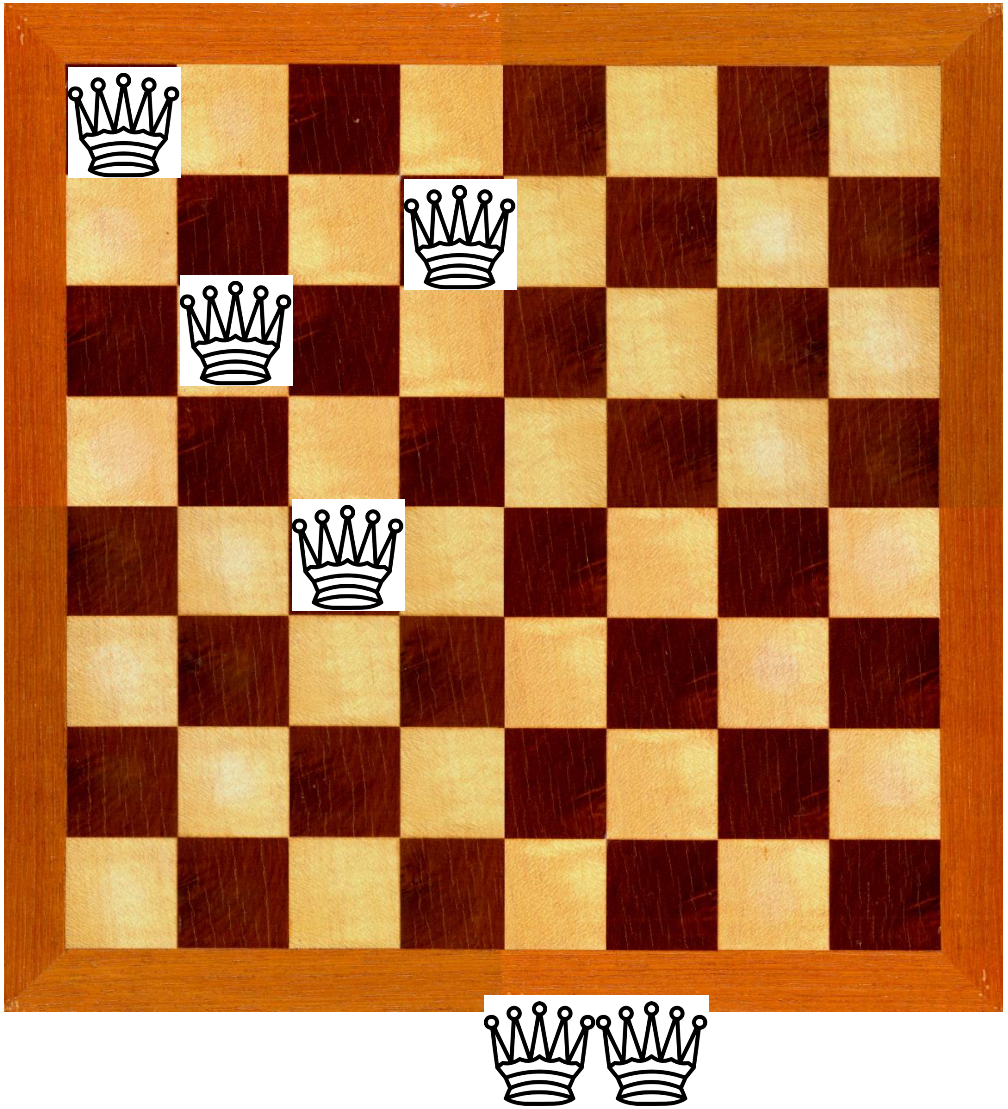
Asks neighbor (queen 5) to change position.



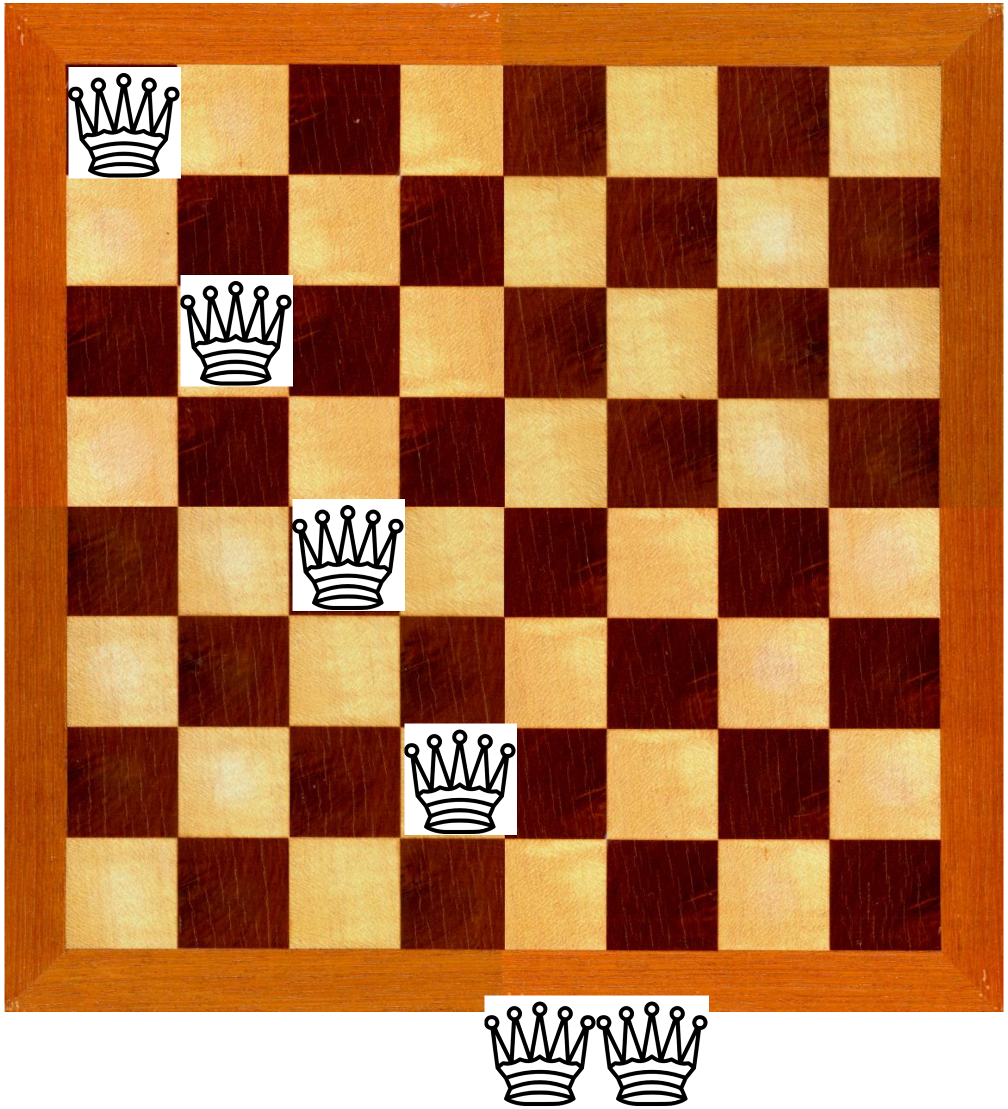
Queen 5 is at the last row.

Asks neighbor (queen 4) to change position.

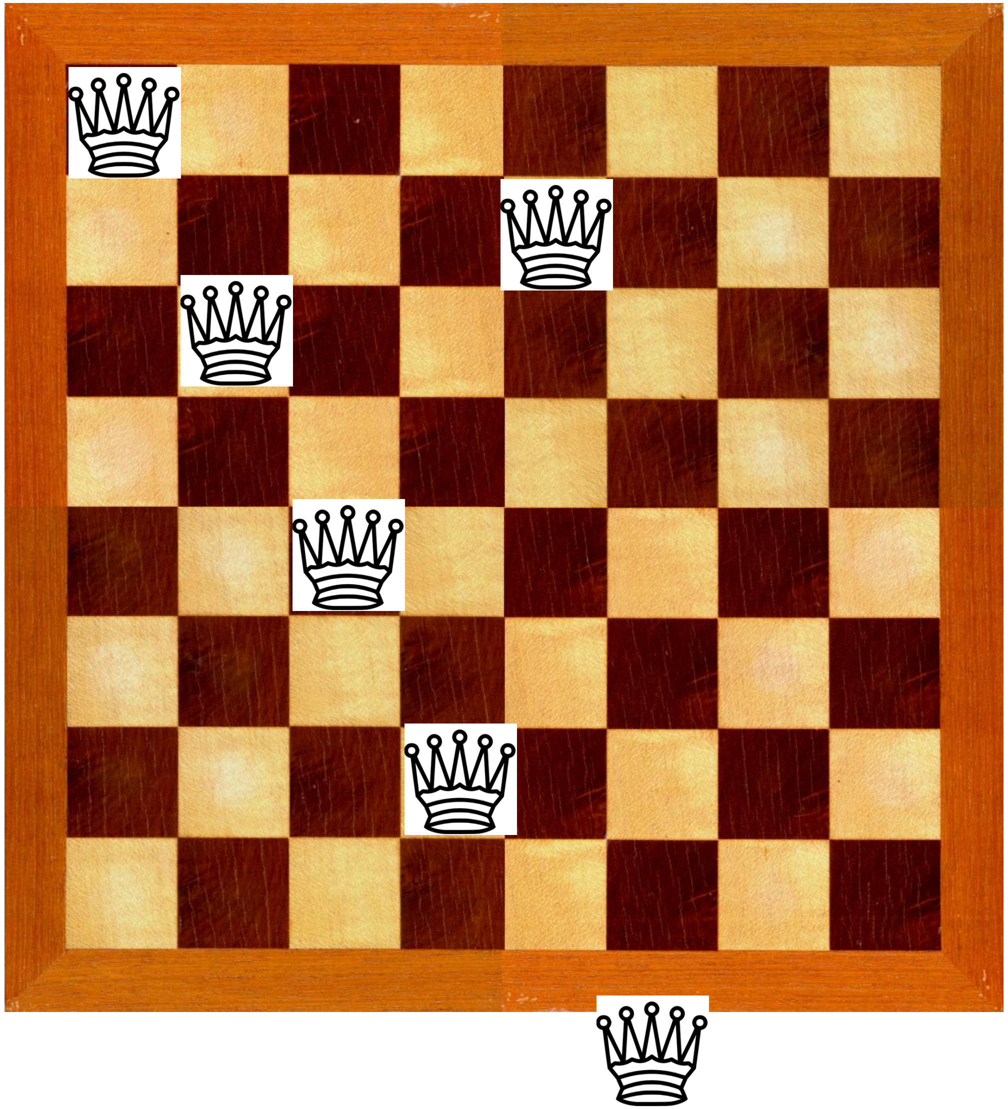




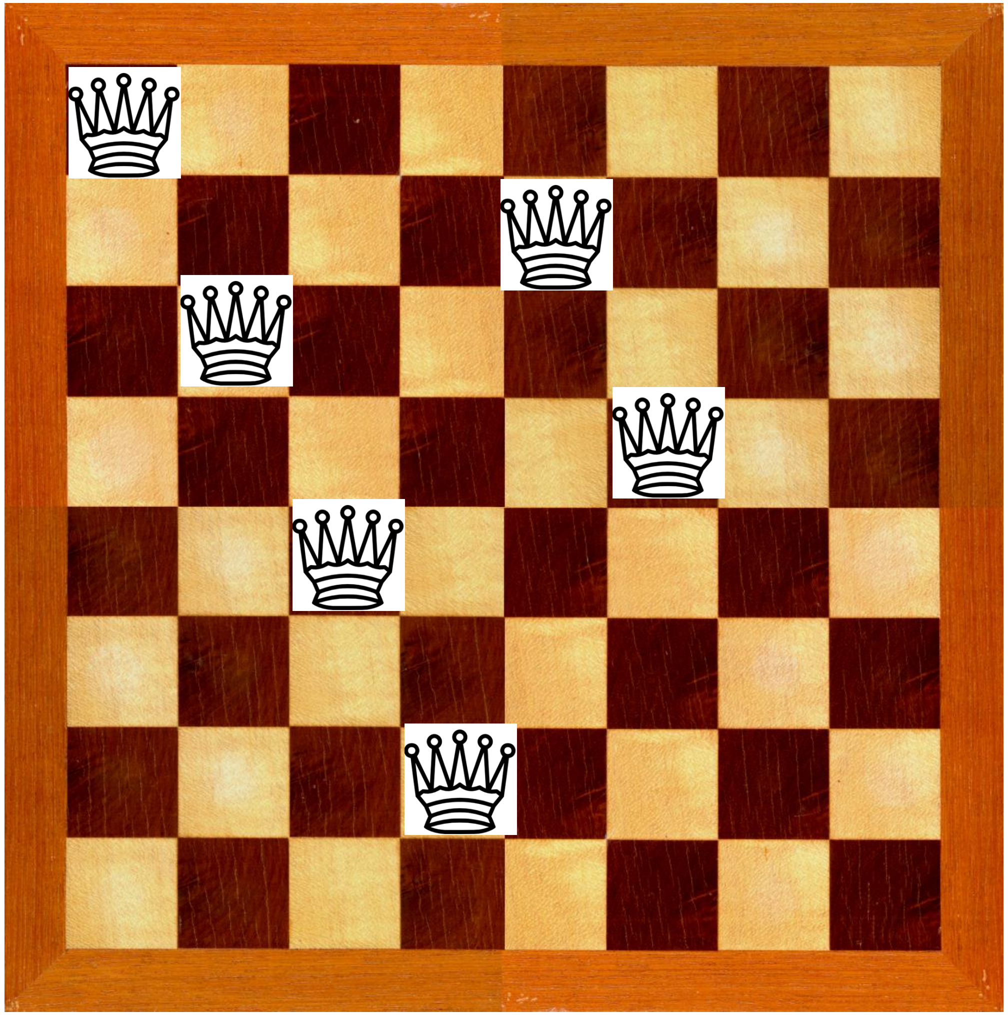
A queen places itself at a safe position in its column.



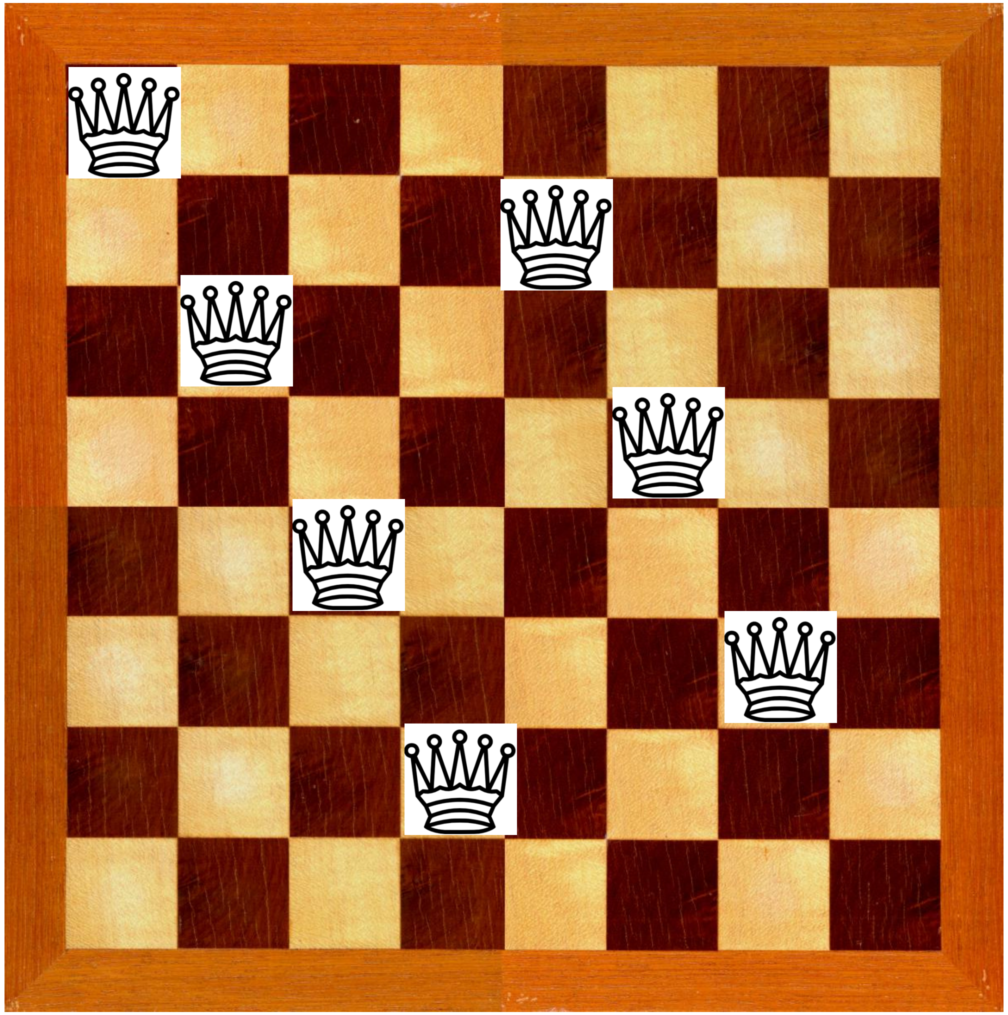
A queen places itself at a safe position in its column.



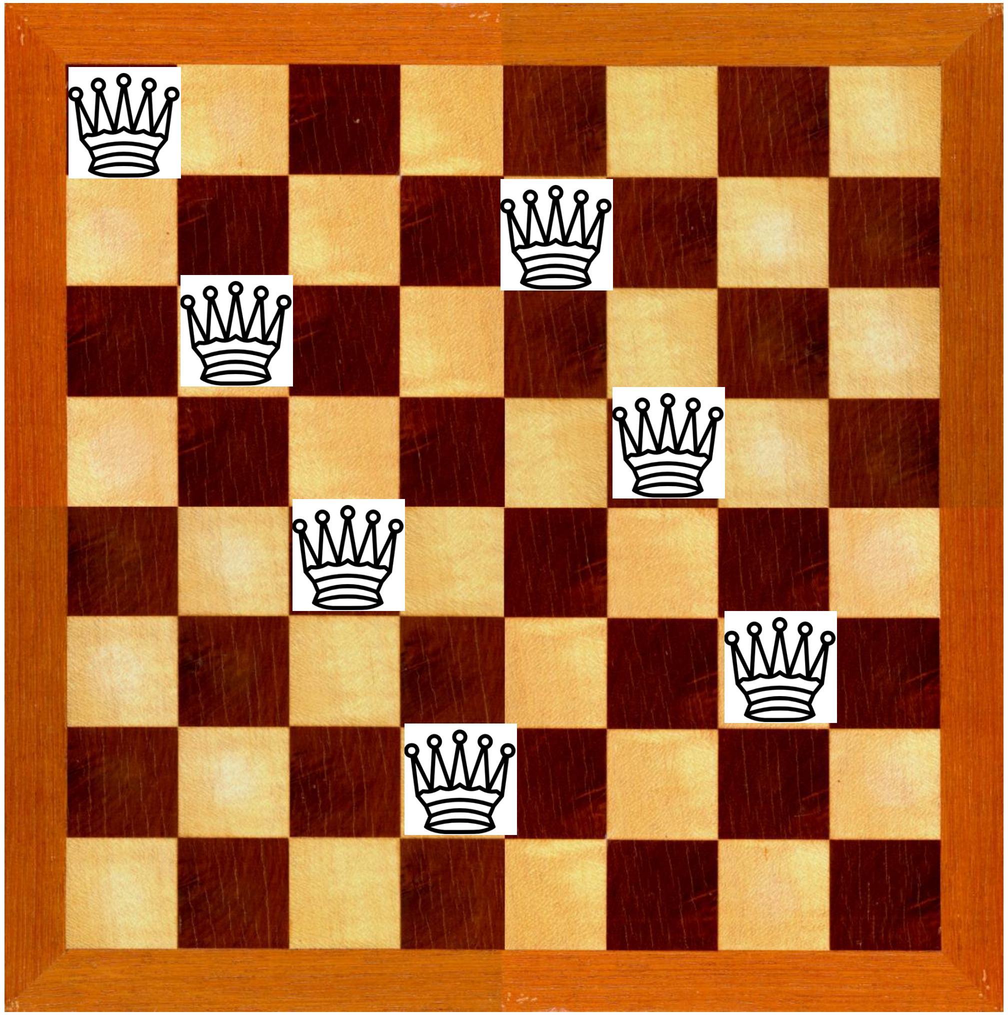
A queen places itself at a safe position in its column.



A queen places itself at a safe position in its column.



No safe position for queen 8 at column 8.



Proceed the search in a similar way…

```
public class Queen{
        …
        public Queen (int column, Queen neighbor) {
                        this.row = 1;
                        this.column = column;
                        this.neighbor = neighbor;
          }
          …..
        public static void main(String args[]){
              Queen lastQueen = null;
              for (int i = 1; i <= N; i++) {  \\ N equals 8
                        lastQueen = new Queen(i, lastQueen);
                        lastQueen.findSolution();
          }
```

```
public boolean findSolution() {
    while (this.neighbor != null  &&  this.neighbor.canAttack(this.row, this.column))
        boolean advanced = this.advance();
            if (!advanced) return false;
    return true;
}
```

```
private boolean canAttack(int testRow, int testColumn) {
        int columnDifference = testColumn – this.column;
        if ((this.row == testRow) ||
                (this.row + columnDifference == testRow) ||
                (this.row - columnDifference == testRow))
                        return true;
        if (this.neighbor != null)
                return neighbor.canAttack(testRow, testColumn);
        return false;
}
```

```
public boolean advance() {
        if (this.row < N) {   \\ N equals 8
                this.row++;
                return this.findSolution();
        }
        if (this.neighbor != null) {
                boolean advanced = this.neighbor.advance());
                if (!advanced) return false;
        }
        else
                return false;
        row = 1;
        return findSolution();
  }
```