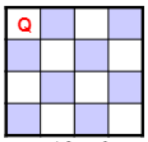
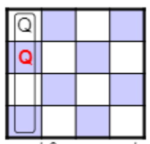
Recursive Strategy for n-Queens

- Consider one row at a time. Within the row, consider one column at a time, looking for a “safe” column to place a queen.
- If we find one, place the queen, and *make a recursive call* to place a queen on the next row.
- If we can’t find one, *backtrack* by returning from the recursive call, and try to find another safe column in the previous row.
- Example for n = 4:
  - row 0:

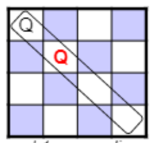


col 0: safe

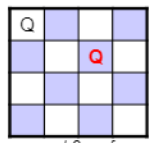
  - row 1:



col 0: same col



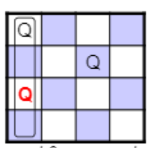
col 1: same diag



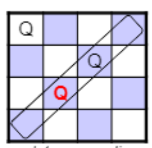
col 2: safe

4-Queens Example (cont.)

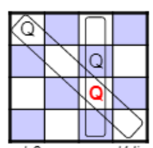
- row 2:



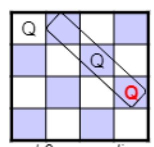
col 0: same col



col 1: same diag

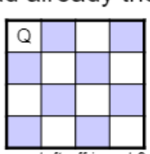


col 2: same col/diag

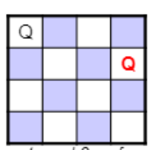


col 3: same diag

- We’ve run out of columns in row 2!
- *Backtrack* to row 1 by returning from the recursive call.
  - pick up where we left off
  - we had already tried columns 0-2, so now we try column 3:



we left off in col 2

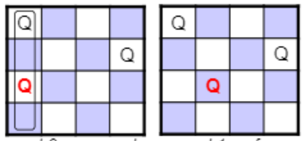


try col 3: safe

- Continue the recursion as before.

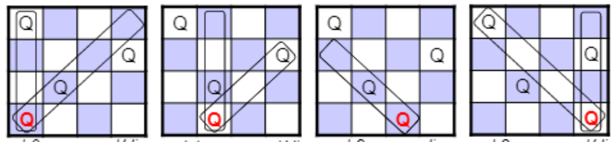
4-Queens Example (cont.)

- row 2:



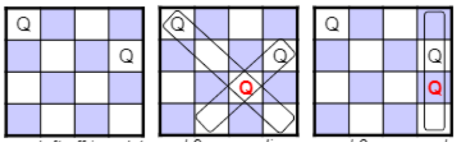
col 0: same col | col 1: safe

- row 3:



col 0: same col/diag | col 1: same col/diag | col 2: same diag | col 3: same col/diag

- Backtrack to row 2:

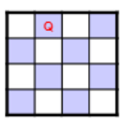


we left off in col 1 | col 2: same diag | col 3: same col

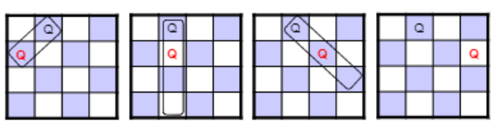
- Backtrack to row 1. No columns left, so backtrack to row 0!

4-Queens Example (cont.)

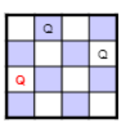
- row 0:



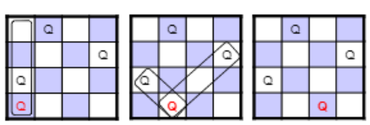
- row 1:



- row 2:



- row 3:



A solution!

findSafeColumn() Method

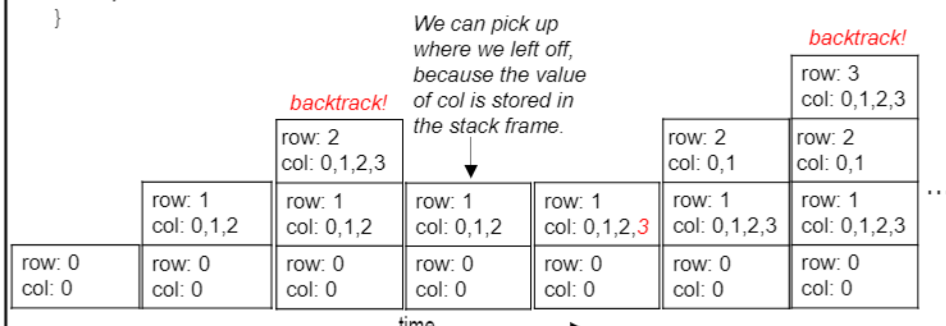
```
public void findSafeColumn(int row) {
    if (row == boardSize) {  // base case: a solution!
        solutionsFound++;
        displayBoard();
        if (solutionsFound >= solutionTarget)
            System.exit(0);
        return;
    }

    for (int col = 0; col < boardSize; col++) {
        if (isSafe(row, col)) {
            placeQueen(row, col);

            // Move onto the next row.
            findSafeColumn(row + 1);

            // If we get here, we've backtracked.
            removeQueen(row, col);
        }
    }
}
```

Note: neither row++ nor ++row will work here.

(see `~cscie119/examples/recursion/Queens.java`)

Tracing findSafeColumn()

```
public void findSafeColumn(int row) {
    if (row == boardSize) {
        // code to process a solution goes here...
    }
    for (int col = 0; col < BOARD_SIZE; col++) {
        if (isSafe(row, col)) {
            placeQueen(row, col);
            findSafeColumn(row + 1);
            removeQueen(row, col);
        }
    }
}
```



Template for Recursive Backtracking

```
void findSolutions(n, other params) {
    if (found a solution) {
        solutionsFound++;
        displaySolution();
        if (solutionsFound >= solutionTarget)
            System.exit(0);
        return;
    }

    for (val = first to last) {
        if (isValid(val, n)) {
            applyValue(val, n);
            findSolutions(n + 1, other params);
            removeValue(val, n);
        }
    }
}
```

Template for Finding a Single Solution

```
boolean findSolutions(n, other params) {
    if (found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val, n)) {
            applyValue(val, n);
            if (findSolutions(n + 1, other params))
                return true;
            removeValue(val, n);
        }
    }

    return false;
}
```

Data Structures for n-Queens

- Three key operations:
  - `isSafe(row, col)`: check to see if a position is safe
  - `placeQueen(row, col)`
  - `removeQueen(row, col)`
- A two-dim. array of booleans would be sufficient:

```
public class Queens {
    private boolean[][] queenOnSquare;
```

- Advantage: easy to place or remove a queen:

```
    public void placeQueen(int row, int col) {
        queenOnSquare[row][col] = true;
    }
    public void removeQueen(int row, int col) {
        queenOnSquare[row][col] = false;
    }
...
```

- Problem: `isSafe()` takes a lot of steps. What matters more?

Additional Data Structures for n-Queens

- To facilitate `isSafe()`, add three arrays of booleans:

```
    private boolean[] colEmpty;
    private boolean[] upDiagEmpty;
    private boolean[] downDiagEmpty;
```

- An entry in one of these arrays is:
  – `true` if there are no queens in the column or diagonal
  – `false` otherwise
- Numbering diagonals to get the indices into the arrays:

upDiag = row + col

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

downDiag =
(boardSize – 1) + row – col

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

Using the Additional Arrays

- Placing and removing a queen now involve updating four arrays instead of just one. For example:

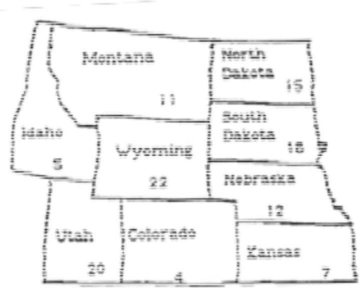
```
public void placeQueen(int row, int col) {
    queenOnSquare[row][col] = true;
    colEmpty[col] = false;
    upDiagEmpty[row + col] = false;
    downDiagEmpty[(boardSize - 1) + row - col] = false;
}
```

- However, checking if a square is safe is now more efficient:

```
public boolean isSafe(int row, int col) {
    return (colEmpty[col]
      && upDiagEmpty[row + col]
      && downDiagEmpty[(boardSize - 1) + row - col]);
}
```

Recursive Backtracking II: Map Coloring

- Using just four colors (e.g., red, orange, green, and blue), we want color a map so that no two bordering states or countries have the same color.
- Sample map (numbers show alphabetical order in full list of state names):



- This is another example of a problem that can be solved using recursive backtracking.