Do Maze!!
Recursion and Backtracking

- All problems considered so far progress steadily towards an answer.
- Consider a maze. Sometimes you need to “backtrack”.
  - RECURSION makes backtracking easy!
- Idea:
  1. Somehow make a copy of where you are,
  2. Try to go forward one step.
     - A. If success,
       - Mark your step on the copy.
       - return to step 1
     - B. If failure
       - throw out copy
       - go some other direction using your original
- Twiddle
  - especially with mazes mark places you have been so you do not retry failed paths
N Queens problem

- Place N queens on an NxN chessboard such that no queen can take another
- Strategy:
  - on row N
    - move across columns trying a spot for OK
      - if found a spot, then recur with N+1
    - if have checked everything in a row and there is no place that is OK
      - backtrack
        - undo placement of queen in row N-1 and continue across that row
N Queens

**setup**

- board just a 2d array of chars
- will do recursion with a private utility function

```java
public class NQueens {
    private char[][] board;
    private int size = 0;

    public NQueens(int siz) {
        size = siz;
        board = new char[siz][siz];
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                board[i][j] = '.';
            }
        }
    }

    private void showBoard() {
        for (int r = 0; r < size; r++) {
            for (int c = 0; c < size; c++) {
                System.out.print(board[r][c]);
            }
            System.out.println("\n");
        }
    }

    public void doQueens() {
        doQueensUtil(0);
    }

    private void doQueensUtil(int row) {
        // Implementation...
    }
}
```
N Queens

recursion

private boolean doQueensUtil(int roww) {
    if (roww >= size)
        return true;
    if (rowOccupied(roww))
        return doQueensUtil(roww + 1);
    for (int col = 0; col < size; col++) {
        board[roww][col] = 'Q';
        if (OKBoard()) {
            boolean v = doQueensUtil(roww + 1);
            if (v)
                return true;
        } else {
            System.out.println("NOT OK");
            showBoard();
            System.out.println("NOT OK " + roww + " " + col);
        }
        board[roww][col] = '-';
    }
    return false;
}
Sudoku

Puzzle solve(Puzzle p, int xloc, int yloc)
if isSolved(p)
    return p
if not isSolvable(p)
    return null
if (yloc>9)
    xloc++
    yloc=0
if (xloc>9)
    return null
if (p(xloc, yloc) != 0)
    return solve(p, xloc, yloc+1)
else
    legalmoves = legalmovesat(p, xloc, yloc)
    foreach legalmove : legalmoves
        set p(xloc, yloc) to legalmove
        np = solve(copy(p), xloc, yloc+1)
        if (np!=null)
            return np
    return null

• Solvable using really stupid recursion
Sudoku

2

Puzzle

at its simplest this could be just a 2d array (specifically 9x9) of int

boolean isSolved(Puzzle p)
return true if the puzzle is completely solved false otherwise

boolean isSolvable(Puzzle p)
return true if the puzzle still might be solvable, false otherwise

List legalmovesat(Puzzle p, int xloc, int yloc)
return the list of numbers that can be legally put into the position given the current board

Puzzle copy(Puzzle p)
return a new instance of puzzle that is an exact copy of the provided puzzle. Importantly, making a change in the copy should have no effect on the original.