I&C SCI 46 Fall 2022: Recursion as a Problem Solving Tool

This handout covers material in lectures 1-3 and should help you follow and organize notes.

Buying Soda in Weird Quantities

Suppose I want to buy $n \geq 12$ cans of soda from a store that sells them only in 4-packs and 5-packs. The store will not allow me to "break apart" a package; for example, if I want 13 cans, I cannot do this by buying two 5-packs and three additional cans, or by buying three 5-packs and discarding two cans.

First, think about how we could achieve getting exactly n cans, for some $n \geq 12$.

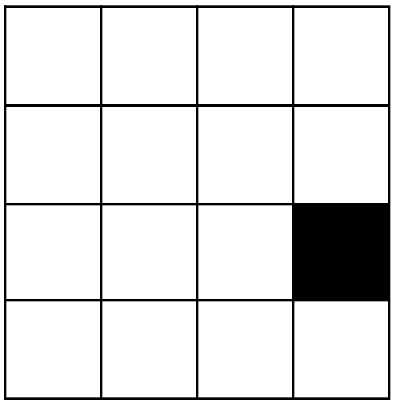
Then, how would you write this function?

```
void buySoda(unsigned n, unsigned & num4Packs, unsigned & num5Packs)
```

Tiling a Chessboard with L-shaped pieces

Suppose I have a $2^n \times 2^n$ chessboard, for some $n \geq 0$. The chessboard is missing one piece. I have an unlimited supply of L-shaped pieces. I want to "tile" the chessboard using these L-shaped pieces, so that every square is covered by exactly one square of a piece, and no portions of the pieces are "hanging off" the board, nor do they cover the initially missing piece.

Example:



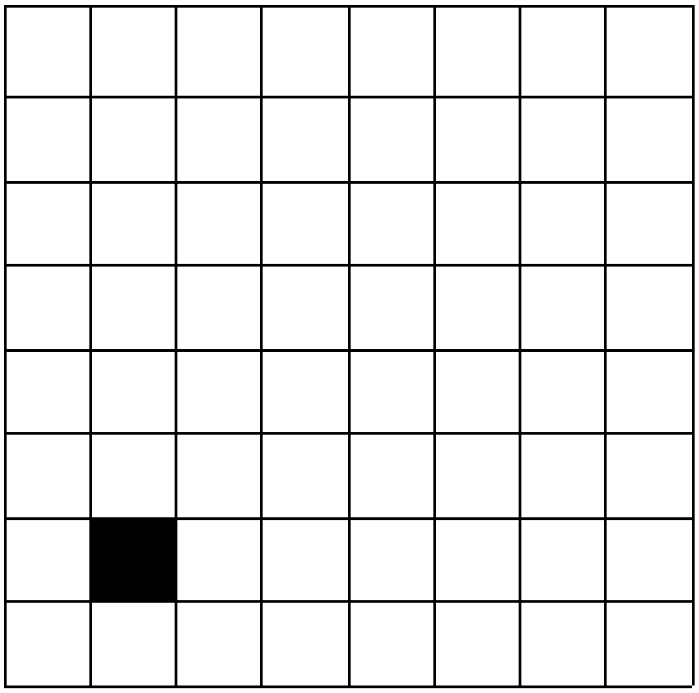
“Types” of Pieces

Each L-shaped piece that can be placed is a 2x2 square with one 1x1 square removed. We refer to these by “types” based on which 1x1 square is removed. The quadrant number removed is the type of the piece. In order to “place” a tile on the board, we have to tell the system where it belongs; this is the X on the chessboard.

```
+--+              +--+        +--+--+         +--+--+
|X |              |X |        |X |  |         |X |  |
+--+--+        +--+--+        +--+--+         +--+--+
|  |  |        |  |  |           |  |        |  |
+--+--+        +--+--+          +--+        +--+

Type 1         Type 2         Type 3         Type 4
```

A Larger Example

Here is an example with *n=3* ; that is, the puzzle is $2^3 \times 2^3$ = 8x8. We will use this to demonstrate how we solve the puzzle.



For this example, which quadrant contains the missing piece?

For this example, where is the first piece we will place? What are the recursive calls?

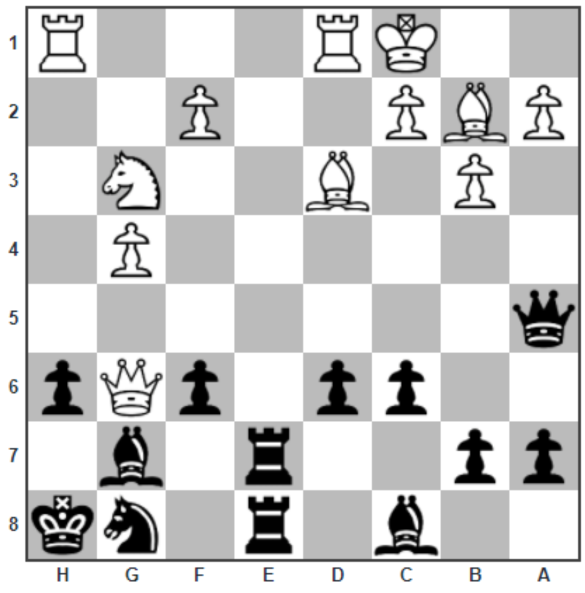
Provided Functions

The code provided has the following functions already implemented for you:

```
// Places a tile of the given type with the identifying square in the designated location.
void placeTile(unsigned type, Point loc);

// returns which quadrant the missing piece is in.
// Assume it's in exactly one (that's always true) and n > 0
unsigned whichQuadrant(Point topLeft, unsigned n, Point missingPiece);

// Gets the top-left of quadrant q from a 2^n x 2^n grid whose
//         top-left point is the first parameter.
Point getTopLeftOfQuadrant(Point topLeft, unsigned n, unsigned q);

// Gets the "internal" corner of quadrant q from a 2^n x 2^n grid whose
//         top-left point is the first parameter.
// An internal corner is the Point closest to the center of the grid.
Point getInternalCorner(Point topLeft, unsigned n, unsigned q);
```

Placing queens on a chessboard

Suppose you have an *n* x *n* chessboard and *n* queens. The queen is a chess piece that threatens any piece on the same row, column, or diagonal she is on, unless another piece is in between the two. Your goal is to place all *n* queens on the board so that no two threaten one another. The queen on g6 in the following diagram threatens four enemy pieces (although there is a better move if it is her turn, that is not the point of this exercise).



For example, here are two attempts at placing four queens on a 4x4 chessboard. Are they valid?

		Q	
	Q		
			Q
Q			

	Q		
			Q
Q			
		Q	

Let's use recursion to write a program that will take as input an unsigned integer *n* and find a placement of *n* queens onto an *n x n* chessboard in such a way that no two threaten one another. We can express our output as a set of columns, where the *i*th queen is placed into the *i*th row in column *queens[i]*. To help us solve the problem, we have vector threats, where *threats[i][j]* is the number of queens on rows prior to *i* that threaten the given square.

The provided code has a print function and a main to set things up. We need to implement the following functions. They have additional parameters to keep the board and threat tracker as well.

- void search (unsigned row, std::vector<unsigned> & queens, std::vector< std::vector<unsigned> > & threats, unsigned n)

  This is the main function for our solution: start a search by placing a queen in the given row. This assumes each previous row has exactly one queen, and will initially be called with row=0.

- void changeThreats(unsigned row, unsigned column, int changestd::vector< std::vector<unsigned> > & threats, unsigned n)
  This is a helper function: given that we have placed or removed a queen on the board at (row, column), we need to update the number of threats to lower rows' squares.

Reinforcement: after lecture, think about how you would adjust the code we wrote for this problem to, rather than output every valid board, instead count how many valid boards solve this problem.