Eric Roberts
CS106B

Handout #23
April 20, 2009

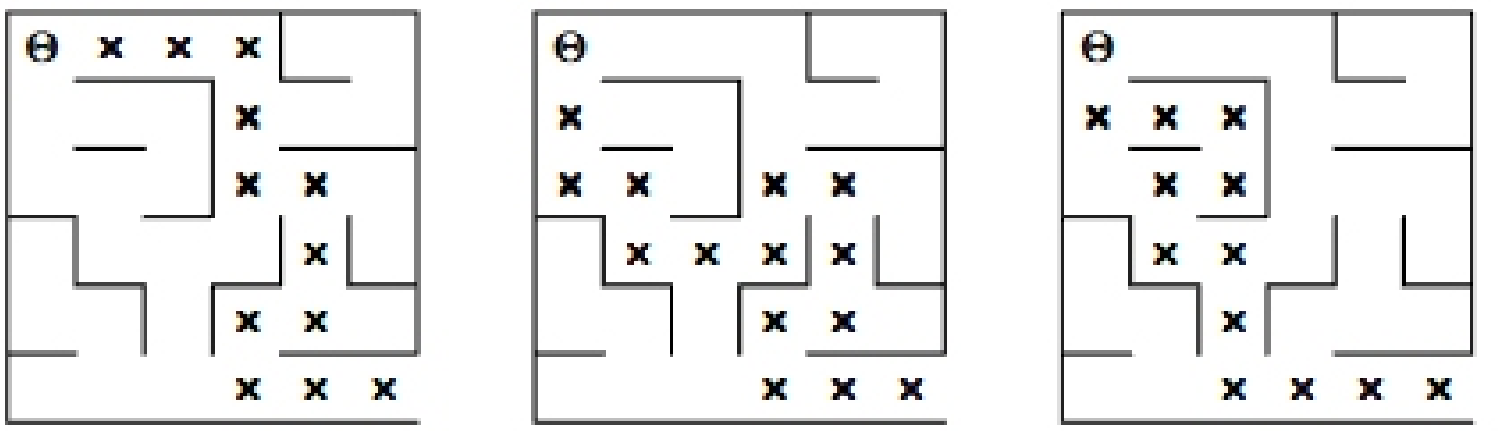
Section Handout #3—Recursion and Big O

Parts of this handout were written by Julie Zelenski and Jerry Cain.

The purpose of this section is to give you some additional practice solving recursive problems and to test your understanding of complexity classes and Big-O notation.

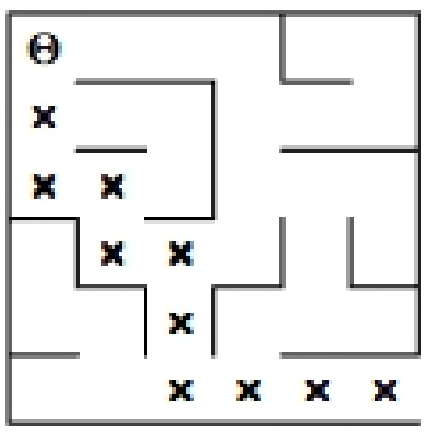
Problem 1. Shortest path (Chapter 7, exercise 2, page 273)

In many mazes, there are multiple paths. For example, the diagrams below show three solutions for the same maze:



length = 13 length = 15 length = 13

None of these solutions, however, is optimal. The shortest path through the maze has a path length of 11:



Write a function

```
int ShortestPathLength(pointT pt);
```

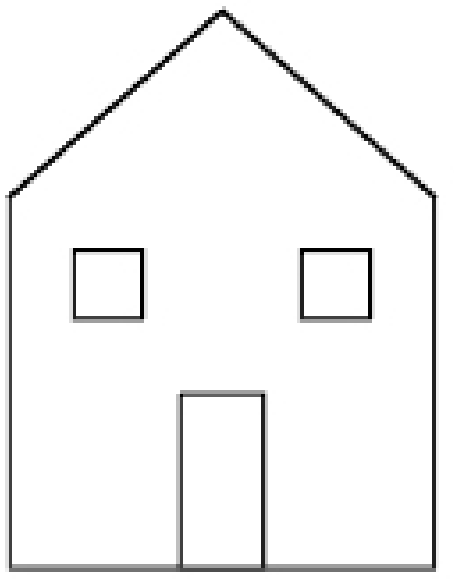
that returns the length of the shortest path in the maze from the specified position to any exit. If there is no solution to the maze, `ShortestPathLength` should return the constant `NO_SOLUTION`, which is defined to have a value larger than the maximum permissible path length, as follows:

```
const int NO_SOLUTION = 10000;
```

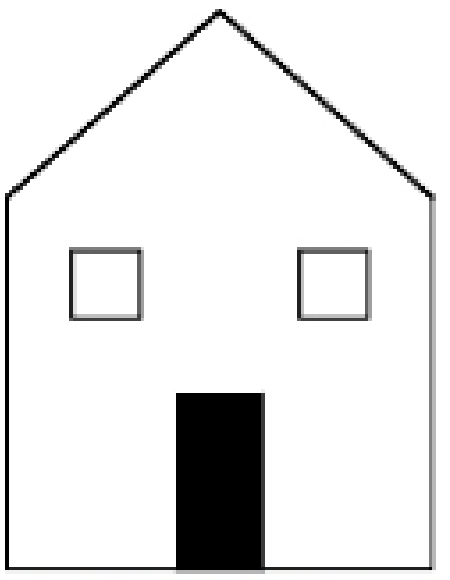
Problem 2. Filling a region (Chapter 7, exercise 6, page 275)

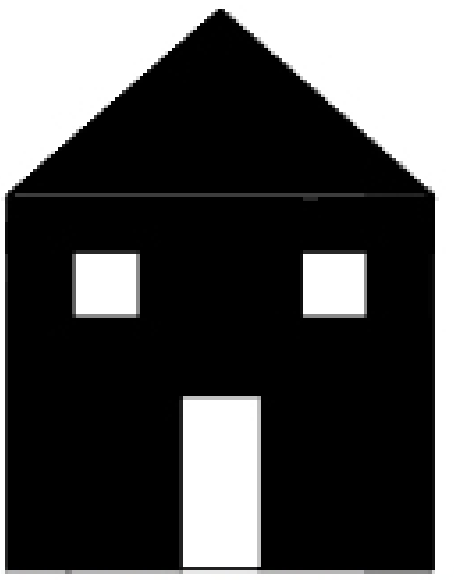
Most drawing programs for personal computers make it possible to fill an enclosed region on the screen with a solid color. Typically, you invoke this operation by selecting a paint-bucket tool and then clicking the mouse, with the cursor somewhere in your drawing. When you do, the paint spreads to every part of the picture it can reach without going through a line.

For example, suppose you have just drawn the following picture of a house:

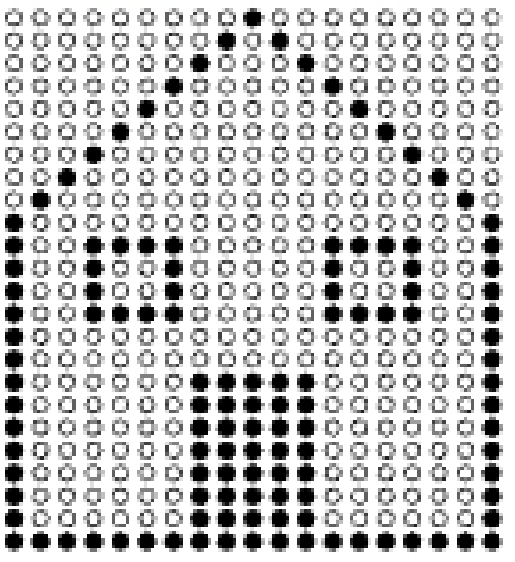


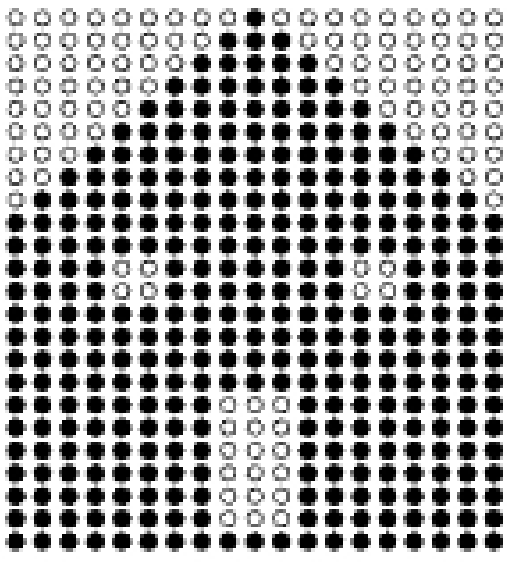
If you select the paint bucket and click inside the door, the drawing program fills the area bounded by the door frame as shown at the left side of the following diagram. If you instead click somewhere on the front wall of the house, the program fills the entire wall space except for the windows and doors, as shown on the right:





In order to understand how this process works, it is important to understand that the screen of the computer is actually broken down into an array of tiny dots called **pixels.** On a monochrome display, pixels can be either white or black. The paint-fill operation consists of painting black the starting pixel (i.e., the pixel you click while using the paint-bucket tool) along with any pixels connected to that starting point by an unbroken chain of white pixels. Thus, the patterns of pixels on the screen representing the preceding two diagrams would look like this:





Write a program that simulates the operation of the paint-bucket tool. To simplify the problem, assume that you have access to the enumerated type

```
enum pixelStateT { WHITE, BLACK };
```

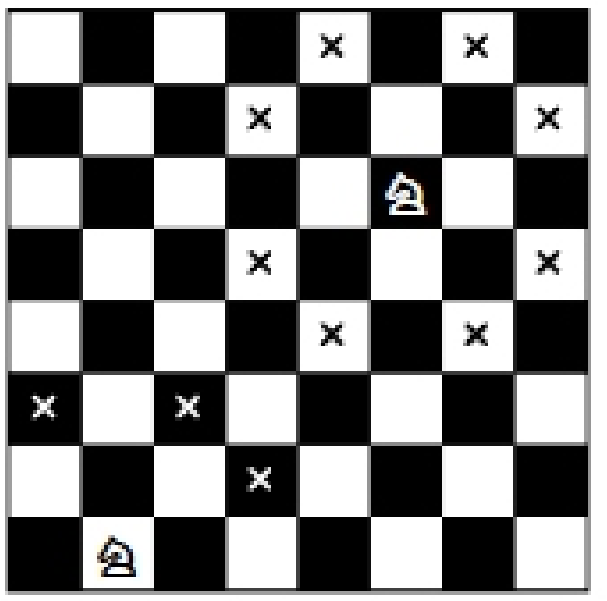
and the following functions:

```
pixelStateT GetPixelState(pointT pt);
void SetPixelState(pointT pt, pixelStateT state);
bool OutsidePixelBounds(pointT pt);
```

The first function is used to return the state of any pixel, given its coordinates in the pixel array. The second allows you to change the state of any pixel to a new value. The third makes it possible to determine whether a particular coordinate is outside the pixel array altogether, so that the recursion can stop at the edges of the screen.

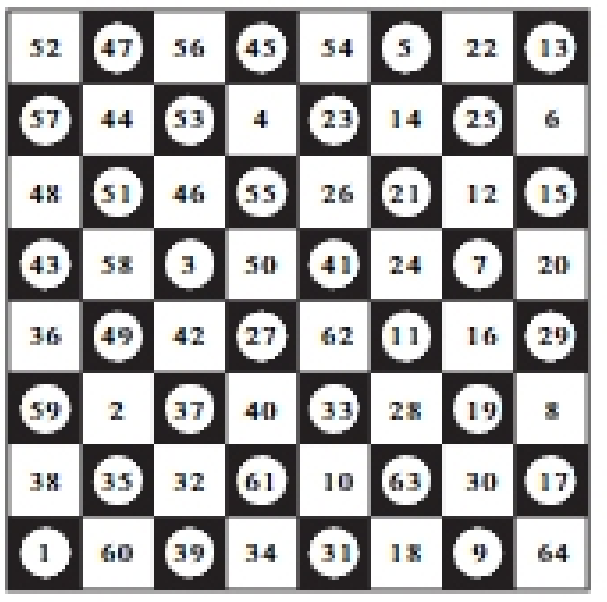
Problem 3. Generating a knight's tour (Chapter 7, exercise 7, page 276)

In chess, a knight moves in an L-shaped pattern: two squares in one direction horizontally or vertically, and then one square at right angles to that motion. For example, the knight in the upper right side of the following diagram can move to any of the eight squares marked with a black cross:



The mobility of a knight decreases near the edge of the board, as illustrated by the bottom knight, which can reach only the three squares marked by white crosses.

It turns out that a knight can visit all 64 squares on a chessboard without ever moving to the same square twice. A path for the knight that moves through all the squares without repeating a square is called a **knight's tour.** One such tour is shown in the following diagram, in which the numbers indicate the order in which the squares were visited:



Write a program that uses backtracking recursion to find a knight's tour.