CMPSCI 187: Programming With Data Structures

Lecture #15: Thinking Recursively
David Mix Barrington
10 October 2012

Thinking Recursively

- Recursive Definitions, Algorithms, and Programs
- Computing Factorials
- Three Questions About a Recursive Algorithm
- Towers of Hanoi
- Grids and Continents (Blobs)
- Counting Continents
- Recursively Marking Continents
- Code for the `Grid` Class

Recursive Definitions, Algorithms, and Programs

- A common error in logic is **circular definition**, as in “good music is music that is good”. In ordinary language, defining a concept in terms of itself gives no further information about the concept and is thus useless.

- But in mathematics and computer science we often have **recursive definitions** of concepts that refer to themselves. A “directory” in most operating systems, for example, is something that contains things that are either files or directories. A “statement” in Java might be an “if statement”, and an “if statement” is made up of the words `if` and `else`, a boolean expression in parentheses, and one or two “statements”. A “postfix expression” is either an operand or two “postfix expressions” followed by an operator. A “stack” is an “empty stack” or a “stack” with an item pushed.

- **Recursive algorithms** are algorithms that call themselves. They are most useful when applied to recursively defined concepts. Much of CMPSCI 250 is devoted to the relationship of recursive definitions and algorithms to **inductive proofs** -- here we will look at the definitions and algorithms.

Computing Factorials

- The **factorial** of a non-negative integer n, written “n!”, is the product of all the numbers from 1 through n, inclusive, or “1 * 2 * ... * n”. (The factorial of 0 is 1, because multiplying together an empty set of numbers is like not multiplying at all, or multiplying by 1.) For example, 5! = 1 * 2 * 3 * 4 * 5 = 120.

- Here is a recursive definition of “factorial”: “0! = 1, and if n > 0, n! = n * (n-1)!”

- If we use this definition to try to calculate 5!, we get that 5! = 5 * 4!. If we apply the definition again to 4!, we get that 5! = 5 * 4 * 3!. Continuing in the same way, we get 5! = 5 * 4 * 3 * 2! = 5 * 4 * 3 * 2 * 1! = 5 * 4 * 3 * 2 * 1 * 0! = 5 * 4 * 3 * 2 * 1, just as in the non-recursive definition.

- The recursive definition is easy to turn into code:

```
public static int factorial (int n) {
   if (n == 0) return 1;
   else return n * factorial (n-1);}
```

More About Factorials

- What happens when we run the recursive code with a parameter of 5? That call makes a call on factorial(4), which calls factorial(3), which calls factorial(2), which calls factorial(1), which calls factorial(0), which returns 1 to the factorial(1), which returns 1 to factorial(2), which returns 2 to factorial(3), which returns 6 to factorial(4), which returns 24 to factorial(5), which returns 120, the right answer.

- The original definition is also easy to convert into code, of course:

```
public static int recursiveFactorial (int n) {
   if (n == 0) return 1;
   else return n * factorial (n-1);}

public static int iterativeFactorial (int n) {
   int x = 1;
   for (int i = 1; i <= n; i++)
       x *= i;
   return x;}
```

Three Questions About a Recursive Algorithm

- For a recursive algorithm to work correctly on a particular input, we need positive answers to the **three questions** that DJW pose in Section 4.2.
- Does the algorithm **have a base case**, where there is no further recursion and it gets the right answer? Our factorial algorithm has a base case of n = 0.
- Does every recursive call **make progress** toward the base case? For example, is the parameter of every recursive call **smaller than** the parameter of the call making it, in some definable way? Our factorial(n) always calls factorial (n-1), so the parameters get smaller as long as we aren't negative.
- Can we show that the general call to the algorithm gets the right answer **if we assume that all the recursive calls get the right answer**? This claim is usually justified by a recursive definition.

Towers of Hanoi

- In the **Towers of Hanoi** game, there are three pegs and n rings of different sizes. Originally all n rings are on the first peg, with the largest at the bottom and the others in order of size above it. We want to move all the rings to another peg, following the rules: (1) we move one ring at a time, and (2) no larger peg may ever be put above a smaller peg. (A legend has it that the world will end as soon as some monks someplace finish the n = 64 version.)

- We can recursively solve the puzzle by defining the following procedure to move k rings from peg A to peg B. This turns out to make 2^k - 1 moves.

```
public void move (peg A, peg B, int k) {
   if (k == 1) move one ring from A to B;
   else {
      move (A, C, k-1);
      move one ring from A to B;
      move (C, B, k-1);}
```

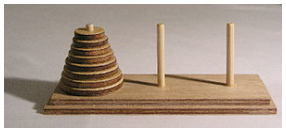

Image from wikipedia.org: "Towers of Hanoi"

Grids and Continents (Blobs)

- In games like *Civilization™*, the computer creates a map to play on, with land areas, water areas, different terrain for each area, and so forth. We're now going to look at a simple version of this where the world is a rectangular grid of squares, each one land or water. Squares are considered **adjacent** if they share a horizontal or vertical edge, not if they just meet at a corner.

- A **continent** (called a **blob** in DJW) is a set of land squares such that you may travel by land from any square on the continent to any other square on the continent, but to no square not on the continent. Again, you cannot jump diagonally across a corner in this process.

- We're going to look at the computational problem of **counting the continents** in a particular rectangular map. We'll be greatly helped by the use of recursion. Programming Project #3 will build on DJW's code for this.

Counting Continents

- In Project #3 we'll ask you to be able to mark the continents on a grid as in the example below. But DJW's code just *counts* the continents, which is simpler.

- The first might idea might be to traverse the grid, counting the squares, but this overcounts whenever two or more squares are on the same continent. The right idea is to traverse, but to **mark** all the squares on a given continent the first time you find one of its squares. Then you just count one every time you find an unmarked land square in the traversal.

```
X-X--X--X--  A-A--B--C--
XXX--X-----  AAA--B-----
--XX-X-----  --AA-B-----
-----XX----  -----BB----
----X---XXX  ----D---EEE
```

Using Recursion to Mark Continents

- The marking algorithm requires us to have a boolean array `visited` that holds `true` for every land square that is part of the same continent as a square that has been traversed. The marking algorithm will set this visited value true for every square reachable from the square denoted by its parameters.

- The plan is just to mark the target square, then recursively mark any of its four neighbors that are in the grid and are unseen land squares.

```
private void markBlob (int row, int col) {
   visited [row][col] = true;
   if ((row - 1 >= 0)  // on grid
      if (grid[row - 1][col]) // land
         if (!visited[row - 1][col]) //not seen
            markBlob(row - 1, col);
   //same for (row + 1, col)
   //same for (row, col - 1)
   //same for (row, col + 1)
```

Three Questions for `markBlob`

- Base Case: Whenever we call `markBlob` on a square that has no unseen land squares as neighbors, it just marks its target square and returns.
- Progress Toward Base Case: Every time we mark a square, we reduce the total number of unseen land squares in the grid, which started out as a finite number. And every call to `markBlob` marks at least one such square, because we only ever call it on an unseen land square.
- General Correctness: We want to be sure that when markBlob (i, j) is called, it marks (i, j) and every unseen land square that can be reached from (i, j) by using unseen land squares. Suppose we are sure that this property is true for all the calls to `markBlob` on neighbors of (i, j). Then if a square is reachable from (i, j), it must be either be (i, j) itself or be reachable from one of its neighbors. So if the neighbor calls do their job, so does the first call.
- In CMPSCI 250 we'll phrase this argument as a **proof by induction**.

Code for the `Grid` Class

- Here are the data fields and constructor for the Grid class. I have altered DJW's code to have the constructor take a seed, for ease in grading later.

```
import java.util.Random;
public class Grid {
   protected int rows, cols;
   protected boolean [ ] [ ] grid; // true = land
   boolean [ ] [ ] visited; // true = seen
   public Grid (int rows, int cols, int pct, int seed) {
      this.rows = rows; this.cols = cols;
      grid = new boolean [rows] [cols];
      int randInt;
      Random rand = new Random (seed);
      for (int i = 0; i < rows; i++)
         for (int j = 0; j < cols; j++) {
            randInt = rand.nextInt (100);
            grid [i][j] = (randInt < pct);}}
```

More Code for the Grid Class

- Here are the `toString` and `blobCount` methods.

```
public String toString( ) {
   String gridString = “”;
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         if (grid[i][j]) gridString += “X”;
         else gridString += “-”;}
      gridString += “\n”;}
   return gridString;}

public int blobCount( ) {
   int count = 0;
   visited = new boolean [rows][cols];
   // initialize visited to all false
   for (int i = 0; i < rows; i++)
      for (int j = 0; j < cols; j++)
         if (grid[i][j] && !visited[i][j]) {
            count++; markBlob(i, j);}
   return count;}
```