# CMPSCI 187: Programming With Data Structures

Lecture 23: Examples of Recursion
4 November 2011
Guest Lecturer: Bobby Simidchieva

# Examples of Recursion

- Recursively Searching a Maze
- Depth-First Search Without the `seen` Field
- The Towers of Hanoi Problem
- Exponentially-Growing Running Times
- Analysis of Recursive Algorithms
- Examples of Time Complexity with Recursion

# Recursively Searching a Maze

- In Section 7.3. L&C return to their maze search example from Chapter 4, from which we adapted Project #2. This time they write a recursive solution, which has somewhat simpler code and does not use an explicit `Stack` object. The increased clarity of reasoning is exemplified by the fact that this new version is actually correct.

- In our setting, the basic idea is to search a cell for a path to the destination by searching each of its *unseen* neighbors for a path to the destination. After all, the path from the source can exist only if a path from one of the neighbors exists.

- There is clearly a base case, since if we are asked for a path from the destination to itself we can just return it. But why can we be sure that this recursive algorithm will ever reach the base case?

# Depth-First Search With the Method Stack

- Let's trace the action of the recursive code below. If the source has any open unseen neighbors, we recurse on the first one, then on that one's first open unseen neighbor, and so on. We are exactly mimicking Project #2's search.

- We only finish processing a cell when we have finished with all its open unseen neighbors. At that point it is correct to give up on its having a path.

```
public SCell [ ] pathFrom (int sx, int sy, int dx, int dy) {
   if (sx == dx && sy == dy) // return one-cell path with s = d
   getCell (sx, sy).setSeen (true);
   SCell [ ] neighbors = moves(sx, sy);
   for (int i = 0; i < neighbors.length; i++) {
      SCell here = moves[i];
      if (here.getSeen( )) break;
      SCell [ ] fromHere = pathFrom
                (here.getXCoord( ), here.getYCoord( ), dx, dy);
      if (fromHere != null) // prepend s to fromHere and return
   }
   return null;}
```

# Depth-First Search Without the `seen` Field

- Marking nodes as seen is absolutely vital to make a depth-first search as well, in any situation where paths might revisit nodes.

- Suppose that in our recursive maze search, we had no seen field. From the source s, we find a neighbor x and recurse on it. But s itself is one of x's neighbors, so at some point in the search of x we will recursively begin searching s again. If nothing has changed since we first began searching s, everything will go as before and we will eventually begin a third search of s, then a fourth, and so on. This is an ungrounded indirect recursion.

- With breadth-first search and no marking of seen nodes, we will still find a path if one exists, as long as each cell has only finitely many neighbors. But we have no way to terminate a failed search (also a problem with DFS above).

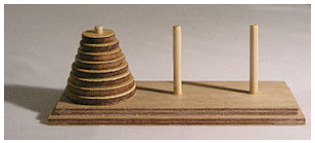
- We'll see more about this in CMPSCI 250 and CMPSCI 311.

# The Towers of Hanoi Problem

- The Towers of Hanoi puzzle was invented by Édouard Lucas in 1883. You have three towers, and start with n disks on one tower. No larger disc may ever go on a smaller disk, and you may move only one disk at a time. The goal is to move all n disks to another tower.

- If n = 0 or n = 1, the solution is easy. We can use this as the base case of a recursion. The idea is this: To move n disks from tower A to tower B, we first move the top n - 1 disks from A to C, move the largest disk from A to B, then move the other n - 1 disks from C to B.

- With n = 2, we move A to C, A to B, C to B.



- With n = 3, we need seven moves in all.

# Exponentially Growing Running Times

- We saw that for n = 0, 1, 2, 3... we need 0, 1, 3, 7... single-disk moves.
- We might recognize this sequence as 1-1, 2-1, 4-1, 8-1,..., or 2^n - 1.
- If f(n) is the number of moves for n disks, we can see from the code that f(n) = f(n-1) + 1 + f(n-1) = 2*f(n-1) + 1. Using induction (a CMPSCI 250 technique) we can *prove* that f(n) = 2^n - 1 is the solution to this **recurrence**.
- In Lucas' original story, some priests are doing the puzzle with n = 64, and the world will end when they finish. Fortunately 2^{64} - 1 is a *very* big number. And it's possible to prove that no other solution uses fewer moves.

```
private void moveTower (int num, int start, int end, int temp) {
   if (num == 1) moveOneDisk (start, end);
   else {
      moveTower (num - 1, start, temp, end);
      moveOneDisk (start, end);
      moveTower (num - 1, temp, end, start);}}
```

# Analysis of Recursive Algorithms

- We've now seen the basic idea of how to analyze the running time of a recursive algorithm. There must be a base case, which we can analyze as before because it contains no recursion. And there should be a parameter that moves toward zero as we approach the base case. We want to argue about a worst-case time for each value of the parameter.

- If we know the worst-case time for all *smaller* values of the parameter, we can figure out the worst-case time for a new value by analyzing the code. This analysis often comes in the form of a recurrence -- an equation defining the new running time in terms of the times for smaller values.

- There are organized techniques for dealing with recurrences, which we'll study in CMPSCI 250 and especially in CMPSCI 311. For now we'll look at some representative examples.

# Examples of Time Complexity With Recursion

- Suppose our code is a tail recursion, with the n case calling the n-1 case, that the rest of the code is O(1), and that the base case is O(1). We get a recurrence f(n) = f(n-1) + O(1) with base case f(0) = O(1), which solves to O(n). Similarly, if we have f(n) = f(n-1) + g(n), and f(0) = h, we get f(n) = n*g(n) + h.

- If we can cut a problem in half with O(1) work, as in Binary Search, we get a recurrence f(n) = f(n/2) + O(1), which solves to f(n) = O(log n).

- The Merge Sort algorithm will give f(n) = O(n log n) by a more complicated analysis.

- With Towers of Hanoi, our recurrence f(n) = 2*f(n-1) + 1 gives us f(n) = O(2^n). Any situation where you *multiply* the time by a constant greater than one, in order to increase n by one, will give you an exponential growth function.