CMPSCI 250: Introduction to Computation

Lecture #14: Induction and Recursion (Still More Induction)
David Mix Barrington
14 March 2013

Induction and Recursion

- Three Rules for Recursive Algorithms
- Proving a Recursive Algorithm Correct by Induction
- Recursively Defined Functions
- The Fibonacci Numbers
- The Behavior of the Euclidean Algorithm
- Recursively Defined Structures
- Recursion on Trees

Three Rules for Recursive Algorithms

- A recursive algorithm is one that calls itself with a different parameter. For example, we might calculate the factorial n! by using a recursive call to compute (n-1)!, then multiply the result by n.

- In CMPSCI 187, I teach three rules -- three properties to check in order to verify the correctness and termination of a recursive algorithm:

- The algorithm must have a base case where it gets an answer without further recursion.

- Every recursive call must make progress towards the base case.

- If every recursive call terminates and gives the correct output, the original call will terminate and give the correct output.

Proving Recursive Algorithms Correct by Induction

- Suppose my recursive algorithm A has one parameter which is a natural number n. Let P(n) be the statement that the method call A(n) terminates with the correct output. How would I prove $\forall n:P(n)$ by strong induction?

- 1) I would prove P(0), which says that A(0) terminates with the correct output.

- 2) I would make sure that all calls from A(n) are to methods A(i) with $i < n$.

- 3) I would prove that *if* the strong inductive hypothesis holds, and thus all the recursive calls terminate with the correct output for them, *then* the original call to A(n) terminates with the correct output.

- The verification rules from CMPSCI 187 are just the steps of the strong induction proof, in the case where there is one parameter that is a natural.

Recursively Defined Functions

- One virtue of recursive algorithms is that they are often easy to prove correct. This is particularly true when the desired function itself has a **recursive definition**.

- We can define the **factorial** function fact(n) by the two rules fact(0) = 1 and fact(n) = n·fact(n-1), where the latter rule holds whenever $n \geq 1$. It's easy to write a recursive method that matches this definition, and prove that method correct. We can also write recursive definitions for addition, multiplication, and powering.

- Often the running time of an algorithm is given by a **recurrence**, which is a kind of recursive definition. We'll see more of this in CMPSCI 311

- With more than one parameter we have to know that the recursive calls don't go on forever, which is saying that the set of parameter values is a well-ordered set and that the calls always go downward in its ordering.

iClicker Question #1: Recursive Powering

- Suppose we want a recursive definition for a function power(x, y), where the intended result is x^y. Which set of rules will give us a correct definition?

- (a) power(x, 0) = 1; power(x, y) = x·power(x, y-1) whenever y ≥ 1

- (b) power(x, 0) = 0; power(x, y) = power(x, 1)·power(x, y-1) whenever y ≥ 1

- (c) power(x, 0) = x; power (x, y) = power(x, 0)·power(x, y) whenever y ≥ 1

- (d) power(x, 0) = 1; power(x, y) = power(x, 0)·power(x, y) whenever y ≥ 1

The Fibonacci Numbers

- The **Fibonacci numbers** have a recursive definition with two base cases. We define fib(0) = 0, fib(1) = 1, and for any $n \geq 1$, fib(n+1) = fib(n) + fib(n-1). So fib(2) = 1 + 0 = 1, fib(3) = 1 + 1 = 2, fib(4) = 2 + 1 = 3, and fib(5) = 3 + 2 = 5.

- We could calculate fib(n) by the following Java method:

```
public int fib (int n) {
   if (n <= 1) return n;
   return fib(n-1) + fib(n-2);}
```

- This is clearly correct, but it is woefully inefficient as you can see if you trace its execution on input 5. It takes about fib(n) steps, so on input 100 it would take longer than the expected age of the universe to compute the answer of 354224848179261915075. If we **memoize** to avoid recalculating the same function value more than once, however, we can compute large values easily.

The Behavior of the Euclidean Algorithm

- We can easily prove by strong induction that the **Euclidean Algorithm** terminates given any two natural numbers as input. Let P(n) be the statement that the EA terminates on input (n, b) when b < n. P(1) is true because the only possible case has b = 0 and this terminates immediately. If we assume P(k) for all k ≤ n and look at the EA on input (n+1, b), we see that after one step we call the EA again on input (b, c) for c = (n+1)%b, and this call must terminate because we know that P(b) is true.

- How many steps does the EA take? Rosen shows by induction on n that if the EA takes n divisions to find gcd(a, b), then b must be at least fib(n+1). He also shows that fib(n+1) is at least α^{n-1}, where α is the number $(1 + \sqrt{5})/2$ which is about 1.61. This shows **Lame's Theorem**, which says that the number of divisions in the EA is at most five times the number of digits in the input numbers. This means that we take O(n) time to test whether two n-digit numbers are relatively prime, making this test practical even for huge numbers.

iClicker Question #2: Fibonacci Bounds

- Let α be the number $(1 + \sqrt{5})/2$, which is about 1.6. We can prove by strong induction that for any n with $n \geq 3$, the Fibonacci number fib(n) is at least α^{n-2}. **What two consequences** of the strong inductive hypothesis will we need to **complete the inductive step**?

- (a) We need to prove "$fib(4) > \alpha^2$" and "$fib(3) > \alpha$" for the base case.

- (b) We need "$fib(3) > \alpha$" and "$fib(n-1) > \alpha^{n-3}$" to prove "$fib(n) > \alpha^{n-2}$".

- (c) We need "$fib(0) = 0$" and "$fib(1) = 1$" to prove "$fib(n+1) = fib(n) + fib(n-1)$".

- (d) We need "$fib(n) > \alpha^{n-2}$" and "$fib(n-1) > \alpha^{n-3}$" to prove "$fib(n+1) > \alpha^{n-1}$".

Recursively Defined Structures

- Many structures in computer science have recursive definitions. This allows us to define functions on those structures recursively, and write recursive methods to implement those functions.

- A **stack** is a data structure that is either an **empty** stack, or a stack with an element **pushed** onto it. We can define the **pop** operation recursively -- a pop from an empty stack is an error, and popping from “S with x pushed onto it” returns the value x and makes the stack equal to S. The **size** of an empty stack is 0, and the size of “S with x pushed onto it” is the size of S plus 1.

- We could write similar definitions for queues and lists, for binary search trees, for arithmetic expressions, and a host of other structures. As in Rosen, we can recursively define strings over an alphabet, and functions like concatenation and length. Each kind of structure has a “law of induction” to say when a definition applies to all possible structures.

iClicker Question #3: Stacks

- Each of the following is an informal description of a recursive method which is meant to **clear** a stack, which means replacing it with an empty stack. **Which method is correct?**
- (a) Pop an element and clear the stack.
- (b) If the stack is empty, do nothing. Otherwise pop an element.
- (c) Create a new stack, pop an element from the old stack, and push that element onto the new stack.
- (d) If the stack is empty, do nothing. Otherwise pop an element and clear the stack.

Recursively Defined Trees

- **Tree structures** are ubiquitous in computer science. There are many different classes of trees, with slight variations in their definitions and a different name in each setting. But Rosen defines three types in Section 5.3, each with a recursive definition.

- A **rooted tree** is either a single vertex (its own root) or a vertex with one or more children and an edge to the root of each child. Rooted trees model directory trees in a file system, and the object hierarchy in Java.

- An **extended binary tree (EBT)** is either the empty set or a vertex with two children, each of which is an EBT. The right and left children are distinguished.

- A **full binary tree (FBT)** is either a single vertex or a vertex with left and right children that are each FBT's -- contrast this with the CMPSCI 187 definition where a full binary tree had to be balanced.

Recursion on Binary Trees

- Each of these tree types can easily be embodied as a Java class, where each object has fields that are pointers to other objects of the same class:

```
public class Tree {
   Node root;
   Tree left;
   Tree right;
```

- We can write instance methods in such a `Tree` class to, for example, count the elements in the calling `Tree` object:

```
public size( ) {
   if ((left == null) && (right == null)) return 1;
   else return left.size( ) + right.size( ) + 1;}
```

- Similarly, we can use recursion to find the depth of the tree or some function of the node values, such as their sum or their maximum if they are numbers.

iClicker Question #4: Binary Trees

- Suppose that every node in a full binary tree (whether a leaf or an internal node) has a numerical value. I want to write a `sum` method that will return the sum of the values of all the nodes in the tree. **Which of these** is the correct **recursive** way to do this?

- (a) Return the sum of the left subtree plus the sum of the right subtree.

- (b) If the tree has only one node, return its value. Otherwise return the root's value plus the sum of the left subtree plus the sum of the right subtree.

- (c) If the tree has only one node, return its value. Otherwise return the value of the root's left child plus the value of the root's right child.

- (d) Set a variable to zero, then visit each node in turn using breadth-first search and add its value to the variable, returning the variable at the end.

Expression Trees

- We've seen both boolean and arithmetic expressions in this course, and both kinds of expressions may be modeled by binary trees. An expression is made up from **atoms** (constants or variables), connected by unary or binary **operators**. The expression tree has an atom at each leaf and an operator at each internal node.

- Expression trees must resolve any ambiguity about the order of operations in the expression. In CMPSCI 187 we discussed how to transform such a tree into a string, and vice versa.

- In the next discussion, the Monday after break, you'll write some Java code for boolean expression trees using a class definition I'll give you. There the value of a leaf will be true or false, and the operator at an internal node will be AND, OR, or NOT. (A NOT node has only a left child, while AND and OR nodes have both left and right children.)