CMPSCI 187: Programming With Data Structures

Lecture #22: Introduction to Recursion
2 November 2011

Introduction to Recursion

- What is Recursion, and Why Recursion?
- Ungrounded Recursion
- The Factorial Function
- Tracing Recursive Code: Computing Sums
- Iterative Alternatives
- Indirect Recursion

What is Recursion, and Why Recursion?

- A method is **recursive** if it calls itself. A method can call any other method, and its own execution is suspended until (or unless) the new method finishes.

- When more than one version of the same method is on the stack, they have separate **contexts** -- separate copies of local variables and parameters. None of them affect any of the others, except by finishing their execution.

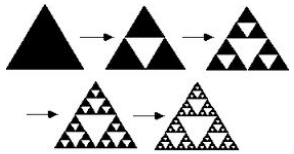
- Recursive code is often the cleanest and simplest way to solve a problem.

- Many mathematical and computing concepts have **recursive definitions**. For example, a stack is either empty, or is another stack with an element pushed onto it. A recursive algorithm can often be written directly from a recursive definition. This is a main theme of CMPSCI 250 -- the close connection between algorithm and definition makes it easier to prove correctness, often by mathematical induction.

More Recursion Examples

- *"Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law."* -- Douglas Hofstadter, in *Gödel, Escher, Bach*

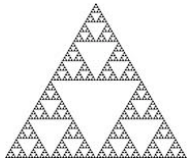
- The Sierpinski Triangle: Three half-size Sierpinski Triangles with a blank inverted triangle in the middle. An example of a **fractal**.



- Fibonacci: f(0) = 0, f(1) = 1, f(n+1) = f(n) + f(n-1)

- Binary search: Find which half, Binary Search it

- Merge Sort: Merge Sort each half, merge results.



- Next Lecture: Maze Search, Towers of Hanoi

- Google "recursion"

Ungrounded Recursion

- *"I just spent all morning in the shower. The directions on the shampoo said: Lather -- Rinse -- Repeat."* (Stephen Wright)

- If method calls itself, and the new call calls the method again, and that one calls the method again, and so on, the execution can only stop if the repeated calls stop. We need a **base case**, a situation where the method finishes without any further recursive calls. Knowing that a recursive algorithm will terminate means knowing that it will reach a base case.

- In principle, an ungrounded recursion would run forever (or until some outside agency cuts it off). Actually, what will typically happen first is that the **method stack**, where the context of each new call is stored, will reach its capacity and cause a runtime error.

The Factorial Function

- A recursive definition: “The factorial of 0 is 1. If n > 0, then the factorial of n is n times the factorial of n - 1.”

- So the factorial of 4 (written “4!”) is 4 * 3!, or 4 * (3 * 2!), or 4 * (3 * (2 * 1!)), or 4 * (3 * (2 * (1 * 0!))), or 4 * (3 * (2 * (1 * 1))), which comes out to 24.

- This is equivalent to the more familiar definition: “n! = 1*2*...*n”. (Why does this one give you “0! = 1”?) Both definitions lead directly to code:

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

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

Tracing Recursive Code: Computing Sums

- Here is a recursive method to sum the first n positive integers. (I changed L&C's method to handle input 0 and rename the input parameter.)

- The recursion is ungrounded if it is ever called with negative input. We could fix this by saying `if (n <= 0)` in the third line.

- Look at what happens if we call `sum(4)`. The first version of `sum` calls `sum(3)`, then the next calls `sum(2)`, then `sum(1)`, then `sum(0)`. When the last call finishes, it returns 0, then the next returns 1, then 3, then 6, then 10. We have computed `sum(4)` as 1 + 2 + 3 + 4 = 10.

```
public int sum (int n) {
   int result;
   if (n == 0) result = 0;
   else result = n + sum (n - 1);
   return result;}
```

Iterative Alternatives

- The non-recursive code below (very similar to the non-recursive `factorial` method) computes the `sum` function with a loop. Is it easier to understand?

- In fact the recursive forms of both `factorial` and `sum` are examples of **tail recursion** -- the method has one parameter and each version of the method calls exactly one other version, at the end of the execution, or is the base case. Any tail recursion can be turned into a non-recursive loop.

- The tradeoff between iteration and recursion is often that recursive algorithms are easier to design and verify, but iterative algorithms are more efficient. We'll see more about this in discussion on Wednesday.

```
public int sum (int n) {
   int x = 0;
   for (int i = 1; i <= n; i++)
      x += i;
   return x;}
```

Indirect Recursion

- If method A calls method B, B calls C, and C calls A again, you have an **indirect recursion**.

- It is still vital that the recursion eventually reaches a base case, and that recursive calls don't proliferate excessively. It would be better if the original call to A led to *one* other call to A at a time, and if some parameter were to change from the first call to the second, moving closer to a base case.

- But if the reader of the program can't tell that recursion is happening at all, they can't carry out this sort of analysis. Code using recursion should be up front about it.

- Example: Zero-parameter constructor calls `this(w)` where `w` is a `String`, then the `String`-parameter constructor calls `this( )`. I tried this, and the compiler caught the "recursive constructor invocation".