

CS231 Fall 2013 Lecture 16

Stephanie R Taylor

October 9, 2013

1 Administrative Topics

- Fill out mid-semester course evaluations.

2 Recursion

One of the most powerful features of modern programming languages is the ability to execute recursive functions. A recursive function is a function that calls itself. The general strategy for a recursive function is to complete some small part of a large task and to use a recursive call to complete the remainder of the task. For example, we can square each number in a list of numbers by squaring the first element, then making a recursive call, passing in the rest of the list. This will create a chain of function calls, each of which will square one element of the list.

The general shape of a recursive method is this:

```
void methodA() {  
    if(<base case>) {  
        <do base thing>;  
        return;  
    }  
    <do something>  
    methodA(); //do recursion  
    <do something>  
}
```

But what is <base **case**>? The base case is the smallest problem the method is designed to solve. The usual way to handle this situation is to pass a parameter to methodA that tells methodA whether to call itself by telling it how much work is left to be done. We use this parameter to determine if we have reached that smallest problem.

Example: Print out the numbers from 1 to 100. How can we do it? One way is to use a for or while loop. But another way is as follows using recursion. We need to keep track of how much work is left to be done after each call, so we pass two parameters indicating the range of numbers to be printed. When that range is just one number, then we can print it and no recursion needs to be done.

```

/** Print the numbers from start to end
 * It assumes start <= end
 * @param start, first number
 * @param end, last number
 */
public static void printNumbers(int start , int end) {
    if (start == end) {
        System.out.println(start);
        return;
    }
    System.out.println(start);
    printNumbers(start+1, end);
}

```

If the user executes printNumbers(1,5), the numbers 1,2,3,4,5 are printed (one per line).

[Do a recursive trace diagram for this using start = 1 and end = 5].

Note the assumption that start <= end. What if we called printNumbers(1,0)? See if you can figure out what would happen. [give them time]. Show trace diagram.

Another example: Adding numbers in an array of integers. Once again, to do it recursively, we need a method that takes at least one parameter telling it whether there is a need to call itself recursively. In this case, like the previous one, we'll pass two extra parameters, indicating the range of values to be added. [Give the method's signature now.] To get the sum of the first 10 integers in the array A, the user would call addUp(A, 0, 9). Try to implement it yourself first. Remember: It needs to return the sum.

```

/** Return the sum of the number from A[start] to A[end]
* It assumes that start <= b.
* @param A, array whose elements will be summed
* @param start, starting index
* @param b, stopping index
*/
public static int addUp(int [] A, int start, int b) {
    if(start == b)
        return A[start];
    else {
        int rest = addUp(A, start+1, b); //compute the sum of the rest
        return A[start] + rest; //add the two values to get the result
    }
}

```

[Draw a picture similar to the one above with methodA, methodB, and methodC.] [Also draw a recursive trace for an array with 3 values] [Also, discuss stackframes and draw stack frames as step through the recursive trace]

Now, what if we want a method addUp(A) that takes only one parameter and adds all the values in A and returns it? [Note: It is okay in Java to create two methods with the same name, as long as the parameter types or order or number are different.] Now we only have one parameter so how do we take care of infinite recursion? [One way: Create a new smaller array for each recursive call. A better way: implement addUp(A) by calling addUp(A, 0, A.length-1);]

Summary Note 1: Write your method so that it uses parameters to tell how much work there is to do, so that, by changing one or more parameters, you get a case with less work to do.

Note 2: Be sure that there is some base case that is always reached in which the method does not call itself. Otherwise, you could end up with infinite recursion.

Note 3: You may find that recursion is a little confusing at first, but once you get used to it, you will see that it is a wonderful technique for repetition. In fact, as you'll see later in this course when we talk about binary trees and as you'll see if you take any further CS courses, in some situations it is very hard to do the repetition without recursion.

Natural places where recursion occurs:

- nested Russian dolls
- Fractals
- Expressions (to compute sum of 2 expressions, first compute the two expressions and then add)
- Product rule for derivatives
- Traversing a directory structure
- Traversing XML or HTML documents
- All permutations of 1..n
- Powerset of 1,...,n
- Traversing a maze (Can I get there from here? I can if I can get there from the spot next to me.)
- When youngsters try to describe an iterative process verbally and informally it is often done recursively. "Grab a piece of wood, chop it into matchsticks, and do the same thing to the rest of the pieces of wood."
- Factorial and Fibonacci and Towers of Hanoi are not good examples

3 Recursion in linked lists

One place where recursion can be very helpful is performing operations in certain abstract data types. You will find recursion almost necessary to do anything with trees. But it also lends itself naturally to linked lists. If we have time, we will write the following two methods in our StringLinkedList class, using recursion. If we don't have time, we will do so on Friday.

- Write a length method
- Write a contains method