RECURSION AND LISTS

Recursion is a useful concept for both procedures and data. You may already be familiar with recursive procedures, but we'll review the concept in this chapter to make sure. Then we'll see that recursion is equally useful when talking about data: In particular, we'll look at one important category of recursive data, the *linked list*.

Recursive procedure

A recursive procedure is a procedure that sometimes calls itself to accomplish its task. One classical example of a recursive procedure is one for computing the *n*th Fibonacci number. The Fibonacci sequence starts as:

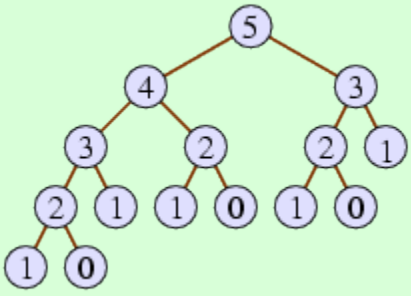
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ….

Each Fibonacci number fib_n is the sum of the two preceding numbers, $fib_{n-1} + fib_{n-2}$, starting with $fib_0 = 0$ and $fib_1 = 1$. We can easily translate this definition into a recursive method.

```
static int fibonacci(int n) {
    if(n <= 1) return n;
    else       return fibonacci(n - 1) + fibonacci(n - 2);
}
```

In talking about recursive procedures such as this, it's useful to be able to diagram the various method calls performed. We'll do this using a recursion tree. (Trees generally are quite important to computer science; Chapter 5 is devoted entirely to them.) The recursion tree for computing `fibonacci(5)` would be as follows.

Figure 2.1: Recursion tree for computing `fibonacci(5)`.



The recursion tree has the original parameter (5 in this case) at the top, representing the original call to the method. In the case of `fibonacci(5)`, there would be two recursive calls, to `fibonacci(4)` and `fibonacci(3)`, so we include 4 and 3 in our diagram below 5 and draw a line connecting them. Of course, `fibonacci(4)` has two recursive calls itself, diagrammed in the recursion tree, as does `fibonacci(3)`. The complete diagram depicts all the recursive calls to `fibonacci` made in the course of computing `fibonacci(5)`. The bottom of the recursion tree depicts those cases when there are no recursive calls — in this case, when `n <= 1`.

Note that every recursive procedure must have at least one condition where there are no recursive calls. Such a condition is called a base case. The `fibonacci` method has one base case, when `n` is less than or equal to 1. Without a base case, the method will never terminate: It will call itself infinitely. With Java, this normally overflows memory, and the program ends up terminating with a *StackOverflowException*.

Though Fibonacci computation is a classical example of recursion, it has a major shortcoming: It's not a compelling example. There are two reasons for this. First, how often do you expect to want to compute Fibonacci numbers? (The Fibonacci sequence is admittedly useful occassionally for analyzing phenomena, but even those cases rarely require computing large Fibonacci numbers.) And second, the above recursive method isn't a good technique for doing it anyway. In fact, if you measure the speed by the number of additions performed, the recursive technique above will take $fib_n - 1$ additions; to see this, you can take the above recursion tree and notice that the overall return value is computed as

$$(((1 + 1) + 1) + (1 + 1)) + ((1 + 1) + 1) .$$

Essentially, we are summing fib_n 1's, which will require $fib_n - 1$ additions. A much faster way is to start with the first two Fibonaccis and to extend the sequence one by one, each time adding the previous two numbers, until we reach the nth Fibonacci. Computing each Fibonacci requires just one addition, so the total number of additions is $n - 1$, which is much less than $fib_n - 1$ for large n.

So let's look at a more compelling example: Suppose we want to list all the subsets of elements in a list of strings named *`elements`*. (Okay, I know what you're thinking: Why would I ever want to list all the subsets of some set? We wouldn't, but the same basic method works any time that we want to search through all subsets. Suppose that we have a list of valuables and their weights, and we want to choose those that fit into a bag without overloading it. We can figure this out using a program that goes through all subsets of the valuables. So, next time that you rob a jewelry store, keep this method in mind.)

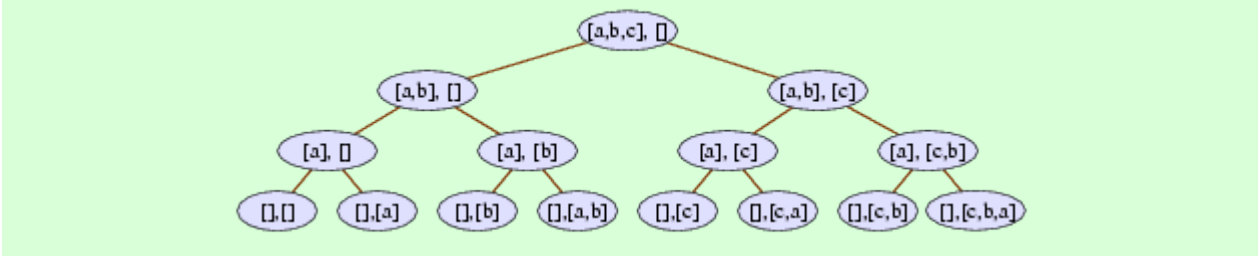
```
static void printSubsets(List<String> elements, List<String> current) {
    if(elements.size() == 0) {
        for(int i = 0; i < current.size(); i++) {
```

```
            System.out.print(" " + current.get(i));
        }
        System.out.println();
    } else {
        String x = elements.remove(elements.size() - 1);
        printSubsets(elements, current); // print subsets without x
        current.add(x);
        printSubsets(elements, current); // print subsets containing x
        elements.add(x);                 // restore current and elements
        current.remove(current.size() - 1);       // to what they were
    }
}
```

We can read this as follows: If `elements` is empty, we simply display the elements of `current`. But if it isn't empty, then we'll remove some element *x* from `elements`, print all the subsets of `elements` following its removal, then add *x* into `current` and print all the subsets of `elements` again, but this time with *x* printed out among the others. Finally, we restore`elements` and `current` to where they were previous to entering the recursive call, by adding *x* back to `elements` and removing *x* from `current`.

A recursion tree can help us to get a handle on how this works. Suppose we called `printSubsets` with `elements` holding three strings, `a`, `b`, and `c`.

Figure 2.2: Recursion tree for computing subsets of [a, b, c].



At the bottom is the case where `elements` is empty and so `current` will be printed to `System.out`. You can see that the `current` parameter goes through all eight subsets of `[a,b,c]`.

Source : http://www.toves.org/books/data/ch02-recur/index.html