RECURSION

Reference:

- Roberts, Eric S., "Thinking Recursively", John Wiley and Sons, 1986.
  An excellent, intuitive book.

The Idea

Recursion isn't hard. You've been doing it your whole life.

For example, how do you look up a word in a dictionary?

```
// Look up w in d.
look up word(Dictionary d, Word w) {

    find w in d

    read the definition of w

    for each word v in the definition
         if you don't understand v, then
              look up word(d, v)

    put your understanding of the words
     together to understand the definition

}
```


Thinking Recursively

Imagine you have to solve one of the problems below, but you are very lazy. You can ask one or more equally lazy friends for help (so you can't ask someone to solve the whole problem — you have to do *some* work!)

Problem: Calculate the value of 13!

Problem: You have a two-pan balance and a pile of 32 quarters. One is counterfeit and weighs slightly less than the others. Find it. How many weighings will it take?

Problem: Given a set of characters, determine all possible permutations of that set.

To calculate the value of 13!

- You can ask a friend what the value of 12! is, (and then later multiply that by 13).
- Your friend can ask another friend what 11! is (and then multiply that by 12).
- ...
- Eventually, someone has to figure out the factorial of 1. This is so simple that they can just answer "it's 1".
- This answer is passed on to the person who needs to figure out 2!. They calculate $2 * 1 = 2$ and pass that answer on to the person who needs to figure out 3!.
- They calculate $3 * 2 = 6$, and pass it on to the person who needs to figure out 4!.
- ...
- Eventually the first person has the value of 13!.

Divide and Conquer

In each case, we

- Repeatedly break down a problem into subproblem(s) that
  - have the same structure as the original problem, but
  - are smaller or simpler to solve.
- Eventually, the subproblem(s) are so simple that they can be solved without further division.
- The solution to the original problem consists of either:
  - the solution to the simplest subproblem, or
  - a combination of the solutions to the solved subproblems.

Question: Identify how `inorderPrint(BSTNode)` follows this form.

The Definition of Recursion

Because a recursive solution to a problem requires that a "smaller" instance of the same problem be solved, methods that are recursive call *themselves*.

Definition: a recursive method is one that is called from within its own body, directly or indirectly.
(E.g. indirectly: method A calls method B which calls method A).

Suspicious?

You may feel "suspicious" about recursion, but don't be: Think of the recursive call as no different from any other method call.

We often examine non-recursive methods, assuming that any other methods they call will "do the right thing".

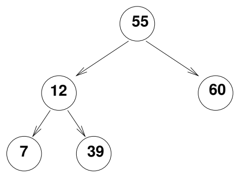
Similarly, for a recursive method, we can consider what it will do *assuming that the recursive call will do the right thing*.

For example, for a non-empty tree, `inorderPrint` prints out the root's value in between printing out the left and right subtrees.

This is correct, assuming that the two recursive calls do the right thing.

Tracing a recursive method

Trace a call to `inorderPrint` with a reference to the root of the following tree. Keep careful track of the call stack.




```
/** Print the tree rooted at t, in order. */
private static void inorderPrint(BSTNode t) {
    if (t != null) {
        inorderPrint(t.left);
        System.out.println(t.key);
        inorderPrint(t.right);
    }
}
```

When you're comfortable with recursion, you'll be able to write, understand, and debug recursive code *without* tracing the recursive calls. But don't hesitate to trace if it helps you!

Writing a Recursive Method

Problem: Compute the height of a given tree.

Get the basic strategy

1. How can you reduce the problem to one or more simpler sub-problems of the same form?

Flow of information

2. What information determines the subproblem (parameter(s)). What information do you want back about the subproblem (return value).

3. Write the method header.

4. Write a method specification that explains exactly what it will do, in terms of the parameters. Include any necessary preconditions.

Base cases

5. When is the answer so simple that we know it without recursing? What is the answer in these base case(s) (also called "degenerate")? Don't stop short of the really simplest *case!*

6. Describe the answer in the other case(s) in terms of the answer on smaller inputs.

Recursive steps

7. Write code for the base case(s).

8. Write code for the recursive case(s).

9. Simplify if possible.

Conclusion

10. Put it all together.

Recursing vs Iterating

Iterating has some similarity to recursing: each iteration of a loop does a bit of the work, and the remaining iterations finish the job.

What's missing is the "coming back" that we can do easily with recursion.

An Example

Consider the task of printing out the objects from an `Iterator` *in reverse order*.

When we get an object from the `Iterator`, it needs to be printed *after* the ones we haven't got yet. Recursion handles this easily:

- if there's a next object:
  - remember it
  - ask for the remaining objects to be printed in reverse order
  - print our remembered object

Write the method:

Another Example: The Fibonacci Sequence

This Fibonacci sequence is:

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

Each number (except the first two) is the sum of the two numbers before it.

We can implement a function to calculate the nth Fibonacci number recursively ...

Combinatorial Explosion

```
/** Return the nth Fibonacci number f_n,
  * starting with f_1 = f_2 = 1.
  * Requires: n >= 1. */
public static int fib(int n) {
    if (n > 2) {
        return fib(n - 1) + fib(n - 2);
    } else {
        return 1;
    }
}
```

Start tracing the calls for `fib(6)`. Do you see any inefficiency?

Iterative Fibonacci

A natural (and faster) iterative version is:

```
public static int fib(int n) {
    if (n == 1) {
        return 1;
    } // else n >= 2.
    int f1 = 1;
    int f2 = 1;
    // f1, f2 are i-1st and ith fibonacci #s.
    for (int i = 2; i != n; ++i) {
        int nextf = f1 + f2;
        f1 = f2;
        f2 = nextf;
    }
    return f2;
}
```

Let's mimic this recursively. Consider:

1. Each iteration gets (is "passed") information via the local variables. Let's make a method header reflecting this:

```
/** n is the index of the desired fibonacci #.
  * f1 and f2 are the i-1st and th fibonacci #s. */
public static int it(int n, int i, int f1, int f2) {
```


2. When does the iteration stop, and what does it do then? This is the base case.

```
    if (i == n) {
        return f2;
    }
```

3. What does the iteration do otherwise?

```
    // Calculate next fibonacci #.
    int nextf = f1 + f2;
    // Pass information to next 'iteration'.
    it(n, ++i, f2, nextf); // will be changed below.
```

4. Deal with the recursion "coming back".

The base case – the last call to `it` – returns a value to the 2nd last call to `it`; at the moment we're throwing away this value.

But we'd like the value to get all the way back to the first caller of `it`. So we have them all return the value they get (this is why we made the return type `int`).

```
    // Pass information to next 'iteration'
    // and return the result.
    return it(n, ++i, f2, nextf);
}
```


The Revised Recursive Fibonacci

The method does the work of the loop in the iterative version. We use it as a helper method:

```
public static int fib(int n) {
    if (n == 1) {
        return 1;
    } // else n >= 2.
    int f1 = 1;
    int f2 = 1;
    return it(n, 2, f1, f2);
}
```

Exercise: Remove all uses of local variables (but not parameters!) from this `fib` and its helper method `it`.

The method is a good example of passing on information during recursion. We do some work (revising `f1` and `f2`) and pass that work on. We also tell the next call where we are (`i`) and something about our goal (`n`).

Such helper methods are common in recursion, though they don't always need to pass so much information.

Hints for Writing Recursive Code

- Start by thinking about how a friend can help by solving a simpler sub-problem.
- Use the 10 questions above to guide.
- Aim for as few base cases as possible.
- Be careful about combining loops and recursion in the same portion of code.
  It *can* be legitimate, but is often a sign of not believing recursion works.
- Design your recursive methods so that all communication is via parameters — don't use globals.
  Doing so is often a sign of trying to work *around* the recursion, rather than with it.
- Remember to think of the parameters as specifying smaller and smaller problems as the method recurses.
- Watch out for "infinite regress".
  In order to stop, a recursive method must eventually execute a non-recursive case.

Non-static Recursive Methods

So far we've seen recursive methods that are static only.

This means that:

- They don't refer to any instance variables.
- We call them via a class rather than an instance. Example:

```
BSTNode r = new BSTNode();
... Build a binary search tree with root r ...
BSTNode.inorderPrint(r);
```

What if instead we want to say "Hey r, you're a root, print your tree," or in Java:

```
r.inorderPrint();
```

Then we'd make `inorderPrint()` an instance method.

How would that look?

A non-static print method

```
class BSTNode {
    public Comparable key;
    public BSTNode left;
    public BSTNode right;

    // Print the tree rooted at me, in order.
    public void inorderPrint() {
        if (left != null) left.inorderPrint();
        System.out.println(key.toString());
        if (right != null) right.inorderPrint();
    }
}
```

The `BSTNode` parameter is now implicit, as "`this`".

We can't call `inorderPrint` on a null reference. (With the static version, we could.)

So inside the method, each recursive call needs an `if` around it. And so might the initial call to `inorderPrint`. The printing method and a call to it come out slightly more awkward.

Exercise: Write `height` in this style.