RECURSION

Reference:

- Shaffer, section 2.4
- You should read chapters 1 and 2 in full

Reference:

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

Thinking Recursively

Imagine that you have to solve one of the problems below, but you are very lazy. You can ask one or more friends for help.
(But 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.

In each case, we

- Repeatedly break down a problem into subproblem(s) that
  - are somewhat smaller or simpler to solve, and
  - have identical structure to the original problem.
- 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.

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., method A may call method B which calls method A).

Examples:

```
int fact(int n) {
   if (n == 0)
      return 1;
   else
      return n * fact(n-1);
}

void BST::print(IntNode* root){
    if (root != 0) {
        print(root->left);
        cout << root->data << " ";
        print(root->right);
    }
}
```

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".

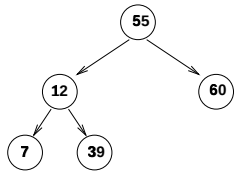
Similarly, for a recursion method, we can consider what it will do *assuming that the recursive call will do the right thing*.

For instance, given a non-empty tree, `BST::print` 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 `BST::print` with a reference to the root of the following tree. Keep careful track of the call stack.




```
void BST::print(IntNode* root){
    if (root != 0) {
        print(root->left);
        cout << root->data << " ";
        print(root->right);
    }
}
```

(When you are comfortable with recursion, you will be able to write, understand, and debug recursive code *without* tracing the recursive calls.)

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 is needed as input and output for the recursion?

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 the base case(s) (also called "degenerate")?

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

Recursive steps

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

8. Simplify if possible.

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

Conclusion

10. Put it all together.

Recursion vs Iteration

Any problem that can be solved iteratively can be solved recursively, and vice versa. So why use recursion?

- Some problems have only a complicated iterative solution, but a simple, elegant recursive solution.
  (Example: tree traversal.)
- In such cases, the recursive program is easier to write, understand, debug, and analyze.

With recursion, the compiler / run-time system keeps track of the method calls, thereby hiding bookkeeping details from the programmer.

Although there may be many method calls, a good compiler can handle recursion efficiently.

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 you when writing a recursive method.
- 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.
  This 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 degenerate (non-recursive) case.