C M S C 2 1 4
Computer Science II
Fall 2002

Recursive Functions

What's a Recursive Function?

Technically, a *recursive function* is a function that makes a call to itself. To prevent infinite recursion, you need an if-else statement (of some sort) where one branch makes a recursive call, and the other branch does not. The branch without a recursive call is usually the *base case* (base cases do not make recursive calls to the function).

Functions can also be *mutually recursive*. For example, function **f()** can call function **g()** and function **g()** can call function **f()**. This is still considered recursion because a function can eventually call itself. In this case, **f()** indirectly calls itself.

Functions can be *tail-recursive*. In a tail-recursive function, none of the recursive call do additional work after the recursive call is complete (additional work includes printing, etc), except to return the value of the recursive call.

The following is typical of a tail-recursive function return.

```
return rec_func( x, y ) ; // no work after recursive call, just return the value of call
```

The following is NOT tail-recursive.

```
return rec_func( x, y ) + 3 ; // work after recursive call, add 3 to result
```

because once **rec_func** is done, it must add 3, and then return that value. Hence, additional work. Hence, not tail-recursive.

It's common, in tail-recursive functions, to have one of the parameters be pass-by-reference (though this is not necessary), which is used to accumulate the answer. Once the base case is reached, you simply return the parameter value, which has the answer. Thus, tail-recursive functions often require an additional parameter, where non tail-recursive functions do not.

The main drawback with recursion is that it can use **O(n)** space (stack space) when a simple loop may only use **O(1)**. For example, printing an array should only require **O(1)** space in addition to the array. You just need a looping variable, which requires **O(1)** space. The recursive solution uses **O(n)** space.

However, a good compiler can determine if a function is tail-recursive, and internally produce code that runs in **O(1)** space, thus giving you the benefits of recursion (i.e., recursion is "neat" and compact to write) but with the space efficiency of a loop. Languages other than C++ (say, ML) often require tail-recursion be converted to loops by the compiler. Typical C++ compilers generally (I believe) do not make this optimization.

When to Use Recursion?

Some people think recursion is so neat, they would like to use it all the time. In general, recursion should be used when you know the number of recursive calls isn't excessive. That number depends somewhat on how

much memory you have. Usually, a 1000 recursive calls should be fine. Stack sizes can now be several megabytes of memory, which allows recursion to go fairly deep without causing a core dump.

Sometimes recursion is used because it produces a cleaner answer compared to the iterative version. For example, nearly all code written for tree-like structures is recursive. Many sorting algorithms are more naturally written recursively as well.

However, recursive solutions can be VERY inefficient, if you aren't careful. For example, the obvious recursive solution to compute the Nth Fibonacci numbers has exponential running time, even though the loop version runs in **O(n)**. This is one good reason to study algorithms (in CMSC 351)---so you know what the running time of a recursive algorithm is, and decide if that solution is worthwhile.

If it sounds like recursion is always bad, realize that recursion often produces solutions that are very compact (requires few lines of typed code). Code with fewer lines are easier to debug than code with many lines. Writing recursive functions can give you greater confidence that you are coding correctly. Often, writing code that ought to be recursive without recursion (i.e., as loops) produces messy code. You may even have to simulate a stack (recursion uses the program stack behind the scenes) to get the behavior you want.

How to Think Recursively

When you want to write a recursive function, begin by writing its prototype. Define what the recursive function does in English (or whatever language you prefer).

For example, consider the following stand-alone (non-member) function:

```
// Sums first n elements of arr
int sum( int arr[], int n ) ;
```

When you make the call:

```
sum( arr, 10 ) ;
```

You know this is summing the first 10 elements of **arr**.

Writing down the purpose of the function may seem like a trivial step, but it's important. Students often fail to write recursive functions correctly because they forget what the function is trying to do.

Then, repeat the following to yourself several times: *Recursion solves a big problem (of size **n**, say) by solving one or more smaller problems, and using the solutions of the smaller problems, to solve the bigger problem*. For example, suppose you are running for some fund-raising marathon. You have collected a large stack of pledges, but want to know what the total amount that people have pledged to you. This stack is rather large, and you'd rather not do the work yourself.

However, you have many willing friends. You divide the stack of pledges and ask your friend "Could you please add up the dollar amount in pledges in this stack? I've only given you half, so there's half the work to do.". As a sneaky person, you give the other half to another friend, and say the same thing. Once both are done, they will give their answer to you, and you add their results.

Thus, you have broken down the problem into two smaller parts, and asked your friends to do the work.

Now those friends are clever, so they divide the stack into two parts (now each has two stacks with size 1/4) and ask two of their friends. When their friends are done, they return their answer, and the result is summed.

Eventually, there is only a stack of two pledges, and these are given to two friends, and those friends, seeing how silly the problem is now, just tell the first friend the only value on the pledge. There's no need to ask any more friends, because you're down to one pledge (this is the base case).

Thus, recursion is all about breaking a problem down, and solving that, and that smaller problem is solved by breaking it down some more, and trying to solve that. Eventually, you reach an easy solution (the base case), and return the solution.

It's interesting to think about what happens to the stack when this happens. As you make each recursive call, the stack grows larger and larger. When you reach the base case, the recursion is done, and the stack becomes smaller and smaller, as it passes the solution back.

Once you have the prototype written, think about solving the next smaller sized problem. Thus, if the call is:

```
sum( arr, n ) ;
```

what's the next smallest size? What about **n - 1**?

```
sum( arr, n - 1 ) ;
```

Suppose someone gave you the answer to this sum. What would you have? This is where it's important to remember how you defined the function. This would give you the sum of the first **n - 1** elements.

Now that you have this solution (which you can assume), what's needed to solve the entire problem? Well, you haven't added **arr[n - 1]**. So, do that.

Finally, you should deal with the base case, which is the smallest problem (in terms of input size, n) you can solve without any recursive calls. It turns out that

```
sum( arr, 0 ) ;
```

is the smallest input size. While having an array of size 0 may not make sense, it's fine. Just let this value be 0 (since 0, summed to any value, just gives you that value). 0 is the additive identity.

Here's the code:

```
int sum( int arr[], int n )
{
  if ( n == 0 )  // base case
    return 0 ;   // no recursive call
  else
    {
      int small = sum( arr, n - 1 ) ; // solve smaller problem
      // use solution of smaller to solve larger
      return small + arr[ n - 1 ] ;
    }
}
```

So, here are the steps to writing a recursive function.

1. Write a prototype for the recursive function.
2. Write a comment that describes what the function does.
3. Determine the base case (there may be more than one), and its solution(s).
4. Determine what smaller problem (or problems) to solve. If it makes it easier for you to follow, save the solutions to the smaller problems to local variables (e.g., **small** in the **sum()** example).

   ASSUME the recursive call works (similar to inductive hypothesis in CMSC 250), i.e., that it will correctly compute the answer.

5. Use the solutions of the smaller problem to solve the larger problem. (If this is done INCORRECTLY, the solutions of the smaller problems will also be computed incorrectly, thus, the assumption in the previous step will fail).

What Makes Recursion Work

Recursion works only when a problem has a recursive structure. This means that the problem's solution is similar for a large input size as well as a small input size. If reducing the input size causes the solution to look different, then it will be hard to use recursion. However, for the same reason, it can be hard to use loops too. Loops also depend on doing the same thing over and over, but on, say, a different index of an array, or a different node in a linked list.

Also, you must be able to use the solutions of the small problem to help solve the larger one. Again, if the solutions aren't useful to you, then recursion isn't going to be useful.

Fortunately, many problems exhibit this kind of behavior. If you can write it in a loop, there are ways to convert it to recursion (and vice versa).

Misconceptions about Recursion

Suppose function **f()** calls **g()** and then function **g()** calls **h()** which then calls **i()**. What happens when **i()** is done?

As you know, the flow of execution goes back to **h()** and when that's done, it goes back to **g()**, when when **g()** is done, it goes back to **f()**.

This is how the call stack works. Each function call causes some memory to be pushed on a stack. Once the call is done, that memory is popped off the stack, and the flow of control goes back to the function that made the call.

Still, even though most students in 214 have no trouble understanding this idea of a function call and its return, they think that recursion is somehow different. In particular, they think that once the base case is reached, the function is magically done. They think no return occurs, and the previous recursive calls disappear. It doesn't. In this respect, recursive calls are just like regular function calls. In fact, they are not implemented any different (except possibly that they can be optimized) from regular function calls.

For example, if function **f()** calls **f()**, which calls **f()**. Then, when the last call to **f()** is completed (presumably the base case), you return back to the second call to **f()**, and when that is done, it goes back to the original call. This is just like non-recursive function calls.

I call this process winding and unwinding. Each recursive call is like walking up one step. Eventually you reach the top step (the base case). When you're done, you back down one step at a time, until you are back at the bottom step (the original call).

Don't forget that even recursive functions return back, just like non-recursive functions do.

The second misconception about recursion is that the base case is only called when the value passed in as argument is the base case value. Technically, this is not wrong. This is when the base case is run.

However, any recursive call (that doesn't end in infinite recursion), eventually calls the base case. Thus, when you call
sum(arr, 10) the base case is eventually called. Some people think the base case only gets called when you call
sum(arr, 0), and while this is true, **sum(arr, 0)** is eventually called when you call **sum(arr, 10)**.

The lesson? Base case contributes to the solution of *any* recursive call (if it doesn't, you have infinite recursion, or a core dump).

Helper Functions

Recursive functions rely heavily on parameters and return values of functions to do their work. Indeed, the basic unit of a recursive function is a function. This may seem obvious, but it really isn't. If you only use

loops, then theoretically, you can write one large **main()** and never have to write another function again. Loops don't really require functions. Recursion does.

When writing recursive functions, you have to write functions (I suppose one could make **main()** recursive, and let it do all the work, but it's not very elegant). This has one good side-effect. Functions tend to be shorter if written recursively than if written using loops.

However, because recursive functions rely on parameters and return values, what happens if the function you write doesn't have the parameters you need.

For example, consider a singly linked list. You wish to print out all the values in the list. The prototype in the class header file looks like:

```
void print() ;
```

Without any parameters, you can't write this recursively. What do you do? You write a recursive helper function.

```
void printRec( Node * curr ) ; // helper
```

print() calls **printRec(first)**, where **printRec()** is the true recursive function.

Often, you need such helpers so you can make recursive calls. The helpers have the parameters you need to make it work. So, **print()**, while not recursive, makes a call to a helper function which is recursive, and does all the work. **print()** is basically a "bootstrap" function, which calls the recursive helper function with the correct initial value as argument.

Common Mistakes With Helpers

Just because it's useful to write recursive functions with helpers, don't overdo it. The most common mistake when writing helper functions is to use the EXACT SAME PROTOTYPE.

Thus, you have **rec_func** which takes two parameter types, which calls **rec_func_help** which takes the same two parameter types! Every semester, someone does this on an exam (writes a recursive function, which calls a helper with the exact same parameter list), leaving graders to scratch their heads wondering why students are doing this.

Why bother writing a helper when the original function would have been fine? Write helpers only because you need a different parameter list. Otherwise, don't use helper functions.

Avoiding Static/Global Variables

Generally, I put static variables in the same category as global variables, and prefer to avoid them when I can. In particular, using static variables to do recursion can lead to problems. For example, static variables are initialized when the function is initially called for the first time. However, it's hard to re-initialize them afterwards.

Consider

```
int sum ( int arr[], int n )
{
   int result = 0 ;
   if ( n == 0 )
      return result ;
   else {
      result += arr[ n - 1 ] ;
      sum( arr, n - 1 ) ;
   }
```

```
}
```

This should work fine the first time you call it. However, the second time, **result** is not 0, and your answer will be wrong.

Instead, you should always update arguments to the recursive function.

You can "cheat" somewhat by using a reference parameter. This isn't a static variable, nor is it global, though it behaves like a global variable. The "pure" form of recursion avoids those kinds of references (arrays are usually fine because arrays are often not modified in the recursive call).

Nevertheless, you should know how to use reference parameters. Almost always, you need helper recursive functions to use reference parameters. The initial function is not recursive, but declares some local variables and initializes it, then calls the recursive function and passes the variables to the reference parameters.

Converting a Loop to Recursion

Converting a loop to recursion is not as hard as it seems. This is *not* how you would normally write a recursive function. To write a recursive function, you generally want to think "recursively".

As mentioned earlier, thinking recursively means solving a large instance of a problem by solving a smaller instance of the problem, assuming that the smaller instance can be solved, and using that solution to solve the bigger problem.

Nevertheless, it's fairly easy to convert a **for-loop** to recursion, should you choose to do so.

Here's a typical loop.

```
for ( <init> ; <cond> ; <update> )
   <body>
```

This can be converted to two functions: the main function and the helper recursive function.

```
void recFunc() {
   int loopVar = <init>

   recHelperFunc( loopVar );
}

void recHelperFunc( int loopVar ) {
   if ( <cond> ) {
      <body>
      <update>
      recHelperFunc( loopVar ); // recursive call
   }
}
```

The loop has been replaced by an if-statement. The recursive call at the end replaces the loop. Since the looping variable is being updated, you can save some memory by passing the variable by reference.

However, the looping variable *can* be passed by value as well. Sometimes, it's nicer to pass by value, because if you pass by reference, then you must pass something that looks like a variable. Passing an expression (e.g., **i + 1**) is not allowed for a non-const reference, but is permitted for a value parameter.

Let's look at how to convert a simple loop.

```
for ( int i = 0 ; i < n ; i++ )
  cout << i << endl;
```

Suppose that **n** is a variable, then, you need two parameters in the recursive helper function.

```
void recFunc() {
   int i = 0 ;
   recHelperFunc( i, n ) ; // recursive call
}

void recHelperFunc( int i, int n ) {
   if ( i < n ) {  // condition of loop
      cout << i << endl ;
      i++;
      recHelperFunc( i, n ) ; // recursive call
   }
}
```

In general, you should avoid using ++ on variables that are passed as arguments. In particular, don't use post-increment or post-decrement, since, in principle, the update won't occur until AFTER the function call, and at that point, you will cause infinite recursion.

It's better to update the value prior to calling the function, and pass the variable in. Alternately, if you are passing by value, use an expression that is not side-effecting. **i++** has a side effect (it updates **i**), where as **i + 1** is not side effecting (it doesn't update **i**).

Recall that when you pass an argument by value, a copy is created. Thus, every single time **recHelperFunc**(), a new copy of **i** and **n** are created.

Admittedly, this is somewhat of a waste because **n** never changes, but recursion often does this.

"Real" Recursion

We can try to write this function more "recursively". That is, to think about how to solve the smaller problem. Initially, you'd think about what kind of function you want. The goal is to print the numbers from 0 to **n - 1**. So you can imagine having a function to do this:

```
void printIt( int n );
```

The recursive call passes in **n**.

Now, what would be a smaller recurisve call?

```
void printIt( n - 1 );
```

This function would print out 0 up to **n - 2**. If this occurred, what would we do to print the rest? You would print **n - 1**. The base case is when **n == 0**.

So, here's how you would write the code in a more traditional recursive way.

```
void printIt( int n ) {
   if ( n == 0 ) { // base case
      cout << 0 << endl;
   }
   else { // recursive case
      printIt( n - 1 );  // recursive call
      cout << n - 1 << endl;
   }
}
```

Notice that you pass one fewer parameter this way, although it doesn't match up with the loop nearly as well. In general, you would "prefer" to do it this way, because it's more "natural" when writing recursive functions.

Here's an analogy. There are two ways to learn a foreign language. Either learn a foreign language by learning it like native speakers speak it, or learn to translate every word in your native language to the foreign language. The second method works, somewhat, but isn't as "fluent" as the first.

For coding, of course, it doesn't matter which method, as long as it works, but nevertheless, the second method we looked at it more "fluent" since the problem is solved from the usual methodology of solving smaller problems and using it to solve bigger problems.

Occasionally, you must use the loop method to come up with a recursive call, because not all problems are nearly as easy to break down into smaller cases.