CS 315 Data Structures Fall 2008

Lab Exercise # 1 August 29, 2008

Goals of the lab:

- Learn the concept of recursive programming
- Understand common errors that beginning programmers make when writing recursive programs
- Learn to trace recursive programs.

What should be submitted?

A hard-copy containing solutions to Problems 1 to 3.

When should it be submitted?

Before the lab ends at 11:45 AM.

Overview of recursion:

You are perhaps familiar with a function that calls another function. When a function calls itself, we say that it is a *recursive* function. There are more complex forms of recursion where a function f calls g, which in turn calls f. (This is known as mutual recursion.)

Recursion is an important programming technique. It is a natural way to write programs when working on a data structure that is recursively defined. (For example, a linked list is a data structure that has a header node and a pointer to a list.) However, the first few examples we will look at do not involve any recursive data structures. There are many problems for which recursive programs are much easier to write than the ones without using recursion. An example is to generate all the permutations of a given set of symbols. There are also some classes of programs (such as backtracking programs) that are much easier to write using recursion. In this course, recursive programming will be used quite extensively. The text-book introduces recursion in Section 1.3 so you this section is the reading assignment for today. *All the examples discussed in today's lab can be easily rewritten using iteration (for or while loop).* However, the real power of recursion will be clear in the next lab where the corresponding iterative versions are considerably harder than their recursive counterparts.

A simple example involving recursion arises in the computation of the sum 1 + 2 + … + n. Suppose we want to write a function f(n) to compute this sum. Here is the iterative way to perform this computation:

```
int f (int n) {
 int sum = 0;
 for (int i = 1; i <= n; ++i)
   sum += i ;
 return sum;
```

}

We can also compute f(n) recursively by noting that f(n) = f(n – 1) + n. (This follows from the fact that f(n) = 1 + 2 + … + n = (1 + 2 + … + n – 1) + n = f(n – 1) + n.) Thus, we can write the function

```
int f(int n) {
 return f(n – 1) + n;}
```

But this program does not work correctly; in fact, it does not work at all. (Test it.) It is not difficult to see why. Suppose we call this program to compute f(2), say.This in turn will trigger a call to f(1), which will trigger a call to f(0), and to f(-1), f(-2) etc. This process never ends so f(2) never gets computed.

The reason is, of course, that we did not provide a way for the program to terminate. Specifically, the formula f(n) = f(n – 1) + n holds only when n >= 2. f(1) should be computed directly without using the formula. This is called the "base case".

Rule 1: Always make sure that your recursive program terminates by providing an exit through the base case(s).

The correct formula for f(n) in the above case is:

$$f(n) = \begin{cases} f(n-1) + n & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

The correct recursive function to compute f(n) is:

```
int f(int n) {
  if (n == 1) return 1; else
  return f(n – 1) + n;
}
```

The following program is a minor modification of the above program. It seems to provide exit from recursion, but it only works for some inputs.

```
int f(int n) {
  if (n == 1) return 1; else
  if (n == 2) return 2;
  return f(n – 3) + n;
}
```

Exercise 1: For what values of n does the above function work correctly? Answer this question without running the program. (Trace the code with some inputs and observe the pattern.)

The next problem is Exercise 1.5 from the text: Write a recursive function g that returns the number of 1's in the binary representation of an input (nonnegative) integer N.

Thus, g(12) = 2 since binary rep. of 12 is 1100. g(15) = 4 since 15 → 1111 etc.

First note that when a number k is even, f(k) = f(k / 2). The reason is: adding a 0 (at the right-end) of k / 2 (in binary) gives k (in binary) so they have the same number of 1's.

What if k is odd? In this case, look at k – 1. What is connection between f (k) and f(k – 1)? We note that f(k) = f(k – 1) + 1. Why? The reason is as follows: k – 1 (in binary) ends with 0 and we get k (in binary) by changing that 0 to 1. Thus, k in binary has 1 more one than does k – 1.

So, in both cases (of k being odd and even), we can compute f(k) by calling f with a smaller argument.

What is the base case? f(0) = 0.

The actual code is as follows:

```
int f(int k) {
  if (k == 0) return 0;
  if (k % 2 == 0) return f(k/2);
  else return f( k – 1) + 1;
}
```

Run this code and make sure that it computes the outputs correctly.

In all the examples above, we note that the recursive calls involve arguments that are smaller than the original parameter. (For example, in the above program, the original argument is k, and the calls involve arguments k – 1 and k/2 both of which are smaller than k.) This property assures us that the recursion is steadily making progress towards the base case at which point it stops.

Rule 2: The arguments used in recursive calls must be such that progress towards the base case is assured. One way to make sure that this condition holds is to make the argument involved in a recursive call to be strictly smaller than the formal parameter.

In page 9 (Figure 1.3) of the text you see an example of a recursive program that does not terminate because the computation of bad(1)involves bad(1) and this creates an infinite loop.

The best way to trace a simple recursive program is as follows: identify the base cases. Usually you can recognize by a path on a branch (if statement) at the start of the code in which small values of the parameter are handled. Start making a table of the output for small values such as n = 1, 2 etc. Continue simulation for the next larger value. When a recursive call is made, you already know the value of the output from the table you are creating. After you have determined the output for a few inputs this way, you can predict what the function does. When a recursive function has more than one

argument, you should first determine if the recursion involves one argument or more than one. I.e., see which arguments change in the calls. Focus only on the argument that changes. If two arguments change, then the table you construct should be two-dimensional.

In Lecture # 2, we presented a recursive procedure to compute x^n. Our procedure was based on the fact that

$$x^n = \begin{cases} (x^2)^{n/2} & \text{if n is even} \\ x * (x^2)^{(n-1)/2} & \text{if n is odd} \end{cases}$$

We can also implement a recursive computation of x^n based on the slightly different formulas given below:

$$x^n = \begin{cases} (x^{n/2})^2 & \text{if n is even} \\ x * (x^{(n-1)/2})^2 & \text{if n is odd} \end{cases}$$

Exercise 2: Implement a recursive function exp(x, n) based on this recursive formula. Compute 3^{40} using your recursive function.

The next recursive function is related to the above problem.

Trace the following recursive program and answer the following questions:

```
int f (int n) {
  if (n == 1) return 0;
  else if (n%2 == 0) return 1 + f(n/2);
  else return 2 + f((n – 1) / 2);
 }
```

(a) Compute f(1000) by hand.
(b) What is $f(2^k)$?
(c) What is $f(2^k - 1)$?

The next example involves computing the binomial coefficient C(n,m) (also sometimes written as $\binom{n}{m}$). C(n, m) is the number of ways to select m people from a group of n people where the order does not matter. Thus, C(6,3) = 20 since there are twenty ways to choose 3 people from a group of 6.

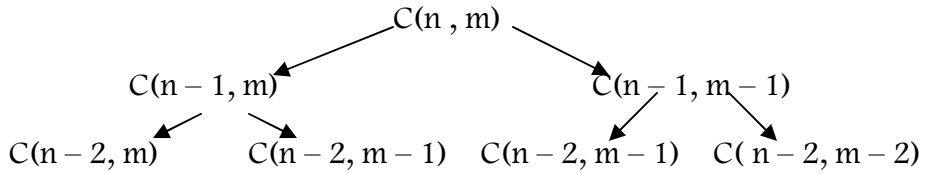
You may recall from CS 242 that C(n, m) = C(n – 1, m) + C(n –1, m –1). (Proof: Look at those selections that include person 1. The number of such selections is C(n – 1, m – 1) because we have already selected person 1 and so we need to choose m – 1 more people from the remaining n – 1. The number of selections that does not include person 1 is C(n – 1, m) by a similar reasoning.) Note the recursive way of thinking that led to this formula. In fact, inductive proofs and recursion are closely related in the sense that an inductive proof can often be converted in a recursive computation. An example of this can be found in page 10 of the text.

Converting the formula *C(n, m) = C(n – 1, m) + C(n –1, m –1)* into a recursive formula seems quite direct. But following rule 1, we need to determine the appropriate exit conditions. Note first of all that if m > n, the answer is 0. (There is no way to select m people from a group of n people if m > n.) So m > n provides one exit condition. So, we can assume that n >= m. In this case, the second call reduces both arguments while the first call reduces only the first argument. In the second call, eventually the second argument will become 0, and *C(x,0)* is 1 for any x. (There is one way to choose 0 people from a group of x people, namely to exclude all of them.) Thus this becomes one exit condition for recursion. The first call reduces the first argument, but not the second. Thus we have the pattern of reduction:

(n, m) → *(n – 1, m)* → *(n – 2, m) …* and this will eventually reach (m, m). So what is C(m, m)? It is easy to see that C(m, m) = 1; the only way to select m out of m is to select all of them. This becomes another exit from recursion. Putting these together, we get the following code for C(n, m):

```
long int C(n, m) {
 if (m > n) return 0; else
 if (m == n || m = 0) return 1;
 else return C(n – 1, m) + C(n – 1, m – 1);
}
```

Although this code is correct, it does not work very efficiently. You can verify this fact by computing C(40, 20). Try it! (The execution will hang for a long time. But even if it completes the execution in a reasonable amount of time, the result will be wrong. This has nothing to do with recursion. This is due to the fact that the numbers involved are too large to be stored in int or even long.) The reason it does not work efficiently is because of redundancy involved. Note the pattern of calls.



Note that the third level the call C(n – 2, m – 1) is being made twice. This kind of redundancy proliferates quickly and this is cause for inefficiency. This is what the book calls the compound interest rule (Rule 4).

Rule 4: Avoid duplicate calls, i.e., a call with a given set of arguments should be made only once.

Exercise 3: Identify redundancy in the recursive computation of Fibonacci numbers. Specifically, how many times a call for F(1) will be made when computing F(10)? Write a recursive function for computing the n-th Fibonacci number that avoids redundancy. Use your function to compute F(50).