CS 315 Data Structures Fall 2013

Lab Assignment # 5, Sept 30, 2013

Goals of the lab:

- Learn the concept of recursion in programming.
- Understand common errors that beginning programmers make when writing recursive programs and learn to avoid them.
- Learn to trace recursive programs.

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 easier to write than the ones without using recursion. An example is to generate all the permutations of a given set of symbols. Another example is an image processing problem that appears in a future lab. There are some classes of programs (such as backtracking programs) that are easier to write using recursion. In this course, recursive programming will be used quite extensively – especially in Chapter 6 on Binary Search Trees, a central topic of this course. The text-book introduces recursion in Chapter 5. Since recursion is a challenging topic, you should read at least Sections 5.1, 5.3 and 5.4 carefully.

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 best way to start learning about recursion is by tracing a recursive program. Consider the following recursive function recf:

```
int recf(int n) {
  if (n == 1 || n == 0) return 1;
 else if (n % 2 == 0)
        return 2* recf(n/2) + 1;
else return recf((n-1)/2) + recf((n+1)/2);
}
```

Exercise 1: What is recf(46)? Calculate this without using the computer.

Our next example is: *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, g(k) = f(k / 2). The reason is: adding a 0 to the binary representation of k / 2 gives the binary representation of k. Thus both k and k/2 have the same number of 1's in binary.

What if k is odd? In this case, look at k – 1 which is an even number. What is relationship between g(k) and g(k – 1)? We claim that g(k) = g(k – 1) + 1. The reason is as follows: k – 1 (in binary) ends with a 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 (k odd and even), we can compute g(k) using recursion.

What about the base case? g(0) = 0.

Putting it all together, we get a recursive function for computing the number of ones in the binary representation of k:

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

Test this code and make sure that it computes the outputs correctly on all the test cases.

In the examples above, we note that the recursive calls involved 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 ensures 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.

The next example involves computing x^n recursively. It is based on the fact that

$x^n = (x^2)^{n/2}$ if n is even
$x * (x^2)^{(n-1)/2}$ if n is odd

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

$x^n = (x^{n/2})^2$ if n is even
$x * (x^{(n-1)/2})^2$ if n is odd

Exercise 2: Implement a recursive function exp(x, n) based on both recursive formulas. Compute 2^{51} using your recursive functions. (That is, run both versions and print the results.) One way to check if your answer is correct is to enter 2^51 is the text box in the web site http://wolframalpha.com

The next recursive function is related to the recursive function g presented above. It calculates the number of multiplications performed by the fast exponential algorithms described above.

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);
  }
```

Exercise 3:

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

The answers to questions 3(b) and 3(c) should be a function of k.

The next example involves computing the binomial coefficient C(n,m) (also 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

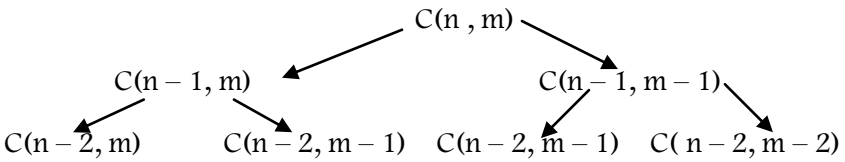
does not matter. Thus, C(6,3) = 20 since there are twenty ways to choose 3 people from a group of 6.

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. Some such examples are presented in Section 5.8 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 long int C(int n, int 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 is not an efficient way to compute C(n, m). You can verify this fact by computing C(100, 50). The reason it does not work efficiently is because of redundancy involved. Note the pattern of calls.



Note that the call C(n – 2, m – 1) is being made twice. This redundancy proliferates quickly and this is the 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 4: Identify redundancy in the recursive computation of Fibonacci numbers using the definition F(n) = F(n – 1) + F(n – 2). Specifically, how many times is a call to F(1) made when computing F(10)? Similarly, compute the number of recursive calls made to compute C(10, 5). You are to solve the problem without using a computer.

The last problem involves implementing the algorithm for computing x where x and n can be as large as a five-digit number. In this case, although x and n can be represented by int, the value x^n can have a lot more than 64 bits - so we will need a linked list representation and the functions we implemented in Lab 4 to solve the next problem.

Exercise 5: Rewrite the function exp(x, n) to compute x^n using the recursive algorithm presented above. (See the formula just above Exercise 2.) Compare the two algorithms by calculating the actual CPU time needed to compute 999^{998} using both the iterative algorithm we implemented in Lab 4 and the recursive version. An important function that is needed in implementing the recursive algorithm is square which takes as input an integer x with no size limit represented as a linked list and returns the list containing x^2. We will discuss the details on how to implement this in the lab.

Exercise 6: The last problem is from the text:

Write a recursive function to add the first *n* terms of the series

$$1+\frac{1}{2}-\frac{1}{3}+\frac{1}{4}-\frac{1}{5}\cdots$$

Run your recursive function for n = 1000 and print the result (as a float).

What should be submitted?

Create a PDF containing solutions to Exercises 1 to 6. For programming problems, include the source code as well as the screen shot of the output for the specified inputs. Make sure to include your name in the text and submit it through Moodle. The lab is due Oct 6, 2013 by mid-night.