Assignment 9

TA C252 / CP-II

Note: If you know recursion you can skip the reading material and start solving the problems given at the end.

RECURSION-A powerful programming technique

Recursion is a powerful programming technique in which complex problems are broken down into simpler problems of the same form, especially when dealing with mathematical functions such as factorialisation that lend themselves naturally to recursion, or with self-similar data structures. Recursion is a great art, enabling programs for which it is easy to verify correctness without sacrificing performance, but it requires the programmer to look at programming in a new light. Imperative programming is often a more natural and intuitive starting place for new programmers which is why most programming introductions focus on imperative languages and methods. But as programs become more complex, recursive programming gives the programmer a better way of organizing code in a way that is both maintainable and logically consistent. The disadvantage to recursion is the amount of memory required to make it work. Do not forget that the program stack grows each time a function call is made. If a function calls itself too many times, your program will run out of memory and crash.

Backtracking is a simple, yet elegant, recursive technique which can be put to a variety of uses. In this article, we will explore this technique in detail, and analyze its usefulness in various applications.

A classic example of recursion

The classic example of recursive programming involves computing factorials. Let us examine a simple function that takes a single integer parameter and returns the factorial of that integer. The factorial of an integer N (written as N!), is defined as N multiplied by all the integers lower than N. Thus 5! is calculated as 5*4*3*2*1. An interesting property of a factorial is that the factorial of a number is equal to the starting number multiplied by the factorial of the number immediately below it. For example, factorial(5) is the same as 5 * factorial(4). You could almost write the factorial function simply as this:

```
int function fact(int n)
{
var retval=1;
for(i=N;i>0;i--){
        retval=retval*i;
        }
return retval;
}
```

The loop is pretty straightforward. *retval* starts off with the value 1, is then multiplied by N, then N-1, and similarly all the integers between N and 1, inclusive. Now, let us write the same function in a slightly different way.

```
int factorial(int n)
{ return n * factorial(n-1);
}
```

The function above is an example of a *recursive* function, because, as you can see in the second 'return' statement, the function calls itself. The term *recur* is from Latin, re means back and currere means to run, to happen again at repeated intervals.

The problem with this function, however, is that it would run forever because there is no place where it stops. The function would continually call factorial. There is nothing to stop it when it hits zero, so it would continue calling factorial on zero and the negative numbers. Therefore, our function needs a condition to tell it when to stop.

Since factorials of numbers less than 1 don't make any sense, we stop at the number 1 and return the factorial of 1 (which is 1). Therefore, the real factorial function will look like this:

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

As you can see, as long as the initial value is above zero, this function will terminate. The stopping point is called the *base case*. A base case is the bottom point of a recursive program where the operation is so trivial as to be able to return an answer directly. All recursive programs must have at least one base case and must guarantee that they will hit one eventually; otherwise the program would run forever or until the program ran out of memory or stack space.

Recursive functions are closely related to inductive definitions of functions in mathematics. In order to evaluate whether an algorithm is a candidate for recursion, we must first try to deduce an inductive definition of the algorithm. For example, the factorial function can be defined inductively in this way :

$$N! = \begin{cases} 1 \text{ if } N=1 \\ N \times (N-1)! \text{ if } N>1 \end{cases}$$

Algorithms that are by nature recursive, like the factorial example above, can be coded either as a loop, as in the first example, or as a recursive function, as in the second example. However recursive functions are generally smaller and more efficient than their looping equivalents.

The Stack

The Stack is an special area of memory in which temporary variables are stored. The Stack acts on the LIFO (Last In First Out) principle. In order to understand how recursive functions use the Stack, we will walk through how the second algorithm above works.

Let us assume we want to find the value of 3!, which is 3 x 2 x 1 = 6. The first time the function is called, N holds the value 3, so the *else* statement is executed. The function knows the value of N, but not of fact(N-1), so it pushes N (value=3) on the stack, and calls itself for the second time with the value 2. This time round too the *else* statement is executed, and N (value=2) is pushed on the stack as the function calls itself for the third time with the value 1. Now the *if* statement is executed as n==1, so the function returns 1. Since the value of fact(1) is now known, it reverts back to it's second execution by popping the last value (2) from the stack and multiplying it by 1. This operation gives the value of fact(2), so the function reverts to it's first execution by popping the next value (3) from the stack, and multiplying it with fact(2), giving the value 6, which is what the function finally returns.

From the above example, we see that

- The function runs 3 times, out of which it calls itself 2 times. The number of times that a function calls itself is known as the *recursive depth* of that function.
- Each time the function calls itself, it stores one or more variables on the Stack. Since the Stack holds a limited amount of memory, functions with a high recursive depth may crash because of non-availability of memory. Such a condition is known as *Stack Overflow*.
- Recursive functions usually have a *terminating condition*. In the above example the function stops calling itself when n==1. If this condition were not present, the function would keep calling itself with the values 3,2,1,0,-1,-2... and so on for infinity. This condition is known as *Endless Recursion*.
- All recursive functions go through 2 distinct phases. The first phase, *Winding*, occurs when the function is calling itself and pushing values on the Stack. The second phase, *Unwinding*, occurs when the function is popping values from the stack.

Basic steps of recursive programs

Every recursive program follows the same basic sequence of steps:

1. Initialize the algorithm. Recursive programs often need a **seed value** to start with. This is accomplished either by using a parameter passed to the function or by providing a gateway function that is nonrecursive but that sets up the seed values for the recursive calculation.
2. Check to see whether the current value(s) being processed match the **base case**. If so, process and return the value.
3. Redefine the answer in terms of a smaller or simpler sub-problem or sub-problems.
4. Run the algorithm on the sub-problem.
5. Combine the results in the formulation of the answer.
6. Return the results.

Using an inductive definition

Sometimes when writing recursive programs, finding the simpler sub-problem can be tricky. Dealing with *inductively-defined data sets*, however, makes finding the sub-problem considerably easier. An inductively-defined data set is a data structure defined in terms of itself -- this is called an *inductive definition*.

For example, linked lists are defined in terms of themselves. A linked list consists of a node structure that contains two members: the data it is holding and a pointer to another node structure (or NULL, to terminate the list). Because the node structure contains a pointer to a node structure within it, it is said to be defined inductively.

With inductive data, it is fairly easy to write recursive procedures. Notice how like our recursive programs, the definition of a linked list also contains a base case -- in this case, the NULL pointer. Since a NULL pointer terminates a list, we can also use the NULL pointer condition as a base case for many of our recursive functions on linked lists.

Linked list example

Let's look at a few examples of recursive functions on linked lists. Suppose we have a list of numbers, and we want to sum them. Let's go through each step of the recursive sequence and identify how it applies to to our summation function:

1. Initialize the algorithm. This algorithm's seed value is the first node to process and is passed as a parameter to the function.
2. Check for the base case. The program needs to check and see if the current node is the NULL list. If so, we return zero because the sum of all members of an empty list is zero.
3. Redefine the answer in terms of a simpler sub-problem. We can define the answer as the sum of the rest of the list plus the contents of the current node. To determine the sum of the rest of the list, we call this function again with the next node.
4. Combine the results. After the recursive call completes, we add the value of the current node to the results of the recursive call.

Here is the c-code and the real code for the function:

```
int sum_list(struct list_node *l)
{
        If (l==NULL)
        return 0;
        return l.data + sum_list(l.next);
}
```

You may be thinking that you know how write this program to perform faster or better without recursion. We will get to the speed and space issues of recursion later on. In the meantime, let's continue our discussion of recursing of inductive data sets.
Suppose we have a list of strings and want to see whether a certain string is contained in that list. The way to break this down into a simpler problem is to look again at the individual nodes.
The sub-problem is this: "Is the search string the same as the one in *this node*?" If so, you have your solution; if not, you are one step closer. What's the base case? There are two:

- If the current node has the string, that's a base case (returning "true").
- If the list is empty, then that's a base case (returning "false").

This program won't always hit the first base case because it won't always have the string being searched for. However, we can be certain that if the program doesn't hit the first base case it will at least hit the second one when it gets to the end of the list.

Comparing loops with recursive functions

Properties	Loops	Recursive functions
Repetition	Execute the same block of code repeatedly to obtain the result; signal their intent to repeat by either finishing the block of code or issuing `acontinue` command.	Execute the same block of code repeatedly to obtain the result; signal their intent to repeat by calling themselves.
Terminating conditions	In order to guarantee that it will terminate, a loop must have one or more conditions that cause it to terminate and it must be guaranteed at some point to hit one of these conditions.	In order to guarantee that it will terminate, a recursive function requires a base case that causes the function to stop recursing.
State	Current state is updated as the loop progresses.	Current state is passed as parameters.

As you can see, recursive functions and loops have quite a bit in common. In fact, loops and recursive functions can be considered interchangeable. The difference is that with recursive functions, you rarely have to modify any variable -- you just pass the new values as parameters to the next function call. This allows you to keep all of the benefits of not having an updateable variable while still having repetitive, stateful behavior.

Practice problems:

```
/*    Program to Print Fibonacci Series using Recursion    */
#include <stdio.h>
int fibbo(int, int, int, int);
main()
{
    int n, f, x=0, y=1, i=3;
printf("\nEnter value of n: ");
```

```
    scanf("%d", &n);
printf("\n%d\t%d", x, y);
    fibbo(x, y, n, i);
getch();
}
fibbo(int x, int y, int n, int i)
{
    int z;
if (i <= n)
{
        z = x + y;
printf("\t%d", z);
        x = y;
y = z;
i++;
fibbo(x,y,n,i);
    }
}
```

/* Program to Find HCF of Two Numbers using Recursion */

```
#include <stdio.h>
int hcf(int, int);
main()
{
        int h, i, a, b;
printf("\nEnter values of two numbers: ");
        scanf("%d %d", &a, &b);
h = hcf(a, b);
printf("\nHCF of numbers is: %d", h);
        getch();
}
int hcf(int a, int b)
{
        if (a%b == 0)
                return b;
        else
                return hcf(b, a%b);
}
```

/* to check a string for anbn using recursion */

```
#include<stdio.h>
#include<string.h>
int compare(char a[],int i,int j);
int main()
{
  char a[10];
  int x,l,yes;
  printf("enter a string\n");
  gets(a);
  l=strlen(a);
  printf("length=%d\n",l);
  x=0;
```

```
  yes=compare(a,x,l-1);
  if(yes)
  printf("string is of given form\n");
  else
  printf("NO\n");


            return 0;
}

int compare(char a[],int i,int j)
{  int k;
            if(i>j)
            k=1;
            else
            {
            if((a[i]=='a')&&(a[j]=='b'))
         k=compare(a,i+1,j-1);


            else k=0;
            }
            return k;
}
```

Resources:
http://www.ibm.com/developerworks/linux/library/l-recurs/index.html

Practice Problem

WAP to implement Mergesort of size n using recursion and also count the number of activation recodes created for merge_sort() function.

WAP to show the disk movement in TOWER OF HANOI problem for 5 disks.

WAP to search for an element in a Binary tree (not Binary Search Tree) using recursion.

–