

Name: _____

CSCI 270

Programming Assignment #4

Summer 2005

Due: Thursday, July 21

Emphasis: Recursion

Part 4.1 – Hand Tracing

Background: A function that calls itself is said to be recursive because it involves a process called *recursion*. In connection with this assignment, you should read Section 10.1 and 10.2 of the course text, which deal with recursion and give several examples. Some problems have a naturally recursive structure, and in those cases recursion is a natural and elegant (but not necessarily efficient) way of expressing the problem. As you will see, recursion has some hidden pitfalls, and so we must be careful when using it.

Example: The *Fibonacci sequence* is one instance of a function with a recursive structure. It gets its name from Leonardo de Piza, whose nickname was Fibonacci. In his book *Liberabaci* (Book of the Abacus) written in 1202, Fibonacci introduced a problem for his readers:

A pair of rabbits is put in a field and if rabbits take one month to become mature and they then produce a new pair each month after that, how many pairs will there be after any given month?

If **r** represents a young pair and **R** a mature pair, the first few generations are:

r → R → Rr → RRr → RRRrr → RRRRRrrr → RRRRRRRRrrrrr

The Fibonacci sequence is made up of the number of rabbits in succeeding generations. It begins

1, 1, 2, 3, 5, 8, 13, 21, ... i.e., it begins with two 1's and each number thereafter is the sum of the two preceding integers.

✍ (1) Calculate the next four numbers in this Fibonacci sequence.

1, 1, 2, 3, 5, 8, 13, 21, _____, _____, _____, _____

A very natural way to define this sequence is with a recursive function like the following:

$\begin{aligned} fib(1) &= 1, fib(2) = 1 \\ \text{For } n > 2, fib(n) &= fib(n-2) + fib(n-1) \end{aligned}$

The basic pattern for such recursive definitions consists of two parts:

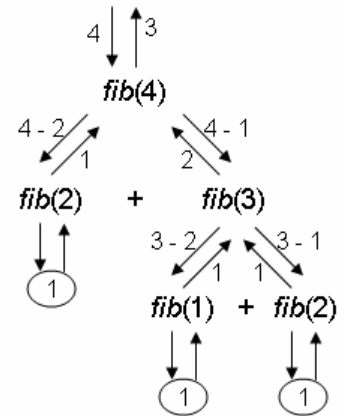
- A **base** (or **anchor**) **case** that specifies the function's value for one or more arguments. In this definition it is the first line: $fib(1) = 1, fib(2) = 1$.
- An **inductive case** that specifies how the value of the function for the current parameter is to be computed in terms of other parameter values that are "closer" to the base case. In this definition it is for


$n > 2$. Note that this inductive case is interesting in that it contains two recursive calls, unlike the factorial and power examples in the text that have just one. This situation is commonly called *double recursion*.

Thus $fib(1)$ gives the value of the first element of the Fibonacci sequence, 1, and $fib(2)$ gives the value of the second element in the sequence, 1. The definition can be expanded recursively; for example,

$$\begin{aligned}
 fib(3) &= fib(3 - 2) + fib(3 - 1) && // \text{by the inductive case} \\
 &= fib(1) + fib(2) \\
 &= 1 + 1 && // \text{by the base case} \\
 &= 2
 \end{aligned}$$

We can recursively expand the definition for $fib(4)$ in a similar way. But a convenient way to picture this repeated application of the definition is as going down the “tree of recursive calls” as pictured at the right, until you hit the base case (shown as ovals in the diagram) and then unwinding back up the tree, evaluating the pending calculations to obtain the value 3 for $fib(4)$.



 (2) Draw a similar “recursion tree” to picture how $fib(5)$ is computed recursively:

When writing recursive functions, you must remember to observe three key things:

1. You must code the base case(s).
2. You must code the inductive case.
3. You must ensure that the recursive calls make progress toward the base case.


In the general case, your algorithm will look something like this:

```
if (base-case)  
    return certain specified value  
else  
    make the recursive call(s) to the function
```

- ✍ (3) Following this general approach, write an algorithm for a recursive version of the Fibonacci function below. Think carefully about it.


My Fibonacci Algorithm (1st Attempt)

Note: In what follows you will use the program `rectester.cpp` to test your recursive function. Get a copy from the class web page under the programming assignments section ([link](#)).


 (4) In the space indicated in `rectester.cpp` write a recursive function called `recFibonacci()` that implements your algorithm. Compile the program and execute it with the base cases (i.e. 1 or 2) only. What does the function return for these cases?

For $n = 1$ `recFibonacci(n)` returns _____
For $n = 2$ `recFibonacci(n)` returns _____

If you run into problems, fix them first.
The base cases should be the easiest
because they don't require recursive calls.


 (5) Now test the recursive part of your code by executing the program with values for n of 3, 4, 6, 6, and 7 and record the results here:

Input	3	4	5	6	7
Result					

 (6) When you are convinced that your function is working correctly, execute the program with the following inputs and record the outputs here.

Input	9	10	15	25	30	35
Result						

You may have noticed that the computations were taking longer and longer. You probably shouldn't try for the 50th or 100th Fibonacci numbers. You can try some larger inputs than those in the preceding table – for example, 40 – if you're prepared to wait a while. We'll discuss the difficulty here later.

 (7) WATCH OUT – What do you think would happen if you removed the base cases from `recFibonacci()`? (You could just comment them out.)

Try it now and see. (Note: Try `Ctrl + C` or `Ctrl + Z` for runaway executions.)
Did what you expect would happen actually happen? _____

Tracing Recursive Functions – Hand Tracing

Tracing is an important skill to develop as a programmer. It involves examining what a program segment does, step by step. This is especially difficult for recursive functions because they call themselves and you may encounter difficulties far from where the problem actually occurs. Moreover, with recursion, evaluation of the function must be postponed and pending function calls must be put on a stack, which rapidly becomes hard to visualize.

You will understand recursion better by tracing some examples. This can be either *hand tracing* – a form of *desk checking* – or *machine tracing*, in which key items of information are output while the function is

executing. In this lab exercise you will do hand tracing first using indentation and alignment to help keep track of the various recursive calls and then tracing the contents of the run-time stack.

✍️ (8) Consider the function shown at the right:

```
int f(int n)
{
    if (n < 2)           // 2
        return 0;       // 3
    else
        return 1 + f(n/2); // 4
}
```

Suppose this function is called by

`x = f(4); // 1`

The following table traces the execution of this function call. The indentation and alignment of the function calls in the right column is intended to show how deep we are into the recursion and which function call we are currently dealing with. The statement numbers in the table come from the inline comments appended to the statements.

Statements Being Executed	ACTION	Current Function
1	Call function f with argument 4	f(4)
2, 4	Inductive step: Call f with argument $4 / 2 = 2$	f(2)
2, 4	Inductive step: Call f with argument $2 / 2 = 1$	f(1)
2, 3	Anchor: Return value 0 to previous function call	
4	Calculate $1 + 0 = 1$ and return to the previous function call	f(2)
4	Calculate $1 + 1 = 2$ and return to the previous function call	f(4)
1	Assign 2 to x	

Now you should make a trace table like the one created above for the statement

`x = f(12); // 1`

Statements Being Executed	ACTION	Current Function

✍ (9) In your function `recFibonacci()` use comment “statement numbers” starting with 2 like those used in the function `f()` to number the instructions and then make a trace table like those in step 8 but for the function call:

```
x = recFibonacci(6); // 1
```

Note: It might be a good idea to abbreviate `recFibonacci()` to something like `rF` to save writing.

Statements Being Executed	ACTION	Current Function

Tracing Recursive Functions – Version 2

Learning how to write and use recursive functions can be difficult for beginning programmers. Tracing step by step how they are implemented using a run-time stack as described in section 10.3 of the text is helpful in understanding how the function works and is also a useful tool for debugging the function.

✍ (10) Consider the function $f()$ from step 8 shown at the right:

Suppose this function is called by

$x = f(4); // 1$

```
int f(int n)
{
    if (n < 2)           // 2
        return 0;       // 3
    else
        return 1 + f(n/2); // 4
}
```

The following table traces the execution of this function call; each activation record (a.r.) consists of:

parameter n	value returned by f	return address [the number following //]
----------------	---------------------	---

Function Call	Run-Time Stack	Action
	_____ (empty)	
f(4)	4 ? 1	
f(2)	2 ? 4 4 ? 1	Push a.r. onto stack
f(1)	1 ? 4 2 ? 4 4 ? 1	Push a.r. onto stack
	2 ? 4 4 ? 1	Assign 0 to return value of f(1), pop a.r. 1 0 4 from stack and return to instruction 4.
	4 ? 1	Assign $1 + 0 = 1$ to return of f(2), pop a.r. 2 1 4 from stack and return to instruction 4.
	_____ (empty)	Assign $1 + 1 = 2$ to return of f(4), pop a.r. 4 2 1 from stack, return to instruction 1, and assign 2 to x.

Make a trace table like that above for the statement

$x = f(12); // 1$

on the following page.

