Lab 8.1 Recursion—Hand Tracing

Project 8.1 Recursion—Machine Tracing

NOTES TO THE INSTRUCTOR

There are two labs and corresponding projects that deal with the material in Chapter 10 of the text *ADTs, Data Structures, and Problem Solving with C++*, 2E. Their purpose is to review recursion and introduce algorithm complexity and to supplement the presentation of recursion and algorithm complexity in the text. They can be used as two different lab-project assignments, or you might select just one pair to assign or perhaps one from each pair:

- Lab Exercise 8.1: Recursion—Hand Tracing
- Project 8.1: Recursion—Machine Tracing

- Lab Exercise 9.1: Algorithm Efficiency
- Project 9.1: Comparing Algorithms

Recursion may have already been introduced in a first programming course, but it is a programming technique that requires *practice, practice, and more practice*. It is easier to understand once stacks have been studied and can be used to explain how recursive function calls are implemented using a run-time stack. This is the approach in Lab Exercise 8.1. Students first write a recursive Fibonacci function and do some problems with it to ensure that they understand how it works. Then they trace the execution of a simpler recursive function and then their recursive Fibonacci function.

Two methods of hand tracing are described in the lab exercise, so if you prefer that they use only one, you should specify which one:

- Version 1: Tracing execution statement by statement in a table, using indentation and alignment to keep track of the depth of recursion and which function invocation is the current one

- Version 2: Tracing execution by recording which activation records are on the run-time stack for each function call

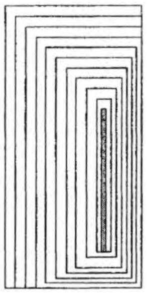
Project 8.1 then describes how to do machine tracing by inserting output statements in the function being traced.

Notes:

1. Lab Exercise 8.1 uses the program `rectester.cpp` to test and trace recursive functions. This file can be downloaded from the website whose URL is given in the preface.

Course Info: ________________ ***Name:*** ________________________________

Lab 8.1 Recursion—Hand Tracing



Background: A function that calls itself is said to be *recursive* because it involves a process called *recursion*. In connection with this lab, you should read Sections 10.1 and 10.2 of the text *ADTs, Data Structures, and Problem Solving*, 2E, 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 Pisa, whose nickname was Fibonacci. In his book *Liber abaci* (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 8 represents a young pair and **8** a mature pair, the first few generations are:

8 → **8** → **8**8 → **88**8 → **888**88 → **88888**888 → **88888888**88888

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.

Calculate the next four numbers in this Fibonacci sequence.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

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

$$\boxed{\begin{array}{l} fib(1) = 1, fib(2) = 1 \\ \text{For } n > 2, fib(n) = fib(n-2) + fib(n-1) \end{array}}$$

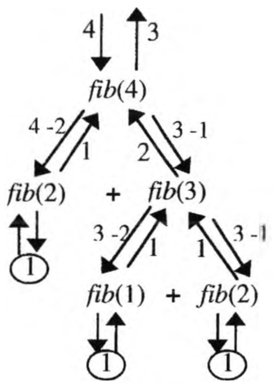
The basic pattern for such recursive definitions consists of two parts:

- 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).



Draw a similar "recursion tree" to picture how *fib*(5) is computed recursively:

See print copy

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
```

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)

if (parameter is less than or equal to 2)
return 1 ;

fib (parameter - 2) + fib(parameter - 1) ;

Note: In what follows you will use the program `rectester.cpp` to test your recursive function. Get a copy using the procedure specified by your instructor. (This file can be downloaded from the website whose URL is given in the preface.)

[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 1

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

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, 5, 6, and 7 and record the results here:

Input	3	4	5	6	7
Result	2	3	5	8	13

[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	20	25	30	35
Result	34	55	610	6765	75025	832040	9227465

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.)

A fatal error occurs by not returning a value

Try it now and see. (*Note: Try Ctrl + C or Ctrl + Z for runaway executions.*)

Did what you expect would happen actually happen? Infinite Recursion

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. Project 8.1 has you do some machine tracing.

✍ [8] Consider the function 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. 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 Call
See print copy		

✍ [9] In your function `recFibonacci()` use commented "statement numbers" starting with 2 like those used in the function `f()` to number the instructions and then make a trace table (continuing on another page if you need more space) 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 Call

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.

✍ $\boxed{10}$ 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 numbers following `//`]

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

Function Call	Run-Time Stack	Action

✍ $\boxed{11}$ In your function `recFibonacci()` with commented "statement numbers" from step 9, make a trace table like those on the preceding page for the function call:

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

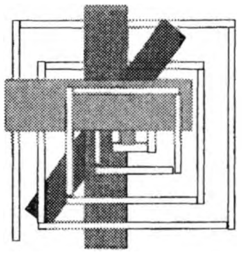
Note: As in step 9, you may abbreviate the function name to `rF` to save some writing.

Function Call	Run-Time Stack	Action

You have finished! Hand in this lab exercise with the answers filled in and attach a copy of `rectester.cpp` that has your prototype and definition of `recFibonacci()` with "commented statement numbers."

Project 8.1 Recursion—Machine-Tracing

Tracing Recursive Functions—Machine Tracing



Hand tracing of the execution of a recursive function as in Lab Exercise 8.1 helps us to better understand the function. It is also helpful in debugging the function. In *machine tracing* we insert pieces of code in the function to report how the function is executing, or we might use a debugger to keep track of how the parameters and other local variables are changing in the recursive calls to the function.

In this project we will use the first approach of adding output statements to the recursive function that will display information similar to that in the trace tables we've done by hand. You will use statements that use indentation to indicate the level of recursion reached for each call and that report when the function is entered and what value it is returning similar to that in Version 1 tracing in Lab 8.1.

$\boxed{1}$ The last part of Lab 8.1 dealt with hand tracing a recursive function. One of the functions was the following function `f()`:

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

Now you are to write statements that produce output like the following:

```
Trace for f(21) is:
-> entering f(21)
  -> entering f(10)
    -> entering f(5)
      -> entering f(2)
        -> entering f(1)
        <- f(1) returns 0
      <- f(2) returns 1
    <- f(5) returns 2
  <- f(10) returns 3
<- f(21) returns 4
```

Here are some ideas about how to do this:

- For the indentation:
  - o At the beginning of the program (by the `#includes`), declare a global integer variable `indent` initialized to –2.
  - o At the beginning of the recursive function that is to be traced, increment `indent` by 2 and then output `indent` spaces followed by a message that the function has been entered and the current values of the parameter(s)—for example, something like "`-> entering f(__)`" but with the blank replaced by the current parameter value(s).

- For the undenting:

  After the function's return value has been calculated and *before* it is returned:

  - o Output a statement indicating that the function is finished and what value it is returning—for example, something like "`<- f(__) returns __`" with the blanks filled in with the parameter value(s) and the return value, respectively.
  - o Decrement `indent` by 2.
  - o Return the function's value.

  Note: You may have to modify your function's `return` statement by separating the computation of the return value from the actual returning of that value; that is, replace a `return` such as

  ```
  return 1 + f(n/2);
  ```

  with

  ```
  int retValue = 1 + f(n/2);
  // Do the outputting and decrementing
  return retValue;
  ```

➜ When you have your indentation approach working for the function `f()`, *print out a copy of your program and execution traces for* `f(1)`, `f(21)`, and `f(1024)` to be handed in.

2 ***Now do it for Fibonacci***

Once you have your indentation approach working for the `f()` function, you should do the same thing as in step 1 with the recursive function `recFibonacci()` that you added to `rectester.cpp` in Lab 8.1.

➜ *Print out a copy of your program and execution traces for* `recFibonacci(1)`, `recFibonacci(4)`, and `recFibonacci(10)` to be handed in.

3 ***Fibonacci—a bad example of recursion***

You will probably find that your output for `recFibonacci(10)` got rather lengthy! Take a close look at it and note all the duplicate function calls—e.g., how many times `recFibonacci(3)` got called.

Now modify your program by adding a global integer variable `numCalls` initialized to 0, and add a statement at the beginning of `recFibonacci()` to increment `numCalls` by 1. Then add a statement in `main()` after the call to `recFibonacci()` to display the value of `numCalls`. Use your program to fill in the table on the grade sheet on the next page.

The results should indicate that recursion is not an appropriate way to compute Fibonacci numbers—especially since it is so easy to do it nonrecursively (see Section 10.4 in the text)!

4 ***Now do it for GCD***

The greatest common divisor problem is described in Exercise 33 of Section 10.1 of the text *ADTs, Data Structures, and Problem Solving with C++*, 2E:

The ***greatest common divisor*** of two integers *a* and *b*, **GCD(*a, b*)**, not both of which are zero, is the largest positive integer that divides both *a* and *b*.

A common method for computing the GCD is known as the *Euclidean Algorithm,* because it was first described by the Greek mathematician Euclid, perhaps better known as the "Father of Geometry":

The ***Euclidean algorithm*** for finding this greatest common divisor of positive integers a and b:

Divide a by b to obtain the integer quotient q and remainder r, so that $a = bq + r$ (if $b = 0$, GCD(a,b) = a). Then GCD(a, b) = GCD(b, r). Replace a with b and b with r and repeat the procedure. Because the remainders are decreasing, eventually a remainder of 0 will result. The last nonzero remainder is the greatest common divisor of a and b.

This method systematically reduces the problem of computing the GCD of two integers to computing the GCD of smaller and smaller integers, until the solution is obvious. Here is an illustration with GCD(2520, 476):

a	b	Quotient q	Remainder r	Comment
2520	476	5	140	Divide 2520 by 476 to get quotient 5 and remainder 140, so 2520= **476** · 5 + **140**. We next compute GCD(476, 140)
476	140	3	56	Dividing 476 by 130 gives quotient 3 and remainder 56, so 476 = **140** · 3 + **56**. We next compute GCD(140, 56)
140	56	2	28	Dividing 140 by 56 gives quotient 2 and remainder 28, so 140 = **56** · 2 + **28**. We next compute GCD(56, 28)
56	28	2	0	Since 28 divides 56 exactly (remainder is 0), their GCD is **28** which is, therefore, the GCD of the original numbers.

The Euclidean algorithm for finding GCD(a, b) is really a recursive one:

Anchor: If b is a divisor of a, return b.

Inductive Step: If $a = b \cdot q + r$ with $r \neq 0$, return GCD(b, r).

For example, the following illustrates how this algorithm computes GCD(2510, 476):

$$\begin{tikzcd}
\underline{\text{GCD(2520, 476)}} \arrow[dr] & & & \\
& \underline{\text{GCD(476, 140)}} \arrow[dr] & & \\
& & \underline{\text{GCD(140, 56)}} \arrow[dr] & \\
& & & \underline{\text{GCD(56, 28)}} \\
& & & \text{Return 28 } (\textit{Anchor}) \arrow[dl] \\
& & \text{Return 28} \arrow[dl] & \\
& \text{Return 28} \arrow[dl] & & \\
\text{Return 28} & & &
\end{tikzcd}$$

Notice that there is something quite different from the two earlier examples of recursive functions: the *value returned by the deepest recursive call is passed back through all of the earlier recursive calls.* This is known as *tail-end recursion,* and it is always possible to write a tail-recursive function nonrecursively, using a loop instead of recursion (see Project 9.1).

Repeat step 2 for a recursive function `gcd()` to compute the greatest common divisor of two integers using the Euclidean Algorithm.

➜ *Print out a copy of your program and execution traces for finding the GCDs of the following pairs of integers:* 512 and 1024; 198 and 1260; 243 and 1024. These are to be handed in.

Course Info: ________________ ***Name:*** ______________________________________

Project 8.1 Grade Sheet

Hand in:

1. This sheet with the following table of function calls to `recFibonacci()` from step 3:

Parameter	1	2	3	4	5	6	7	8	9	10
No. of calls										

Hm-m-m-m... an interesting pattern of numbers. I wonder why that is???

2. The three printouts called for in steps 1, 2, and 4

Attach this grade sheet to your printouts

Category	Points Possible	Points Received
The table of function calls in step 3	20	________
Correctness (including following instructions)		
• The recursive functions	40	________
• The trace outputs	40	________
Program design and structure	20	________
Style (white space, alignment, etc.)	10	________
Total	**130**	________