Armstrong State University
Engineering Studies
MATLAB Marina – Recursion Primer

Prerequisites

The Functions Primer assumes knowledge of the MATLAB IDE, MATLAB help, arithmetic operations, built in functions, scripts, variables, arrays, logic expressions, conditional structures, iteration, and functions. Material on these topics is covered in the MATLAB Marina Introduction to MATLAB module, MATLAB Marina Variables module, MATLAB Marina Arrays module, MATLAB Marina Logic Expressions module, MATLAB Marina Conditional Structures module, MATLAB Marina Iteration module, and MATLAB Marina functions module.

Learning Objectives

1. Be able to recognize recursive function implementations.
2. Be able to describe how recursive functions use a stack.
3. Be able to describe how recursive functions operate.

Terms

Important terms you will encounter in this module include: recursion, stack

MATLAB keywords and functions

function

Recursion

Recursion allows repetition of a code segment by repeated function calls rather than iteration. Recursion (and function calls) requires the use of a stack.

Stack

A stack is a last in first out (LIFO) data structure along with push and pop operations. Stacks are used to store function parameter values, temporary variable values, and the location in memory to return to after a function call. The push operation puts a new object on the top of the stack. The pop operation takes the top object off the stack.

Recursion, due to its repetitive function calls, greatly utilizes the stack and if not careful can result in stack overflow (stack runs out of space to hold things).

Recursive Functions

Recursion involves a function calling a clone of itself. There must be a way for the recursion to terminate otherwise one has endless recursive calls which leads to a stack overflow and termination of the program. Each successive recursive call must move closer to the terminating condition. Recursive solutions to problems can be elegant and intuitive. However, in general non recursive solutions are more efficient in terms of speed and resource utilization.

Recursive functions typically consist of two parts: a base or terminating case and a recursive case. Recursive functions will usually call themselves to solve a smaller version of the same problem, eventually the smaller problems are reduced to the base case, and results are returned back to the calling function eventually reaching the original calling function.

Consider finding the nth Fibonacci number in the Fibonacci series. The Fibonacci series can be defined as follows: f(0) = 0, f(1) = 1, f(n) = f(n-1)+f(n-2), so the Fibonacci series is

n	0	1	2	3	4	5		n
fibonacci(n)	0	1	1	2	3	5		fib(n-1) + fib(n-2)

Table 1, Finbonacci Sequence

The base case is Fibonacci of zero or one, the other cases are defined recursively (n is computed from n-1 and n-2). A recursive solution for determining Fibonacci numbers is
Base case: fibonacci(0) = 0, fibonacci(1) = 1
Recursive Case (n >= 2): fibonacci(n) = fibonacci (n-1) + fibonacci (n-2)

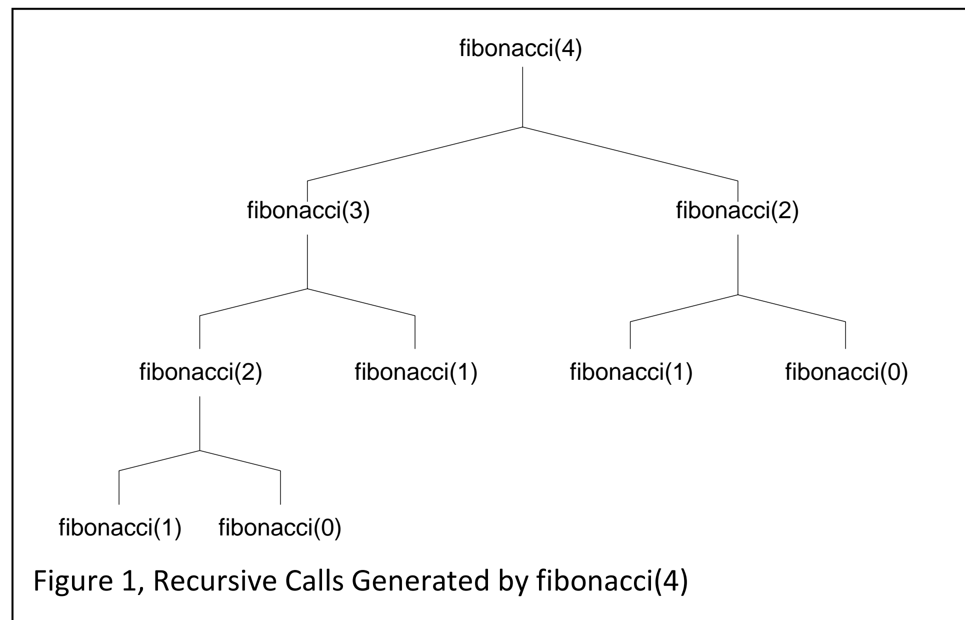

Figure 1, Recursive Calls Generated by fibonacci(4)

Consider a sample call, fibonacci(4), Figure 1 shows the recursive calls generated
Call 1: fibonacci(4) = fibonacci(3) + fibonacci(2)_1
Call 2: fibonacci(3) = fibonacci(2)_2 + fibonacci(1)_1
Call 3: fibonacci(2)_2 = fibonacci(1)_2 + fibonacci(0)_1
Call 4: fibonacci(1)_2 =1, returns value for call 3
Call 5: fibonacci(0)_1 = 0, returns value for call 3
Call 6: fibonacci(2)_1 = fibonacci(1)_3 + fibonacci(0)_2
Call 7: fibonacci(1)_3 = 1, returns value for call 6
Call 8: fibonacci(0)_2 = 0, returns value for call 6

Once the base cases are reached, the results are returned back up the chain until the original call `fibonacci(4)` generates its results. MATLAB functions for recursive and non-recusive (iterative) solutions to the Fibonacci number problem are shown in Figures 2a and 2b.

```
function result = fibonacciRecursive(n)
if (n == 0)
    result = 0;
elseif (n == 1)
    result = 1;
else
    result = fibonacci_rec(n-1) + fibonacci_rec(n-2);
end
end
```

Figure 2a, Recursive Solution to Fibonacci Problem (omitting comments)

```
function fib_n = fibonacciIterative(n)
if (n == 0)
    fib_n  = 0;
elseif (n == 1)
    fib_n  = 1;
else
    fib_n_1 = 1;
    fib_n_2 = 0;
    for k = 2:1:n
        fib_n = fib_n_1 + fib_n_2;
        fib_n_2 =  fib_n_1;
        fib_n_1 =  fib_n;
    end
end
end
```

Figure 2b, Non-Recursive Solution to Fibonacci Problem (omitting comments)

Since Fibonacci(n) requires the two Fibonacci numbers before it, one non-recursive solution to generating Fibonacci numbers involves iteratively generating all the Fibonacci numbers up to and including n
Start with fibonacci(0) and fibonacci (1)
fibonacci (k) = fibonacci (k-1) + fibonacci (k-2), k = 2 to n

Which is more efficient? Recursive functions do not usually require local variables (can save memory). Recursive functions, however, generate additional function calls which require overhead (copies of variables, stack use, transfer of control). So recursive functions are often more inefficient since it takes time to put values on the stack. The iterative solution, however, may be difficult to find or may be confusing.

Recursive Problems

Some examples of recursive problems are:

- Finding the shortest distance between two nodes on a graph
- Determining the folder structure of a computer system
- Tower of Hanoi game
- Reversing text
- Binary searches
- Determining greatest common divisor

Last modified Thursday, October 02, 2014

This work by Thomas Murphy is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.