Editors: Isabel Beichl, isabel.beichl@nist.gov
Julian V. Noble, jvn@virginia.edu

RECURSES!

By Julian V. Noble

THIS COLUMN IS ABOUT RECURSION: FUNCTIONS, SUBROUTINES, AND EVEN WHOLE COMPUTER LANGUAGES DEFINED IN TERMS OF THEMSELVES. RECURSION IS A DIRECT and elegant way to translate certain mathematical relations into programs, and it's a great technique for discovering efficient algorithms. Given its utility, you might wonder why people seldom use it. Here are just some of the reasons why:

- Not all computer languages permit recursion (although most do today). Recursion seems arcane to scientific programmers of my generation who grew up with Fortran, which forbade it. (Fortunately, Fortran 90 and later versions have remedied this.)
- Not all languages that permit recursion make it easy, especially for non-textbook examples (as I recently learned when translating some algorithms to C).
- Although useful, recursion is never essential: by theorem, a recursive algorithm can always be re-expressed non-recursively.[1]
- Recursive programs are generally believed to execute slower or to use more memory than their non-recursive equivalents. Poor examples have reinforced this negative impression; ditto for languages that implement recursion inefficiently.

Correctly used, recursion is so valuable you should use it whenever it makes programs clearer or briefer. In this installment of Computing Prescriptions, I explain when recursion is appropriate and when it is a bad idea. I also show how you might find it useful.

How Recursion Works

Typically, recursive program *A* can call itself or subprogram *B*, which in turns calls program *A* (or *B* could call *C*, which then calls *A*). The first form is called *direct* recursion; the second is *indirect* (see Figure 1).

To understand how a subroutine can call itself, we must first explore how subroutine calls are compiled to machine code. Compilers have several options for compiling argument lists:

- They can reserve space in a memory area assigned to the called subprogram. Calling the subprogram means copying each argument to its new location. Exiting pastes the results back to the memory space in the calling program. This is called *passing by value*.
- A more streamlined method (*pass by reference*) merely passes the arguments' addresses, keeping in memory only one copy of the argument itself.
- The arguments can be pushed onto the machine stack (see Figure 2).

The compiler builds stack-relative pointers into each subprogram that indicate where on the stack its arguments are located. The compiler also installs instructions that reset the stack pointer when the subprogram exits, thereby dropping the arguments from the stack and reclaiming the temporary memory. Neither passing by value nor by reference lends itself to recursion, because these methods store the arguments in fixed memory locations that must be either shared between, or duplicated within, calling and called subprograms.

Self-reference requires the called subprogram's arguments to be physically distinct from the calling program's. Passing arguments in distinct *stack frames* (one for each call) keeps the arguments separate. Calling the recursive function `fib(n)`, for example, pushes n onto the stack; `fib` then tests whether $n \le 0$ or $n = 1$. In these cases, it returns 0 or 1 and terminates, popping the stack as appropriate. If $n > 1$, `fib` pushes $n - 1$ onto the stack and transfers control to the entry point of its own code, thereby calling itself. When it returns the answer from that call, `fib` then pushes $n - 2$ onto the stack and calls itself again. Finally, it adds the second result to the first and exits. Manifestly, using a stack puts the arguments of calling and called subprograms in different places, even when the programs themselves are the same and occupy the same memory.

Inefficient Recursion

The Fibonacci sequence, like the game of golf, offers scope for folly on a grand scale. Its defining relation is

$$F_{n+1} = F_n + F_{n-1}, \tag{1}$$

where ($F_0 = 0$ and $F_1 = 1$) and its direct translation to pseudo code[2] are both recursive:

```
FUNCTION fib(n)
  CASE
    n <= 0 OF RETURN 0 ENDOF
    n  = 1 OF RETURN 1 ENDOF
    n  > 1 OF RETURN fib(n - 1) + fib(n - 2)
                     ENDOF
  ENDCASE
```

Anyone who has tried this knows it's mighty slow compared with the explicit loop

```
FUNCTION fib(n)
  F0 = 0
  F1 = 1
  CASE  n <= 0 OF RETURN F0       ENDOF
        n = 1  OF RETURN F1       ENDOF
        n > 1  OF
          FOR K = 2 TO n
              F = F1 + F0
              F0 = F1
              F1 = F
          NEXT
          RETURN F                ENDOF
  ENDCASE
```

Why is the iterative loop, whose running time is $\mathcal{O}(n)$ (that is, linear in n), so much faster? The time to compute F_n recursively satisfies the linear difference equation

$$T_{n+1} = T_n + T_{n-1}, \tag{2}$$

whose solution has the form.

$$T_n = a\left[\frac{1}{2}\left(\sqrt{5}+1\right)\right]^n + b\left[\frac{1}{2}\left(\sqrt{5}-1\right)\right]^n \tag{3}$$

The first term dominates for large n, hence, the running time increases exponentially with n. (Interestingly, we can compute—recursively—the nth Fibonacci number in $\mathcal{O}(\log_2 n)$ time,[3,4] but space limitations defer that discussion to another column.) From this, we learn recursion is inefficient when it replaces the original problem with two similar problems of nearly the same size as the original, because this leads to exponential growth of the running time.

Caesarian Recursion: Divide and Conquer

Julius Caesar discovered that it was easier to solve a prob-

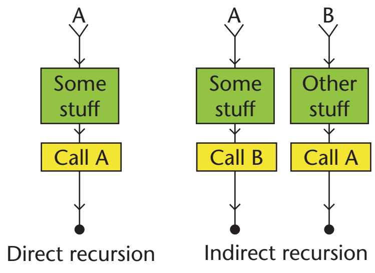

Figure 1. Direct and indirect recursion. The subroutine on the left calls itself—that is, it places parameters on the stack and transfers control to its entry point (labeled *A*). The pair of subroutines on the right call each other—*A* calls *B*, which in turn calls *A*, closing the recursive loop.

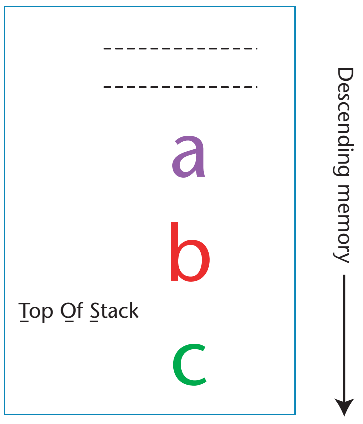

Figure 2. The machine stack. A CPU stack is a contiguous area of memory, organized as a last-in, first-out buffer. Most CPUs extend stacks downward from high memory, to separate them from program and data (which usually extend upward from low memory). The figure shows several successive items placed on the stack, with *c* (top of stack or TOS) the lowest item in physical memory and *b* and *a* in the two next higher memory cells (although we say they are lower on the stack). To push *c* onto the stack, the pointer to TOS is decremented and *c* moves to the new TOS cell; to pop *c* from the stack, *c* moves to its new location and the TOS pointer increments, so that TOS now contains item *b*. To drop *c* from the stack, just increment the TOS pointer.

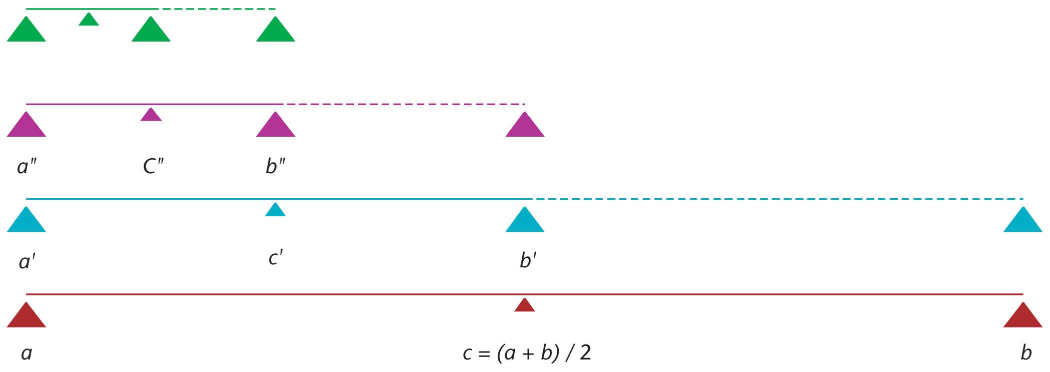

Figure 3. Successive subintervals. Defined by their endpoints (large triangles), these subintervals are passed as arguments in recursive calls of the function `integral`. (Table 1 also displays these subintervals.)

lem of size N by dividing it into two problems of size $N/2$ and tackling them one at a time (rather than by attacking the original problem head-on). If we can combine the solutions to the subproblems in time $\mathcal{O}(N)$, and if the running time of the basic algorithm is a power N^α, $\alpha > 1$, divide and conquer is a winning strategy because $N + 2(N/2)^\alpha < N^\alpha$ if $N > N_{\min} = (1 - 2^{1-a})^{-1/(a-1)}$ (for example, with $\alpha = 2$, $N_{\min} = 2$). These considerations suggest recursion is most efficient when it replaces the original problem with (one or two) similar problems roughly half the original size. That is, we should consider recursion only when dividing (and, one hopes, conquering).

Let's look at some examples. Binary search replaces the original problem with the same problem, only half as big. Usually applied to finding an item in a sorted list, it is also good for locating an isolated root[5] in the interval $L \le x \le U$ via these steps:

- If $f(L) \cdot f(U) > 0$, exit: error message
- If $|L - U| < \varepsilon$, exit: root found
- Let $M = (L + U)/2$
- If $f(M) \cdot f(L) > 0$, let $L = M$; else let $U = M$
- Recurse

At each pass, the interval to be searched halves, so $\mathcal{O}(-\lg\varepsilon)$ steps are needed.

A standard sorting algorithm, MergeSort, helps overworked professors alphabetize piles of graded papers:

- Divide the items into two equal subpiles
- Sort those
- Merge the subpiles (merging requires N operations)

MergeSorting each subpile lets us subdivide them until each pile contains only one item, hence, we can call it sorted. Because MergeSort replaces the original problem with two of half that size, its running time satisfies the relation

$$T_N = 2T_{N/2} + \lambda N, \tag{4}$$

whose solution is $T_N = \lambda N \lg N$, of the same order as QuickSort or HeapSort. A recursive implementation looks like

```
SUBROUTINE MergeSort(list)
  length(list) < 1?
  IF    Error message
  ELSE
        length(list) > 1?
        IF   partition(list, list1, list2)
             MergeSort(list1)
             MergeSort(list2)
             merge(list1, list2, list)
        ENDIF
  ENDIF
```

The venerable Euclidean algorithm for finding the greatest common divisor (gcd) of two integers is also most easily stated recursively:

$$\gcd(a,b) = \begin{cases} a, & b = 0 \\ \gcd(b, a \bmod b), & b > 0 \end{cases} \tag{5}$$

In words, the greatest common divisor of positive integers a and b is the same as that of the integers b and $(a, \bmod b)$.[6] Another important fact is that for integers a and b, integers x,y exist such that

$$\gcd(a,b) = xa + yb. \tag{6}$$

Certain encryption algorithms use x and/or y. Because we can define them recursively,[7]

$$\begin{aligned} d &= \gcd(a,b) = xa + yb = d' \equiv \gcd(b, a \bmod b) \\ &= x'b + y'(a \bmod b) \\ &= x'b + y'\left(a - \left\lceil \frac{a}{b} \right\rceil b\right); \end{aligned} \tag{7}$$

that is,

$$d = d'$$
$$x = y'$$
$$y = x' - \left\lceil \frac{a}{b} \right\rceil y', \tag{8}$$

we can express Equation 6 entirely recursively:

```
FUNCTION xgcd(a,b,x,y)  \ gcd = x*a + y*b
  b=0 ?
  IF     gcd = a
         x = 1
         y = 0
  ELSE   c = [a/b]        \ integer division
         gcd = xgcd(b, a mod b, x', y')
                          \ recurse
         x = y'           \ recurse
         y = x' - c*y'
  END IF
  RETURN gcd
```

I programmed this in Forth, exhibiting the stack at each entry and exit as `xgcd` calls itself:

```
99 78 xgcd
xgcd  [2] 99 78
xgcd  [2] 78 21
xgcd  [2] 21 15
xgcd  [2] 15 6
xgcd  [2] 6 3
xgcd  [2] 3 0
exit  [3] 3 1 0
exit  [3] 3 0 1
exit  [3] 3 1 -2
exit  [3] 3 -2 3
exit  [3] 3 3 -11
exit  [3] 3 -11 14  ok
```

(You can visit www.phys.virginia.edu/classes/551.jvn.fall01/CiSE_progs/Cprogs.html for C and Forth versions of all my examples.) If you lack a Forth compiler but want to experiment, you can choose a public-domain one from the wide selection (listed by CPU and operating system) at www.forth.org/compilers.htm.

Recursive versions of Euclid's algorithm are highly efficient. Although the algorithm can be expressed iteratively,[8] iteration requires more data movement, is less clear, and is often slower.

As a final application of divide and conquer, let's consider adaptive numerical integration. We want to evaluate

$$I = \int_a^b f(x)dx = \sum_{n=1}^{N} f(x_n) w_n + R_N \, , \tag{9}$$

where R_N is the error term, and x_n and w_n are a numerical quadrature formula's points and weights. For fixed absolute precision $|R_N| < \varepsilon$, the most efficient strategy for estimating I concentrates the x_n where $f(x)$ varies most rapidly. This is the adaptive aspect. The justly renowned book *Numerical Recipes*[9] suggests doing this by converting Equation 9 to a differential equation

$$\frac{dF}{dx} = f(x) \, , \tag{10}$$

where $F(a) = 0$. We integrate Equation 10 with a canned adaptive solver. Unfortunately, this approach does not specify the result's absolute precision ab initio. As an alternative, compute I_{L+R} with a standard formula on the entire interval, then I_L and I_R on each half of that interval. If $|I_L + I_R - I_{L+R}| < \varepsilon$, accumulate the result; otherwise, subdivide each half-interval in turn and repeat (see Figure 3).

Here is pseudo code for Simpson's Rule with Richardson extrapolation:[10]

```
FUNCTION simpson (a,b,dummy)
   f1 = dummy(a)
   f2 = dummy((a+b)/2))
   f3 = dummy(b)
   RETURN (f1 + 4*f2 + f3) * (b - a)/3

FUNCTION integral (a,b,eps,dummy)
   c = (a + b) / 2
   OldInt = simpson(a,b,dummy)
   NewInt = simpson(a,c,dummy)
                 + simpson(c,b,dummy)
   ABS(OldInt - NewInt) < eps ?
   IF RETURN NewInt + (NewInt - OldInt)/15
                            \ extrapolate
   ELSE RETURN   integral(a,c,dummy,eps/2)
                 + integral(c,b,dummy,
                  eps/2) \ recurse
   ENDIF
```

Table 1. Intermediate output from adaptive recursive integration.

a	b	ε	$\int_a^b dx\sqrt{x}$
0.0	1.953125E–03	1.953125E–06	5.677541E–05
1.953125E–03	3.906250E–03	1.953125E–06	1.052158E–04
3.906250E–03	0.0078125	3.906250E–06	2.975953E–04
0.0078125	0.015625	7.812500E–06	8.417267E–04
0.015625	0.03125	1.562500E–05	2.380762E–03
0.03125	0.0625	3.125000E–05	6.733813E–03
0.0625	0.125	0.0000625	0.0190461
0.125	0.25	0.000125	5.387051E–02
0.25	0.5	0.00025	0.1523688
0.5	1.0	0.0005	0.4309641

$$\int_0^1 dx\sqrt{x} = 0.6666653$$

Table 1 shows the intermediate output obtained from integrating $\sqrt{x}$ from $x = 0$ to $x = 1$ with the precision set to 0.001.

You've probably noticed that the program evaluates the function multiple times at adjoining endpoints. We can fix this by passing such function values as arguments (on the stack), but this might make the stack too deep—and dramatically conclude program execution. Other ways to get around this exist, as we will see in a future column.

Recursive Descent Parsing

Consider the preceding examples, bad and good, as warm-up exercises. Now we can look at how recursion comes into its own. Most modern languages (with the notable exceptions of Lisp and Forth) have built-in support for formula translation. A long time ago, I added this facility to Forth because a formula is self-documenting, whereas its translation to postfix form is as indecipherable as its translation to machine language. (My FORmula TRANslator—a few hundred lines of Forth—resides at www.phys.virginia.edu/classes/551.jvn.fall01/programs.htm.)

Formula translation is an instance of rule-based programming. We can specify a "Forth-tran" formula by rules stated in a specialized format devised by John Backus (one of Fortran's designers) and Peter Naur (one of Algol's designers):

```
"Forth-tran" Rules in BNF (Backus-Naur Format)

NOTATION:
|             -> "or"
+             -> "unlimited repetitions"
Q             -> "empty set"
&             -> + | -
%             -> * 1/

NUMBERS:
fp#           -> {-|Q}{digit.digit+ |.digit
                   digit+} exponent
exponent      -> {dDeE {&|Q} digit {digit|Q}
                   {digit|Q} | Q}

FORMULAS:
assignment    -> id = expression
id            -> name|name {+ Forth}+ -curly
                   braces balance!
name          -> letter {letter|digit}+
arglist       -> (expression {, expression}+)
function      -> name arglist
expression    -> term | expression & term
term          -> factor | term % factor
factor        -> id | fp# | (expr) | f^f |
                   function
```

In everyday language, the line defining `expression` reads "An *expression* consists of either a single *term*; or of an expression, followed by a plus or minus sign, followed by a *term*." Similarly, "A *term* is a *factor*; or a *term*, followed by a multiply or divide sign, followed by a *factor*." A factor can be one of five different things, two of them elementary (a number or a variable) and three more complicated: an expression enclosed in parentheses, a factor raised to the power of another factor, and a function.

I have stated these rules in Backus-Naur format (BNF) to show you how recursive the definitions are. That is, *expression* is defined in terms of *term* and *expression* as shown graphically in Figure 4.

For the subroutine in Figure 4 to work correctly, the function `find` must not find an operator "hidden" in parentheses or in a floating-point number's exponent field. That is, `find` must locate only "exposed" operators. (This is best accomplished by defining `find` as a finite-state machine.) The subroutine `factor` is defined with a `CASE` or `SWITCH` statement:

```
SUBROUTINE factor(beg, end)
 CASE
  id         OF  do_id(beg, end)      ENDOF
  fp#        OF  do_fp(beg, end)      ENDOF
  f^f        OF  do_power(beg, end)   ENDOF
  (expr)     OF  expression(beg+1, end-1)
                                      ENDOF
  function   OF  do_funct(beg, end)   ENDOF
 END CASE
```

We can write the entire program recursively like this so that it generates the appropriate output from the input text. Note especially that because `term` calls `factor`, which then calls `expression`, the recursion is indirect but mirrors the rules precisely.

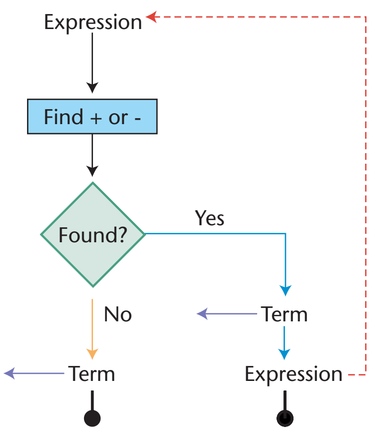

Figure 4. A graphic representation of pseudo code for the subroutine *`expression`*. Starting from the right end of the string, *`expression`* looks for an exposed + or – sign. If it finds one, it breaks the string there and treats it as an expression plus a term; if it finds no exposed sign, it treats the whole string as a term. Thus it embodies both direct recursion (when *`expression`* calls itself), and indirect recursion (when *`expression`* calls *`term`*).

Symbolic manipulation

Because I plan to discuss computer algebra in a future column, I won't dwell on it here. An algebra program uses the same principles as a compiler: it translates statements in one language to another, perhaps performing manipulations during the process. The key again is to work out the rules and implement them. For example, the rules for differentiating a function are

$$\begin{aligned} &\text{diff}\,[\alpha f(x) + \beta g(x)] = \alpha\,\text{diff}\,[f(x)] + \beta\,\text{diff}\,[g(x)] \\ &\text{diff}\,[f(x)\cdot g(x)] = g(x)\,\text{diff}\,f(x) + f(x)\,\text{diff}\,g(x) \\ &\text{diff}\,f(g(x)) = \text{diff}\,g(x)\,\text{diff}\,f(g), \end{aligned} \tag{11}$$

to which we add a library of derivatives of known functions. Many algebraic operations (such as differentiation) are innately recursive, hence, we can translate them directly to a recursive program. I'll also defer those details to a future column.

I hope these illustrations have shown you some of the power and elegance of recursion. To see the difference between recursive and iterative versions of several codes, please check my Web site. Expect lots more recursive applications in future articles.

References

1. R.L. Kruse, *Data Structures and Program Design*, 2nd ed., Prentice-Hall, 1987.
2. B. Einarsson and Y. Shokin, *Fortran 90 for the Fortran 77 Programmer*, version 2.3, 1996, www.nsc.liu.se/~boein/f77to90.
3. E.B. Escott, "Procédé Expéditif pour Calculer un Terme très Éloigné dans la Série de Fibonacci" ("Fast Algorithm for Calculating Remote Terms in the Fibonacci Series"), *L'Intermédiaire des Mathématiciens* (*Mathematician's J.*), vol. 7, 1900, pp. 172–175.
4. J.C.P. Miller and D.J. Spencer Brown, "An Algorithm for Evaluation of Remote Terms in a Linear Recurrence Sequence," *Computer J.*, vol. 9, 1966, pp. 188–190.
5. F.S. Acton, *Numerical Methods that Work*, Mathematical Assoc. America, 1990, p. 179.
6. G. Birkhoff and S. MacLane, *A Survey of Modern Algebra*, 5th ed., A.K. Peters Ltd., 1996.
7. T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.
8. D.E. Knuth, *The Art of Computer Programming*, 3rd ed., Addison Wesley Longman, 1997, p. 13ff.
9. W.H. Press et al., *Numerical Recipes: The Art of Scientific Computing*, Cambridge Univ. Press, 1986.
10. A. Ralston, *A First Course in Numerical Analysis*, McGraw-Hill, 1965, p. 118ff.

Julian Noble is a physics professor at the University of Virginia. His interests are eclectic, both in and out of physics. His teaching philosophy is "no black boxes." Contact him at the Dept. of Physics, Univ. of Virginia, PO Box 400714, Charlottesville, VA 22904-4714; jvn@virginia.edu.