Sections 3.1: Recursive Definitions

February 9, 2025

Abstract

In this section and the next we examine multiple applications of recursive definition and illustrate its usefulness with many examples. Recursion is one of the coolest ideas in the whole world: it has been voted "most likely to land you in an infinite loop", however....

we might say that induction is a recursive method of proof.

1 Recursive Definitions

A **recursive definition** is a close relative of mathematical induction. There are two elements to the definition:

(a) A basis case (or cases) is given, and

(b) an inductive or recursive step describes how to generate additional cases from known ones.

Example: the Factorial function sequence:

(a) $F(0) = 1$, and

(b) $F(n) = nF(n-1)$, $n \geq 1$.

$F(0)=1 \quad = 0!$

$F(1) = 1 \cdot F(0)$

$= 1 \quad = 1!$

Note: This method of defining the Factorial function obviates the need to "explain" that $F(0) = 0! = 1$. For that reason, it's better than defining the Factorial function as "the product of the first n positive integers," which it actually is from $n = 2$ on. Defined as "the product", even $F(1) = 1! = 1$ seems weird!

TABLE 3.1

Recursive Definitions

What Is Being Defined	Characteristics
Recursive sequence	The first one or two values in the sequence are known; later items in the sequence are defined in terms of earlier items.
Recursive set	A few specific items are known to be in the set; other items in the set are built from combinations of items already in the set.
Recursive operation	A "small" case of the operation gives a specific value; other cases of the operation are defined in terms of smaller cases.
Recursive algorithm	For the smallest values of the arguments, the algorithm behavior is known; for larger values of the arguments, the algorithm invokes itself with smaller argument values.

Figure 1: Table 3.1, p. 171. In this section we encounter examples of several different objects which are defined recursively.

- **sequences** – an enumerated list of objects (like factorials)

  Example: Fibonacci numbers - Example 2, p. 159 - history, #37, p. 175 – let's have a look at those....)

$$\begin{array}{rl} F(1) & 1 \\ F(2) & 1 \\ F(n) = F(n-2) + F(n-1) & n > 2 \end{array}$$

$F(3) = F(2) + F(1)$
$= 1 + 1 = 2$
$F(4) = 2 + 1 = 3$
$F(5) = 3 + 2 = 5$

I'm very fond of lisp (my variant is called xlisp, and xlispstat). Here is a recursive definition for Fibonaccis, in lisp:

algorithm

```
(defun fib(n)
  (if (not (and (integerp n) (> n 0))) (error "Only natural numbers are allowed"))
  (case n
        ;; the following two cases are the base cases:
        (1 1)
        (2 1)
        ;; and, if we're not in a base case, then we should use recursion.
        ;; This means that function fib actually invokes itself:
        (t (+ (fib (- n 1)) (fib (- n 2))))
        ;; but, because the argument decreases, we'll eventually hit the
        ;; ''basement, or base cases''.
        )
  )
> (fib 5)
5
> (mapcar #'fib (iseq 1 8))
(1 1 2 3 5 8 13 21)
```

Note, however, that this is a horrible way to compute Fibonacci numbers. If you try

(fib 55),

it will first compute (fib 54) and (fib 53).

Then (fib 54) will likewise compute (fib 53) (but we're already scheduled to do that!), and so on. Very wasteful. It will only take us a little while to drive a computer to its knees (if it only had knees...:)

```
> (time (fib 20))
The evaluation took 0.02 seconds; 0.00 seconds in gc.
6765
> (time (fib 30))
The evaluation took 2.85 seconds; 0.05 seconds in gc.
832040
> (time (fib 35))
The evaluation took 31.61 seconds; 0.70 seconds in gc.
9227465
```

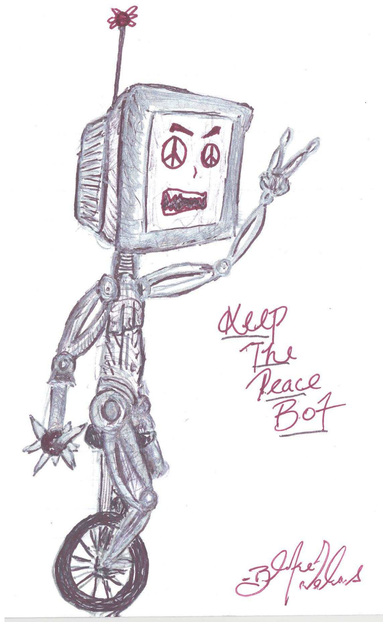

Figure 2: Computers DO have knees! But just two fingers.... Thanks to Blake Nelms, Math for Liberal Arts student.

Upshot: Recursive definitions of functions may be **easy to create or code**, but they may also be tremendously **wasteful**!

Here's a better way: "fibb" produces a *pair* of fibonacci numbers at each calculation by making a *single* call to itself, thus avoiding the needless proliferation of pointless repetitions of "fib":

F(n) & F(n-1)

```
(defun fibb(n)
  (if (not (and (integerp n) (> n 0))) (error "Only natural numbers are allowed"))
  (case n
        (1 '(1 0))
        (2 '(1 1))
        (t (let ((temp (fibb (- n 1))))
             (list (sum temp) (first temp))
             )
           )
        )
    )
  )
> (time (fibb 30))
The evaluation took 0.00 seconds; 0.00 seconds in gc.
(832040 514229)
> (time (fibb 35))
The evaluation took 0.00 seconds; 0.00 seconds in gc.
(9227465 5702887)
```

Note ⊙ case!

add the two previous

retain the previous

We'll be proving various facts about Fibonacci numbers. Pay careful attention to the differences in Examples 3 and 4 (p. 160–): I love mathematics because there's always more than one way to show something – but these examples illustrate why you want to stop and think about strategy before you attempt a proof! Let's take a look....

By the way, Fibonacci numbers appear systematically in Pascal's Triangle, the rows of which are recursively defined.

Example: #32, p. 174 (This example illustrates – like Example 3 – that you sometimes need more than one base case in an induction proof.)

$P(k+1): L(k+1) = F((k+1)+1) + F((k+1)-1)$

32. The *Lucas sequence* is defined by

$$L(1) = 1$$
$$L(2) = 3$$
$$L(n) = L(n-1) + L(n-2) \text{ for } n \geq 2$$

a. Write the first five terms of the sequence.

b. Prove that $\underbrace{L(n) = F(n+1) + F(n-1)}_{P(n)}$ for $n \geq 2$ where F is the Fibonacci sequence.

$P(2): L(2) = 3 \overset{?}{=} F(3) + F(1) = 2 + 1$ ✓

$P(3): L(3) = 4 \overset{?}{=} F(4) + F(2) = 3 + 1$ ✓

Assume $P(r)$ $\forall r$ / $1 \leq r \leq k$ & consider $P(k+1)$:

LHS: $L(k+1) = L(k) + L(k-1)$
$= F(k+1) + F(k-1) + F(k) + F(k-2)$
$= F(k+2) + F(k)$
$= F((k+1)+1) + F((k+1)-1)$ ✓ the RHS of $P(k+1)$

Often we may be able to find a "closed-form" solution to a recurrence relation (in fact, one exists for the Fibonacci sequence). We'll focus on that in the next section.

sets

Example: finite length and palindromic strings - Example 6 and Practice 6 and 7, pp. 163)

EXAMPLE 6 The set of all (finite-length) strings of symbols over a finite alphabet A is denoted by A^*. The recursive definition of A^* is

1. The **empty string** λ (the string with no symbols) belongs to A^*.
2. Any single member of A belongs to A^*.
3. If x and y are strings in A^*, so is xy, the **concatenation** of strings x and y.

Parts 1 and 2 constitute the basis, and part 3 is the recursive step of this definition. Note that for any string x, $x\lambda = \lambda x = x$.

PRACTICE 6 If $x = 1011$ and $y = 001$, write the strings xy, yx, and $yx\lambda x$.

PRACTICE 7 Give a recursive definition for the set of all binary strings that are **palindromes**, strings that read the same forward and backward.

P - the set of Palindromic strings.

(check out Demetri Martin's Palindromic Poem)

Base: $\lambda, 0, 1$ are the base cases

Example: wffs We also used a recursive definition to create the set of all valid wffs: propositions are wffs, and, given two wffs P and Q,

- $P \wedge Q$ and $P \vee Q$,
- $P \to Q$ and $P \longleftrightarrow Q$, and
- P' and Q'

are also wffs. (Notice that there's some redundancy in our definition.)

Inductive: If x & y are palindromes, so is yxy.

- **operations**

  Example: string concatenation - Practice 8, p. 165

  PRACTICE 8 Let x be a string over some alphabet. Give a recursive definition for the operation x^n (concatenation of x with itself n times) for $n \geq 1$. ■

- **algorithms**

  Example: BinarySearch - Practice 10, p. 170. Example #14, p. 170, describes the author's definition of "middle" when you have an even number of elements – it's the top of the left half.

  The list elements are 3, 7, 8, 10, 14, 18, 22, 34.

  PRACTICE 10 In a binary search of the list in Example 14, name the elements against which x is compared if x has the value 8.