Sections 3.1: Recursive Definitions

February 1, 2021

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....

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 $0! = 1$

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

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 is from $n = 2$ on. Defined as "the product", even $F(1) = 1! = 1$ seems weird....

Every natural numbers can be written in a unique way as:

1) A product of prime numbers

2) A sum of distinct powers of two (binary)

3) A sum of non-consecutive Fibonacci numbers

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

- **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....)

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

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

```
(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) ⎭ Anchor (base cases)
        ;; 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))))    * two more calcs
        ;; 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),

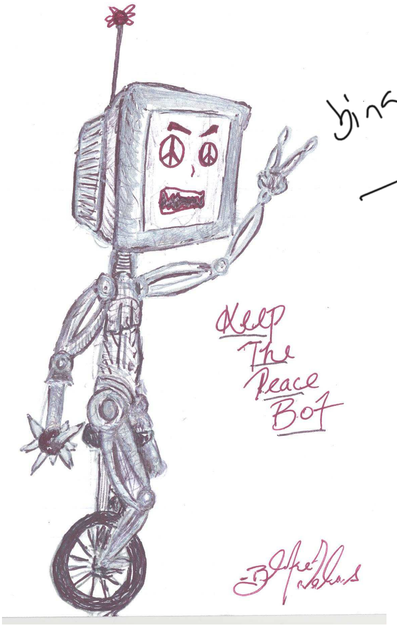
it will first compute (fib 54) and (fib 53).

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
```

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":



binary!

Figure 1: Computers DO have knees! But just two fingers.... Thanks to Blake Nelms, Math for Liberal Arts student.

```
(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)
```

..., -1, 1, 0, 1, 1, 2, 3

We'll be proving various facts about Fibonacci numbers. Pay careful attention to the differences in examples 3 and 4: 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 (of course!).

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

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)

  (check out Demetri Martin's Palindromic Poem)

  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.)

- **operations**

  Example: string concatenation - Practice 8, p. 165

Lucas Numbers:

$L(1) = 1$

$L(2) = 3$

$L(n) = L(n-1) + L(n-2)$ $n \geq 2$

a. 1, 3, 4, 7, 11, 18, 29, ...

b. Prove:

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

for $n \geq 2$

By induction (2nd):

$P(2): L(2) = 3 = 2+1 = F(3) + F(1)$ ✓

$P(3): L(3) = 4 = 3+1 = F(4) + F(2)$ ✓

Consider $P(k+1)$:

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

Start with the LHS:

$$L(k+1) = \underline{L(k)\checkmark + L(k-1)\checkmark}$$

$$= [F(k+1) + F(k-1)] + [F(k-1+1) + F(k-1-1)]$$

$$= [F(k+1) + F(k)] + [F(k-1) + F(k-2)]$$

$$= F((k+1)+1) + F((k-1)+1)$$

$$= F((k+1)+1) + F((k+1)-1) \checkmark$$

- **algorithms**

Example: BinarySearch - Practice 10, p. 170

Check out Example #14, p. 170, for the author's definition of "middle" when you have an even number of elements – it's the top of the left half. For sorted lists.

3, 7, 8, 10, 18, 22, 34

11?

18, 22, 34

11?

"stopper"

18

11?

base case