How Many Recursive Calls Does a Recursive Function Make?

John S. Robertson
Department of Mathematics and Computer Science
Georgia College & State University
Milledgeville, Georgia 31061 USA
jroberts@mail.gcsu.edu

Abstract

The calculation of the Fibonacci sequence using recursion gives rise to an interesting question: How many times does a recursive function call itself? This paper presents one way to examine this question using difference equations with initial conditions, or discrete dynamical systems (DDS). We show that there is a linear relationship between the Fibonacci numbers themselves and the number of recursive calls. This relationship generalizes to any type of DDS of second-order, and DDS of higher-order.

Introduction

Sometimes interesting theoretical problems crop in the most unlikely place. Recently I was introducing Computer Science I students to recursive functions. One of the classic problems encountered in this area is the calculation of the n-th Fibonacci number.

The Fibonacci numbers are defined by the following recurrence relation:

$$F(n) = F(n-1) + F(n-2), \quad n = 2, 3, \ldots \qquad (1)$$

We define $F(0) = F(1) = 1$, which are called initial conditions. Together with those conditions, we sometimes call Eq. (1) a discrete dynamical system (DDS) [1]. It is straightforward to implement a solution to this DDS using recursion.

One day, while my class was in the laboratory, one of my students observed that her computer took an inordinate amount of time to calculate F(30) using recursion (the machine was a 100 MHz Pentium) and she was concerned that something was wrong with her code. There wasn't, of course, and I suggested that she modify her program to print out not only the n-th Fibonacci number, but also the number of times the function defining the Fibonacci number was called. She used a global variable to keep track of the calls and we show her code in Figure 1.

```
#include <iostream.h>
long calls;
long F(long n) {
    calls++;
    if (n <= 1)
      return 1;
    else
      return F(n - 1) + F(n - 2);
}

int main() {
    long i, c, d;
    for (i = 0; i <= 30; i++) {
    calls = 0;
    d = F(i);
    c = calls;
    cout << "i = " << i << " F(i) = "
         << d << " calls = " << c << endl;
}
                    return 0;
            }
```

Figure 1. C++ code used to calculate the n-th Fibonacci number and to track the number of recursive made to the defining function.

Table 1 summarizes the output of the program. Quite evidently, the number of calls that function makes to itself grows much faster than n. Further, when n = 30, the program makes over 2.6x106 recursive calls. That is a tall computational order, even for a Pentium Processor.

i	F(i)	calls
0	1	1
1	1	1
2	2	3
3	3	5
4	5	9
5	8	15
6	13	25
7	21	41
8	34	67
9	55	109
10	89	177
11	144	287
12	233	465
13	377	753
14	610	1219
15	987	1973
16	1597	3193
17	2584	5167
18	4181	8361
19	6765	13529
20	10946	21891
21	17711	35421
22	28657	57313
23	46368	92735
24	75025	150049
25	121393	242785
26	196418	392835
27	317811	635621
28	514229	1028457
29	832040	1664079
30	1346269	2692537

Table 1: Summary of output of program given in Figure 1.

Introductory texts usually do not say much about the growth of the number of recursive calls. Although it is not hard to convince a student that the growth increases exponentially (i.e. something like O(2n)), is it possible to *calculate* the exact number of calls for a given n? This is what my student asked me. The answer turns out to be yes, and the result is only slightly more complicated than calculating the n-th Fibonacci number.

Analysis

There are several ways to solve Eq. (1), i.e. to determine an explicit function dependence of the n-th Fibonacci number. The actual solution, which, if fairly complicated, is not important to this discussion, but see [1, p. 212-215] and many other sources for a description of the solution method.

Now to determine the number of recursive calls needed to calculate F(n), we let G(n) be the number of calls made by the recursive program in calculating F(n). Examining the code we see that G(0) = 1 since just one call to the function is needed to compute F(0). Moreover, G(1) = 1 as well. Now, to compute G(n), we will need to make an initial call, and then G(n – 1) calls to calculate F(n – 1) as well as G(n – 2) calls to calculate F(n – 2). In other words,

$$G(n) = G(n - 1) + G(n - 2) + 1 \quad (2)$$

Together with the conditions G(0) = 1 and G(1) = 1, Eq. (2) specifies another DDS. In fact, it is very similar to the DDS in Eq. (1), with the addition of the constant term on the right-hand side. (This is called a constant nonhomogeneous term.) It is possible to solve this DDS using the same method as was used to solve Eq. (1). The exact solution method is rather long and intricate and, in any case, tends to obscure an interesting relationship between G(n) and F(n). It turns out that we can solve Eq. (2) in a much more elegant way.

Noting the form of the DDS, let us suppose that G(n) depends on F(n) in some way; in other words, G(n) is a function of F(n). A good first conjecture would be to assume a linear dependence of G(n) on F(n), so that

$$G(n) = a\,F(n) + b \quad (3)$$

where a and b are constants yet to be determined. Since we know that G(0) = G(1) = 1, while F(0) = F(1) =1, this tells us that 1 = a + b. Then by substituting Eq. (3) into (2) and using the fact F(n) = F(n – 1) + F(n – 2), we obtain that b = –1. Therefore, a = 2 and we have the remarkable result that

$$G(n) = 2\,F(n) - 1 \quad (4)$$

In other words, the number of recursive calls required to compute the n-th Fibonacci number is twice that number less one, which Table 1 confirms. Since the Fibonacci numbers grow exponentially (see [1]), we see that the number of recursive calls grows exponentially as well (ample evidence that recursion is not an efficient way to attack second order DDS!).

Extensions

Our result is even more interesting when we note that G(n) in no way depends on the coefficients of the difference equation in general. In fact, if properly code the recursive function, the same result will hold for any second order DDS—even nonlinear ones! In fact, the number of calls turns out to be an upper bound for an arbitrary second-order DDS, since difference equations of the form

$$H(n) = a\,H(n - 2) \quad (5)$$

will evidently require less computational effort. In fact, in this instance, the expression for the number of calls is simpler:

$$C(n) = C(n - 2) + 1$$

with C(0) = C(1) = 1. The solution is

$$C(n) = 3/4 + (-1)n / 4 + n / 2$$

so that in this case, the number of recursive calls grows linearly.

Higher-Order DDS

Similar results are obtainable for a third order DDS. In this case, we first consider the third-order analog to Eq. (1):

$$T(n) = T(n - 1) + T(n - 2) + T(n - 3)$$

with T(0) = 1, T(1) = 1, and T(2) = 1, and suppose that we code it in a manner analogous to the code given in Figure 1. Then *any* third order DDS computed recursively (assuming no coefficient vanishes) will have a growth function D(n) for which D(0) = 1, D(1) = 1, D(2) = 1, and

$$D(n) = D(n - 1) + D(n - 2) + D(n - 3) + 1,$$

Not surprisingly, there is an intimate relationship between D(n) and T(n). If we again suppose that

$$D(n) = a\,T(n) + b,$$

we find that

$$D(n) = 3\,T(n) / 2 - 1/2$$

In fact, this type of relationship continues as the order of the DDS takes on any value of n.

We see then that a straightforward question asked by a student about a straightforward programming problem, one routinely assigned, contains some quite beautiful mathematics. Although we did not delve the techniques used to solve DDS here, this mathematics is certainly within the grasp of first and second year students. In any case, we hope that this discussion might serve to encourage computer science majors to begin exploring the mathematics that lies at the theoretical foundations of the discipline at an early stage.

Acknowledgment

A. del Portillo provided a number of useful suggestions about this work.

References

[1] D.C. Arney, F.R. Giordano, and J.S. Robertson, *Discrete Dynamical Systems: Mathematics, Methods, and Models*, McGraw-Hill, New York, 1998.