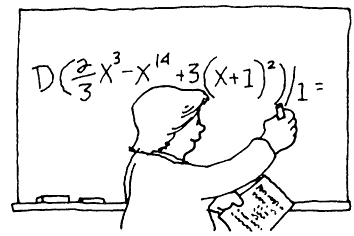
CLASSROOM CAPSULES



A Classroom Capsule is a short article that contains a new insight on a topic taught in the earlier years of undergraduate mathematics. Please submit manuscripts prepared according to the guidelines on the inside front cover to Frank Flanigan.

Fibonacci Numbers, Recursion, Complexity, and Induction Proofs

Elmer K. Hayashi, Wake Forest University, Winston-Salem, NC 27109

In this paper, we compare the complexities of three methods for computing the nth Fibonacci number recursively. The methods are not new, see [1], but the examples and proofs given are interesting, instructive, and probably unfamiliar to many teachers and students. We give simple proofs of the complexity of all three algorithms (if induction proofs can be called simple). Many books will warn students not to use our first algorithm, and we provide a proof that shows why the algorithm should not be used. Our second algorithm illustrates the use of binary halving to improve the performance of an algorithm. Our third algorithm shows how parameters may be used effectively with a recursive algorithm.

The Fibonacci sequence is defined as follows:

$$F_1 = 1, \qquad F_2 = 1, \quad \text{and} \quad F_n = F_{n-1} + F_{n-2} \quad \text{for all } n > 2.$$

For each method that we describe, we will define a function $f(n)$ that returns the value F_n. We will measure the complexity of the method by counting the number of times f (or in the last algorithm, a second function g) must be called recursively in order to compute F_n.

The Fibonacci sequence grows exponentially. Note that $F_3 = 2$ is twice as large as $F_2 = 1$, and $F_4 = 3$ is 1.5 times larger than F_3. Now if you suppose that

$$F_{k-1} \geq 1.5 \cdot F_{k-2}$$

and

$$F_{k-2} \geq 1.5 \cdot F_{k-3},$$

then

$$F_{k-1} + F_{k-2} \geq 1.5 \cdot (F_{k-2} + F_{k-3})$$

or

$$F_k \geq 1.5 \cdot F_{k-1}.$$

By induction, for all $n > 2$, we have

$$F_n \geq 1.5 \cdot F_{n-1} \geq (1.5)^2 \cdot F_{n-2} \geq \cdots \geq (1.5)^{n-2} \cdot F_2 = (1.5)^{n-2}.$$

Thus F_n grows faster than the exponential function $(1.5)^{n-2}$. The reader may wish to prove that $F_n \geq (1.61)^{n-2}$ for all n, but $F_n \geq (1.62)^{n-2}$ does not hold for all n. For a different approach, see [2].

A Method with exponential complexity. The first method for computing F_n merely uses the definition directly, and perhaps not surprisingly turns out to be the slowest method.

```
function f(n)
    if n = 1 or n = 2 then
        return 1
    else if n > 2 then
        return f(n − 1) + f(n − 2)
    end if
end function
```

Note that if we call $f(1)$ or $f(2)$, then the function immediately returns 1. Thus computing F_1 requires $F_1 = 1$ function call, and computing F_2 requires $F_2 = 1$ function call. If we call $f(3)$, then the function returns $f(2) + f(1)$, so $f(2)$ and $f(1)$ must also be called. Thus computing F_3 requires 3 function calls. Since $3 > 2 = F_3$, we conjecture that computing F_n requires at least F_n function calls. We have already shown the conjecture is true for $n = 1$, 2, and 3. We assume that computing F_{k-1} requires at least F_{k-1} function calls, and that computing F_{k-2} requires at least F_{k-2} function calls where $k > 2$. Then when we call $f(k)$ it will return $f(k-1) + f(k-2)$. Hence to compute F_k, we call $f(k)$ which in turn calls $f(k-1)$ and $f(k-2)$. Using the induction hypothesis, this will require at least $1 + F_{k-1} + F_{k-2} > F_k$ calls. By mathematical induction, we conclude that for every positive integer n, computing F_n will take at least F_n calls of the function f. Applying the result in the previous section, it follows that the time to compute F_n grows exponentially with n. You may enjoy trying to find a formula for the exact number of calls needed to compute F_n by this method.

A Method with polynomial complexity. We can improve on the first method in the same way that binary search improves on linear search. We first notice that we can skip the computation of F_{n-1} as follows

$$F_n = F_{n-1} + F_{n-2} = (F_{n-2} + F_{n-3}) + F_{n-2} = 2 \cdot F_{n-2} + F_{n-3}.$$

Continuing the above process, we can skip the computation of F_{n-2} by replacing F_{n-2} with $F_{n-3} + F_{n-4}$, and then skip F_{n-3}, etc. By expressing F_n in terms of $F_{\lfloor n/2 \rfloor + d}$, $d = -1$, 0, or, 1, we can substantially cut our work. We claim for $n > 2$ and $2 \leq k \leq n$,

$$F_n = F_k \cdot F_{n-k+1} + F_{k-1} \cdot F_{n-k}.$$

The proof for fixed $n > 2$ is by induction on k. We first note that $F_n = F_{n-1} + F_{n-2} = F_2 \cdot F_{n-1} + F_1 \cdot F_{n-2}$, i.e. the result is true when $k = 2$. Now suppose we have

$$F_n = F_k \cdot F_{n-k+1} + F_{k-1} \cdot F_{n-k} \quad \text{for some } k,$$

then

$$\begin{aligned} F_n &= F_k \cdot (F_{n-k} + F_{n-k-1}) + F_{k-1} \cdot F_{n-k} \\ &= (F_k + F_{k-1}) \cdot F_{n-k} + F_k \cdot F_{n-k-1} \\ &= F_{k+1} \cdot F_{n-(k+1)+1} + F_{(k+1)-1} \cdot F_{n-(k+1)}. \end{aligned}$$

It follows by mathematical induction that the result is valid for all k for which all subscripts are positive. Let $k = \lfloor (n+1)/2 \rfloor$ be the greatest integer not exceeding (n + 1)/2. Note that if n is even, then $k = n - k$, and if n is odd, then $k = n - k + 1$ and $k - 1 = n - k$.

We can now define our function f as follows:

```
function f(n)
      if n = 1 or n = 2 then
            return 1
      else if n > 2 then
            k = ⌊(n + 1)/2⌋
            if n is even then
                  return f(k)·(f(n − k + 1) + f(k − 1))
            else
                  return f(k)² + f(k − 1)²
            end if
      end if
end function
```

As before, F_1 and F_2 require one call each to compute. Since $F_3 = F_2^2 + F_1^2$, computing F_3 will take 3 function calls, one to $f(3)$, one to $f(2)$, and one to $f(1)$. Similarly, computing $F_4 = F_2 \cdot (F_3 + F_1)$ will require 6 calls, and $F_5, \ldots, F_9$ will require 5, 11, 10, 15, and 12 function calls, respectively. Examining the data gathered so far, it is not hard to conjecture that the number of function calls required to compute F_n is bounded above by n^2 (the complexity is probably more on the order of $n^{1.4}$, but I have not been able to prove this). In the proof, our induction hypothesis requires that we assume the truth of our conjecture for all $k \le n$. Then if n is odd, F_n is computed by calling $f((n+1)/2)$ and $f((n-1)/2)$, and hence by our induction hypothesis requires no more than $1 + ((n+1)/2)^2 + ((n-1)/2)^2 = (n^2+3)/2 < n^2$ function calls when $n > 2$. If n is even, F_n is computed by calling $f(n/2)$, $f((n+2)/2)$, and $f((n-2)/2)$, hence computing F_n requires no more than $1 + (n/2)^2 + ((n+2)/2)^2 + ((n-2)/2)^2 = 3n^2/4 + 3 < n^2$ function calls when $n \ge 4$. By induction, it follows that the computation of F_n never requires more than n^2 function calls. The same method of proof can be used to show that F_n can be computed with no more than $O(n^{1.585})$ function calls since $1 + (n/2)^a + ((n+2)/2)^a + ((n-2)/2)^a < n^a$ will hold for all large n if $a > \log(3)/\log(2)$.

A Method with linear complexity. The final method uses two parameters to remember the last two Fibonacci numbers, and hence eliminates repetitive computation of the same Fibonacci number.

```
function f(n)
    if n = 1 then
        return 1
    else if n > 1 then
        return g(1, 1, n − 1)
    end if
end function

function g(a, b, n)
    if n = 1 then
        return a
    else
        return g(a + b, a, n − 1)
    end if
end function
```

Since each recursive call reduces the n parameter by 1, and a value is returned when this parameter reaches 1, it is clear that a call to f results in $n-1$ calls to g and 1 call to f. It remains to be shown that F_n is the value that is returned. If $n=1$, then $1=F_1$ is returned by f. If $n=2$, then f calls g with parameters 1, 1, 1, and so $1=F_2$ is returned by g and hence by f. Note that in any first call to g, the first two parameters are $a=F_2=1$, and $b=F_1=1$. Now if we assume that on the kth call of g, the first two parameters are $a=F_{k+1}$ and $b=F_k$, then $a=F_{k+1}$ is returned if the third parameter is 1, and otherwise g is called again with the first two parameters $a+b=F_{k+1}+F_k=F_{k+2}$ and $a=F_{k+1}$. By induction, it follows that F_n will be returned after 1 call to f and $n-1$ calls to g.

References

1. Giles Brassard and Paul Bratley, *Algorithmics, Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ, 1968, pp. 16–18.
2. Udi Manbar, *Introduction to Algorithms*, Addison-Wesley, Reading, MA, 1989, pp. 46–50.

---o---

Distance from a Point to a Plane with a Variation on the Pythagorean Theorem

Abdus Sattar Gazdar, University of New England, Armidale 2350 NSW Australia

In this capsule we give a short and direct derivation of the standard formula

$$|Aa + Bb + Cc + D| / \sqrt{A^2 + B^2 + C^2} \tag{1}$$

for the distance between a point $P(a, b, c)$ and the plane LMN:

$$Ax + By + Cz + D = 0. \tag{2}$$