Peergrade #4: Mathematical induction

Alessandro Bruni

October 26, 2019

Exercise 1. *Let $n \in \mathbb{N}$ and $n > 6$. Prove by simple induction that $3^n < n!$.*
(Remember that $n!$ is called the factorial *of n, and is defined as $n! = \Pi_{i=1}^{n} i = 1 \cdot 2 \cdot \ldots \cdot n$)*

Solution 1. *Let $P(n) := 3^n < n!$. We prove the property for the base case $P(7)$:*

$$3^7 = 2187 < 7! = 5040$$

For the inductive case: assume the I.H. $P(n)$ to prove $P(n+1)$:

$$\begin{aligned} 3^{n+1} =& 3^n \cdot 3 \\ <& n! \cdot 3 && \text{(By I.H. } 3^n < n!\text{)} \\ <& n! \cdot (n+1) = (n+1)! \end{aligned}$$

thus proving the theorem.

Exercise 2. *Prove that $f_0^2 + f_1^2 + f_2^2 + \cdots + f_n^2 = f_n \cdot f_{n+1}$.*
That is, for any $n \in \mathbb{N}$, the sum of the squares of the first n Fibonacci numbers is equal to the product of $f_n \cdot f_{n+1}$.

Solution 2. *Let $P(n) := \sum_{i=0}^{n} f_i^2 = f_n \cdot f_{n+1}$.*
Base case $P(0)$:

$$\sum_{i=0}^{0} f_i^2 = 1^2 = f_0 \cdot f_1 = 1 \cdot 1$$

Inductive case: assume $P(n)$ to prove $P(n+1)$:

$$\begin{aligned} \sum_{i=0}^{n+1} f_i^2 &= \left(\sum_{i=0}^{n} f_i^2 \right) + f_{n+1}^2 \\ &= f_n \cdot f_{n+1} + f_{n+1}^2 && \text{by I.H.} \\ &= f_{n+1} \cdot (f_n + f_{n+1}) && \text{regrouping the equation} \\ &= f_{n+1} \cdot f_{n+2} && \text{by definition of the Fibonacci function} \end{aligned}$$

Exercise 3. *Give iterative and recursive algorithms for the nth term of the sequence defined by*

$$\begin{aligned} a_0 &= 1 \\ a_1 &= 3 \\ a_2 &= 5 \\ a_n &= a_{n-1} \cdot a_{n-2}^2 \cdot a_{n-3}^3 \end{aligned}$$

Which one is more efficient?

Solution 3. *There are multiple valid solutions to this exercise: pseudocode or real code are both valid options.*

In "math-style" pseudocode, here is the solution for the recursive and iterative functions, respectively f and g:

$$f_n = \begin{cases} 1 & \text{if } n = 0 \\ 3 & \text{if } n = 1 \\ 5 & \text{if } n = 2 \\ f_{n-1} \cdot f_{n-2}^2 \cdot f_{n-3}^3 & \text{otherwise, if } n > 2 \end{cases}$$

$$g_n = \begin{cases} (1, \mathbf{F}, \mathbf{F}) & \text{if } n = 0 \\ (3, 1, \mathbf{F}) & \text{if } n = 1 \\ (5, 3, 1) & \text{if } n = 2 \\ (x \cdot y^2 \cdot z^3, x, y) & \text{otherwise, if } n > 2 \text{ and } (x, y, z) = g_{n-1} \end{cases}$$

Here the symbol $\mathbf{F}$ *(read: bottom) stands for no result, since the recursive relation is actually undefined for parameters below zero. In the actual code we'll use* null *or equivalents.*

We know that they are "equivalent", as one can prove by induction that for $n > 2$ we have $g_n = (f_n, f_{n-1}, f_{n-2})$.

Here are two Python functions that correspond to the pseudocode definitions:

```
def f(n):
    if n == 0: return 1
    if n == 1: return 3
    if n == 2: return 5
    return f(n-1) * f(n-2)**2 * f(n-3)**3

def g(n):
    if n == 0: return (1, None, None)
    if n == 1: return (3, 1, None)
    if n == 2: return (5, 3, 1)
    (x,y,z) = g(n-1)
    return (x * y**2 * z**3, x, y)
```

Another way to express g in pseudocode is with a for-loop:

```
def g(n):
    (x, y, z) = (1, 3, 5)
    for i in range(1,n):
        (x, y, z) = (y, z, z * y**2 * x**3)
    return x
```

Arternatively, here is Java code that uses the BigInteger class, necessary to get the correct results:

```
import java.math.BigInteger;
public class RecIter {
    public static void main(String[] args) {
BigInteger k = new BigInteger(args[0]);
System.out.println(f(k));
System.out.println(g(k));
    }
    static final BigInteger n0, n1, n2, n3, n5;
    static {
n0 = new BigInteger("0");
n1 = new BigInteger("1");
n2 = new BigInteger("2");
n3 = new BigInteger("3");
n5 = new BigInteger("5");
    }

    static BigInteger f(BigInteger n) {
if (n.equals(n0)) return n1;
if (n.equals(n1)) return n3;
if (n.equals(n2)) return n5;
BigInteger x = f(n.subtract(n1));
BigInteger y = f(n.subtract(n2));
BigInteger z = f(n.subtract(n3));
return x.multiply((y.pow(2)).multiply(z.pow(3)));
    }

    static Triple g(BigInteger n) {
if (n.equals(n0)) return new Triple(n1,null,null);
if (n.equals(n1)) return new Triple(n3,n1,null);
if (n.equals(n2)) return new Triple(n5,n3,n1);
Triple t = g(n.subtract(n1));
return new Triple(t.x.multiply((t.y.pow(2)).multiply(t.z.pow(3))),t.x,t.y);
    }
```

```
}

class Triple {
    public BigInteger x,y,z;
    public Triple(BigInteger x, BigInteger y, BigInteger z) {
this.x = x; this.y = y; this.z = z;
    }
    public String toString() {
return "(" + x + "," + y + "," + x + ")";
    }
}
```

Execution times may vary, but you might have observed that with growing parameters both functions slow down "a lot": this is due to the fact that the libraries we use may cache internal results[1].

If we were instead to check how many times the functions f and g are called, we get that if we choose $n = 20$, then $f(n)$ requires 128287 calls to the function f, wherease $g(n)$ requires only 19 calls to the function g. If the time to compute the operations inside the function were constant (for example if we were to use simple integers or floats in Java instead), then f would take considerably more time than g.

Obviously your response may vary depending on your implementation and observations, but it should be coherent with what presented so far.

[1] Caching is a technique in programming that stores intermediate results as a way to speed up computation, if these results are needed again