12. This is the inefficient method.

\[
\text{procedure } \text{power}(x, n, m : \text{positive integers})
\]
\[
\text{if } n = 1 \text{ then return } x \mod m
\]
\[
\text{else return } (x \cdot \text{power}(x, n - 1, m)) \mod m
\]

14. This is actually quite subtle. The recursive algorithm will need to keep track not only of what the mode actually is, but also of how often the mode appears. We will describe this algorithm in words, rather than in pseudocode. The input is a list \(a_1, a_2, \ldots, a_n\) of integers. Call this list \(L\). If \(n = 1\) (the base case), then the output is that the mode is \(a_1\) and it appears 1 time. For the recursive case \((n > 1)\), form a new list \(L'\) by deleting from \(L\) the term \(a_n\) and all terms in \(L\) equal to \(a_n\). Let \(k\) be the number of terms deleted. If \(k = n\) (in other words, if \(L'\) is the empty list), then the output is that the mode is \(a_n\) and it appears \(n\) times. Otherwise, apply the algorithm recursively to \(L'\), obtaining a mode \(m\), which appears \(t\) times. Now if \(t \geq k\), then the output is that the mode is \(m\) and it appears \(t\) times; otherwise the output is that the mode is \(a_n\) and it appears \(k\) times.

16. The sum of the first one positive integer is 1, and that is the answer the recursive algorithm gives when \(n = 1\), so the basis step is correct. Now assume that the algorithm works correctly for \(n = k\). If \(n = k + 1\), then the \text{else} clause of the algorithm is executed, and \(k + 1\) is added to the (assumed correct) sum of the first \(k\) positive integers. Thus the algorithm correctly finds the sum of the first \(k + 1\) positive integers.

18. We use mathematical induction on \(n\). If \(n = 0\), we know that \(0! = 1\) by definition, so the \text{if} clause handles this basis step correctly. Now fix \(k \geq 0\) and assume the inductive hypothesis—that the algorithm correctly computes \(k!\). Consider what happens with input \(k + 1\). Since \(k + 1 > 0\), the \text{else} clause is executed, and the answer is whatever the algorithm gives as output for input \(k\), which by inductive hypothesis is \(k!\), multiplied by \(k + 1\). But by definition, \(k! \cdot (k + 1) = (k + 1)!\), so the algorithm works correctly on input \(k + 1\).

20. Our induction is on the value of \(y\). When \(y = 0\), the product \(xy = 0\), and the algorithm correctly returns that value. Assume that the algorithm works correctly for smaller values of \(y\), and consider its performance on \(y\). If \(y\) is even (and necessarily at least 2), then the algorithm computes 2 times the product of \(x\) and \(y/2\). Since it does the product correctly (by the inductive hypothesis), this equals \(2(x \cdot y/2)\), which equals \(xy\) by the commutativity and associativity of multiplication. Similarly, when \(y\) is odd, the algorithm computes 2 times the product of \(x\) and \((y-1)/2\) and then adds \(x\). Since it does the product correctly (by the inductive hypothesis), this equals \(2(x \cdot (y-1)/2) + x\), which equals \(xy - x + x = xy\), again by the rules of algebra.

22. The largest in a list of one integer is that one integer, and that is the answer the recursive algorithm gives when \(n = 1\), so the basis step is correct. Now assume that the algorithm works correctly for \(n = k\). If \(n = k + 1\), then the \text{else} clause of the algorithm is executed. First, by the inductive hypothesis, the algorithm correctly sets \(m\) to be the largest among the first \(k\) integers in the list. Next it returns as the answer either that value or the \((k + 1)\)st element, whichever is larger. This is clearly the largest element in the entire list. Thus the algorithm correctly finds the maximum of a given list of integers.

24. We use the hint.

\[
\text{procedure } \text{twopower}(n : \text{positive integer}, a : \text{real number})
\]
\[
\text{if } n = 1 \text{ then return } a^2
\]
\[
\text{else return } \text{twopower}(n - 1, a)^2
\]

26. We use the idea in Exercise 24, together with the fact that \(a^n = (a^{n/2})^2\) if \(n\) is even, and \(a^n = a \cdot (a^{(n-1)/2})^2\) if \(n\) is odd, to obtain the following recursive algorithm. In essence we are using the binary expansion of \(n\) implicitly.