4.1.4. *The sequence of Fibonacci numbers F_n is defined as $F_1 = 1$, $F_2 = 1$, and for $n > 2$, $F_n = F_{n-1} + F_{n-2}$. Write an algorithm for obtaining F_{100}.*

set F_1 **to** 1;
set F_2 **to** 1;
set i **to** 3;
while $i \leq 100$ **do**
  set F_i **to** $F_{i-1} + F_{i-2}$;
  set i **to** $i + 1$;
return F_{100};

We see that this algorithm will proceed by first initializing F_1 and F_2, then, in turn, setting i to 3 and calculating F_i, then setting i to 4 and calculating F_i, and so forth, until it has calculated F_i when i is 100, at which point it can stop.

4.1.6. *Suppose we have two words of arbitrary length. If a computer can compare any two letters, describe an algorithm for determining which word comes first in alphabetical order.*

Input: words $a_1a_2a_3 \ldots a_n$ and $b_1b_2b_3 \ldots b_m$
set i **to** 1;
repeat
  if $i > n$ *and* $i > m$ **then** decide that $a = b$;
  if $i > n$ *and* $i \leq m$ **then** decide that $a < b$;
  if $i \leq n$ *and* $i > m$ **then** decide that $a > b$;
  if $i \leq n$ *and* $i \leq m$ **then**
    if $a_i < b_i$ **then** decide that $a < b$;
    if $a_i > b_i$ **then** decide that $a > b$;
    if $a_i = b_i$ **then set** i **to** $i + 1$ and make no decision;
until *we have made a judgment* ;

This algorithm is an expression of how we would compare words. The process is simply a matter of comparing the first letters (a_1 to b_1), and returning the result of that comparison if unequal, and moving onto the next letter if impossible. i records which letter we are on at each stage of the process, so it starts at 1, since we start by comparing the first letters, and increments when a letter fails to provide an adequate comparison. The conditions at the beginning indicate what to do when we run out of letters from one or the other string. If we run out on one, it means that string is shorter (e.g. "ant" is lexicographically smaller than "anteater"); if we run out on both, it means we've exhausted both words without finding a way in which they differ, in which case the two words are identical.

4.1.10. *Describe an algorithm that inputs a positive integer N and outputs its binary repre-*

sentation.

Input: integer N
set i **to** 0;
repeat
  set i **to** $i+1$;
  if N *is even* **then set** a_i to 0;
  if N *is odd* **then set** a_i to 1 and **set** N **to** $N-1$;
  set N to $\frac{N}{2}$;
until $N = 0$;
output $a_i a_{i-1} a_{i-2} \cdots a_0$;

This algorithm works by repeated division of N by 2, recording the remainder in each division in the a_is. We can see how this works on, for instance, 13: division by 2 yields 6 with a remainder of 1, another division yields 3 with a remainder of 0, another division yields 1 with a remainder of 1, and another division yields 0 with a remainder of 1. Our remainders are, in order: 1, 0, 1, 1. Reversing those gives the binary representation 1101. This works because, if $f(n)$ is the binary representation of a number n, we generate the binary representation with the recursion:

$$f(n) = \begin{cases} f(\frac{n}{2})0 & \text{if } n \text{ is even} \\ f(\frac{n-1}{2})1 & \text{if } n \text{ is odd} \end{cases}$$

The above algorithm could, in fact, be framed as a recursive procedure.

4.1.12. *There are n gossips, each of whom knows some information that the others do not. Whenever two gossips call each other, they always tell each other all the information that they know.*

(a) *Describe a sequence of calls that will result in each gossip hearing all the information.*

Let us call our gossips $g_1, g_2, \ldots, g_n$. The easiest way to distribute the information would be to collect all the information in one place, then transmit it all from there. We will use g_1 as our collection point, so we can have g_1 call each of g_2, g_3, $\ldots$, g_n in turn, collecting each of their pieces of information. After these $n-1$ calls, both g_1 and g_n will know everything (before calling g_n, g_1 knew everything except g_ns secret; when they shared information, they both got a complete set of secrets). g_1 now needs to share these secrets with all $n-2$ of the uninformed gossips, and can do so by calling each of g_2, $\ldots$, g_{n-1} in turn. Thus, all secrets are shared in $2n-3$ calls if g_1 calls the following people in order: $g_2, g_3, \ldots, g_n, g_2, g_3, \ldots, gn-1$.

(b) *Show that for $n \geq 4$, everyone can learn all the gossip in $2n-4$ calls.*

Above we saw that it was possible in $2n-3$ calls: shaving one call off of that naive strategy requires some subtlety! We may note that 4 gossips can share every piece of information in 4 calls with the following sequence: g_1 calls g_2, g_3 calls g_4, g_1 calls g_3, and g_2 calls g_4. Now we can show that $2(n+1)-4$ suffices to solve the

problem on $n+1$ people with an inductive argument: have g_{n+1} first share their information with one of the others (have g_{n+1} call g_1), then have $g_1, g_2, \ldots g_n$ share their information (using $2n-4$ calls, by the inductive hypothesis), and finally, have g_1 call g_{n+1} back. This takes a total of $1+(2n-4)+1 = 2(n+1)-4$ calls, demonstrating our inductive step.

It is far more difficult to show that $2n-4$ is actually the least number of calls possible. This problem was presented in 1971 by Tijdeman, and was solved four different ways in the following decade. This problem is described, and references are provided for full proofs in the notes on OEIS A058992.

4.1.14. *In a group of nine coins there is one counterfeit coin that is lighter than the others. Describe an algorithm for finding the counterfeit coin in two weighings on an equal arm balance.*

Let us divide the coins into three equally-sized sets A_1,A_2, and A_3. A single weighing in which we compare A_1 to A_2 will tell us which group contains the counterfeit. If the sets are of unequal weight, whichever is lighter contains the counterfeit; if the two sets are of equal weight, the counterfeit is in neither and must thus be in A_3.

Having determined the value of i such that A_i contains the counterfeit, let us denote the three elements of A_i by x_1, x_2, and x_3. Now let us do a second weighing comparing x_1 to x_2. If one is lighter, it is clearly counterfeit. If neither of them are lighter, the counterfeit is neither of them and must thus be x_3.

This algorithm can be generalized to determine the counterfeit coin from among 3^n coins in n weighings.

4.2.4. *If $0 < b_1 < b_2$, show that b_1^n is $O(b_2^n)$, but that b_2^n is not $O(b_1^n)$.*

Since raising positive numbers to a positive power preserves inequalities, $b_1^n < b_2^n$ for $n > 0$. To present this pedantically, $b_1^n < 1 \cdot 1 \cdot b_2^n$ for $n > 0$. Thus, for sufficiently large n (in this case, any $n > 0$) b_1^n is bounded above by a constant multiple of b_2^n (in this case, b_2^n times 1). Thus, b_1^n is $O(b_2^n)$.

Showing that b_2^n is not $O(b_1^n)$, on the other hand, cannot be done by simply giving an example, because we want to show that *no* constant multiple of b_1^n is an upper bound on b_2^n for large n. We can demonstrate this most easily using a proof by contradiction: suppose some multiple of b_1^n is a long-term upper bound on b_2^n: that is to say, there is some k and N so that for $n > N$, $b_2^n < kb_1^n$.

We shall show that this is impossible by inspecting the situation under which $b_2^n \geq kb_1^n$. Algebraic rearrangement of this expression shows that it is true whenever $k \leq \left(\frac{b_2}{b_1}\right)^n$. Taking the (natural) logarithm of both sides, we find that $k \leq n \ln \frac{b_2}{b_1}$. Since $\frac{b_2}{b_1} > 1$, its logarithm is positive, so division by it preserves the inequality. Thus, $\frac{k}{\ln \frac{b_2}{b_1}} \leq n$. Thus, if $n > \frac{k}{\ln \frac{b_2}{b_1}}$, then $kb_2^n \geq b_1^n$. So for any k we might choose, a sufficiently large n causes b_2^n to exceed b_1^n. This means that no constant multiple of b_1^n is a long-term upper bound on b_2^n, so b_2^n is not $O(b_1^n)$.

4.2.5. *If* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k < \infty$*, show that* $f(n)$ *is* $O(g(n))$*.*

Let us state that limit in a classical theoretically justifiable form: given $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k$, we know that for any ϵ we might choose, there is a value of N such that if $n > N$, $k - \epsilon < \frac{f(n)}{g(n)} < k + \epsilon$. Let us arbitrarily choose $\epsilon = 1$ (we can choose pretty much anything, but this makes the math work well). Thus, we know there is some N such that for all $n > N$, $\frac{f(n)}{g(n)} < k + 1$. We also have a lower bound on the ratio, but it's not useful to us. Algebraically rearranging the above expression, we get that for $n > N$, $f(n) < (k+1)g(n)$. We have thus shown that, for sufficiently large n (namely, $n > N$), it is the case that $f(n)$ is bounded above by a constant multiple of $g(n)$ (namely, $(k+1)$ times $g(n)$), so $f(n)$ must be $O(g(n))$.

4.2.8. *For each of the following pairs of functions, prove that* $f(n)$ *is not* $O(g(n))$*.*

In this problem, it is most effective to use what is essentially the converse of the statement proven in problem 4.2.5: if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, then $f(n)$ is *not* $O(g(n))$. This proceeds along quite similar lines to the proof of problem 4.2.5: the definition of the limit asusres us that for any E we might choose, there is a value of N such that if $n > N$, $\frac{f(n)}{g(n)} > E$. Thus, for any value of E we might choose, there is an n such that if $n > N$, $f(n) > Eg(n)$. Thus, any constant multiple of $g(n)$ we choose (represented above by $Eg(n)$ will be exceeded for sufficiently large n by $f(n)$. Thus, no constant multiple of $g(n)$ serves as a long-term upper bound on $f(n)$, so $f(n)$ is not $O(g(n))$.

(a) $f(n) = n^{m+1}$ *and* $g(n) = n^m$*.*
Since $\frac{f(n)}{g(n)} = n$, it follows that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, so by the lemma proven above, $f(n)$ is not $O(g(n))$.

(b) $f(n) = n^{3/2}$ *and* $g(n) = n \log n$*.*
We use algebraic simplification combined with L'Hôpital's rule on the indeterminate form $\frac{\infty}{\infty}$:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^{3/2}}{n \log n} = \lim_{n\to\infty} \frac{\sqrt{n}}{\log n} = \lim_{n\to\infty} \frac{\frac{1}{2\sqrt{n}}}{\frac{1}{n}} = \lim_{n\to\infty} \frac{\sqrt{n}}{2} = \infty$$

so by the lemma proven above, $f(n)$ is not $O(g(n))$.

(c) $f(n) = 2^n$ *and* $g(n) = n^{\log n}$*.*
This ratio is more difficult to show goes to infinity than most of the others. We can rephrase the ratio $\frac{f(n)}{g(n)}$ as $2^{n - \log_2 n \log n}$; it then suffices to show that $n - \log_2 n \log n$ gets arbitrarily large as n gets very large. We can do so by looking at $n = 2^s$ for some s; then, since $\log n < \log_2 n$, it follows that $n - \log_2 n \log n > 2^s - s^2$. We know 2^s is not $O(s^2)$ (using an argument similar to the one presented in part (d) of this problem, so 2^s will eventually exceed any multiple of s^2, so for any k, $2^s - s^2$ will eventually exceed $(k-1)s^2$, which can be arbitrarily large, so $\lim_{s\to\infty} 2^s - s^2 = \infty$, and thus $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

(d) *$f(n) = 3^n$ and $g(n) = n^3$.*

We use L'Hôpital's rule several times on the indeterminate form $\frac{\infty}{\infty}$:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3^n}{n^3} = \lim_{n\to\infty} \frac{(\ln 3)3^n}{3n^2} = \lim_{n\to\infty} \frac{(\ln 3)^2 3^n}{6n} = \lim_{n\to\infty} \frac{(\ln 3)^3 3^n}{6} = \infty$$

so by the lemma proven above, $f(n)$ is not $O(g(n))$.

4.2.10. (a) *Show that x^{2^k} can be computed using k multiplications.*

We can get x^2 by multiplying $x \cdot x$; armed with that knowledge, we can get x^4 by multiplying $x^2 \cdot x^2$; then we can get x^8 from $x^4 \cdot x^4$. An inductive argument serves to upgrade our demonstration to a proof:

The base case $k = 0$ is trivial: we already have x, so we get $x^{2^0} = x^1$ with no multiplications at all. If we assume x^{2^k} can be calculated in k multiplications, we verify the inductive step that we can get $x^{2^{(k+1)}}$ in $k+1$ multiplications like so: find x^{2^k} (doable in k multiplications by our inductive hypothesis), and then perform the multiplication $x^{2^k} \cdot x^{2^k}$ to get $\left(x^{2^k}\right)^2 = x^{2\cdot 2^k} = x^{2^{k+1}}$, so, having done k multiplications and then a final multiplication, we have performed $k+1$ multiplications total.

(b) *Describe an $O(\log n)$ algorithm for computing x^n.*

We saw above that we can get x raised to power-of-two powers very efficiently; thus, our best bet is to decompose n efficiently into powers of 2. So, for instance, to calculate x^{13}, the most efficient algorithm would be to calculate x^2, x^4, and x^8, then compute the product $x \cdot x^4 \cdot x^8$. Our choice of exponents is motivated by the binary representation of 13, which encodes the most compact way of representing 13 as a sum of powers of 2 (i.e. $13 = 1101_2 = 8+4+1$). Eahc operation involved in this process (calculating a binary representation for n, raising x to the $2^{\lfloor \log_2 n \rfloor}$ power, and multiplying fewer than $\lfloor \log_2 n \rfloor$ numbers together) is completable in $O(\log n)$ time, so the entire process is $O(\log n)$. However, to make it more explicit, here is an actual, highly optimized algorithm, which is cannibalized from our algorithm in problem 4.1.10:

Input: integer n, number x
set y **to** x;
set p **to** 1;
repeat
  if n *is odd* **then set** p to $p \cdot y$ and **set** n **to** $n-1$;
  set y to $y \cdot y$;
  set n to $\frac{n}{2}$;
until $n = 0$;
output p;

We can see how this would work by looking at, for instance, entering $n = 42$:

- We initialize p to be 1 and y to be x.

- On our first loop, n is even, so we don't change p. We set y to $y \cdot y = x \cdot x = x^2$, and set n to $\frac{42}{2} = 21$.
- On our second loop, n is odd, so we set p to $p \cdot y = 1 \cdot x^2 = x^2$ and n to $21 - 1 = 20$. We then set y to $y \cdot y = x^2 \cdot x^2 = x^4$, and set n to $\frac{20}{2} = 10$.
- On our third loop, n is even, so we don't change p. We set y to $y \cdot y = x^4 \cdot x^4 = x^8$, and set n to $\frac{10}{2} = 5$.
- On our fourth loop, n is odd, so we set p to $p \cdot y = x^2 \cdot x^8 = x^{10}$ and n to $5 - 1 = 4$. We then set y to $y \cdot y = x^8 \cdot x^8 = x^{16}$, and set n to $\frac{4}{2} = 2$.
- On our fifth loop, n is even, so we don't change p. We set y to $y \cdot y = x^{16} \cdot x^{16} = x^{32}$, and set n to $\frac{2}{2} = 1$.
- On our sixth loop, n is odd, so we set p to $p \cdot y = x^{10} \cdot x^{32} = x^{42}$ and n to $1 - 1 = 0$. We then set y to $y \cdot y = x^{32} \cdot x^{32} = x^{64}$, and set n to $\frac{0}{2} = 0$.
- Since $n = 0$, we terminate and return p, whose current value is the desired x^{42}.

We see that the loop's interior can be performed in constant time (since it consists of a two comparisons and four assignments); it is easy to see that in addition the loop iterates only at most $\lceil \log_2(n) \rceil + 1$ times, since each iteration of the loop divides it by 2, so $\lceil \log_2 n \rceil$ iterations should reduce it to 1, and thence a single iteration will take it to zero, where it terminates. Thus, this algorithm takes $O(\log n)$ time, since the number of steps is a constant times an approximately logarithmic function.

4.2.12. *Let the sequence f_r be defined by $f_1 = 1$, $f_2 = 1$, and $f_r = f_{r-1} + f_{r-2}$ for $r \geq 2$. Show that if $m \leq n \leq f_k$, the Euclidean algorithm requires at most k steps to compute the greatest common divisor of m and n. Deduce that the number of steps required by the Euclidean algorithm is $O(\log n)$.*

Let us define $g(m, n)$ to be the number of steps required to perform the Euclidean algorithm m and n given $m < n$. We know the Euclidean algorithm consists of the single step of determining the remainder r under division of n by m, and then instantiating the Euclidean algorithm on r and m. Thus, $g(m, n) = 1 + g(r, m)$, and in addition $g(0, m) = 0$, since the GCD of 0 and any m is known to be simply m. We thus need to solve (or at least find adequate bounds on) what is essentially a multivariate recurrence: $g(m, n) = 1 + g(r, m)$. Since $m < n$, we know that n goes into m at least once, so we are guaranteed that $r < n - m$. Let us suppose the Euclidean algorithm takes k steps to complete. Then, if it has intermediary remainders $r_{k-1}, r_{k-2}, r_{k-3}, \ldots, r_1, 0$, our justification of the number of steps taken follows:

$$g(m, n) = 1 + g(r_{k-1}, m) = 2 + g(r_{k-2}, r_{k-1}) = 3 + g(r_{k-3}, r_{k-2}) = \cdots = k + g(r_1, 0) = k + 0$$

And we know, by the inequality produced above, that:

$$\begin{aligned} r_{k-1} &< n - m \\ r_{k-2} &< m - r_{k-1} \\ r_{k-3} &< r_{k-1} - r_{k-2} \\ &\vdots \\ r_{i-2} &< r_i - r_{i-1} \\ &\vdots \\ 0 &< r_2 - r_1 \end{aligned}$$

Rearraging the above, we see that each $r_i > r_{i-1} + r_{i-2}$. Since we assume r_1 and r_2 are nonzero (since the algorithm would terminate earlier if they were, we have $r_1 > 1$, $r_2 > 1$, and the recurrence-inequality $r_i > r_{i-1} + r_{i-2}$. Since each of these inequalities errs on the side of making things larger than their corresponding Fibonacci number, it follows that $r_i > f_i$. Therefore, if the Euclidean algorithm takes k steps to terminate, it must be the case that the first remainder (r_k in the above notation) exceeds f_k, which is only possible if $m > f_k$. The contrapositive of this statement guarantees that if $m < f_k$ (or if $n < f_k$), then the Euclidean algorithm will terminate in fewer than k steps.

We know from solving the Fibonacci recurrence that f_k grows approximately as $\left(\frac{1+\sqrt{5}}{2}\right)^k$; we can in fact guarantee that $f_k < \left(\frac{1+\sqrt{5}}{2}\right)^k$, so the Euclidean algorithm takes only k iterations if $n < \left(\frac{1+\sqrt{5}}{2}\right)^k$; the resultant value of k is $\left\lceil \frac{\log n}{\log \frac{1+\sqrt{5}}{2}} \right\rceil$, which is $O(\log n)$.