# Chapter 31

Michelle Bodnar, Andrew Lohr

May 9, 2017

**Exercise 31.1-1**

By the given equation, we can write $c = 1 \cdot a + b$, with $0 \geq b < a$. By the definition of remainders given just below the division theorem, this means that $b$ is the remainder when c is divided by a, that is $b = c \mod a$.

**Exercise 31.1-2**

Suppose that there are only finitely many primes $p_1, p_2, \dots, p_k$. Then $p = p_1 p_2 \cdots p_k + 1$ isn't prime, so there must be some $p_i$ which divides it. However, $p_i \cdot (p_1 \cdots p_{i-1} p_{i+1} \cdots p_k) < p$ and $p \cdot (p_1 \cdots p_{i-1} p_{i+1} \cdots p_k + 1) > p$, so $p_i$ can't divide $p$. Since this holds for any choice of $i$, we obtain a contradiction. Thus, there are infinitely many primes.

**Exercise 31.1-3**

$a|b$ means there exists $k_1 \in \mathbb{Z}$ so that $k_1 a = b$. $b|c$ means there exists $k_2 \in \mathbb{Z}$ so that $k_2 b = c$. This means that $(k_1 k_2) a = c$. Since the integers are a ring, $k_1 k_2 \in \mathbb{Z}$, so, we have that $a|c$.

**Exercise 31.1-4**

Let $g = \gcd(k, p)$. Then $g|k$ and $g|p$. Since $p$ is prime, $g = p$ or $g = 1$. Since $0 < k < p$, $g < p$. Thus, $g = 1$.

**Exercise 31.1-5**

By Theorem 31.2, since $gcd(a, n) = 1$, there exist integers $p, q$ so that $pa + qn = 1$, so, $bpa + bqn = b$. Since $n|ab$, there exists an integer $k$ so that $kn = ab$. This means that $knp + pqn = (k + q)pn = b$. Since $n$ divides the left hand side, it must divide the right hand side as well.

**Exercise 31.1-6**

Observe that $\binom{p}{k} = \frac{p!}{k!(p-k)!} = \frac{p(p-1)\cdots(p-k+1)}{k!}$. Let $q = (p-1)(p-2)\cdots(p-k+1)$. Since $p$ is prime, $k! \not| p$. However, we know that $\binom{p}{k}$ is an integer because it is also a counting formula. Thus, $k!$ divides $pq$. By Corollary 31.5, $k!|q$. Write $q = ck!$. Then we have $\binom{p}{k} = pc$, which is divisible by $p$.

By the binomial theorem and the fact that $p|\binom{p}{k}$ for $0 < k < p$,

$$(a+b)^p = \sum_{k=0}^{p} \binom{p}{k} a^k b^{p-k} \equiv a^p + b^p (\text{mod } p).$$

**Exercise 31.1-7**

First, suppose that $x = yb + (x \mod b)$, $(x \mod b) = za + ((x \mod b) \mod a)$, and $ka = b$. Then, we have $x = yka + (x \mod b) = (yk+z)a + ((x \mod b) \mod a)$. So, we have that $x \mod a = ((x \mod b) \mod a)$.

For the second part of the problem, suppose that $x \mod b = y \mod b$. Then, by the first half of the problem, applied first to x and then to b, $x \mod a = (x \mod b) \mod a = (y \mod b) \mod a = y \mod a$. So, $x \equiv y \mod a$.

**Exercise 31.1-8**

We can test in time polynomial in $\beta$ whether or not a given $\beta$-bit number is a perfect $k$th power. Then, since two to the $\beta$th power is longer than the number we are given, we only need to test values of $k$ between 2 and $\beta$, thereby keeping the runtime polynomial in $\beta$.

To check whether a given number $n$ is a perfect $k$th power, we will be using a binary search like technique. That is, if we want to find the k-th root of a number, we initially know that it lies somewhere between 0 and the number itself. We then look at the number of the current range of number under consideration, raise it to the $k$th power in time polynomial in $\beta$. We can do this by the method of repeated squaring discussed in section 31.6. Then, if we get a number larger than the given number when we perform repeated squaring, we know that the true $k$th root is in the lower half of the range in consideration, if it is equal, it is the midpoint, if larger, it is the upper half. Since each time, we are cutting down the range by a factor of two, and it is initially a range of length $\Theta(2^\beta)$, the number of times that we need to raise a number the the $k$th power is $\Theta(\beta)$. Putting it all together, with the $O(\beta^3)$ time exponentiation, we get that the total runtime of this procedure is $O(\beta^5)$.

**Exercise 31.1-9**

For (31.6), we see that $a$ and $b$ in theorem 31.2 which provides a characterization of gcd appear symmetrically, so swapping the two won't change anything.

For (31.7), theorem 31.2 tells us that gcd's are defined in terms of integer linear combinations. If we had some integer linear combination involving a and b, we can changed that into one involving (-a) and b by replacing the multiplier of a with its negation.

For (31.8), by repeatedly applying (31.6) and (31.7), we can get this equality for all four possible cases based on the signs of both $a$ and $b$.

For(31.9), consider all integer linear combinations of $a$ and 0, the thing we multiply by will not affect the final linear combination, so, really we are just taking the set of all integer multiples of $a$ and finding the smallest element. We can never decrease the absolute value of a by multiplying by an integer ($|ka| = |k||a|$), so, the smallest element is just what is obtained by multiplying by 1, which is $|a|$.

For (31.10), again consider possible integer linear combinations $na + mka$, we can rewrite this as $(n+km)a$, so it has absolute value $|n+km||a|$. Since the first factor is an integer, we can't have it with a value less than 1 and still have a positive final answer, this means that the smallest element is when the first factor is 1, which is achievable by setting $n = 1, m = 0$.

**Exercise 31.1-10**

Consider the prime factorization of each of $a$, $b$, and $c$, written as $a = p_1p_2 \dots p_k$ where we allow repeats of primes. The gcd of $b$ and $c$ is just the product of all $p_i$ such that $p_i$ appears in both factorizations. If it appears multiple times, we include it as many times as it appears on the side with the fewest occurrences of $p_i$. (Explicitly, see equation 31.13 on page 934). To get the gcd of $\gcd(b, c)$ and $a$, we do this again. Thus, the left hand side is just equal to the intersection (with appropriate multiplicities) of the products of prime factors of $a$, $b$, and $c$. For the right hand side, we consider intersecting first the prime factors of $a$ and $b$, and then the prime factors of $c$, but since intersections are associative, so is the gcd operator.

**Exercise 31.1-11**

Suppose to a contradiction that we had two different prime decomposition. First, we know that the set of primes they both consist of are equal, because if there were any prime $p$ in the symmetric difference, $p$ would divide one of them but not the other. Suppose they are given by $(e_1, e_2, \dots, e_r)$ and $(f_1, f_2, \dots, f_r)$ and suppose that $e_i < f_i$ for some position. Then, we either have that $p_i^{e_i+1}$ divides $a$ or not. If it does, then the decomposition corresponding to $\{e_i\}$ is wrong because it doesn't have enough factors of $p_i$, otherwise, the one corresponding to $\{f_i\}$ is wrong because it has too many.

**Exercise 31.1-12**

Standard long division runs in $O(\beta^2)$, and one can easily read off the remainder term at the end.

## Exercise 31.1-13

First, we bump up the length of the original number until it is a power of two, this will not affect the asymptotics, and we just imagine padding it with zeroes on the most significant side, so it does not change its value as a number. We split the input binary integer, and split it into two segments, a less significant half $\ell$ and an more significant half $m$, so that the input is equal to $m2^{\beta/2} + \ell$. Then, we recursively convert $m$ and $\ell$ to decimal. Also, since we'll need it later, we compute the decimal versions of all the values of $2^{2^i}$ up to $2^\beta$. There are only $\lg(\beta)$ of these numbers, so, the straightforward approach only takes time $O(\lg^2(\beta))$ so will be overshadowed by the rest of the algorithm. Once we've done that, we evaluate $m2^{\beta/2} + \ell$, which involves computing the product of two numbers and adding two numbers, so, we have the recurrence

$$T(\beta) = 2T(\beta/2) + M(\beta/2)$$

Since we have trouble separating $M$ from linear by a $n^\epsilon$ for some epsilon, the analysis gets easier if we just forget about the fact that the difficulty of the multiplication is going down in the subcases, this concession gets us the runtime that $T(\beta) \in O(M(\beta)\lg(\beta))$ by master theorem.

Note that there is also a procedure to convert from binary to decimal that only takes time $\Theta(\beta)$, instead of the given algorithm which is $\Theta(M(\beta)\lg(\beta)) \in \Omega(\beta \lg(\beta))$ that is rooted in automata theory. We can construct a deterministic finite transducer between the two languages, then, since we only need to take as many steps as there are bits in the input, the runtime will be linear. We would have states to keep track of the carryover from each digit to the next.

## Exercise 31.2-1

First, we show that the expression given in equation (31.13) is a common divisor. To see that we just notice that

$$a = (\prod_{i=1}^{r} p_i^{e_i - \min(e_i, f_i)}) \prod_{i=1}^{r} p_i^{\min(e_i, f_i)}$$

and

$$b = (\prod_{i=1}^{r} p_i^{f_i - \min(e_i, f_i)}) \prod_{i=1}^{r} p_i^{\min(e_i, f_i)}$$

Since none of the exponents showing up are negative, everything in sight is an integer.

Now, we show that there is no larger common divisor. We will do this by showing that for each prime, the power can be no higher. Suppose we had some common divisor $d$ of $a$ and $b$. First note that $d$ cannot have a prime factor that doesn't appear in both $a$ or $b$, otherwise any integer times $d$ would also have that factor, but being a common divisor means that we can write both $a$ and $b$ as an integer times $d$. So, there is some sequence $\{g_i\}$ so that $d = \prod_{i=1}^{r} p_i^{g_i}$.

Now, we claim that for every $i$, $g_i \leq \min(e_i, f_i)$. Suppose to a contradiction that there was some $i$ so that $g_i > \min(e_i, f_i)$. This means that $d$ either has more factors of $p_i$ than $a$ or than $b$. However, multiplying integers can't cause the number of factors of each prime to decrease, so this is a contradiction, since we are claiming that $d$ is a common divisor. Since the power of each prime in $d$ is less than or equal to the power of each prime in $c$, we must have that $d \leq c$. So, $c$ is a GCD.

**Exercise 31.2-2**

We'll create a table similar to that of figure 31.1:

| $a$ | $b$ | $\lfloor a/b \rfloor$ | $d$ | $x$ | $y$ |
|---|---|---|---|---|---|
| 899 | 493 | 1 | 29 | -6 | 11 |
| 493 | 406 | 1 | 29 | 5 | -6 |
| 406 | 87 | 4 | 29 | -1 | 5 |
| 87 | 58 | 1 | 29 | 1 | -1 |
| 58 | 29 | 2 | 29 | 0 | 1 |
| 29 | 0 | - | 29 | 1 | 0 |

Thus, $(d, x, y) = (29, -6, 11)$.

**Exercise 31.2-3**

Let $c$ be such that $a = cn + (a \mod n)$. If $k = 0$, it is trivial, so suppose $k < 0$. Then, EUCLID(a+kn,n) goes to line 3, so returns $EUCLID(n, a \mod n)$. Similarly, $EUCLID(a, n) = EUCLID((a \mod n) + cn, n) = EUCLID(n, a \mod n)$. So, by correctness of the Euclidean algorithm,

$$\begin{aligned} gcd(a + kn, n) &= EUCLID(a + kn, n) \\ &= EUCLID(n, a \mod n) \\ &= EUCLID(a, n) \\ &= gcd(a, n) \end{aligned}$$

**Exercise 31.2-4**

**Algorithm 1** ITERATIVE-EUCLID(a,b)

```
1: while $b > 0$ do
2:     $(a, b) = (b, a \mod b)$
3: end while
4: return $a$
```

**Exercise 31.2-5**

We know that for all k, if $b < F_{k+1} < \phi^{k+1}/\sqrt{5}$, then it takes fewer than $k$ steps. If we let $k = \log_\phi b + 1$, then, since $b < \phi^{\log_\phi b + 2}/\sqrt{5} = \frac{\phi^2}{\sqrt{5}} \cdot b$, we have that it only takes $1 + \log_\phi(b)$ steps.

We can improve this bound to $1+\log_\phi(b/\gcd(a,b))$. This is because we know that the algorithm will terminate when it reaches $\gcd(a,b)$. We will emulate the proof of lemma 31.10 to show a slightly different claim that Euclid's algorithm takes $k$ recursive calls, then $a \geq \gcd(a,b)F_{k+2}$ and $b \geq \gcd(a,b)F_{k+!}$. We will similarly do induction on $k$. If it takes one recursive call, and we have $a > b$, we have $a \geq 2\gcd(a,b)$ and $b = \gcd(a,b)$.

Now, suppose it holds for $k-1$, we want to show it holds for $k$. The first call that is made is of EUCLID$(b, a \mod b)$. Since this then only needs $k-1$ recursive calls, we can apply the inductive hypothesis to get that $b \geq \gcd(a,b)F_{k+1}$ and $a \mod b \geq \gcd(a,b)F_k$. Since we had that $a > b$, we have that $a \geq b + (a \mod b) \geq \gcd(a,b)(F_{k+1}+F_k) = \gcd(a,b)F_{k+2}$ completing the induction.

Since we have that we only need $k$ steps so long as $b < \gcd(a,b)F_{k+1} < gcd(a,b)\phi^{k+1}$. we have that $\log_\phi(b/\gcd(a,b)) < k+1$. This is satisfied if we set $k = 1+\log_\phi(b/\gcd(a,b))$

**Exercise 31.2-6**

Since $F_{k+1} \mod F_k = F_{k-1}$ we have $\gcd(F_{k+1},F_k) = \gcd(F_k,F_{k-1})$. Since $\gcd(2,1)=1$ we have that $\gcd(F_{k+1},F_k)=1$ for all $k$. Moreover, since $F_k$ is increasing, we have $\lfloor F_{k+1}/F_k \rfloor = 1$ for all $k$. When we reach the base case of the recursive calls to EXTENDED-EUCLID, we return $(1,1,0)$. The following returns are given by $(d,x,y) = (d',y',x'-y')$. We will show that in general, EXTENDED-EUCLID$(F_{k+1},F_k)$ returns $(1,(-1)^{k+1}F_{k-2},(-1)^kF_{k-1})$. We have already shown $d$ is correct, and handled the base case. Now suppose the claim holds for $k-1$. Then EXTENDED-EUCLID$(F_{k+1},F_k)$ returns $(d',y',x'-y')$ where $d'=1$, $y'=(-1)^{k-1}F_{k-2}$, $x'=(-1)^kF_{k-3}$, so the algorithm returns

$$\begin{aligned}(1,(-1)^{k-1}F_{k-2},(-1)^kF_{k-3}-(-1)^{k-1}F_{k-2}) &= (1,(-1)^{k+1}F_{k-2},(-1)^k(F_{k-3}+F_{k-2})\\ &= (1,(-1)^{k+1}F_{k-2},(-1)^kF_{k-1})\end{aligned}$$

as desired.

**Exercise 31.2-7**

To show it is independent of the order of its arguments, we prove the following swap property, for all $a,b,c$, $\gcd(a,\gcd(b,c)) = \gcd(b,\gcd(a,c))$. By applying these swaps in some order, we can obtain an arbitrary ordering on the variables (the permutation group is generated by the set of adjacent transpositions). Let $a_i$ be the power of the $i$th prime in the prime factor decomposition of $a$, similarly define $b_i$ and $c_i$. Then, we have that

$$\begin{aligned}\gcd(a,\gcd(b,c)) &= \prod_i p_i^{\min(a_i,\min(b_i,c_i))}\\ &= \prod_i p_i^{\min(a_i,b_i,c_i)}\\ &= \prod_i p_i^{\min(b_i,\min(a_i,c_i))}\\ &= \gcd(b,\gcd(a,c))\end{aligned}$$

To find the integers $\{x_i\}$ as described in the problem, we use a similar approach as for EXTENDED-EUCLID.

**Exercise 31.2-8**

From the gcd interpretation given in equation (31.13), it is clear that $\text{lcm}(a_1, a_2) = \frac{a_1 \cdot a_2}{\gcd(a_1,a_2)}$. More generally, $\text{lcm}(a_1, \ldots, a_n) = \frac{a_1 \cdots a_n}{\gcd(\cdots(\gcd(\gcd(a_1,a_2),a_3),\ldots),a_n)}$. We can compute the denominator recursively using the two-argument gcd operation. The algorithm is given below, and runs in $O(n)$ time.

**Algorithm 2** LCM($a_1, \ldots, a_n$)

```
$x = a_1$
$g = a_1$
for $i = 2$ to $n$ do
    $x = x \cdot a_i)$
    $g = \gcd(g, a_i)$
end for
return $x/g$
```

**Exercise 31.2-9**

For two numbers to be relatively prime, we need that the set of primes that occur in both of them are disjoint. Multiplying two numbers results in a number whose set of primes is the union of the two numbers multiplied. So, if we let $p(n)$ denote the set of primes that divide $n$. By testing that $\gcd(n_1n_2, n_3n_4) = \gcd(n_1n_3, n_2n_4) = 1$. We get that $(p(n_1) \cup p(n_2)) \cap (p(n_3) \cup (n_4)) = (p(n_1) \cup p(n_3)) \cap (p(n_2) \cup (n_4)) = \emptyset$. Looking at the first equation, it gets us that $p(n_1) \cap p(n_3) = p(n_1) \cap p(n_4) = p(n_2) \cap p(n_3) = p(n_2) \cap p(n_4) = \emptyset$. The second tells, among other things, that $p(n_1) \cap p(n_2) = p(n_3) \cap p(n_4) = \emptyset$. This tells us that the sets of primes of any two elements are disjoint, so all elements are relatively prime.

A way to view this that generalizes more nicely is to consider the complete graph on $n$ vertices. Then, we select a partition of the vertices into two parts. Then, each of these parts corresponds to the product of all the numbers corresponding to the vertices it contains. We then know that the numbers that any pair of vertices that are in different parts of the partition correspond to will be relatively prime, because we can distribute the intersection across the union of all the prime sets in each partition. Since partitioning into two parts is equivalent to selecting a cut, the problem reduces to selecting $\lg(k)$ cuts of $K_n$ so that every edge is cut by one of the cuts. To do this, first cut the vertex set in as close to half as possible. Then, for each part, we recursively try to cut in in close to half, since the parts are disjoint, we can arbitrarily combine cuts on each of them into a single cut of the original graph. Since the number of time you need to divide $n$ by two to get 1 is $\lfloor \lg(n) \rfloor$, we have that that is the number of times we need to take gcd.

**Exercise 31.3-1**

| $+_4$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 |
| 1 | 1 | 2 | 3 | 0 |
| 2 | 2 | 3 | 0 | 1 |
| 3 | 3 | 0 | 1 | 2 |

| $\cdot_5$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 4 |
| 2 | 2 | 4 | 1 | 3 |
| 3 | 3 | 1 | 4 | 2 |
| 4 | 4 | 3 | 2 | 1 |

Then, we can see that these are equivalent under the mapping $\alpha(0) = 1$, $\alpha(1) = 3$, $\alpha(2) = 4$, $\alpha(3) = 2$.

**Exercise 31.3-2**

The subgroups of $\mathbb{Z}_9$ and $\{0\}$, $\{0, 3, 6\}$, and $\mathbb{Z}_9$ itself. The subgroups of $\mathbb{Z}_{13}^*$ are $\{1\}$, $\{1, 3, 9\}$, $\{1, 3, 4, 9, 10, 12\}$, $\{1, 5, 8, 12\}$, $\{1, 12\}$ and $\mathbb{Z}_{13}^*$ itself.

**Exercise 31.3-3**

Since $S$ was a finite group, every element had a finite order, so, if $a \in S'$, there is some number of times that you can add it to itself so that you get the identity, since adding any two things in $S'$ gets us something in $S'$, we have that $S'$ has the identity element. Associativity is for free because is is a property of the binary operation, no the space that the operation draws it's arguments from. Lastly, we can see that it contains the inverse of every element, because we can just add the element to itself a number of times equal to one less than its order. Then, adding the element to that gets us the identity.

**Exercise 31.3-4**

The only prime divisor of $p^e$ is $p$. From the definition of the Euler phi function, we have

$$\phi(p^e) = p^e\left(1 - \frac{1}{p}\right) = p^{e-1}(p-1).$$

**Exercise 31.3-5**

To see this fact, we need to show that the given function is a bijection. Since the two sets have equal size, we only need to show that the function is onto. To see that it is onto, suppose we want an element that maps to $x$. Since $\mathbb{Z}_n^*$ is a finite Abelian group by theorem 31.13, we can take inverses, in particular, there exists an element $a^{-1}$ so that $aa^{-1} = 1 \mod n$. This means that $f_a(a^{-1}x) = aa^{-1}x \mod n = (aa^{-1} \mod n)(x \mod n) = x \mod n$. Since we can find an element that maps to any element of the range and the sizes of domain and range are the same, the function is a bijection. Any bijection from a set to itself is a permutation by definition.

**Exercise 31.4-1**

First, we run extended Euclid on $35, 50$ and get the result $(5, -7, 10)$. Then, our initial solution is $-7 * 10/5 = -14 = 36$. Since $d = 5$, we have four other solutions, corresponding to adding multiples of $50/5 = 10$. So, we also have that our entire solution set is $x = \{6, 16, 26, 36, 46\}$.

**Exercise 31.4-2**

If $ax \equiv ay \mod n$ then $a(x-y) \equiv 0 \mod n$. Since $\gcd(a,n) = 1$, $n$ doesn't divide $a$ unless $n = 1$, in which case the claim is trivial. By Corollary 31.5, since $n$ divides $a(x-y)$, $n$ divides $x - y$. Thus, $x \equiv y \mod n$. To see that the condition $\gcd(a,n)$ is necessary, let $a = 3$, $n = 6$, $x = 6$, and $y = 2$. Note that $\gcd(a,n) = \gcd(3,6) = 3$. Then $3 \cdot 6 \equiv 3 \cdot 2 \mod 6$, but $6 \neq 2 \mod 6$.

**Exercise 31.4-3**

it will work. It just changes the initial value, and so changes the order in which solutions are output by the program. Since the program outputs all values of $x$ that are congruent to $x_0 \mod n/b$, if we shift the answer by a multiple of $n/b$ by this modification, we will not be changing the set of solutions that the procedure outputs.

**Exercise 31.4-4**

The claim is clear if $a \equiv 0$ since we can just factor out an $x$. For $a \neq 0$, let $g(x) = g_0 + g_1x + \ldots + g_{t-1}x^{t-1}$. In order for $f(x)$ to equal $(x-a)g(x)$ we must have $g_0 = f_0(-a)^{-1}$, $g_i = (f_i - g_{i-1})(-a)^{-1}$ for $1 \leq i \leq t-1$ and

$g_{t-1} = f_t$. Since $p$ is prime, $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$ so every element, including $-a$, has a multiplicative inverse. It is easy to satisfy each of these equations as we go, until we reach the last two, at which point we need to satisfy both $g_{t-1} = (f_{t-1} - g_{t-2})(-a)^{-1}$ and $g_{t-1} = f_t$. More compactly, we need $f_t = (f_{t-1} - g_{t-2})(-a)^{-1}$. We will show that this happens when $a$ is a zero of $f$.

First, we'll proceed inductively to show that for $0 \leq k \leq t-1$ we have $a^{k+1}g_k = -\sum_{i=0}^{k} f_i a^i$. For the base case we have $ag_0 = -f_0$, which holds. Assume the claim holds for $k-1$. Then we have

$$\begin{aligned} a^{k+1}g_k &= a^{k+1}(f_k - g_{k-1})(-a)^{-1} \\ &= -a^k f_k + a^k g_{k-1} \\ &= -a^k f_k - \sum_{i=0}^{k-1} f_i a^i \\ &= \sum_{i=0}^{k} f_i a^i \end{aligned}$$

which completes the induction step. Now we show that we can satisfy the equation given above. It will suffice to show that $-a^t f_t = a^{t-1}(f_{t-1} - g_{t-2})$.

$$\begin{aligned} a^{t-1}(f_{t-1} - g_{t-2}) &= a^{t-1}f_{t-1} - a^{t-1}g_{t-2} \\ &= a^{t-1}f_{t-1} + \sum_{i=0}^{t-2} f_i a^i \\ &= \sum_{i=0}^{t-1} f_i a^i \\ &= -a^t f_t \end{aligned}$$

where the second equality is justified by our earlier claim and the last equality is justified because $a$ is a zero of $f$. Thus, we can find such a $g$.

It is clear that a polynomial of degree 1 can have at most 1 distinct zero modulo $p$ since the equation $x = -a$ has at most 1 solution by Corollary 31.25. Now suppose the claim holds for $t > 1$. Let $f$ be a degree $t+1$ polynomial. If $f$ has no zeros then we're done. Otherwise, let $a$ be a zero of $f$. Write $f(x) = (x-a)g(x)$. Then by the induction hypothesis, since $g$ is of degree $t$, $g$ has at most $t$ distinct zeros modulo $p$. The zeros of $f$ are $a$ and the zeros of $g$, so $f$ has at most $t+1$ distinct zeros modulo $p$.

**Exercise 31.5-1**

These equations can be viewed as a single equation in the ring $\mathbb{Z}_5^+ \times Z_11^+$, in particular $(x_1, x_2) = (4, 5)$. This means that $x$ needs to be the element in $\mathbb{Z}_55^+$ that corresponds to the element $(4, 5)$. To do this, we use the process described in the proof of Theorem 31.27. We have $m_1 = 11$, $m_2 = 5$, $c_1 = 11(11^{-1} \mod 5) = 11$, $c_2 = 5(5^{-1} \mod 11) = 45$. This means that the corresponding solution is $x = 11 \cdot 4 + 45 \cdot 5 \mod 55 = 44 + 225 \mod 55 = 269 \mod 55 = 49 \mod 55$. So, all numbers of the form $49 + 55k$ are a solution.

**Exercise 31.5-2**

Since $9 \cdot 8 \cdot 7 = 504$, we'll be working mod 504. We also have $m_1 = 56$, $m_2 = 63$, and $m_3 = 72$. We compute $c_1 = 56(5) = 280$, $c_2 = 63(7) = 441$, and $c_3 = 72(4) = 288$. Thus, $a = 280 + 2(441) + 3(288) \mod 504 = 10 \mod 504$. Thus, the desired integers $x$ are of the form $x = 10 + 504k$ for $k \in \mathbb{Z}$.

**Exercise 31.5-3**

Suppose that $x \equiv a^{-1} \mod n$. Also, $x_i \equiv x \mod n_i$ and $a_i \equiv a \mod n_i$. What we then want to show is that $x_i \equiv a_i^{-1} \mod n_i$. That is, we want that $a_i x_i \equiv 1 \mod n_i$. To see this, we just use equation 31.30. To get that $ax \mod n$ corresponds to $(a_1x_1 \mod n_1, \dots, a_kx_k \mod n_k)$. This means that 1 corresponds to $(1 \mod n_1, \dots, 1 \mod n_k)$. This is telling us exactly what we needed, in particular, that $a_ix_i \equiv 1 \mod n_i$.

**Exercise 31.5-4**

Let $f(x) = f_0 + f_1x + \ldots + f_dx^d$. Using the correspondence of Theorem 31.27, $f(x) \equiv 0 \mod n$ if and only if $\sum_{i=0}^{d} f_{i_j} x_j^i \equiv 0 \mod n_j$ for $j = 1$ to $k$. The product formula arises by constructing a zero of $f$ via choosing $x_1, x_2, \dots, x_k$ such that $f(x_j) \equiv 0 \mod n_j$ for each $j$, then letting $x$ be the number associated to $(x_1, \dots, x_k)$.

**Exercise 31.6-1**

| element | order |
|---|---|
| 1 | 1 |
| 2 | 10 |
| 3 | 5 |
| 4 | 5 |
| 5 | 5 |
| 6 | 10 |
| 7 | 10 |
| 8 | 10 |
| 9 | 5 |
| 10 | 2 |

The smallest primitive root is 2, and has the following values for $ind_{11,2}(x)$

| $x$ | $\text{ind}_{11,2}(x)$ |
|---|---|
| 1 | 10 |
| 2 | 1 |
| 3 | 8 |
| 4 | 2 |
| 5 | 4 |
| 6 | 9 |
| 7 | 7 |
| 8 | 3 |
| 9 | 6 |
| 10 | 5 |

**Exercise 31.6-2**

To perform modular exponentiation by examining bits from right to left, we'll keep track of the value of $a^{2^i}$ as we go. See below for an implementation:

**Algorithm 3** MODULAR-EXPONENTIATION-R-to-L$(a, b, n)$

```
$c = 0$
$d = 1$
$s = a$
let $\langle b_k, b_{k-1}, \ldots, b_0 \rangle$ be the binary representation of $b$
for $i = 0$ to $k$ do
    if $b_i == 1$ then
        $d = s \cdot d \mod n$
        $c = 2^i + c$
    end if
    $s = s \cdot s \mod n$
end for
return $d$
```

**Exercise 31.6-3**

Since we know $\phi(n)$, we know that $a^{\phi(n)} \equiv 1 \mod n$ by Euler's theorem. This tells us that $a^{-1} = a^{\phi(n)-1}$ because $aa^{-1} =\equiv aa^{\phi(n)-1} \equiv a^{\phi(n)} \equiv 1 \mod n$. Since when we multiply this expression times $a$, we get the identity, it is the inverse of $a$. We can then compute $n^{\phi(n)}$ efficiently, since $\phi(n) < n$, so can be represented without using more bits than was used to represent $n$.

**Exercise 31.7-1**

For the secret key's value of $e$ we compute the inverse of $d = 3 \bmod \phi(n) = 280$. To do this, we first compute $\phi(280) = \phi(2^3)\phi(7)\phi(5) = 4 \cdot 6 \cdot 4 = 96$. Since any number raised to this will be one mod 280, we will raise it to one less than this. So, we compute

$$\begin{aligned}
3^{95} &\equiv 3(3^2)^{47} \\
&\equiv 3(9(9^2)^{23}) \\
&\equiv 3(9(81(81^2)^{11})) \\
&\equiv 3(9(81(121(121^2)^5))) \\
&\equiv 3(9(81(121(81(81^2)^2 \\
&\equiv 3 \cdot 9 \cdot 81 \cdot 121 \cdot 81 \cdot 81 \\
&\equiv 3 \cdot 9 \cdot 121 \\
&\equiv 187 \mod 280
\end{aligned}$$

Now that we know our value of $e$, we compute the encryption of $M = 100$ by computing $100^{187} \mod 319$, which comes out to an encrypted message of 122

**Exercise 31.7-2**

We know $ed = 1 \mod \phi(n)$. Since $d < \phi(n)$ and $e = 3$, we have $3d - 1 = k(p-1)(q-1)$ for $k = 1$ or 2. We have $k = 1$ if $3d - 1 < n$ and $k = 2$ if $3d - 1 > n$. Once we've determined $k$, $p + q = n - (3d-1)/k + 1$, so we can now solve for $p + q$ in time polynomial in $\beta$. Replacing $q - 1$ by $(p+q) - p - 1$ in our earlier equation lets us solve for $p$ in time polynomial in $\beta$ since we need only perform addition, multiplication, and division on numbers bounded by $n$.

**Exercise 31.7-3**

$$\begin{aligned}
P_A(M_1)P_A(M_2) &\equiv M_1^e M_2^e \\
&\equiv (M_1 M_2)^e \\
&\equiv P_A(M_1 M_2) \mod n
\end{aligned}$$

So, if the attacker can correctly decode $\frac{1}{100}$ of the encrypted messages, he does the following. If the message is one that he can decrypt, he is happy, decrypts it and stops. If it is not one that he can decrypt, then, he picks a random element in $\mathbb{Z}_m$, say $x$ encrypts it with the public key, and multiplies that by the encrypted text, he then has a $\frac{1}{100}$ chance to be able to decrypt the new message. He keeps doing this until he can decrypt it. The number of steps needed follows a geometric distribution with a expected value of 100. Once he's stumbled upon one that he could decrypt, he multiplies by the inverses of all the elements that he multiplied by along the way. This recovers the final answer, and also can be done efficiently, since for every $x$, $x^{n-2}$ is $x^{-1}$ by Lagrange's theorem.

**Exercise 31.8-1**
Suppose that we can write $n = \prod_{i=1}^k p_i^{e_i}$, then, by the Chinese remainder theorem , we have that $\mathbb{Z}_n \cong \mathbb{Z}_{p_1^{e_1}} \times \cdots \times \mathbb{Z}_{p_1^{e_1}}$. Since we had that $n$ was not a prime power, we know that $k \geq 2$. This means that we can take the elements $x = (p_1^{e_1} - 1, 1, \dots, 1)$ and $y = (1, p_2^{e_2} - 1, 1, \dots, 1)$. Since multiplication in the product ring is just coordinate wise, we have that the squares of both of these elements is the all ones element in the product ring, which corresponds to 1 in $\mathbb{Z}_n$. Also, since the correspondence from the Chinese remainder theorem was a bijection, since $x$ and $y$ are distinct in the product ring, they correspond to distinct elements in $\mathbb{Z}_n$. Thus, by taking the elements corresponding to $x$ and $y$ under the Chinese remainder theorem bijection, we have that we have found two squareroots of 1 that are not the identity in $\mathbb{Z}_n$. Since there is only one trivial non-identity squareroot in $\mathbb{Z}_n$, one of the two must be non-trivial. It turns out that both are non-trivial, but that's more than the problem is asking.

**Exercise 31.8-2**

Let $c = \gcd(\cdots(\gcd(\gcd(\phi(p_1^{e_1}), \phi(p_2^{e_2})), \phi(p_3^{e_3})), \dots), \phi(p_r^{e_r}))$. Then we have $\lambda(n) = \phi(e_1^{e_1}) \cdots \phi(p_r^{e_r})/c = \phi(n)/c$. Since the lcm is an integer, $\lambda(n) | \phi(n)$.

Suppose $p$ is prime and $p^2 | n$. Since $\phi(p^2) = p^2(1 - \frac{1}{p}) = p^2 - p = p(p-1)$, we have that p must divide $\lambda(n)$. however, since $p$ divides $n$, it cannot divide $n-1$, so we cannot have $\lambda(n) | n-1$.

Now, suppose that is the product of fewer than 3 primes, that is $n = pq$ for some two distinct primes $p < q$. Since both $p$ and $q$ were primes, $\lambda(n) = lcm(\phi(p), \phi(q)) = lcm(p-1, q-1)$. So, mod $q-1$, $\lambda(n) \equiv 0$, however, since $n - 1 = pq - 1 = (q-1)(p) + p - 1$, we have that $n - 1 \equiv p - 1 \mod q - 1$. Since $\lambda(n)$ has a factor of $q-1$ that $n-1$ does not, meaning that $\lambda(n) \not| n-1$.

**Exercise 31.8-3**

First, we prove the following lemma. For any integers $a, b, n$, $\gcd(a, n) \cdot \gcd(b, n) \geq \gcd(ab, n)$. Let $\{p_i\}$ be an enumeration of the primes, then, by Theorem 31.8, there is exactly one set of powers of these primes so that $a = \prod_i p_i^{a_i}$, $b = \prod_i p_i^{b_i}$, and $n = \prod_i p_i^{n_i}$.

$$\gcd(a, n) = \prod_i p_i^{\min(a_i, n_i)}$$
$$\gcd(b, n) = \prod_i p_i^{\min(b_i, n_i)}$$
$$\gcd(ab, n) = \prod_i p_i^{\min(a_i + b_i, n_i)}$$

We combine the first two equations to get:

$$\begin{aligned}\gcd(a,n)\cdot\gcd(b,n) &= \left(\prod_i p_i^{\min(a_i,n_i)}\right)\cdot\left(\prod_i p_i^{\min(b_i,n_i)}\right)\\ &= \prod_i p_i^{\min(a_i,n_i)+\min(b_i,n_i)}\\ &\geq \prod_i p_i^{\min(a_i+b_i,n_i)}\\ &= \gcd(ab,n)\end{aligned}$$

Since $x$ is a non-trivial squareroot, we have that $x^2 \equiv 1 \mod n$, but $x \neq 1$ and $x \neq n-1$. Now, we consider the value of $\gcd(x^2-1,n)$. By theorem 31.9, this is equal to $\gcd(n, x^2-1 \mod n) = \gcd(n, 1-1) = \gcd(n,0) = n$. So, we can then look at the factorization of $x^2-1 = (x+1)(x-1)$ to get that

$$\gcd(x+1,n)\gcd(x-1,n) \geq n$$

However, we know that since $x$ is a nontrivial squareroot, we know that $1 < x < n-1$ so, neither of the factors on the right can be equal to $n$. This means that both of the factors on the right must be nontrivial.

**Exercise 31.9-1**

The Pollard-Rho algorithm would first detect the factor of 73 when it considers the element 84, when we have $x_{12}$ because we then notice that $\gcd(814-84, 1387) = 73$.

**Exercise 31.9-2**

Create an array $A$ of length $n$. For each $x_i$, if $x_i = j$, store $i$ in $A[j]$. If $j$ is the first position of $A$ which must have its entry rewritten, set $t$ to be the entry originally stored in that spot. Then count how many additional $x_i$'s must be computed until $x_i = j$ again. This is the value of $u$. The running time is $\Theta(t+u)$.

**Exercise 31.9-3**

Assuming that $p^e$ divides $n$, by the same analysis as sin the chapter, it will take time $\Theta(p^{e/2})$. To see this, we look at what is happening to the sequence mod $p^n$.

$$
\begin{aligned}
x'_{i+1} &= x_{i+1} \mod p^e \\
&= f_n(x_i) \mod p^e \\
&= ((x^2-1) \mod n) \mod p^e \\
&= (x^2-1) \mod p^e \\
&= (x'_i)^2 - 1 \mod p^e \\
&= f_{p^e}(x'_i)
\end{aligned}
$$

So, we again are having the birthday paradox going on, but, instead of hoping for a repeat from a set of size $p$, we are looking at all the equivalence classes mod $p^e$ which has size $p^e$, so, we have that the expected number of steps before getting a repeat in that size set is just the squareroot of its size, which is $\Theta(\sqrt{p^e}) = \Theta(p^{e/2})$.

**Exercise 31.9-4**

Taking the idea suggested in the exercise statement, suppose that we store the product of a batch of B consecutive values of $x_i$. We should also recall that the number of recursive calls that GCD makes is on the order of the log of the smaller of the two numbers. Since we don't have control over what value between 0 and n is taken by $x_i$, we should consider that our product of consecutive terms is larger than $n$. So, we will make $O(\beta)$ many recursive calls. We will compute The $B$ many values of $x_i$ all before computing GCD, or checking against $k$. However, this means that we need to change line Line 11 of POLLARD-RHO to checking ig $i > k$, and setting y equal to $x_k$, which will still be remembered as it has to of been in the last $B$ computed. The part that needs justification is that

$$\gcd\left(\prod_{k=0}^{B-1} y - x_{i+k}, n\right)$$

will be non-trivial iff one of $\gcd(y - x_{i+k}, n)$ are nontrivial. We need show that for all $a, b$, we have $1 < \gcd(ab, n)$ if and only if $1 < \gcd(a, n)$ or $1 < \gcd(b, n)$. Once we have that then we will be able to easily show what we originally needed by induction. For the "if" direction, suppose that, without loss of generality, $d = \gcd(a, n)$ and $1 < d < n$. This means that $d|n$, and $d|a$, but $a|ab$, so, by 31.1-3, we have $d|ab$. Any divisor is a lower bound for the gcd, so we have that $1 < \gcd(ab, n)$. For the "only if" direction, suppose that we have $\gcd(ab, n) = d > 1$. Then, there is some prime number dividing $d$. This prime has to belong to either $a$ or $b$. So, we have a positive divisor for one of $a$ or $b$. Since the prime also divides $n$, we again get a lower bound on either $\gcd(a, n)$ or $\gcd(b, n)$. In picking $B$, we need to balance the cost of computing gcd more times with the cost of computing it on larger number. Since we want to think of $x_i$ being (pseudo) randomly distributed, the only control we have on the value of $y - x_i$ is that it is at most $n$. This means that the value of $P$ in our algorithm can be as large as $n^B$. There was also some extra cost in multiplying together the values of $x_i$. The fastest we can multiply to $m$ bit numbers is $O(m \lg(m))$, so, to compute $P$, we take up time $\sum_m O(m \lg(m)) = O(Bm \lg(m))$. We would then have to balance this cost against the fact that regardless of $B$, there will be at most $O(\lg(n))$ many recursive calls to compute the gcd. Another consideration, not on runtime, but on space complexity is that we need to store the $b$ most recent values of $x_i$, so that when we try to set $y$ to a new value, the $x_k$ that it is referring to is still stored.

**Algorithm 4** BATCH-POLLARD-RHO(n,B)

```
$i = 1$
$x_1 = \text{RANDOM}(0, n-1)$
$y = x_1$
$k = B$
while TRUE do
    $P = 1$
    for $k = 1$ to $k = B$ do
        $x_{i+k} = (x_{i+k-1}^2 - 1) \mod n$
        $P = P \cdot (y - x_{i+k})$
    end for
    $i = i + B$
    $d = \gcd(P, n)$
    if $d \neq 1$ and $d \neq n$ then
        print $d$
    end if
    if $i >= k$ then
        $y = x_k$
        $k = 2k$
    end if
end while
```

**Problem 31-1**

a. If $a$ and $b$ are both even, then we can write them as $a = 2(a/2)$ and $b = 2(b/2)$ where both factors in each are integers. This means that, by Corollary 31.4, $\gcd(a, b) = 2 \gcd(a/2, b/2)$.

b. If $a$ is odd, and $b$ is even, then we can write $b = 2(b/2)$, where $b/2$ is an integer, so, since we know that 2 does not divide $a$, the factor of two that is in $b$ cannot be part of the gcd of the two numbers. This means that we have $\gcd(a, b) = \gcd(a, b/2)$. More formally, suppose that $d = \gcd(a, b)$. Since $d$ is a common divisor, it must divide $a$, and so, it must not have any even factors. This means that it also divides $a$ and $b/2$. This means that $\gcd(a, b) \leq \gcd(a, b/2)$. To see the reverse, suppose that $d' = \gcd(a, b/2)$, then it is also a divisor of $a$ and $b$, since we can just double whatever we need to multiply it by to get $b/2$. Since we have inequalities both ways, we have equality.

c. If $a$ and $b$ are both odd, then, first, in analog to theorem 31.9, we show that $\gcd(a, b) = \gcd(a - b, b)$. Let $d$ and $d'$ be the gcd's on the left and right respectively. Then, we have that there exists $n_1, n_2$ so that $n_1a + n_2b = d$, but then, we can rewrite to get $n_1(a - b) + (n_1 + n_2)b = d$. This gets us $d \geq d'$. To see the reverse, let $n_1', n_2'$ so that $n_1'(a - b) + n_2'b = d'$. We rewrite to get $n_1'a + (n_2' - n_1')b = d'$, so we have $d' \geq d$. This means that $\gcd(a, b) = \gcd(a - b, b) = \gcd(b, a - b)$. From there, we fall into the case of part b. This is because the first argument is odd, and the second is the difference of two odd numbers, hence is even. This means we can halve the second argument without changing the quantity. So, $\gcd(a, b) = \gcd(b, (a - b)/2) = \gcd((a - b)/2, b)$.

d. See the algorithm BINARY-GCD(a,b)

**Algorithm 5** BINARY-GCD(a,b)

```
if $a \mod 2 \equiv 1$ then
    if $b \mod 2 \equiv 1$ then
        return BINARY-GCD($(a - b)/2, b$)
    else
        return BINARY-GCD($a, b/2$)
    end if
else
    if $b \mod 2 \equiv 1$ then
        return BINARY-GCD($a/2, b$)
    else
        return 2·BINARY-GCD($a/2, b/2$)
    end if
end if
```

**Problem 31-2**

a. We can imagine first writing $a$ and $b$ in their binary representations, and then performing long division as usual on these numbers. Each time we compute a term of the quotient we need to perform a multiplication of $a$ and that term, which takes $\lg b$ bit operations, followed by a subtraction from the first $b$ terms of $a$ which takes $\lg b$ bit operations. We repeat this once for each digit of the quotient which we compute, until the remainder is smaller than $b$. There are $\lg q$ bits in the quotient, plus the final check of the remainder. Since each requires $\lg b$ bit operations to perform the multiplication and subtraction, the method requires $O((1 + \lg q) \lg b)$ bit operations.

b. The reduction requires us to compute $a \mod b$. By carrying out ordinary "paper and pencil" long division we can compute the remainder. The time to do this, by part a, is bounded by $k(1+\lg q)(\lg b)$ for some constant $k$. In the worst case, $q$ has $\lg a - \lg b$ bits. Thus, we get the bound $k(1+\lg a-\lg b)(\lg b)$. Next, we compute $\mu(a,b)-\mu(b, a \mod b) = (1+\lg a)(1+\lg b)-(1+\lg b)(1+\lg(a \mod b)$. This is smallest when $a \mod b$ is small, so we have a lower bound of $(1+\lg a)(1+\lg b)-(1+\lg b)$. Assuming $\lg b \geq 1$, we can take $c = k$ to obtain the desired inequality.

c. As shown in part (b), EUCLID takes at most $c(\mu(a,b) - \mu(b, amodb))$ operations on the first recursive call, at most $c(\mu(b, a \mod b) - \mu(a \mod b, b \mod (a \mod b))$ operations on the second recursive call, and so on. Summing over all recursive calls gives a telescoping series, resulting in $c\mu(a,b)+O(1) = O(\mu(a,b))$. When applies to two $\beta$-bit inputs the runtime is $O(\mu(a,b)) = O((1+\beta)(1+\beta)) = O(\beta^2)$.

**Problem 31-3**

a. Mirroring the proof in chapter 27, we first notice that in order to solve $FIB(n)$, we need to compute $FIB(n-1)$ and $FIB(n-2)$. This means that the recurrence it satisfies is

$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$

We find it's solution using the substitution method. Suppose that the $\Theta(1)$ is bounded above by $c_2$ and bounded below by $c_1$. Then, we'll inductively assume that $T(k) \leq cF_k - c_2 k$ for $k < n$. Then,

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ &\leq cF_{n-1} - c_2(n-1) + cF_{n-2} - c_2(n-2) + c_2 \\ &= cF_n - c_2 n + (4-n)c_2 \\ &\leq cF_n - c_2 n \end{aligned}$$

Where the last inequality only holds if we have that $n \geq 4$,but since small values can just be absorbed into the constants, we are allowed to assume this.

To show that $T \in \Omega(F_n)$, we again use the substitution method. Suppose that $T(k) \geq cF_k + c_1 k$ for $k < n$. Then.

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ &\geq cF_{n-1} + c_1(n-1) + cF_{n-2} + c_1(n-2) + c_1 \\ &= cF_n + c_1 n + (n-4)c_1 \\ &\geq cF_n - c_1 n \end{aligned}$$

Again, this last inequality only holds if we have $n \geq 4$, but small cases can be absorbed into the constants, we may assume that $n \geq 4$.

b. This problem is the same as exercise 15.1-5.

c. For this problem, we assume that all integer multiplications and additions can be done in unit time. We will show first that

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^k = \begin{pmatrix} F_{k-1} & F_k \\ F_k & F_{k+1} \end{pmatrix}$$

Where we start We will proceed by induction. Then,

$$\begin{aligned}
\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{k+1} &= \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^k \\
&= \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} F_{k-1} & F_k \\ F_k & F_{k+1} \end{pmatrix} \\
&= \begin{pmatrix} F_k & F_{k-1}+F_k \\ F_{k-1}+F_k & F_k+F_{k+1} \end{pmatrix} \\
&= \begin{pmatrix} F_k & F_{k+1} \\ F_{k+1} & F_{k+2} \end{pmatrix}
\end{aligned}$$

completing the induction. Then, we just show that we can compute the given matrix to the power $n-2$ in time $O(\lg(n))$, and look at it's bottom right entry. We will use a technique similar to section 31.6, that is, we will use the idea of iterated squaring in order to obtain high powers quickly. First, we should note that using 8 multiplications and 4 additions, we can multiply any two square matrices. This means that matrix multiplications can be done in constant time, so, we only need to bound the number of those in our algorithm. Run the algorithm MATRIX-POW(A,n-2) and extract the bottom left argument. We can see that this algorithm only takes time $O(\lg(n))$ because in each step, we are halving the value of $n$, and within each step, we are only performing a constant amount of work, so the solution to

$$T(n) = T(n/2) + \Theta(1)$$

is $O(\lg(n))$ by the master theorem.

**Algorithm 6** MATRIX-POW(A,n)

```
if $n\%2 = 1$ then
    return $A\cdot$MATRIX-POW($A^2, \frac{n-1}{2}$)
else
    return MATRIX-POW($A^2, n/2$)
end if
```

d. Here, we replace the assumption of unit time additions and multiplications with having it take time $\Theta(\beta)$ to add and $\Theta(\beta^2)$ to multiply two $\beta$ bit numbers. For the naive approach, We are adding a number which is growing exponentially each time, so, the recurrence becomes

$$T(n) = T(n-1) + T(n-2) + \Theta(n)$$

Which has the same solution $2^n$. Which can be seen by a substitution argument. Suppose that $T(k) \leq c2^k$ for $k < n$. Then,

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + \Theta(\lg(n)) \\ &\leq c(\frac{1}{2} + \frac{1}{4})2^n + \Theta(\lg(n)) \\ &= c2^n - c2^{n-2} + \Theta(\lg(n)) \\ &\leq c2^k \end{aligned}$$

Since we had that it was $\Omega(2^n)$ in the case that the term we added was $\Theta(1)$, and we have upped this term to $\Theta(\lg(n))$, we still have that $T(n) \in \Omega(2^n)$. This means that $T(n) \in \Theta(2^n)$.

Now, considering the memoized version. We have that our solution has to satisfy the recurrence

$$M(n) = M(n-1) + \Theta(n)$$

This clearly has a solution of $\sum_{i=2}^{n} n \in \Theta(n^2)$ by equation (A.11) where it is trivial to obtain $\int x\ dx$.

Finally, we reanalyze our solution to part (c). For this, we have that we are performing a constant number of both additions and multiplications. This means that, because we are multiplying numbers that have value on the order of $\phi^n$, hence have order $n$ bits, our recurrence becomes

$$P(n) = P(n/2) + \Theta(n^2)$$

Which has a solution of $\Theta(n^2)$.

Though it is not asked for, we can compute Fibonacci in time only $\Theta(n\lg(n))$ because multiplying integers with $\beta$ bits can be done in time $\beta\lg(\beta)$ using the fast Fourier transform methods of the previous chapter.

**Problem 31-4**

a. Since $p$ is prime, Theorem 31.32 implies that $\mathbb{Z}_p^*$ is cyclic, so it has a generator $g$. Thus, $g^2, g^4, \ldots, g^{p-1}$ are all distinct. Moreover, each one is clearly a quadratic residue so there are at least $(p-1)/2$ residues. Now suppose that $a$ is a quadratic residue and $a = g^{2k+i}$ for some $i$. Then we must also have $a = x^2$ for some $x$, and $x = g^m$ for some $m$, since $g$ is a generator. Thus, $g^{2m} = g^{2k+i}$. By the discrete logarithm theorem we must have $2m = 2k+1 \mod \phi(p)$. However, this is impossible since $\phi(p) = p-1$ which is even, but $2m$ and $2k+1$ differ by an odd amount. Thus, precisely the elements of the form $g^{2i}$ for $i = 1, 2, \ldots, (p-1)/2$ are quadratic residues.

b. If $a$ is a quadratic residue modulo $p$ then there exists $x$ such that $a^{(p-1)/2} = (x^2)^{(p-1)/2} = x^{p-1} = 1 = \left(\frac{a}{p}\right)$. On the other hand, suppose $a$ is not a quadratic residue. Then $a = g^{2i+1}$ for some $i$ and $a^{(p-1)/2} = \left(g^{2i+1}\right)^{(p-1)/2} = \left(g^{(p-1)/2}\right)^{2i+1} = (-1)^{2i+1} = -1 \mod p$. To see why $g^{(p-1)/2} = -1$, recall that Theorem 31.34 tells us that $g = \pm 1$. Since $g$ is a generator, powers of $g$ are distinct. Since $g^{p-1} = 1$, we must have $g^{(p-1)/2} = -1$.

   To determine whether a given number $a$ is a quadratic residue modulo $p$, we simply compute $a^{(p-1)/2} \mod p$ and check if it is 1 or -1. We can do this using the MODULAR-EXPONENTIATION function, and the number of bit operations is $O((\lg p)^3)$.

c. If $p = 4k+3$ and $a$ is a quadratic residue then $a^{2k+1} = 1 \mod p$. Then we have $(a^{k+1})^2 = a^{2k+2} = aa^{2k+1} = a$, so $a^{k+1}$ is a square root of $a$. To find the square root, we use the MODULAR-EXPONENTIATION algorithm which has $O((\lg p)^3)$ bit operations.

d. Run the algorithm of part b repeatedly until a non-quadratic residue is found. Since only half the elements of $\mathbb{Z}_p^*$ are residues, after $k$ runs this approach will find a nonresidue with probability $1 - 2^{-k}$. The expected number of runs is $\sum_{k=1}^{\infty} k \cdot 2^{-k} = 2$, so the expected number of bit operations is $O((\lg p)^3)$.