Chapter 2

Cryptographers' favorites

This chapter and the following chapter are copied verbatim from the "The Laws of Cryptography with Java Code", [Wag] with permission from Prof Neal Wagner. The book is well worth reading and contains a lot of information that is relevant to this course. You can find the book at

http://www.cs.utsa.edu/~wagner/lawsbookcolor/laws.pdf

2.1 Exclusive-Or

The function known as **exclusive-or** is also represented as `xor` or a plus sign in a circle, $\oplus$. The expression $a \oplus b$ means either a or b *but not both*. Ordinary *inclusive-or* in mathematics means either one or the other *or both*. The exclusive-or function is available in C / C++ / Java for bit strings as a hat character: ^. (Be careful: the hat character is often used to mean exponentiation, but Java, C, and C++ have no exponentiation operator. The hat character also sometimes designates a control character.) In Java ^ also works as exclusive-or for boolean type.

Law XOR-1:
The cryptographer's favorite function is *Exclusive-Or*.

Exclusive-or comes up constantly in cryptography. For example, the exclusive-or of a pseudo-random bit stream with a message bit stream is one simple form of encryption. Decryption is then just the exclusive-or of the resulting stream with the same pseudo-random stream. (See Chapter 10 in [Wag]: The One-Time Pad.)

Recall that the boolean constant **`true`** is often written as a **`1`** and **`false`** as a **`0`**. Exclusive-or is the same as *addition mod 2*, which means ordinary addition, followed by taking the remainder on division by 2.

For single bits 0 and 1, Table 2.1 gives the definition of their exclusive-or.

Exclusive-Or		
a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.1: Definition of Exclusive-Or

The exclusive-or function has many interesting properties, including the following, which hold for any bit values or bit strings a, b, and c:

$a \oplus a = 0$
$a \oplus 0 = a$
$a \oplus 1 = \sim a$, where $\sim$ is bit complement.
$a \oplus b = b \oplus a$ (commutativity)
$a \oplus (b \oplus c) = (a \oplus b) \oplus c$ (associativity)
$a \oplus a \oplus a = a$
if $a \oplus b = c$, then $c \oplus b = a$ and $c \oplus a = b$.

Beginning programmers learn how to exchange the values in two variables **`a`** and **`b`**, using a third temporary variable **`temp`** and the assignment operator **`=`** :

```
temp = a;
a = b;
b = temp;
```

The same result can be accomplished using **`xor`** without an extra temporary location, regarding **`a`** and **`b`** as bit strings. (A Java program that demonstrates interchange using exclusive-or is on page 161 of [Wag]).

```
a = a xor b;
b = a xor b;
a = a xor b;
```

2.2 Logarithms

By definition, $y = \log_b x$ means the same as $b^y = x$. One says: "y is the logarithm of x to base b," or "y is the log base b of x." Stated another way, $\log_b x$ (also known as y) is the *exponent* you raise b to in order to get x. Thus $b^{(\log_b x)} = x$. In more mathematical terms, the logarithm is the inverse function of the exponential.

Law LOG-1:
The cryptographer's favorite logarithm is *log base 2*.

One uses logs base 2 in cryptography (as well as in most of computer science) because of the emphasis on binary numbers in these fields.

So $y = \log_2 x$ means the same as $2^y = x$, and a logarithm base 2 of x is the exponent you raise 2 to in order to get x. In symbols: if $y = \log_2 x$, then $x = 2^y = 2^{\log_2 x}$. In particular $2^{10} = 1024$ means the same as $\log_2 1024 = 10$. Notice that $2^y > 0$ for all y, and inversely $\log_2 x$ is not defined for $x \leq 0$.

Here are several other formulas involving logarithms:

$$\begin{aligned}
&\log_2(ab) = \log_2 a + \log_2 b, \text{ for all } a, b > 0\\
&\log_2(a/b) = \log_2 a - \log_2 b, \text{ for all } a, b > 0\\
&\log_2(1/a) = \log_2(a^{-1}) = -\log_2 a, \text{ for all } a > 0\\
&\log_2(a^r) = r \log_2 a, \text{ for all } a > 0, r\\
&\log_2(a + b) = \text{(Oops! No simple formula for this.)}
\end{aligned}$$

Table 2.2 gives a few examples of logs base 2.

Some calculators, as well as languages like Java, do not directly support logs base 2. Java does not even support logs base 10, but only logs base e, the "natural" log. However, a log base 2 is just a fixed constant times a natural log, so they are easy to calculate if you know the "magic" constant. The formulas are:

$$\begin{aligned}
\log_2 x &= \log_e x / \log_e 2, \text{ (mathematics)}\\
&= \texttt{Math.log(x)/Math.log(2.0);} \text{ (Java).}
\end{aligned}$$

The magic constant is: $\log_e 2 = 0.69314\ 71805\ 59945\ 30941\ 72321$, or $1/\log_e 2 = 1.44269\ 50408\ 88963\ 40735\ 99246$. (Similarly, $\log_2 x = \log_{10} x / \log_{10} 2$, and $\log_{10} 2 = 0.30102999566398114$.)

A Java program that demonstrates these formulas is found on page 162 of [Wag].

Logarithms base 2	
$x = 2^y = 2^{\log_2 x}$	$y = \log_2 x$
$1,073,741,824$	30
$1,048,576$	20
$1,024$	10
8	3
4	2
2	1
1	0
$1/2$	-1
$1/4$	-2
$1/8$	-3
$1/1,024$	-10
0	$-\infty$
< 0	undefined

Table 2.2: Logarithms base 2

Here is a proof of the above formula:

$2^y = x$, or $y = \log_2 x$ (then take $\log_e$ of each side)
$\log_e(2^y) = \log_e x$ (then use properties of logarithms)
$y \log_e 2 = \log_e x$ (then solve for y)
$y = \log_e x / \log_e 2$ (then substitute $\log_2 x$ for y)
$\log_2 x = \log_e x / \log_e 2$.

Law LOG-2:
The log base 2 of an integer x tells how many bits it takes to represent x in binary.

Thus $\log_2 10000 = 13.28771238$, so it takes 14 bits to represent 10000 in binary. (In fact, $10000_{10} = 10011100010000_2$.) Exact powers of 2 are a special case: $\log_2 1024 = 10$, but it takes 11 bits to represent 1024 in binary, as 10000000000_2.

Similarly, $\log_{10}(x)$ gives the number of decimal digits needed to represent x.

2.3 Groups

A *group* is a set of *group elements* with a *binary operation* for combining any two elements to get a unique third element. If one denotes the group operation by $\#$, then the above says that

for any group elements a and b, $a\#b$ is defined and is also a group element. Groups are also *associative*, meaning that $a\#(b\#c) = (a\#b)\#c$, for any group elements a, b, and c. There has to be an *identity element* e satisfying $a\#e = e\#a = a$ for any group element a. Finally, any element a must have an *inverse* a' satisfying $a\#a' = a'\#a = e$.

If $a\#b = b\#a$ for all group elements a and b, the group is *commutative*. Otherwise it is *non-commutative*. Notice that even in a non-commutative group, $a\#b = b\#a$ might sometimes be true — for example if a or b is the identity.

A group with only finitely many elements is called *finite*; otherwise it is *infinite*.

Examples:

1. The *integers* (all whole numbers, including 0 and negative numbers) form a group using ordinary addition. The identity is 0 and the inverse of a is $-a$. This is an infinite commutative group.

2. The *positive rationals* (all positive fractions, including all positive integers) form a group if ordinary multiplication is the operation. The identity is 1 and the inverse of r is $1/r = r^{-1}$. This is another infinite commutative group.

3. The *integers mod n* form a group for any integer $n > 0$. This group is often denoted Z_n. Here the elements are $0, 1, 2, \ldots, n-1$ and the operation is addition followed by remainder on division by n. The identity is 0 and the inverse of a is $n - a$ (except for 0 which is its own inverse). This is a finite commutative group.

4. For an example of a non-commutative group, consider 2-by-2 non-singular matrices of real numbers (or rationals), where the operation is matrix multiplication:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

   Here a, b, c, and d are real numbers (or rationals) and $ad - bc$ must be non-zero (non-singular matrices). The operation is matrix multiplication. The above matrix has inverse

$$\frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

   and the identity is

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

   This is an infinite non-commutative group.

5. The chapter on decimal numbers in [Wag] gives an interesting and useful example of a finite *non*-commutative group: the *dihedral* group with ten elements.

Law GROUP-1:
The cryptographer's favorite group is the *integers mod n*, Z_n.

In the special case of $n = 10$, the operation of addition in Z_{10} can be defined by $(x + y) \mod 10$, that is, divide by 10 and take the remainder. Table 2.3 shows how one can also use an *addition table* to define the integers modulo 10:

+	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	0
2	2	3	4	5	6	7	8	9	0	1
3	3	4	5	6	7	8	9	0	1	2
4	4	5	6	7	8	9	0	1	2	3
5	5	6	7	8	9	0	1	2	3	4
6	6	7	8	9	0	1	2	3	4	5
7	7	8	9	0	1	2	3	4	5	6
8	8	9	0	1	2	3	4	5	6	7
9	9	0	1	2	3	4	5	6	7	8

Table 2.3: Addition in the integers mod 10, Z_{10}.

2.4 Fields

A *field* is an object with a lot of structure, which this section will only outline. A field has two operations, call them $+$ and (though they will not necessarily be ordinary addition and multiplication). Using $+$, all the elements of the field form a commutative group. Denote the identity of this group by 0 and denote the inverse of a by $-a$. Using , all the elements of the field except 0 must form another commutative group with identity denoted 1 and inverse of a denoted by a^{-1}. (The element 0 has no inverse under .) There is also the *distributive identity*, linking $+$ and : $a*(b+c) = (a*b)+(a*c)$, for all field elements a, b, and c. Finally, one has to exclude *divisors of zero*, that is, non-zero elements whose product is zero. This is equivalent to the following cancellation property: if c is not zero and $a * c = b * c$, then $a = b$.

Examples:

1. Consider the *rational numbers* (fractions) Q, or the *real numbers* $\mathbb{R}$, or the *complex numbers* $\mathbb{C}$, using ordinary addition and multiplication (extended in the last case to the complex numbers). These are all infinite fields.

2. Consider the *integers mod p*, denoted Z_p, where p is a prime number (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ...). Regard this as a group using + (ordinary addition followed by remainder on division by p). The elements with 0 left out form a group under (ordinary multiplication followed by remainder on division by p). Here the identity is clearly 1, but the inverse of a non-zero element a is not obvious. In Java, the inverse must be an element x satisfying $(x*a)\%p == 1$. It is always possible to find the unique element x, using an algorithm from number theory known as the *extended Euclidean algorithm*. This is the topic in the next chapter, but in brief: because p is prime and a is non-zero, the greatest common divisor of p and a is 1. Then the extended Euclidean algorithm gives ordinary integers x and y satisfying $x * a + y * p = 1$, or $x * a = 1 - y * p$, and this says that if you divide $x * a$ by p, you get remainder 1, so this x is the inverse of a. (As an integer, x might be negative, and in this case one must add p to it to get an element of Z_p.)

Law FIELD-1:
The cryptographer's favorite field is the *integers mod p*, denoted Z_p, where *p* is a prime number.

The above field is the only one with p elements. In other words, the field is unique up to renaming its elements, meaning that one can always use a different set of symbols to represent the elements of the field, but it will still be essentially the same.

There is also a unique finite field with p^n elements for any integer $n > 1$, denoted $GF(p^n)$. Particularly useful in cryptography is the special case with $p = 2$, that is, with 2^n elements for $n > 1$. The case $2^8 = 256$ is used, for example, in the new U.S. Advanced Encryption Standard (AES). It is more difficult to describe than the field Z_p. The chapter about multiplication for the AES will describe this field in more detail, but here are some of its properties in brief for now: It has 256 elements, represented as all possible strings of 8 bits. Addition in the field is just the same as bitwise exclusive-or (or bitwise addition mod 2). The zero element is 00000000, and the identity element is 00000001. So far, so good, but multiplication is more problematic: one has to regard an element as a degree 7 polynomial with coefficients in the field Z_2 (just a 0 or a 1) and use a special version of multiplication of these polynomials. The details will come in the later chapter on the AES.

Law FIELD-2:
The cryptographer's *other* favorite field is *GF(2^n)*.

2.5 Fermat's Theorem

In cryptography, one often wants to raise a number to a power, modulo another number. For the integers mod p where p is a prime (denoted Z_p), there is a result know as Fermat's Theorem, discovered by the 17th century French mathematician Pierre de Fermat, 1601-1665.

Theorem (Fermat): If p is a prime and a is any non-zero number less than p, then

$$a^{p-1} \mod p = 1$$

Law FERMAT-1:
The cryptographer's favorite theorem is Fermat's Theorem.

Table 2.4 illustrates Fermat's Theorem for $p = 13$. Notice below that the value is always 1 by the time the power gets to 12, but sometimes the value gets to 1 earlier. The initial run up to the 1 value is shown in bold italic in the table. The lengths of these runs are always numbers that divide evenly into 12, that is, 2, 3, 4, 6, or 12. A value of a for which the whole row is bold italic is called a *generator*. In this case 2, 6, 7, and 11 are generators.

p	**a**	a^1	a^2	a^3	a^4	a^5	a^6	a^7	a^8	a^9	a^{10}	a^{11}	a^{12}
13	2	***2***	***4***	***8***	***3***	***6***	***12***	***11***	***9***	***5***	***10***	***7***	***1***
13	3	***3***	***9***	***1***	*3*	*9*	*1*	*3*	*9*	*1*	*3*	*9*	*1*
13	4	***4***	***3***	***12***	***9***	***10***	***1***	*4*	*3*	*12*	*9*	*10*	*1*
13	5	***5***	***12***	***8***	***1***	*5*	*12*	*8*	*1*	*5*	*12*	*8*	*1*
13	6	***6***	***10***	***8***	***9***	***2***	***12***	***7***	***3***	***5***	***4***	***11***	***1***
13	7	***7***	***10***	***5***	***9***	***11***	***12***	***6***	***3***	***8***	***4***	***2***	***1***
13	8	***8***	***12***	***5***	***1***	*8*	*12*	*5*	*1*	*8*	*12*	*5*	*1*
13	9	***9***	***3***	***1***	*9*	*3*	*1*	*9*	*3*	*1*	*9*	*3*	*1*
13	10	***10***	***9***	***12***	***3***	***4***	***1***	*10*	*9*	*12*	*3*	*4*	*1*
13	11	***11***	***4***	***5***	***3***	***7***	***12***	***2***	***9***	***8***	***10***	***6***	***1***
13	12	***12***	***1***	*12*	*1*	*12*	*1*	*12*	*1*	*12*	*1*	*12*	*1*

Table 2.4: Fermat's theorem for $p = 13$

Because a to a power mod p always starts repeating after the power reaches $p-1$, one can reduce the power mod $p-1$ and still get the same answer. Thus no matter how big the power x might be,

$$a^x \mod p = a^{x \bmod (p-1)} \mod p.$$

Thus modulo p in the expression requires modulo $p-1$ in the exponent. (Naively, one might expect to reduce the exponent mod p, but this is not correct.) So, for example, if $p = 13$ as above, then

$$a^{29} \mod 13 = a^{29 \bmod 12} \mod 13 = a^5 \mod 13.$$

The Swiss mathematician Leonhard Euler (1707-1783) discovered a generalization of Fermat's Theorem which will later be useful in the discussion of the RSA cryptosystem.

Theorem (Euler): If n is any positive integer and a is any positive integer less than n with no divisors in common with n, then

$$a^{\phi(n)} \mod n = 1,$$

where $\phi(n)$ is the *Euler phi function*:

$$\phi(n) = n(1 - 1/p_1) \ldots (1 - 1/p_m),$$

and $p_1, \ldots, p_m$ are all the prime numbers that divide evenly into n, including n itself in case it is a prime.

p	**a**	a^1	a^2	a^3	a^4	a^5	a^6	a^7	a^8	a^9	a^{10}	a^{11}	a^{12}	a^{13}	a^{14}
15	2	***2***	***4***	***8***	***1***	*2*	*4*	*8*	*1*	*2*	*4*	*8*	*1*	*2*	*4*
15	3	***3***	***9***	***12***	***6***	***3***	***9***	***12***	***6***	***3***	***9***	***12***	***6***	***3***	***9***
15	4	***4***	***1***	*4*	*1*	*4*	*1*	*4*	*1*	*4*	*1*	*4*	*1*	*4*	*1*
15	5	***5***	***10***	***5***	***10***	***5***	***10***	***5***	***10***	***5***	***10***	***5***	***10***	***5***	***10***
15	6	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***
15	7	***7***	***4***	***13***	***1***	*7*	*4*	*13*	*1*	*7*	*4*	*13*	*1*	*7*	*4*
15	8	***8***	***4***	***2***	***1***	*8*	*4*	*2*	*1*	*8*	*4*	*2*	*1*	*8*	*4*
15	9	***9***	***6***	***9***	***6***	***9***	***6***	***9***	***6***	***9***	***6***	***9***	***6***	***9***	***6***
15	10	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***
15	11	***11***	***1***	*11*	*1*	*11*	*1*	*11*	*1*	*11*	*1*	*11*	*1*	*11*	*1*
15	12	***12***	***9***	***3***	***6***	***12***	***9***	***3***	***6***	***12***	***9***	***3***	***6***	***12***	***9***
15	13	***13***	***4***	***7***	***1***	*13*	*4*	*7*	*1*	*13*	*4*	*7*	*1*	*13*	*4*
15	14	***14***	***1***	*14*	*1*	*14*	*1*	*14*	*1*	*14*	*1*	*14*	*1*	*14*	*1*

Table 2.5: Euler's theorem for $n = 15$ and $\phi(n) = 8$

If n is a prime, then using the formula, $\phi(n) = n(1-1/n) = n(\frac{n-1}{n}) = n-1$, so Euler's result is a special case of Fermat's. Another special case needed for the RSA cryptosystem comes when the modulus is a product of two primes: $n = pq$. Then $\phi(n) = n(1-1/p)(1-1/q) = (p-1)(q-1)$. Table 2.5 illustrates Euler's theorem for $n = 15 = 3 \cdot 5$, with $\phi(15) = 15 \cdot (1-1/3) \cdot (1-1/5) = (3-1) \cdot (5-1) = 8$. Notice here that a 1 is reached when the power gets to 8 (actually in this simple case when the power gets to 2 or 4), but only for numbers with no divisors in common with 15. For other base numbers, the value never gets to 1.

Tables 2.5 and 2.4 were generated by the Java program on page 163 of [Wag].

In a way similar to Fermat's Theorem, arithmetic in the exponent is taken mod $\phi(n)$, so that, assuming a has no divisors in common with n,

$$a^x \mod n = a^{x \bmod \phi(n)} \mod p.$$

If $n = 15$ as above, then $\phi(n) = 8$, and if neither 3 nor 5 divides evenly into a, then $\phi(n) = 8$. Thus for example,

$$a^{28} \mod 15 = a^{28 \bmod 8} \mod 15 = a^4 \mod 15.$$

The proof in Chapter 14 of [Wag] that the RSA cryptosystem works depends on the above fact.

2.6 Summary of topics

In this section, we introduced "Cryptographers favorites"

Supplemental questions for chapter 2

1. For any bit string a, what is $a \oplus a \oplus a \oplus a \oplus a$ equal to?
2. Prove in two ways that the three equations using exclusive-or to interchange two values work. One way should use just the definition of **`xor`** in the table, and the other way should use the properties of **`xor`** listed above. (On computer hardware that has an **`xor`** instruction combined with assignment, the above solution may execute just as fast as the previous one and will avoid the extra variable.)
3. Use the notation $\vee$ to mean "inclusive-or", $\wedge$ to mean "and", and $\sim$ to mean "not". With this notation, show, using either truth tables or algebraically that

$$\begin{aligned} a \oplus b &= (a \wedge \sim b) \vee (\sim a \wedge b), \text{ and} \\ &= (a \vee b) \wedge (\sim (a \wedge b)) \end{aligned}$$

4. Show how to use exclusive-or to compare the differences between two bit strings.
5. Given a bit string a, show how to use another *mask* bit string m of the same length to reverse a fixed bit position i, that is, change 0 to 1 and 1 to 0, but just in position i.
6. How many bits are needed to represent a number that is 100 decimal digits long? How many decimal digits are needed to represent a number that is 1000 bits long? How many decimal digits are needed to represent a number that is 100 decimal digits long?
7. Write a Java function to return the log base b of a, where $b > 1$ and $a > 0$.
8. In the example of 2-by-2 matrices, verify that the product of a non-zero element and its inverse is the identity.

Further study

- The Laws of Cryptography with Java Code *[Wag]*
  http://www.cs.utsa.edu/~wagner/lawsbookcolor/laws.pdf

Chapter 3

Cryptographers' favorite algorithms

This chapter is also copied verbatim from the "The Laws of Cryptography with Java Code", [Wag] with permission from Prof Neal Wagner.

3.1 The extended Euclidean algorithm

The previous section introduced the field known as the *integers mod p*, denoted Z_p or $GF(p)$. Most of the field operations are straightforward, since they are just the ordinary arithmetic operations followed by remainder on division by p. However the multiplicative inverse is not intuitive and requires some theory to compute. If a is a non-zero element of the field, then a^{-1} can be computed efficiently using what is known as *the extended Euclidean algorithm*, which gives the greatest common divisor (gcd) along with other integers that allow one to calculate the inverse. It is the topic of the remainder of this section.

Law GCD-1:
The cryptographer's first and oldest favorite algorithm is the *extended Euclidean algorithm*, which computes the greatest common divisor of two positive integers *a* and *b* and also supplies integers *x* and *y* such that *x*a + y*b = gcd(a, b)*.

The Basic Euclidean Algorithm to give the gcd: Consider the calculation of the greatest common divisor (gcd) of 819 and 462. One could factor the numbers as: $819 = 3 \cdot 3 \cdot 7 \cdot 13$ and $462 = 2 \cdot 3 \cdot 7 \cdot 11$, to see immediately that the gcd is $21 = 3 \cdot 7$. The problem with this method is that there is no efficient algorithm to factor integers. In fact, the security of the RSA cryptosystem relies on the difficulty of factoring, and we need an extended gcd algorithm to implement RSA. It

turns out that there is another better algorithm for the gcd — developed 2500 years ago by Euclid (or mathematicians before him), called (surprise) the *Euclidean algorithm*.

The algorithm is simple: just repeatedly divide the larger one by the smaller, and write an equation of the form "larger = smaller * quotient + remainder". Then repeat using the two numbers "smaller" and "remainder". When you get a 0 remainder, then you have the gcd of the original two numbers. Here is the sequence of calculations for the same example as before:

$$\begin{array}{rcrcrl} 819 & = & 462 & \cdot & 1 + & 357 \quad \text{(Step 0)} \\ 462 & = & 357 & \cdot & 1 + & 105 \quad \text{(Step 1)} \\ 357 & = & 105 & \cdot & 3 + & 42 \quad \text{(Step 2)} \\ 105 & = & 42 & \cdot & 2 + & 21 \quad \text{(Step 3, so GCD = 21)} \\ 42 & = & 21 & \cdot & 2 + & 0 \quad \text{(Step 4)} \end{array}$$

The proof that this works is simple: a common divisor of the first two numbers must also be a common divisor of all three numbers all the way down. (Any number is a divisor of 0, sort of on an honorary basis.) One also has to argue that the algorithm will terminate and not go on forever, but this is clear since the remainders must be smaller at each stage.

Here is Java code for two versions of the GCD algorithm: one iterative and one recursive. (There is also a more complicated *binary* version that is efficient and does not require division.)

Java function: gcd (two versions)

```
public static long gcd1(long x, long y) {
   if (y == 0) return x;
   return gcd1(y, x%y);
}

public static long gcd2(long x, long y) {
   while (y != 0) {
      long r = x % y;
      x = y; y = r;
   }
   return x;
}
```

A complete Java program using the above two functions is on page 165 of [Wag].

The Extended GCD Algorithm: Given the two positive integers 819 and 462, the extended Euclidean algorithm finds unique integers a and b so that $a \cdot 819 + b \cdot 462 = \gcd(819, 462) = 21$. In this case, $(-9) \cdot 819 + 16 \cdot 462 = 21$.

For this example, to calculate the magic a and b, just work backwards through the original equations, from step 3 back to step 0 (see above). Below are equations, where each shows two numbers a and b from a step of the original algorithm, and corresponding integers x and y (in **bold**), such that $x \cdot a + y \cdot b = gcd(a, b)$. Between each pair of equations is an equation that leads to the next equation.

```
1*105+(-2)*42=21                                        (from Step 3 above)
(-2)*357+(-2)(-3)*105=(-2)*42=(-1)*105+21                 (Step 2 times -2)
(-2)*357+7*105=21                 (Combine and simplify previous equation)
7*462+(7)(-1)*357=7*105=2*357+21                          (Step 1 times 7)
7*462+(-9)*357=21                 (Combine and simplify previous equation)
(-9)*819+(-9)(-1)*462=(-9)*357=(-7)*462+21                 (Step 0 * (-9))
(-9)*819+16*462=21(Simplify -- the final answer)
```

It's possible to code the extended gcd algorithm following the model above, first using a loop to calculate the gcd, while saving the quotients at each stage, and then using a second loop as above to work back through the equations, solving for the gcd in terms of the original two numbers. However, there is a much shorter, tricky version of the extended gcd algorithm, adapted from D. Knuth.

Java function: GCD (extended gcd)

```
public static long[] GCD(long x, long y) {
   long[] u = {1, 0, x}, v = {0, 1, y}, t = new long[3];
   while (v[2] != 0) {
      long q = u[2]/v[2];
      for (int i = 0; i < 3; i++) {
         t[i] = u[i] - v[i]*q; u[i] = v[i]; v[i] = t[i];
      }
   }
   return u;
}
```

A complete Java program using the above function is on page 166 of [Wag].

The above code rather hides what is going on, so with additional comments and checks, the code is rewritten below. Note that at every stage of the algorithm below, if **w** stands for any of the three vectors **u**, **v** or **t**, then one has **x*w[0] + y*w[1] = w[2]**. The function **check** verifies that this condition is met, checking in each case the vector that has just been changed. Since at the end, **u[2]** is the gcd, **u[0]** and **u[1]** must be the desired integers.

Java function: GCD (debug version)

```
public static long[] GCD(long x, long y) {
   long[] u = new long[3];
   long[] v = new long[3];
   long[] t = new long[3];
   // at all stages, if w is any of the 3 vectors u, v or t, then
   //   x*w[0] + y*w[1] = w[2]  (this is verified by "check" below)
   // vector initializations: u = {1, 0, u}; v = {0, 1, v};
   u[0] = 1; u[1] = 0; u[2] = x; v[0] = 0; v[1] = 1; v[2] = y;
   System.out.println("q\tu[0]\tu[1]\tu[2]\tv[0]\tv[1]\tv[2]");

   while (v[2] != 0) {
      long q = u[2]/v[2];
      // vector equation:  t = u - v*q
      t[0] = u[0] - v[0]*q; t[1] = u[1] - v[1]*q; t[2] = u[2] - v[2]*q;
```

```
      check(x, y, t);
      // vector equation:  u = v;
      u[0] = v[0]; u[1] = v[1]; u[2] = v[2]; check(x, y, u);
      // vector equation:  v = t;
      v[0] = t[0]; v[1] = t[1]; v[2] = t[2]; check(x, y, v);
      System.out.println(q + "\t"+ u[0] + "\t" + u[1] + "\t" + u[2] +
                             "\t"+ v[0] + "\t" + v[1] + "\t" + v[2]);
   }
   return u;
}

public static void check(long x, long y, long[] w) {
   if (x*w[0] + y*w[1] != w[2]) {
      System.out.println("*** Check fails: " + x + " " + y);
      System.exit(1);
   }
}
```

Here is the result of a run with the data shown above:

```
q       u[0]    u[1]    u[2]    v[0]    v[1]    v[2]}

1       0       1       462     1       -1      357
1       1       -1      357     -1      2       105
3       -1      2       105     4       -7      42
2       4       -7      42      -9      16      21
2       -9      16      21      22      -39     0

gcd(819, 462) = 21
(-9)*819 + (16)*462 = 21
```

Here is a run starting with 40902 and 24140:

```
q       u[0]    u[1]    u[2]    v[0]    v[1]    v[2]}

1       0       1       24140   1       -1      16762
1       1       -1      16762   -1      2       7378
2       -1      2       7378    3       -5      2006
3       3       -5      2006    -10     17      1360
1       -10     17      1360    13      -22     646
2       13      -22     646     -36     61      68
9       -36     61      68      337     -571    34
2       337     -571    34      -710    1203    0

gcd(40902, 24140) = 34
(337)*40902 + (-571)*24140 = 34
```

A complete Java program with the above functions, along with other example runs appears on page 167 of [Wag].

3.2 Fast integer exponentiation (raise to a power)

A number of cryptosystems require arithmetic on large integers. For example, the RSA public key cryptosystem uses integers that are at least 1024 bits long. An essential part of many of the algorithms involved is to raise an integer to another integer power, modulo an integer (taking the remainder on division).

Law EXP-1:
Many cryptosystems in modern cryptography depend on a fast algorithm to perform integer exponentiation.

It comes as a surprise to some people that in a reasonable amount of time one can raise a 1024 bit integer to a similar-sized power modulo an integer of the same size. (This calculation can be done on a modern workstation in a fraction of a second.) In fact, if one wants to calculate x^{1024} (a 10-bit exponent), there is no need to multiply x by itself 1024 times, but one only needs to square x and keep squaring the result 10 times. Similarly, 20 squarings yields $x^{1048576}$ (a 20-bit exponent), and an exponent with 1024 bits requires only that many squarings if it is an exact power of 2. Intermediate powers come from saving intermediate results and multiplying them in. RSA would be useless if there were no efficient exponentiation algorithm.

There are two distinct fast algorithms for raising a number to an integer power. Here is pseudo-code for raising an integer x to power an integer Y:

Java function: exp (first version)

```
int exp(int x, int Y[], int k) {
   //  Y = Y[k] Y[k-1] ... Y[1] Y[0]  (in binary)
   int y = 0, z = 1;
   for (int i = k; i >= 0; i--) {
      y = 2*y;
      z = z*z;
      if (Y[i] == 1) {
         y++;
         z = z*x;
      }
   }
   return z;
}
```

The variable y is only present to give a loop invariant, since at the beginning and end of each loop, as well as just before the if statement, the invariant $x^y = z$ holds, and after the loop terminates $y = Y$ is also true, so at the end, $z = x^Y$. To find $x^y \mod n$ one should add a remainder on division by n to the two lines that calculate z.

Here is the other exponentiation algorithm. It is very similar to the previous algorithm, but differs in processing the binary bits of the exponent in the opposite order. It also creates those bits as it goes, while the other assumes they are given.

Java function: exp (second version)

```
int exp(int X, int Y) {
   int x = X, y = Y, z = 1;
   while (y > 0) {
      while (y%2 == 0) {
         x = x*x;
         y = y/2;
      }
      z = z*x;
      y = y - 1;
   }
   return z;
}
```

The loop invariant at each stage and after the each iteration of the inner while statement is:

$$z * x^y = X^Y.$$

Here is a mathematical proof that the second algorithm actually calculates X^Y. Just before the while loop starts, since $x = X$, $y = Y$, and $z = 1$, it is obvious that the loop invariant is true. (In these equations, the $=$ is a mathematical equals, not an assignment.)

Now suppose that at the start of one of the iterations of the while loop, the invariant holds. Use x', y', and z' for the new values of x, y, and z after executing the statements inside one iteration of the inner while statement. Notice that this assumes that y is even. Then the following are true:

$$\begin{aligned} x' &= x * x \\ y' &= y/2 \text{ (exact integer because y is even)} \\ z' &= z \\ z' * (x')^{y'} &= z * (x * x)^{y/2} = z * x^y = X^Y. \end{aligned}$$

This means that the loop invariant holds at the end of each iteration of the inner while statement for the new values of x, y, and z. Similarly, use the same prime notation for the variables at the end of the while loop.

$$\begin{aligned} x' &= x \\ y' &= y - 1 \\ z' &= z * x \\ z' * (x')^{y'} &= z * x * (x)^{y-1} = z * x^y = X^Y. \end{aligned}$$

So once again the loop invariant holds. After the loop terminates, the variable y must be 0, so that the loop invariant equation says:

$$X^Y = z * x^y = z * x^0 = z.$$

For a complete proof, one must also carefully argue that the loop will always terminate.

A test of the two exponentiation functions implemented in Java appears on page 169 of [Wag].

3.3 Checking for probable primes

For 2500 years mathematicians studied prime numbers just because they were interesting, without any idea they would have practical applications. Where do prime numbers come up in the real world? Well, there's always the 7-Up soft drink, and there are sometimes a prime number of ball bearings arranged in a circle, to cut down on periodic wear. Now finally, in cryptography, prime numbers have come into their own.

Law PRIME-1:
A source of large random prime integers is an essential part of many current cryptosystems.

Usually large random primes are created (or found) by starting with a random integer n, and checking each successive integer after that point to see if it is prime. The present situation is interesting: there are reasonable algorithms to check that a large integer is prime, but these algorithms are not very efficient (although a recently discovered algorithm is guaranteed to produce an answer in running time no worse that the number of bits to the twelth power). On the other hand, it is very quick to check that an integer is "probably" prime. To a mathematician, it is not satisfactory to know that an integer is only probably prime, but if the chances of making a mistake about the number being a prime are reduced to a quantity close enough to zero, the users can just discount the chances of such a mistake.

Tests to check if a number is probably prime are called *pseudo-prime* tests. Many such tests are available, but most use mathematical overkill. Anyway, one starts with a property of a prime number, such as Fermat's Theorem, mentioned in the previous chapter: if p is a prime and a is any non-zero number less than p, then $a^{p-1} \mod p = 1$. If one can find a number a for which Fermat's Theorem does not hold, then the number p in the theorem is *definitely not a prime*. If the theorem holds, then p is called *a pseudo-prime with respect to* a, and it might actually be a prime.

So the simplest possible pseudo-prime test would just take a small value of a, say 2 or 3, and check if Fermat's Theorem is true.

> **Simple Pseudo-prime Test:** If a very large random integer p (100 decimal digits or more) is not divisible by a small prime, and if $3^{p-1} \mod p = 1$, then the number is prime except for a vanishingly small probability, which one can ignore.

One could just repeat the test for other integers besides 3 as the base, but unfortunately there are non-primes (called *Carmichael numbers*) that satisfy Fermat's theorem for all values of a even though they are not prime. The smallest such number is $561 = 3 \cdot 11 \cdot 17$, but these numbers become very rare in the larger range, such as 1024-bit numbers. Corman et al.estimate that the chances of a mistake with just the above simple test are less than 10^{-41}, although in practice

commercial cryptosystems use better tests for which there is a proof of the low probability. Such better tests are not really needed, since even if the almost inconceivable happened and a mistake were made, the cryptosystem wouldn't work, and one could just choose another pseudo-prime.

Law PRIME-2:
Just one simple pseudo-prime test is enough to test that a very large random integer is probably prime.

3.4 Summary of topics

In this section, we introduced “Cryptographers favorite algorithms”

Supplemental questions for chapter 3

1. Prove that the long (debug) version of the Extended GCD Algorithm works.

   (a) First show that **`u[2]`** is the gcd by throwing out all references to array indexes **`0`** and **`1`**, leaving just **`u[2]`**, **`v[2]`**, and **`t[2]`**. Show that this still terminates and just calculates the simple gcd, without reference to the other array indexes. (This shows that at the end of the complicated algorithm, **`u[2]`** actually is the gcd.)

   (b) Next show mathematically that the three special equations are true at the start of the algorithm, and that each stage of the algorithm leaves them true. (One says that they are left *invariant*.)

   (c) Finally deduce that algorithm is correct.

Further study

- The Laws of Cryptography with Java Code *[Wag]*
  http://www.cs.utsa.edu/~wagner/lawsbookcolor/laws.pdf
