4 Number Theory and Cryptography

Introduction

Mathematica includes numerous functions for exploring number theory. In this chapter we will see how to use *Mathematica*'s computational abilities to compute and solve congruences, represent integers in bases other than ten, explore arithmetic algorithms in those bases, check whether or not a number is prime, and compute discrete logarithms. We will also see how *Mathematica* can help explore several of the applications described in the textbook, in particular, hashing functions, pseudorandom numbers, check digits, and, of course, cryptography.

4.1 Divisibility and Modular Arithmetic

In this section we will use *Mathematica* to explore divisibility of integers and modular arithmetic. We will see how to compute quotients and remainders in integer division, how to test integers for the divisibility relationship, and how to perform computations in modular arithmetic. This section will conclude with an illustration of how to create infix addition and multiplication operators for modular arithmetic and a demonstration of how *Mathematica* can be used to compute addition and multiplication tables.

Quotient, Remainder, and Divisibility

Mathematica's functions `Quotient` and `Mod` compute the quotient and remainder, respectively, obtained when you divide two integers. For example, consider 99 divided by 13.

```
In[1]:= Quotient[99, 13]
Out[1]= 7
In[2]:= Mod[99, 13]
Out[2]= 8
```

These results indicate that 99 divided by 13 results in a quotient of 7 and a remainder of 8. That is, $99 = 13 \cdot 7 + 8$.

Note that the textbook uses the notation $a \bmod m$ to represent the remainder when a is divided by m, using mod like an operator. To enter this in *Mathematica*, you must use the functional notation **`Mod[a, m]`**.

Another function, `QuotientRemainder`, produces a list with first element the quotient and second element the remainder.

```
In[3]:= QuotientRemainder[99, 13]
Out[3]= {7, 8}
```

Checking Divisibility

To test whether one integer divides another, you can, of course, check to see if the remainder is 0 or not. For example, the following shows that $3 \mid 132$.

```
In[4]:= Mod[132, 3]
Out[4]= 0
```

Mathematica also provides the function `Divisible` for testing divisibility. The `Divisible` function returns true if its first argument is divisible by the second.

```
In[5]:= Divisible[132, 3]
Out[5]= True
In[6]:= Divisible[99, 13]
Out[6]= False
```

Offset

Recall from the division algorithm that the remainder must always be nonnegative, even when the dividend is negative. *Mathematica*'s `Mod` function respects that convention by default, with **`Mod[a, m]`** returning a value between 0 and $m - 1$.

```
In[7]:= Mod[-27, 5]
Out[7]= 3
```

However, there are times when it is more useful to allow negative values. For example, consider the following question: "It is now 11:00 AM. What time will it be 142 hours from now?" If we compute $142 \bmod 24$,

```
In[8]:= Mod[142, 24]
Out[8]= 22
```

We see that the time will be the same 142 hours from now as 22 hours from now. But it's also the case that $142 \equiv -2 \pmod{24}$, which means that the time 142 hours from now is the same as the time 2 hours earlier, i.e., 9:00 AM. You can see that this congruence is somewhat more convenient.

To support this, the `Mod` function accepts an optional third argument, called the offset. The offset specifies a lower bound on the `Mod` function's output. Specifically, for an integral offset d, **`Mod[a, m, d]`** will return the value congruent to $a \bmod m$ that lies between d and $d + m - 1$. For example, by specifying an offset of 1, the result will be betwen 1 and m, so a computation that would ordinarily return 0 will return the value of the modulus instead.

```
In[9]:= Mod[132, 3, 1]
Out[9]= 3
```

The value 1 is a common offset, as it ensures a positive result that can then be used as an index into a list. Another common offset is the fraction $-m/2$. This offset ensures that the output is the integer closest to 0 that is congruent to the first argument modulo the second. For example, returning to the

time example above, to obtain the result -2 we use the offset $-24/2$.

```
In[10]:= Mod[142, 24, -24 / 2]
Out[10]= -2
```

Congruences

The first argument to Mod can be any algebraic expression. For example, you can compute $3 + 4 \cdot 9^2 \bmod 5$ as follows.

```
In[11]:= Mod[3 + 4 * 9^2, 5]
Out[11]= 2
```

To test a congruence, for example to confirm that $428 \equiv 530 \pmod{17}$, you must apply the Mod function to both values and test them using the Equal (==) relation.

```
In[12]:= Mod[428, 17] == Mod[530, 17]
Out[12]= True
```

You may not include the Equal (==) relation within the argument to Mod.

Solving Congruences

Mathematica can solve congruences by using the Solve function in conjunction with the Modulus option. To do so, give the congruence or list of simultaneous congruences as the first argument to Solve and identify the modulus using the Modulus option. As an example, consider Exercise 13a from Section 4.1 of the textbook. Under the assumption that $a \equiv 4 \pmod{13}$, we are to solve $c \equiv 9\,a \pmod{13}$. We solve this using *Mathematica* as follows.

```
In[13]:= Solve[{a == 4, c == 9 * a}, Modulus → 13]
Out[13]= {{a → 4, c → 10}}
```

Note that the congruences are entered as a list. This tells *Mathematica* that they must be simultaneously satisfied. Also observe that we must use the Equal (==) relation, not Set (=), when specifying the congruences.

If there are no solutions to the congruence, then Solve will return the empty list.

```
In[14]:= Solve[n^2 == 3, Modulus → 4]
Out[14]= {}
```

Arithmetic Modulo *m*

In this subsection we'll define operators based on the definitions of $+_m$ and $\cdot_m$ given in the text. Our goal will be to get as close as possible to being able to enter $7 +_{11} 9$ and have *Mathematica* return 5.

The usual style of writing arithmetic operators in between the operands is referred to as infix notation. There are operators that *Mathematica* recognizes as infix operators, but which do not have built-in definitions. We can take advantage of this to create our own infix operators by providing definitions for these undefined operators.

You can see all of *Mathematica*'s operators in the table of operator precedence in the Operator Input Forms tutorial. Those operators with a triangular mark in the far right of the table have built-in defini-

tions. Those without such a mark are available for your own use. Here, we will make use of the CirclePlus (⊕) and CircleTimes (⊗) operators.

To enter these as infix operators, you need to use their aliases. To enter ⊕, type ESC c+ ESC, and to enter ⊗, type ESC c* ESC.

Observe that if you enter an expression using one of the operators, and apply the FullForm function, you can see that *Mathematica* interprets the infix operator as an application of the named function.

```
In[15]:= 2 ⊕ 3 // FullForm
Out[15]//FullForm=
CirclePlus[2, 3]
```

Also note that while these functions are currently undefined, they do have a defined precedence. So, for example, $2 \oplus 3 \otimes 4$ is interpreted as $2 \oplus (3 \otimes 4)$, which can be verified by inspecting the functional expression obtained with FullForm.

```
In[16]:= 2 ⊕ 3 ⊗ 4 // FullForm
Out[16]//FullForm=
CirclePlus[2, CircleTimes[3, 4]]
```

To define the operator, you can issue the definition using either the functional or infix form. Below, we define addition in functional form and multiplication in infix form. Regardless of the form you provide the definition, *Mathematica* will properly evaluate both forms.

```
In[17]:= CirclePlus[a_, b_] := Mod[a + b, 11]
In[18]:= a_ ⊗ b_ := Mod[a * b, 11]
```

Now we can compute $7 +_{11} 9$ and $3 \cdot_{11} 5$ using the ⊕ and ⊗ operators.

```
In[19]:= 7 ⊕ 9
Out[19]= 5
In[20]:= 3 ⊗ 5
Out[20]= 4
```

Addition and Multiplication Tables

We conclude this section by producing addition and multiplication tables.

We will create the tables, unsurprisingly, using the Table function. The first argument will be the sum or product of two variables within the Mod function. The Table repetition arguments will specify that the variables range from 0 to one less than the modulus. A call to TableForm will make the result readable.

Here is the addition table modulo 5.

```
In[21]:= Table[Mod[a + b, 5], {a, 0, 4}, {b, 0, 4}] // TableForm
```

Out[21]//TableForm=

0	1	2	3	4
1	2	3	4	0
2	3	4	0	1
3	4	0	1	2
4	0	1	2	3

We can improve the table a bit more by adding column and row headings. This is done using the TableHeadings option to TableForm. By setting the TableHeadings option to a list consisting of two sublists corresponding to the desired labels for the rows and then the columns, *Mathematica* will display those labels as headings. The keywork None can be used in place of one of the sublists so as to omit the corresponding set of labels.

```
In[22]:= TableForm[Table[Mod[a + b, 5], {a, 0, 4}, {b, 0, 4}],
           TableHeadings → {{0, 1, 2, 3, 4}, {0, 1, 2, 3, 4}}]
```

Out[22]//TableForm=

	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

Using that example as a model, it is easy to create general functions that will accept a modulus and display the addition or multiplication table.

```
In[23]:= AdditionTable[m_] := Module[{a, b},
           TableForm[Table[Mod[a + b, m], {a, 0, m - 1}, {b, 0, m - 1}],
            TableHeadings → {Range[0, m - 1], Range[0, m - 1]}]]
```

```
In[24]:= MultiplicationTable[m_] := Module[{a, b},
           TableForm[Table[Mod[a * b, m], {a, 0, m - 1}, {b, 0, m - 1}],
            TableHeadings → {Range[0, m - 1], Range[0, m - 1]}]]
```

Here is the multiplication table modulo 5.

```
In[25]:= MultiplicationTable[5]
```

Out[25]//TableForm=

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

4.2 Integer Representations and Algorithms

In this section we will see how *Mathematica* can be used to explore representations of integers in various bases and to explore algorithms for computing with integers. We will begin by looking at *Mathematica*'s built-in functions for converting between bases. Then we'll focus our attention on binary representations of integers and how to implement algorithms for addition and multiplication on binary representations. We restrict our attention to positive integers throught this section.

Base Conversion

Mathematica provides support for converting from one base representation to another via the functions `IntegerDigits`, `IntegerString`, and `FromDigits`.

The `IntegerDigits` function is used to convert a positive integer, expressed in base ten, into a list of the digits of the integer expressed in a specified base. The first argument to `IntegerDigits` is the base ten integer. If no other arguments are provided, the function returns the list of the digits of the integer.

```
In[26]:= IntegerDigits[1234]
Out[26]= {1, 2, 3, 4}
```

Note that the most significant digit is first in the list and the least significant is last. That is, the "one's digit" is the last element in the list.

By providing a base as a second argument, the `IntegerDigits` function will output the list of the digits in the representation of the integer in that base. The following indicates that $(1234)_{10} = (103\,102)_4$.

```
In[27]:= IntegerDigits[1234, 4]
Out[27]= {1, 0, 3, 1, 0, 2}
```

For bases larger than ten, `IntegerDigits` does not make use of letters for digits with values larger than 9. Rather, it uses the base ten representation of the value of such digits. For example, in Example 5 of the textbook, it is shown that $(177\,130)_{10} = (2\,\mathrm{B3EA})_{16}$. The output of `IntegerDigits` reports 10, 11, and 14 rather than A, B, and E.

```
In[28]:= IntegerDigits[177 130, 16]
Out[28]= {2, 11, 3, 14, 10}
```

The `IntegerDigits` function also accepts a third argument to specify a minimum length of the output. If the representation of the integer in the given base has fewer digits than specified by the third argument, zeros will be added on the left. The following shows the representation of 123 in binary (base 2), with ten digits.

```
In[29]:= IntegerDigits[123, 2, 10]
Out[29]= {0, 0, 0, 1, 1, 1, 1, 0, 1, 1}
```

The `IntegerString` function is very similar to `IntegerDigits`, accepting the same arguments and having a very similar effect. The difference is that the output of `IntegerString` is a string

rather than a list. This means that the output has a more typical appearance. Contrast the output below to the corresponding example above.

```
In[30]:= IntegerString[1234, 4]

Out[30]= 103102
```

Note that, despite appearances, the output is in fact a string object, and not a numerical object.

```
In[31]:= Head[%]

Out[31]= String
```

As a result, you cannot manipulate the output of IntegerString using numerical operators. The IntegerDigits function is much more useful if you wish to work with the output. However, IntegerString produces much more readable output. It also follows the usual convention of using letters for digits larger than 9.

```
In[32]:= IntegerString[177 130, 16]

Out[32]= 2b3ea
```

The IntegerDigits and IntegerString functions are both used to take a base ten representation of a positive integer and return a representation in another base. For the reverse, we use the FromDigits function.

The first argument of FromDigits can be either a list, like the output of IntegerDigits, or a string, like the output of IntegerString. With no second argument, *Mathematica* will assume that the input is intended to be base ten and will convert the list or string into the corresponding integer.

```
In[33]:= FromDigits[{1, 2, 3, 4}]

Out[33]= 1234

In[34]:= FromDigits["1234"]

Out[34]= 1234
```

The second argument specifies the base in which the first argument is given. For example, to convert $(103\,012)_4$ back to base ten, you enter either of the following.

```
In[35]:= FromDigits[{1, 0, 3, 1, 0, 2}, 4]

Out[35]= 1234

In[36]:= FromDigits["103102", 4]

Out[36]= 1234
```

For bases larger than ten, you can use letters to represent digits larger than nine when using a string as the first argument to FromDigits. Note that either lower or upper case are acceptable.

```
In[37]:= FromDigits["2B3EA", 16]

Out[37]= 177 130
```

Finally, note that both IntegerString and FromDigits can, in place of the base, accept the string **"Roman"**, in which case the function will convert to or from a Roman numeral representation.

```
In[38]:= IntegerString[2013, "Roman"]
Out[38]= MMXIII
In[39]:= FromDigits["MMXIII", "Roman"]
Out[39]= 2013
```

The BaseForm function should also be mentioned. Unlike the functions above, BaseForm only affects how a number is displayed by *Mathematica*, not its type. That is, BaseForm does not change an integer into a list or string, it only displays it as if it were in a different base. Its arguments are a base ten integer and a base.

```
In[40]:= BaseForm[177 130, 16]
Out[40]//BaseForm=
         2b3ea16
```

In the other direction, you can use the double-caret (^^) notation to enter a number in a specified base. Enter the base, followed by the representation of the number in that base. For example, to enter $(A3)_{16}$ you type the following.

```
In[41]:= 16^^a3
Out[41]= 163
```

Converting Between Two Non-ten Bases

Given a positive integer in a base other than ten, you can convert it to another by using base ten as an intermediary. For example, to convert $(123)_5$ to base 3, you proceed as follows. First convert to base ten using FromDigits.

```
In[42]:= FromDigits["123", 5]
Out[42]= 38
```

Then use IntegerString to convert to base 3.

```
In[43]:= IntegerString[38, 3]
Out[43]= 1102
```

The result indicates that $(123)_5 = (1102)_3$.

We can combine these two steps into a single function that takes as its arguments a string representing the integer, the starting base, and the final base. The body of the function is simply a composition of FromDigits and IntegerString.

```
In[44]:= convertString[n_, b1_, b2_] :=
          IntegerString[FromDigits[n, b1], b2]
In[45]:= convertString["123", 5, 3]
Out[45]= 1102
```

It is left to the reader to define a similar function that outputs a list of digits rather than a string.

Binary Addition

In this subsection, we will implement Algorithm 2 from Section 4.2, addition of integers. Our function will accept two binary representations given as lists of 0s and 1s with the most significant digit first.

The first task for our function will be to make sure that the binary representations are of the same length. To do this, we compute the maximum of the lengths of the two lists, which will be stored as **n**, and then add as many 0s to the list as are necessary to make both lists that length. We also initialize a sum list **S** to the list of all 0s of that same length.

To add 0s to the input list, we use the PadLeft function, which takes a list and a length and returns the list of the desired length obtained by adding 0s on the left side of the list. As you might suspect, there is also a PadRight function. To initialize **S**, we use ConstantArray, which, given an expression and a length creates the list of the desired length all of whose elements are the given expression. We illustrate these functions below.

```
In[46]:= PadLeft[{1, 2, 3}, 7]
Out[46]= {0, 0, 0, 0, 1, 2, 3}
In[47]:= ConstantArray["x", 5]
Out[47]= {x, x, x, x, x}
```

Once these initial tasks are completed, we follow Algorithm 2. Note that the indices used must be modified to match that used by *Mathematica*. The loop variable j, as presented in the textbook ranges from 0 to $n-1$, where 0 is the index of the least significant (the "one's") digit and $n-1$ is the index of the most significant digit. In this implementation, the least significant digit in the input values, **A** and **B**, will be in the last position, which has index equal to their length, **n**. And the most significant digit will be in position 1. Consequently, our For loop will have variable **j** ranging from **n** to **1**.

The last difference between our implementation and Algorithm 2 is the use of the PrependTo function to add a 1 at the beginning of the sum, in case a carry requires the result have an additional digit.

```
In[48]:= addition[a_List, b_List] := Module[{n, A, B, S, c, j, d},
           n = Max[Length[a], Length[b]];
           A = PadLeft[a, n];
           B = PadLeft[b, n];
           S = ConstantArray[0, n];
           c = 0;
           For[j = n, j ≥ 1, j--,
            d = Floor[(A[[j]] + B[[j]] + c) / 2];
            S[[j]] = A[[j]] + B[[j]] + c - 2 * d;
            c = d
           ];
           If[c == 1, PrependTo[S, 1]];
           S
          ]
In[49]:= addition[{1, 0, 1, 0}, {1, 1, 1, 0, 1, 0}]
Out[49]= {1, 0, 0, 0, 1, 0, 0}
```

Binary Multiplication

Finally, we will implement a multiplication algorithm, presented as Algorithm 3 in Section 4.2. Once

again, our function will accept the binary representations of positive integers as the inputs. This time, however, it is not necessary for them to have the same length.

The shift that occurs when $b_j = 1$ will be accomplished as the following example illustrates. To shift the list {1, 1, 1, 1} by 5 places, we must add five 0s on the end of the list. We do this by using ConstantArray to create the list of 5 0s and then Join to combine the original list with this list of zeros.

```
In[50]:= shiftExample = {1, 1, 1, 1}
Out[50]= {1, 1, 1, 1}
In[51]:= Join[shiftExample, ConstantArray[0, 5]]
Out[51]= {1, 1, 1, 1, 0, 0, 0, 0, 0}
```

We will store the partial products using an indexed variable. Recall from Section 2.3 of this manual that we can store an object in an indexed variable, **c**, by making an assignment to the symbol **c[i]** for index **i**.

The product **p** will be initialized to **{0}**, a binary representation of 0. The addition in the final loop will be performed by the **addition** function we created above.

As noted above, there is a discrepancy between the way indices are used in the pseudocode in the text and the indices used in *Mathematica*. For this function, we will mirror the textbook with the loop variable **j** ranging from **0** to **n-1**, where **n** is the number of digits in the second number. We interpret **j** as being the number of digits beyond the least significant digit, which has position **n**. Within the body of the loop, we inspect location **n-j**.

Here is our implementation of Algorithm 3.

```
In[52]:= multiplication[a_List, b_List] := Module[{n, j, c, p},
           n = Length[b];
           For[j = 0, j ≤ n - 1, j++,
            If[b[[n - j]] == 1,
             c[j] = Join[a, ConstantArray[0, j]],
             c[j] = {0}
            ]
           ];
           p = {0};
           For[j = 0, j ≤ n - 1, j++,
            p = addition[p, c[j]]
           ];
           p
          ]
```

We test our function using Example 10 from Section 4.2.

```
In[53]:= multiplication[{1, 1, 0}, {1, 0, 1}]
Out[53]= {1, 1, 1, 1, 0}
```

4.3 Primes and Greatest Common Divisors

In this section we will see how to use *Mathematica* to find primes, find prime factorizations, and compute greatest common divisors and least common multiples. We will also use *Mathematica*'s capabilities to explore the distribution of primes.

Primes

We will first introduce some of *Mathematica*'s functions for testing whether a number is prime and for finding primes.

Testing for Primality

The `PrimeQ` function accepts a single argument, an integer to be tested, and returns true or false.

```
In[54]:= PrimeQ[5]
Out[54]= True
In[55]:= PrimeQ[10]
Out[55]= False
In[56]:= PrimeQ[2^13 - 1]
Out[56]= True
```

Unlike the trial division algorithm discussed in the book, which checks all possible divisors to see if a number is prime or composite, `PrimeQ` uses a probabilistic primality test. This probabilistic test gains much faster performance at the cost of a small possibility that the command will return an incorrect result. There is no known example of an integer for which `PrimeQ` is incorrect and any such example must be exceptionally large. So, despite there being a chance of error, `PrimeQ` is in fact very reliable.

Listing Primes

The function `Prime` accepts as input a positive integer *i* and outputs the *i*th prime number.

```
In[57]:= Prime[1]
Out[57]= 2
In[58]:= Prime[2]
Out[58]= 3
In[59]:= Table[Prime[i], {i, 20}]
Out[59]= {2, 3, 5, 7, 11, 13, 17, 19, 23,
          29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71}
In[60]:= Prime[100 000]
Out[60]= 1 299 709
```

Mathematica also provides the `NextPrime` function, which returns the smallest prime larger than the input value. For example, to find the first prime number larger than 1000, enter the following.

```
In[61]:= NextPrime[1000]
Out[61]= 1009
```

The NextPrime function can also be given an optional second argument, which must be an integer. **NextPrime[*n*, *k*]** returns the *k*th prime larger then *n*. If the second argument is negative, the function instead returns the prime smaller than *n*. For example, the following expressions find the third prime after 1000 and the prime before it.

```
In[62]:= NextPrime[1000, 3]
Out[62]= 1019
In[63]:= NextPrime[1000, -1]
Out[63]= 997
```

Prime Factorization

To compute the prime factorization of an integer, we can use the *Mathematica* function FactorInteger. Several examples of using FactorInteger follow.

```
In[64]:= FactorInteger[100]
Out[64]= {{2, 2}, {5, 2}}
In[65]:= FactorInteger[123 456 789]
Out[65]= {{3, 2}, {3607, 1}, {3803, 1}}
In[66]:= FactorInteger[-987 654 321]
Out[66]= {{-1, 1}, {3, 2}, {17, 2}, {379 721, 1}}
```

The output of FactorInteger is a list of pairs. Each pair consists of a prime number and the multiplicity, or exponent, of that prime in the prime factorization. Note that in the last example, with negative input, one of the members of the list is the pair **{-1,1}**, indicating that $(-1)^1$ is in the factorization. The output above indicates that $100 = 2^2 \cdot 5^2$, $123\,456\,789 = 3^2 \cdot 3607^1 \cdot 3803^1$, and $-987\,654\,321 = (-1)^1 \cdot 3^2 \cdot 17^2 \cdot 379\,721^1$.

Note that FactorInteger can also accept an optional argument to limit the effort *Mathematica* will exert in trying to factor. Entering **Automatic** as the second argument limits the function to factors that it can find easily. You can also give a positive integer as the second argument, and *Mathematica* will then find at most that many distinct factors.

```
In[67]:= FactorInteger[236 914 830 635 411 777 378 758 175 934 586 404 476 822]
Out[67]= {{2, 1}, {197, 1}, {509, 1}, {10 459 723, 3}, {32 129 861, 2}}
In[68]:= FactorInteger[
          236 914 830 635 411 777 378 758 175 934 586 404 476 822, 3]
Out[68]= {{394, 1}, {509, 1},
          {1 181 349 070 215 370 924 270 532 326 421 800 507, 1}}
```

The first expression above factors the given integer into its complete prime factorization. The second is limited to finding only 3 distinct factors. After finding the first two prime factors, 394 and 509, it

reports the remainder as the last factor with exponent 1. This gives you a way to have *Mathematica* only perform the easy parts of factorizations, which can help ensure that your functions run quickly during development. Then, when you are ready to let the function take all the time it needs, you can remove the limitation.

The Distribution of Primes

The Prime Number Theorem (Theorem 4 in Section 4.3 of the text) tells us that the number of primes not exceeding x is approximated by the function $\frac{x}{\ln(x)}$. In this subsection, we will use *Mathematica*'s graphing capabilities to graph the number of primes not exceeding x.

Recall from Section 3.3 of this manual that we can graph a list of points by using the ListPlot function applied to a list of x-y pairs. Our x values will be the integers from 2 to 1000 (we omit 1 since there are no primes less than or equal to 1).

To find the number of primes not exceeding x, we use the function PrimePi. The function $\pi(x)$ is the standard notation for the number of primes less than or equal to x. The symbol PrimePi distinguishes this function in *Mathematica* from the mathematical constant which is given the symbol Pi. To calculate the number of primes less than or equal to 1000, for example, we enter the following.

```
In[69]:= PrimePi[1000]
Out[69]= 168
```

We use the Table function to produce the list of pairs $\{x, \pi(x)\}$, which we will graph.

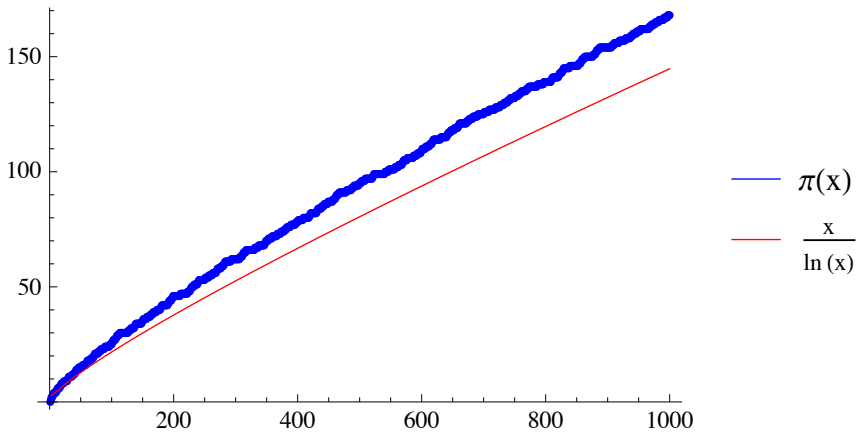
```
In[70]:= piList = Table[{x, PrimePi[x]}, {x, 1000}];
```

As we did in the solution to Computer Project 9 of Chapter 3, we will graphically compare the values of $\pi(x)$ to the function $\frac{x}{\ln(x)}$. We will define two graphics objects and then combine them with the Show function. Refer to Chapter 3 of this manual, particularly Section 3.3 and the solution to Computer Project 9, for detailed information about the commands ListPlot and Plot that we use here.

We also use the legending functions Legended and LineLegend. For a single application of Plot, you can use the PlotLegends option to very easily add a legend to the plot. That option is not available to Show, however. Instead, you must use the more general function Legended and manually construct the legend with LineLegend (or one of the related functions such as PointLegend or SwatchLegend). In this context, Legended can be thought of as a wrapper containing a graphics object, or more precisely an expression that produces a graphics object, as the first argument, and a call to a function that creates a legend as the second argument. The LineLegend function generally takes two arguments: the first a list of colors and the second a list of labels for the color in the corresponding position.

```
In[71]:= piPlot = ListPlot[piList, PlotStyle → Blue];
In[72]:= xlnxPlot = Plot[x / Log[x], {x, 2, 1000}, PlotStyle → Red];
```

```
In[73]:= Legended[Show[piPlot, xlnxPlot],
           LineLegend[{Blue, Red}, {"π(x)", "x/ln (x)"}]]
```



Notice that while the blue line representing $\pi(x)$ seems to remain above the red line representing $\frac{x}{\ln(x)}$, it is fairly clear from the graph that they grow at the same rate.

Greatest Common Divisor and Least Common Multiple

Mathematica provides the functions GCD and LCM for computing the greatest common divisor and the least common multiple of integers. To compute the greatest common divisor of two integers, you apply the GCD function to them.

```
In[74]:= GCD[6, 9]
Out[74]= 3
```

You can also compute the greatest common divisor of more than two integers. For more than 2 integers, the greatest common divisor is defined to be the largest integer that is a divisor of all of them. For example, 3 divides 6, 9, and 12, so:

```
In[75]:= GCD[6, 9, 12]
Out[75]= 3
```

The LCM command finds the least common multiple of two or more integers. For example,

```
In[76]:= LCM[6, 9]
Out[76]= 18
In[77]:= LCM[12, 18, 33]
Out[77]= 396
```

Relatively Prime

Recall from the text that two numbers are said to be relatively prime if their greatest common divisor is 1. For example, consider 10 and 21.

In[78]:= **GCD[10, 21]**

Out[78]= 1

Since GCD returned 1, we conclude that 10 and 21 are relatively prime.

The *Mathematica* function CoprimeQ, applied to two integers, returns true if they are relatively prime and false if not.

In[79]:= **CoprimeQ[3, 6]**

Out[79]= False

In[80]:= **CoprimeQ[22, 15]**

Out[80]= True

Also recall that a list of integers $a_1, a_2, \ldots, a_n$ are said to be pairwise relatively prime if $\gcd(a_i, a_j) = 1$ whenever $1 \le i < j \le n$. That is, when every pair is relatively prime. The CoprimeQ function, applied to more than two integers, tests this property.

Observe that 14, 39, and 55 are pairwise relatively prime.

In[81]:= **CoprimeQ[14, 39, 55]**

Out[81]= True

However, 42, 165, and 182 are not pairwise relatively prime, since 42 and 182 are both even, even though the common GCD of all three integers is 1.

In[82]:= **GCD[42, 165, 182]**

Out[82]= 1

In[83]:= **CoprimeQ[42, 165, 182]**

Out[83]= False

The Extended Euclidean Algorithm

While GCD is useful for calculating the greatest common divisor of integers, it is sometimes desirable to be able to express the greatest common divisor as an integral combination of the integers. Specifically, given integers a and b, we may wish to express $\gcd(a, b)$ as $s \cdot a + t \cdot b$ where s and t are integers. The fact that such integers always exist is known as Bézout's Theorem, given in the text as Theorem 6 of Section 4.3. In the preamble to Exercise 41 in the text, the extended Euclidean algorithm is described, which produces not only the greatest common divisor but also the integers s and t.

In *Mathematica*, the function ExtendedGCD is an implementation of the extended Euclidean algorithm. This function accepts two or more integers as arguments. It returns a list whose first element is the greatest common divisor of the arguments and whose second element is a sublist whose members are the coefficients required to obtain the GCD. As an example, consider 252 and 198, the values used in Example 17 of the textbook.

In[84]:= **ExtendedGCD[252, 198]**

Out[84]= {18, {4, -5}}

The results above indicate that $\gcd(252, 198) = 18 = 4 \cdot 252 - 5 \cdot 198$. Note that the order of the coefficients is the same as the order of the arguments to the function.

4.4 Solving Congruences

In this section, we will see how *Mathematica* can be used to solve congruences. We will begin the section by looking at how to find inverses and solve linear congruences. We will then consider the Chinese Remainder Theorem. Next, we will use *Mathematica* to find pseudoprimes, and we conclude with an exploration of primitive roots and discrete logarithms.

Modular Inverses

Example 1 of Section 4.4 of the text demonstrates how Bézout coefficients can be used to find the inverse of an integer modulo a number. In the previous section of this manual, we saw that the ExtendedGCD function can be used to obtain the Bézout coefficients.

Finding Inverses with ExtendedGCD

For example, to find the inverse of 264 modulo 3185, we need to find s so that $s \cdot 264 + t \cdot 3185 = 1$, provided that 264 and 3185 are relatively prime.

Recall that ExtendedGCD applied to two integers returns a structure of the form {*gcd*, {*s*, *t*}} where *gcd* is the greatest common divisor of the two integers, and *s* and *t* are the Bézout coefficients. Knowing that this is always the form of the output, *Mathematica* allows us to assign the result of ExtendedGCD to a structure of that form. This has the effect of assigning the symbols to the corresponding numbers in the output.

```
In[85]:= {gBezout, {sBezout, tBezout}} = ExtendedGCD[264, 3185]
Out[85]= {1, {374, -31}}
```

Since the first element is 1, we know that 264 and 3185 are relatively prime. Also, the assignment has caused the Bézout coefficients to be stored in the symbols.

```
In[86]:= sBezout
Out[86]= 374
In[87]:= tBezout
Out[87]= -31
```

This indicates that $1 = 374 \cdot 264 + (-31) \cdot 3185$. And thus 374 is the inverse of 264 modulo 3185. We can confirm this by computing the product modulo 3185.

```
In[88]:= Mod[374 * 264, 3185]
Out[88]= 1
```

Finding Inverses with PowerMod

Mathematica provides a simpler way to compute the modular inverse. The textbook uses the notation $\overline{a}$ to indicate the modular inverse of an integer. An alternate notation is a^{-1}, which calls to mind the notation used in algebra for reciprocals, as in $3^{-1} = \frac{1}{3}$.

The PowerMod function computes powers of integers in modular arithmetic. It takes three arguments: the base integer, the exponent, and the modulus. For example, the following computes $3^4 \pmod 5$.

```
In[89]:= PowerMod[3, 4, 5]
Out[89]= 1
```

For positive exponents, PowerMod is more efficient than applying Mod with the exponent computed in the argument. Moreover, PowerMod accepts negative (and even rational) exponents. In particular, applying PowerMod with second argument -1 computes the modular inverse.

```
In[90]:= PowerMod[264, -1, 3185]
Out[90]= 374
```

Note that if the integer and the modulus are not relatively prime, no inverse exists and an error is generated.

```
In[91]:= PowerMod[4, -1, 10]

         PowerMod::ninv : 4 is not invertible modulo 10. ≫

Out[91]= PowerMod[4, -1, 10]
```

Solving Congruences

We saw in Section 4.1 of this manual that the Solve function with Modulus option can be used for solving congruences. We can use this command to solve linear congruences such as $4x \equiv 3 \pmod{11}$.

```
In[92]:= Solve[4 * x == 3, Modulus → 11]
Out[92]= {{x → 9}}
```

The first argument to Solve is the congruence expressed with an Equal (==) symbol. Following the equation is a rule setting the Modulus option to the modulus. *Mathematica* returns a list whose elements express the solutions to the congruence. If there is no solution, *Mathematica* returns an empty list.

The following attempts to solve $4x \equiv 1 \pmod{10}$, which is the same as finding an inverse for 4 modulo 10 and has no solution.

```
In[93]:= Solve[4 * x == 1, Modulus -> 10]
Out[93]= {}
```

It is also possible to have multiple solutions. For example, $3x \equiv 9 \pmod{12}$.

```
In[94]:= Solve[3 * x == 9, Modulus → 12]
Out[94]= {{x → 3 + 4 C[1]}}
```

The symbol **C[1]** is used by *Mathematica* to stand for an arbitrary integer. This output indicates that any value of x of the form $3 + 4 \cdot C$ will solve the congruence. You can obtain a specific solution by substituting a particular integer for the symbol **C[1]**, using ReplaceAll (**/.**).

```
In[95]:= Solve[3 * x == 9, Modulus → 12] /. C[1] → 1
Out[95]= {{x → 7}}
```

The Chinese Remainder Theorem

The text describes two approaches to solving systems of congruences of the form

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ &\vdots \\ x &\equiv a_n \pmod{m_n} \end{aligned}$$

Mathematica provides an implementation of the Chinese remainder theorem as the function ChineseRemainder. The function takes two arguments: a list of the values $\{a_1, a_2, \ldots, a_n\}$ first and a list of the moduli $\{m_1, m_2, \ldots, m_n\}$ second. The result is the smallest positive integer that satisfies all of the congruences. As an example, we solve the congruences

$$\begin{aligned} x &\equiv 2 \pmod{3} \\ x &\equiv 4 \pmod{5} \\ x &\equiv 6 \pmod{7} \\ x &\equiv 10 \pmod{11} \end{aligned}$$

```
In[96]:= ChineseRemainder[{2, 4, 6, 10}, {3, 5, 7, 11}]
Out[96]= 1154
```

Creating a Function

We will create a function for solving systems of congruences. This implementation will be based on the construction given in the proof of the Chinese remainder theorem. While this will be less efficient than *Mathematica*'s built-in ChineseRemainder function, implementing the algorithm can help you to better understand the proof of the theorem.

Our function, which we call **crTheorem**, will accept the same arguments as ChineseRemainder: two lists, **a** and **m**, representing the values and the moduli of the congruences. It will begin with two tests to check that the lists are the same length and that the moduli are in fact pairwise relatively prime, as is required by the assumptions of the theorem. We use CoprimeQ from Section 4.3 of this manual to check that the moduli are pairwise relatively prime. If these tests fail, the following messages are generated.

```
In[97]:= crTheorem::argsize =
           "Arguments must be lists of the same size.";
In[98]:= crTheorem::argcp = "Moduli must be pairwise relatively prime.";
```

We will embed the tests inside a small function in order to make the **crTheorem** function easier to read.

```
In[99]:= crTestArgs[a_, m_] := Check[
           If[Length[a] ≠ Length[m], Message[crTheorem::argsize]];
           If[Not[Apply[CoprimeQ, m]], Message[crTheorem::argcp]];
           True,
           False]
```

Most of the work of **crTestArgs** is done in the two If statements. The first compares the lengths of **a** and **m** and the second checks whether the moduli are relatively prime. If the lists are of different lengths or the moduli are not relatively prime, the appropriate message is issued. Note that the Apply function is used in the second If statement in order to apply the function CoprimeQ, which expects integer arguments, to the list **m**.

The Check function is used to "listen" for messages. It evaluates its first argument, which in this case is the three lines ending with the expression True. If no messages are generated, then the result of the Check is the outcome of that evaluation. In this case, if the two If statements do not raise messages, then the outcome of the Check is True. However, if any messages are raised while evaluating the first argument to Check, then Check returns its second argument instead. In this case, this means that if any messages are raised, then the result of the function will be False.

As you can see below, with valid input, **crTestArgs** results in True, but it also is able to produce both error messages and return False for arguments that violate the rules.

```
In[100]:= crTestArgs[{1, 2, 3}, {5, 7, 11}]
Out[100]= True
In[101]:= crTestArgs[{1, 2}, {4, 5, 6}]
          crTheorem::argsize : Arguments must be lists of the same size.
          crTheorem::argcp : Moduli must be pairwise relatively prime.
Out[101]= False
```

When we define the **crTheorem** function, we will use patterns in the arguments to ensure that the arguments are lists of integers, and we will give **crTestArgs** as a Condition (**/;**), as shown below.

```
crTheorem[a : {__Integer}, m : {__Integer}] /;
  crTestArgs[a, m] := ...
```

The syntax **a:{__Integer}**, and likewise for **m**, names the argument and imposes the pattern that it be a sequence of integers enclosed in braces, that is, a list of integers. On the right hand side of the Condition (**/;**) operator, we apply **crTestArgs** to the arguments before the SetDelayed (**:=**) operator. With a function definition of this form, when you enter a call to **crTheorem**, *Mathematica* will first check to see that the arguments match the specified patterns. If not, for instance if you provide a different number of arguments or attempt to use anything other than two lists of integers, *Mathematica* will simply return the expression unevaluated. Assuming that you enter two lists of integers, then *Mathematica* will apply **crTestArgs**. If this function returns **False**, then *Mathematica* will return the **crTheorem** call unevaluated. Moreover, in this case, *Mathematica* will not attempt to evaluate the body of the function. Only after the arguments have matched the pattern and **crTestArgs** has returned **True**, will *Mathematica* evaluate the body of the function definition.

While the above is a bit more complicated seeming than placing the tests in the body of the function and causing them to terminate execution with a Return or an Abort, it is a much more elegant way to ensure that the function is robust.

Turning now to the main work of **crTheorem**, it begins by setting **p** equal to the product of the moduli. (Note that **p** corresponds to m in the statement of the theorem in the text. This is the only notational difference between our function and the text.) Note that we use the Apply (**@@**) operator with the Times (*****) function in order to compute this product. Times (*****) is the functional version of the multiplication operator, and combining it with Apply (**@@**) results in the product of the elements of the list **m**.

The function then needs to compute M_k and y_k. We use the indexed variables **M** and **y** for this. Note that this creates a subtle syntactic point to pay attention to. Specifically, the third element of the list **m**

is accessed via **m[[3]]**, while M_3 is referred to by **M[3]**. The values are computed within a For loop. The values for **M** are calculated by the formula $M_k = \frac{P}{m_k}$. For **y**, we use the fact that the y_k are the inverses of M_k modulo m_k. That is, $y_k \equiv M_k^{-1} \pmod{m_k}$. Finally, we compute the result $x = a_1 M_1 y_1 + a_2 M_2 y_2 + \cdots + a_n M_n y_n$ using the Sum function and return $x \pmod{p}$. Here is the function.

In[102]:=
```
crTheorem[a : {__Integer}, m : {__Integer}] /;
  crTestArgs[a, m] :=
 Module[{p, M, y, i, x},
  p = Times @@ m;
  For[i = 1, i ≤ Length[a], i++,
   M[i] = p / m[[i]];
   y[i] = PowerMod[M[i], -1, m[[i]]]
  ];
  x = Sum[a[[i]] * M[i] * y[i], {i, Length[a]}];
  Mod[x, p]
 ]
```

Note that our function produces the same result as ChineseRemainder did above.

In[103]:=
```
crTheorem[{2, 4, 6, 10}, {3, 5, 7, 11}]
```

Out[103]= 1154

Pseudoprimes

Recall from the text that a pseudoprime to the base b is a composite number n such that $b^{n-1} \equiv 1 \pmod{n}$. We will write a function to find pseudoprimes. Our function will accept two arguments, the base b and a maximum value for n, and will return a list of the pseudoprimes that it identifies.

The algorithm is fairly straightforward. We will use a For loop beginning at 3, ending with the specified maximum and increasing by 2 each time (so as to skip even integers). Within the loop, we test whether the congruence holds and whether the number is composite, using PrimeQ. Note that if the congruence fails, *Mathematica* will not bother testing primality. If it is composite, then Sow is invoked. The Reap surrounding the loop collects the pseudoprimes into a list. As we have done in the past, we use **[[2,1]]** to access the list of pseudoprimes without the additional information produced by Reap.

In[104]:=
```
findPseudoprimes[b_Integer, max_Integer] := Module[{n},
  Reap[
    For[n = 3, n ≤ max, n = n + 2,
     If[PowerMod[b, n - 1, n] == 1 && Not[PrimeQ[n]], Sow[n]]
    ]
   ][[2, 1]]
 ]
```

Note that we used the PowerMod function rather than the Power (^) operator. The PowerMod

function performs modular exponentiation intelligently, using techniques such as those discussed in Section 4.2 of the text for performing efficient modular exponentiation.

Here are the pseudoprimes to the base 2 up to 100 000.

```
In[105]:= findPseudoprimes[2, 100 000]

Out[105]= {341, 561, 645, 1105, 1387, 1729, 1905, 2047, 2465, 2701, 2821,
  3277, 4033, 4369, 4371, 4681, 5461, 6601, 7957, 8321, 8481,
  8911, 10 261, 10 585, 11 305, 12 801, 13 741, 13 747, 13 981, 14 491,
  15 709, 15 841, 16 705, 18 705, 18 721, 19 951, 23 001, 23 377,
  25 761, 29 341, 30 121, 30 889, 31 417, 31 609, 31 621, 33 153,
  34 945, 35 333, 39 865, 41 041, 41 665, 42 799, 46 657, 49 141,
  49 981, 52 633, 55 245, 57 421, 60 701, 60 787, 62 745, 63 973,
  65 077, 65 281, 68 101, 72 885, 74 665, 75 361, 80 581, 83 333,
  83 665, 85 489, 87 249, 88 357, 88 561, 90 751, 91 001, 93 961}
```

Primitive Roots and Discrete Logarithms

Mathematica includes several functions for computing primitive roots and discrete logarithms.

Primitive Roots

Mathematica provides a function, `PrimitiveRoot`, that computes primitive roots. It takes a single argument, the modulus, and returns the smallest positive primitive root for that modulus. For example, the smallest positive primitive root of 13 is 2.

```
In[106]:= PrimitiveRoot[13]

Out[106]= 2
```

Note that *Mathematica*'s `PrimitiveRoot` function applies to some non-prime moduli as well. *Mathematica* uses a definition of primitive root that is more general than the definition in the text. Specifically, an integer r is a primitive root modulo an integer n if every positive integer that is both less than n and relatively prime to n can be obtained as a power of r.

To obtain a list of all of the primitive roots of a prime, not just the first, we will make use of the `MultiplicativeOrder` function. The multiplicative order of r modulo p is defined to be the smallest positive integer m such that $r^m \equiv 1 \pmod{p}$. Equivalently, one can say that the multiplicative order of r modulo p is the number of distinct powers of r, that is, the number of different values of $r^k \bmod p$. We leave it to the reader to prove the equivalence.

The `MultiplicativeOrder` function accepts the element and modulus as arguments and returns the multiplicative order. For example, to compute the multiplicative order of 8 modulo 13, you enter the following.

```
In[107]:= MultiplicativeOrder[8, 13]

Out[107]= 4
```

We conclude from this that $8^1 \pmod{13}$, $8^2 \pmod{13}$, $8^3 \pmod{13}$, and $8^4 \pmod{13}$ are all distinct, with $8^4 \equiv 1 \pmod{13}$, but that $8^5 \equiv 8^1 \pmod{13}$. We can verify this by computing the values.

```
In[108]:= Table[PowerMod[8, k, 13], {k, 4}]
Out[108]= {8, 12, 5, 1}
In[109]:= PowerMod[8, 5, 13]
Out[109]= 8
```

Recall from Definition 3 of the textbook that a primitive root modulo a prime p is an integer r such that every nonzero element of $\mathbf{Z}_p$ is a power of r. Since there are p elements of $\mathbf{Z}_p$, there are $p-1$ nonzero elements. Consequently, being a primitive root modulo a prime p is identical to having multiplicative order $p-1$. The fact that, as we calculated above, the multiplicative order of 8 modulo 13 is 4, not $13-1=12$ implies that 8 is not a primitive root modulo 13. However, 6 has multiplicative order 12 modulo 13.

```
In[110]:= MultiplicativeOrder[6, 13]
Out[110]= 12
```

Consequently, 6 is a primitive root modulo 13.

To summarize, for r to be a primitive root modulo a prime p, it is necessary and sufficient that the multiplicative order of r modulo p be equal to $p-1$. This observation provides us with a convenient way to list all of the primitive roots for a given prime. We just consider each possible r from 2 to $p-1$ and calculate their multiplicative order with `MultiplicativeOrder`. Those whose order is $p-1$ are included in the list. Here is the function.

```
In[111]:= allPrimitiveRoots[p_?PrimeQ] :=
           Select[Range[2, p - 1], MultiplicativeOrder[#, p] == p - 1 &]
```

We make two comments about this function. First, we ensure that the argument is prime by using the PatternTest syntax: the pattern, in this case `Blank` (`_`) is followed by a question mark and the name of a function, `PrimeQ`, that performs a test.

Second, the `Select` function is used to obtain, given a list and a condition, the sublist of those elements meeting the condition. `Select` requires two arguments. First, the initial list. Second, a function in one argument that returns **`True`** for the desired elements. For the second argument, you can either provide the name of a function that accepts a single argument, or, as in the above, the function can be given as a pure `Function` (**`&`**). Recall that a pure function is terminated with an ampersand (**`&`**) and uses a `Slot` (**`#`**) for its argument.

Applying the **`allPrimitiveRoots`** function to 13 produces the list of all primitive roots modulo 13.

```
In[112]:= allPrimitiveRoots[13]
Out[112]= {2, 6, 7, 11}
```

Discrete Logarithms

The `MultiplicativeOrder` function can also be used to find discrete logarithms. Recall that the discrete logarithm of a modulo a prime p to the base r is a number e such that $r^e \equiv a \pmod{p}$. The multiplicative order of r modulo p, as we defined it above, is the smallest positive m such that $r^m \equiv 1 \pmod{p}$. Notice that the congruences defining these two concepts are similar. Indeed, the discrete logarithm problem is simply more general than the multiplicative order problem, replacing the

specific 1 with an arbitrary a.

The `MultiplicativeOrder` function accepts a third argument which generalizes it to include the computation of discrete logarithms. Recall that the first argument of `MultiplicativeOrder` is the value r, the second is the prime p. By providing a third argument, a, `MultiplicativeOrder` will compute the discrete logarithm of a modulo p to the base r. That is, $\log_r a$ is computed by **`MultiplicativeOrder[r, p, a]`**.

To compute the discrete logarithm of 3 modulo 11 to the base 2, you enter the following.

```
In[113]:= MultiplicativeOrder[2, 11, 3]
Out[113]= 8
```

4.5 Applications of Congruences

In this section we will see how *Mathematica* can be used to further explore the applications of congruences discussed in the text. In particular, we will see how to use a hashing function to store student information in a list. We will create a pseudorandom number generator. And we will write a function that will check the validity of an ISBN.

Hashing Functions

The first application we will explore is the hashing function. Suppose that a small school wants to store information about its students. In particular, each student has a unique four digit identification number and a GPA, which is a real number between 0 and 4.

Initial Examples

Each student record will be stored as a list with first element the student ID and the second element the student's GPA. Here are three example students.

```
In[114]:= student1 = {7319, 3.21};
          student2 = {2908, 2.89};
          student3 = {6578, 3.42};
```

Our student records are going to be stored in a list. Because the school is small, it will suffice to allocate space for 57 records in the school's database and so we create a list with 57 entries all initialized to 0.

```
In[117]:= studentRecords = Table[0, {57}]
Out[117]= {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
           0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
           0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

In order to store a student record in the list (which represents the school's database), we need to apply a hashing function to the unique student ID. The hashing function we'll use is $h(k) = k \bmod 57 + 1$. Note that the addition of 1 is to occur after the computation of $k \bmod 57$. It is included in our function because the indices in our **`studentRecords`** list run from 1 to 57 while the values of $k \bmod 57$ range from 0 to 56.

The following function accepts a student ID as input and returns the result of applying the hashing

function to the ID number.

```
In[118]:= calculateHash[id_Integer] := Mod[id, 57] + 1
```

For example,

```
In[119]:= calculateHash[student1[[1]]]
Out[119]= 24
```

This indicates that **student1**'s record should be stored in location 24. We store a record in a particular location in the usual way.

```
In[120]:= studentRecords[[24]] = student1
Out[120]= {7319, 3.21}
```

Note that accessing location 24 returns the list containing the student's data.

```
In[121]:= studentRecords[[24]]
Out[121]= {7319, 3.21}
```

To access the ID and GPA of the student stored in location 24, we can use a second pair of double brackets with 1 or 2 to access the ID or GPA.

```
In[122]:= studentRecords[[24]][[1]]
Out[122]= 7319
```

Or we can include the record number and the index of the particular piece of data in a single Part (**[[…]]**) operation.

```
In[123]:= studentRecords[[24, 1]]
Out[123]= 7319
```

We can store **student2**'s information in the same way.

```
In[124]:= calculateHash[student2[[1]]]
Out[124]= 2
In[125]:= studentRecords[[2]] = student2
Out[125]= {2908, 2.89}
```

If we try to store **student3**'s data, we find that a collision occurs.

```
In[126]:= calculateHash[student3[[1]]]
Out[126]= 24
```

Since **student3** has the same hash value as **student1** did, we look for the next free location. Check location 25.

```
In[127]:= studentRecords[[25]] == 0
Out[127]= True
```

Since location 25 is still equal to 0, we know that it has not been used and we store **student3**'s record in location 25.

```
In[128]:= studentRecords[[25]] = student3
Out[128]= {6578, 3.42}
```

Printing Records

Before going any further, take a look at the current state of **studentRecords**.

```
In[129]:= studentRecords
Out[129]= {0, {2908, 2.89}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, {7319, 3.21}, {6578, 3.42}, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

This is not very easy to read. We need to write a function to print out the data in a more useful format. To do this, we loop through the elements of the list and, for those that are non-zero, print the index and the data stored in that position. Note that because we are potentially comparing numbers to lists, we must use UnsameQ (=!=) rather than Unequal (!=).

```
In[130]:= printRecords[database_List] := Module[{},
            For[i = 1, i ≤ Length[database], i++,
             If[database[[i]] =!= 0, Print[i, " ", database[[i]]]]
            ]
           ]
In[131]:= printRecords[studentRecords]
          2 {2908, 2.89}
          24 {7319, 3.21}
          25 {6578, 3.42}
```

A Function for Storing New Records

Now we'll write a function **storeRecord** to automate the process of adding records to the database. **storeRecord** will accept two arguments, the ID and GPA of a student, and will add that student's record to the **studentRecords** list (the database).

The first step in implementing **storeRecord** will be to assign to a local variable, which we'll call **record**, the list representing the student record. Then **storeRecord** needs to determine the location in the **studentRecords** list in which the record will be stored. In particular, it will need to avoid collision. To do this, we'll use something similar to the linear probing function defined in the text. Beginning with $i = 0$, we'll calculate $h(k + i) = (k + i) \bmod 57 + 1$. We will store that value in the local name **hash** and check to see if **studentRecords[[hash]]** is 0. If so, then we know the list does not already have a record stored in that location and we can stop our search for an open position. The Break function causes the loop in which it is contained to terminate. If the location is not empty, we increment i and continue looking. Once we have found an open position, we only need to assign our record to that position. We give Null as the final expression so that the function does not display anything when the record is successfully stored.

Note that for this function we chose not to include the database as a parameter, but instead we've described the function in relation to the **studentRecords** list that we began above. This can result in a significant improvement in performance, especially when the list of records is long, because the database does not have to be passed as an argument to the function and then returned from it each time

a new record is to be stored. The disadvantage, of course, is that in order to use a different name for the database, we have to revise the **storeRecord** function.

Here is the completed **storeRecord** function.

```
In[132]:= storeRecord[id_Integer, gpa_Real] := Module[{record, hash, i},
           record = {id, gpa};
           For[i = 0, i ≤ 56, i++,
            hash = calculateHash[id + i];
            If[studentRecords[[hash]] === 0, Break[]]
           ];
           studentRecords[[hash]] = record;
           Null
          ]
```

Now we add a few records.

```
In[133]:= storeRecord[2216, 1.98]

In[134]:= storeRecord[1325, 3.14]

In[135]:= storeRecord[7061, 3.51]
```

Look again at **studentRecords**.

```
In[136]:= printRecords[studentRecords]

          2 {2908, 2.89}

          15 {1325, 3.14}

          24 {7319, 3.21}

          25 {6578, 3.42}

          51 {2216, 1.98}

          52 {7061, 3.51}
```

Retrieving Records

We now have functions for storing a student record in our database and for printing all of the records. But we also need a way to retrieve the record for a particular student. Indeed, one of the benefits of hash functions is that they provide an efficient way to look up records — given the unique key, we need only apply the hash function to determine the memory location in which the record is stored (subject to collision, of course).

Our **retrieveRecord** function will accept a student ID number as its input and return the list storing the student's record. Most of the work will take place within the same `For` loop as was in the **storeRecord** function. This time, we enclose the loop in a `Catch`, as the means of short-circuiting the loop and passing the result, either `Null` or the record, out of the function. We test to make sure the location we're looking in is non-zero. If the location is 0, that tells us that the entry does not exist and the procedure will display the following message and return `Null`.

```
In[137]:= retrieveRecord::missing = "Desired record does not exist.";
```

Assuming the location is not 0, we check to see if the ID of the record in that position is the ID we're looking for. If so, we return the student data. If the ID is not the one we're searching for, it must have

been the case that our record was pushed down the line because of a collision and we continue the loop.

```
In[138]:= retrieveRecord[id_Integer] := Module[{hash, i},
           Catch[
            For[i = 0, i ≤ 56, i++,
             hash = calculateHash[id + i];
             If[studentRecords[[hash]] === 0,
              Message[retrieveRecord::missing];
              Throw[Null]
             ];
             If[studentRecords[[hash]][[1]] == id,
              Throw[studentRecords[[hash]]]
             ]
            ]
           ]
          ]

In[139]:= retrieveRecord[1325]

Out[139]= {1325, 3.14}

In[140]:= retrieveRecord[7061]

Out[140]= {7061, 3.51}
```

Pseudorandom Numbers

Many applications require sequences of random numbers, which are important in cryptography and in generating data for computer simulations. It is impossible to produce a truly random stream of numbers using software only, since software employs algorithms. Anything that can be generated by an algorithm is, by definition, not random. Fortunately, for most applications, it is sufficient to generate a stream of pseudorandom numbers. This is a stream of numbers that, while not truly random, exhibits some of the same properties of a random number stream. Effective algorithms for generating pseudorandom numbers can be based on modular arithmetic. We will implement a linear congruential method, as described in the text.

We must choose four integers: the modulus m, the multiplier a with $2 \le a < m$, the increment c with $0 \le c < m$, and the seed x_0 with $0 \le x_0 < m$. Then we can create a sequence of pseudorandom numbers using the recursive formula $x_{n+1} = (a \cdot x_n + c) \bmod m$. It is common to have the seed chosen based on some physical property accessible by the computer, for instance the time. Alternately, the seed can be based on some truly random physical process, such as radioactive decay. For this example, we will generate a seed by multiplying by 1000 the result of the `SessionTime` function, which gives the total number of seconds since the beginning of the *Mathematica* session. We apply `Floor` to be certain that we obtain an integer.

```
In[141]:= Floor[1000 * SessionTime[]]

Out[141]= 7603
```

We will write two functions that generate random student IDs and GPAs that we can use to add some random records to our **studentRecords** from above. We first write the function **randomIDs**,

which will accept a positive integer as input to control the number of IDs to generate. It will return a sequence of that number of random student IDs.

Recall that a student ID, in the context described above, is a four-digit number. So our random numbers must be between 1000 and 9999. We can obtain such numbers by generating random integers between 0 and 8999 and adding 1000. So our modulus will be 8999. We will choose a multiplier of 57 and an increment of 328. (These values were chosen for no particular reason, but in practice the choice of c and a can be an important consideration. See the references in the textbook for more information.) The seed will be determined from `SessionTime` as described above.

The function is straightforward.

In[142]:=
```
randomIDs[n_Integer] := Module[{m = 8999, a = 57, c = 328, x, i},
  x = Mod[Floor[1000 * SessionTime[]], m];
  Reap[
    For[i = 1, i ≤ n, i++,
     Sow[x = Mod[a * x + c, m]]
    ]
   ][[2, 1]]
 ]
```

We generate 10 random IDs by applying the procedure to 10.

In[143]:=
```
someIDs = randomIDs[10]
```

Out[143]=
```
{3578, 6296, 8239, 2003, 6511, 2496, 7615, 2431, 3910, 7222}
```

To generate GPAs, the approach will be essentially the same. We use the pure multiplicative generator mentioned in the text with modulus $2^{31} - 1$, multiplier 7^5, and increment 0. This will produce integers between 0 and $2^{31} - 2$. To obtain numbers between 0 and 4, we'll divide the random integer by $2^{31} - 2$ and multiply by 4.

In[144]:=
```
randomGPAs[n_Integer] := Module[{m = 2^31 - 1, a = 7^5, x, i},
  x = Mod[Floor[1000 * SessionTime[]], m];
  Reap[
    For[i = 1, i ≤ n, i++,
     x = Mod[a * x, m];
     Sow[Round[(x / (m - 1)) * 4, 0.01]]
    ]
   ][[2, 1]]
 ]
```

In[145]:=
```
someGPAs = randomGPAs[10]
```

Out[145]=
```
{0.25, 3.13, 3.87, 2.61, 0.1, 2.06, 0.07, 0.06, 1.6, 2.92}
```

Note that we use `Round` to round the random number to the nearest hundredth, so that out output has at most two digits after the decimal place. The second argument to `Round` being 0.01 means that the result will be rounded to the nearest multiple of 0.01.

Now we add the random students to **`studentRecords`**.

```
In[146]:= For[i = 1, i ≤ 10, i++,
           storeRecord[someIDs[[i]], someGPAs[[i]]]
          ]

In[147]:= printRecords[studentRecords]

          2 {2908, 2.89}
          9 {2003, 2.61}
          14 {6511, 0.1}
          15 {1325, 3.14}
          24 {7319, 3.21}
          25 {6578, 3.42}
          27 {6296, 3.13}
          32 {8239, 3.87}
          35 {7615, 0.07}
          36 {3910, 1.6}
          38 {2431, 0.06}
          41 {7222, 2.92}
          45 {3578, 0.25}
          46 {2496, 2.06}
          51 {2216, 1.98}
          52 {7061, 3.51}
```

Check Digits

We conclude this section with a function to check the validity of an ISBN. Recall that the ISBN-10 code consists of 10 digits, the last of which is computed by the formula

$$x_{10} = \sum_{i=1}^{9} i \cdot x_i \pmod{11}$$

The symbol X is used in case $x_{10} = 10$.

Our **checkISBN** function will accept the ISBN as a string. It is necessary that we use strings in case the ISBN contains X as the check digit. Consider the ISBN below.

```
In[148]:= isbnExample = "0073383090"

Out[148]= 0073383090
```

In *Mathematica*, in order to access a character within a string, you use the StringTake function. The first argument to StringTake is the string. In order to obtain a single character, you provide as the second argument a list with the position of the character as the sole element. For example, the third

character of our example is obtained as follows.

```
In[149]:= StringTake[isbnExample, {3}]
Out[149]= 7
```

Note that the output is still a string, however.

```
In[150]:= StringTake[isbnExample, {3}] // FullForm
Out[150]//FullForm=
"7"
```

In order to perform arithmetic, we need to turn the character into an integer. To do this, we can use the ToExpression function. When ToExpression is applied to a string, *Mathematica* interprets the string as *Mathematica* input. In this example, the string "7" will be interpreted as if we had entered 7 on an input line.

```
In[151]:= ToExpression[StringTake[isbnExample, {3}]] // FullForm
Out[151]//FullForm=
7
```

Conversely, the function ToString will convert an expression into a string. The following converts the number 7 into the string "7".

```
In[152]:= ToString[7] // FullForm
Out[152]//FullForm=
"7"
```

Our function will compute the sum indicated by the formula above using Sum. Recall that the first argument to Sum is an expression in terms of an index variable and the second argument is the range for the variable. Once the value of x_{10} is determined, we compare it to the check digit. This is only slightly complicated by the fact that a check digit of 10 corresponds to the symbol X. Note that we must compare the check digit with the value of x_{10} as strings in both the case that the $x_{10} = 10$ and when $x_{10} < 10$. This is because the last digit may be X, whether it should be or not, and applying the ToExpression function to the string "X" will result in the symbol **X**, which may in fact be assigned to an expression.

```
In[153]:= checkISBN[isbn_String] := Module[{i, check},
            check = Mod[
              Sum[i * ToExpression[StringTake[isbn, {i}]], {i, 9}], 11];
            Which[
             check < 10,
             ToString[check] == StringTake[isbn, {10}],
             check == 10,
             StringTake[isbn, {10}] == "X"
            ]
           ]
```

Recall that a Which statement evaluates the first argument and if it is true, returns the value of the second argument. If the first argument is false, then it moves to the third argument and evaluates that test, and so on.

```
In[154]:= checkISBN[isbnExample]
Out[154]= True
In[155]:= checkISBN["084930149X"]
Out[155]= False
In[156]:= checkISBN["232150031X"]
Out[156]= True
```

4.6 Cryptography

In this, the final section of Chapter 4, we will see how *Mathematica* can be used to encode and decode strings using two of the approaches described in the textbook. Specifically, we will see how to implement a classical affine cypher and the RSA system.

Encoding Strings

Before we can implement the encryption algorithms, we need to encode strings as numbers. In this manual, we will deviate slightly from the convention used in the textbook. Instead of assigning the letter A to 0, B to 1, and so on with Z assigned to 25, we will assign the space character to 0, A to 1, B to 2, and so on with Z set to 26. We will then work modulo 27 instead of 26.

Some Functions for Working with Strings

Mathematica contains a variety of functions for working with strings. We have already seen, in the previous section, the `StringTake` function. This function is used to obtain a substring of a given string. Its first argument is the initial string, and the second argument specifies what part of the string to return.

If the second argument to `StringTake` is a positive integer, n, the result will be the first n characters. For example, the following produces the first 5 characters in the string "The quick brown fox".

```
In[157]:= StringTake["The quick brown fox", 5]
Out[157]= The q
```

If the second argument is a negative integer, say $-n$, the result is the final n characters.

```
In[158]:= StringTake["The quick brown fox", -5]
Out[158]= n fox
```

To obtain the character in a particular location, as was done in Section 4.5, the second argument is given as a list containing the desired position. For example, to obtain the character in position 8, enter the following.

```
In[159]:= StringTake["The quick brown fox", {8}]
Out[159]= c
```

With a pair of integers in a list given as the second argument, `StringTake` returns the substring between the given two positions. The following produces the substring consisting of characters 5 through 8.

```
In[160]:= StringTake["The quick brown fox", {5, 8}]
Out[160]= quic
```

The ToUpperCase function makes all of the letters in its input string upper case.

```
In[161]:= ToUpperCase["The quick brown fox"]
Out[161]= THE QUICK BROWN FOX
```

The ToUpperCase command is useful in this context because it means we only have to work with the 26 uppercase letters and the space character instead of the full 53 characters, including both upper and lower case letters and space.

The next function we will need is the Characters function and its inverse StringJoin (**<>**). The Characters function takes a string and returns a list of characters.

```
In[162]:= Characters["THE QUICK BROWN FOX"]
Out[162]= {T, H, E,  , Q, U, I, C, K,  , B, R, O, W, N,  , F, O, X}
```

The StringJoin function does the opposite. Given two or more strings as arguments, it joins them into one string.

```
In[163]:= StringJoin["The quick", " ", "brown fox"]
Out[163]= The quick brown fox
```

StringJoin also has an operator form, **<>**.

```
In[164]:= "The quick" <> " " <> "brown fox"
Out[164]= The quick brown fox
```

It can also accept a list of strings as its arguments and will return the string formed by joining the members of the list.

```
In[165]:= StringJoin[{"T", "H", "E", " ", "Q", "U", "I", "C",
           "K", " ", "B", "R", "O", "W", "N", " ", "F", "O", "X"}]
Out[165]= THE QUICK BROWN FOX
```

Mapping Characters to Integers

To represent the function that maps characters to integers, and its inverse, we will use two indexed variables, **charToNum** and **numToChar**. In the **charToNum** variable, the space character and capital letters will serve as the indices with the corresponding integers the entries. The **numToChar** variable will be the reverse.

To define these two indexed variables, rather than entering each individual assignment manually, we will apply the MapThread function. Recall that MapThread is used to apply a function to lists of arguments. The first argument to MapThread is a function. In this case, we will create a pure function that sets a value of the indexed variable. The second argument to MapThread is a list of lists, with the inner lists containing the arguments. The elements of the first sublist are values of the first argument of the function, the elements of the second sublist values of the second argument and so forth. The result of MapThread is that the function is evaluated on corresponding pairs of elements from the lists.

Here, MapThread is used to define **charToNum**. Note that the parentheses are needed in the pure

function, since Function (&) has greater precedence than Set (=).

```
In[166]:= MapThread[(charToNum[#1] = #2) &,
            {Characters[" ABCDEFGHIJKLMNOPQRSTUVWXYZ"], Range[0, 26]}];
```

After executing this MapThread expression, **charToNum**, issued with a character as an index, returns the appropriate value.

```
In[167]:= charToNum["F"]
Out[167]= 6
```

The **numToChar** indexed variable is created in the same way, but reversed.

```
In[168]:= MapThread[(numToChar[#1] = #2) &,
            {Range[0, 26], Characters[" ABCDEFGHIJKLMNOPQRSTUVWXYZ"]}];
In[169]:= numToChar[6]
Out[169]= F
```

Converting Between a String and a Numerical Representation

We now have the tools needed to encode a string as a list of numbers and a decode the numerical representation as a string.

In the **stringToNums** function, we will first apply ToUpperCase and Characters to produce a list of uppercase characters. Then we will use the Map (/@) function to apply the **charToNum** table to each character. When Map (/@) is applied to a function (in this case a function represented by an indexed variable) and a list, it returns the list obtained by applying the function to each element of the list.

```
In[170]:= stringToNums[s_String] := Module[{charList},
            charList = Characters[ToUpperCase[s]];
            Map[charToNum, charList]
          ]
In[171]:= stringToNums["The quick brown fox"]
Out[171]= {20, 8, 5, 0, 17, 21, 9, 3, 11, 0, 2, 18, 15, 23, 14, 0, 6, 15, 24}
```

The **numsToString** function begins with a list of integers and returns the string.

```
In[172]:= numsToString[numList : {__Integer}] := Module[{charList},
            charList = Map[numToChar, numList];
            StringJoin[charList]
          ]
In[173]:= numsToString[{8, 5, 12, 12, 15, 0, 23, 15, 18, 12, 4}]
Out[173]= HELLO WORLD
```

Now that we have the ability to convert strings into a numerical representation and back again, we are ready to implement our encryption algorithms.

Classical Cryptography

We will now implement an affine cipher in *Mathematica*. Recall from the text that a general affine cipher has the form

$$f(p) = (a \cdot p + b) \pmod{27}$$

where p is an integer corresponding to a character that is to be encrypted. We will refer to the pair (a, b) as the key to the cipher. For decryption to be feasible, the key must be chosen so that f is a bijection. This amounts to choosing an a that is relatively prime to 27. (Note that the text uses a modulus of 26 where we use 27 because we are considering space to be an encodable character.)

Encrypting a string requires three simple steps. First, the string is transformed into its numerical representation via **stringToNums**. Second, the function f is applied to each number. And third, the **numsToString** function transforms the result back into a string. Our **affineCipher** function accepts as input a string and values of a and b.

We ensure that the argument a is relatively prime to 27 by imposing a Condition (**/;**) and creating a message if it is not.

```
In[174]:= affineCipher::arga =
            "Second argument must be relatively prime to 27.";
```

Recall that following the name and arguments of a function definition with the condition operator **/;** and an expression that results in true or false allows you to create functions that will not execute on invalid arguments.

```
In[175]:= affineCipher[s_String, a_Integer, b_Integer] /;
            If[CoprimeQ[a, 27], True,
             Message[affineCipher::arga]; False] := Module[{S, T},
            S = stringToNums[s];
            T = Map[Mod[a * # + b, 27] &, S];
            numsToString[T]
           ]
```

Note the use of Map (**/@**) to apply the function $f(p)$, defined as a pure function, to each character.

We now use the cipher to encrypt "The quick brown fox" with the key (5, 3).

```
In[176]:= affineCipher["The quick brown fox", 5, 3]

Out[176]= VPACG URDCMLXJSCFXO
```

To decrypt the message, we use the same function. The discussion following Example 4 in Section 4.5 of the text indicates that decrypting amounts to solving $c \equiv (a \cdot p + b) \pmod{27}$ for p. As the text shows, we obtain $p \equiv a^{-1}(c - b) \pmod{27} \equiv a^{-1}\, c - a^{-1}\, b \pmod{27}$. In other words, to decrypt a message encrypted using the key (a, b), we use the same procedure but with key $\left(a^{-1}, -a^{-1}\, b\right)$.

First, compute the inverse of $a = 5$.

```
In[177]:= PowerMod[5, -1, 27]

Out[177]= 11
```

And then $-a^{-1}\, b$, being sure to include the negative.

```
In[178]:= Mod[-11 * 3, 27]
Out[178]= 21
```

Thus the decryption key is (11, 21).

```
In[179]:= affineCipher["VPACG URDCMLXJSCFXO", 11, 21]
Out[179]= THE QUICK BROWN FOX
```

RSA Encryption

We will now see how to use *Mathematica* to implement the RSA cryptosystem. Implementing the RSA system involves two steps: key generation and the encryption algorithm.

To construct keys in the RSA system, we need to find pairs of large primes, say with 200 digits each. Since messages can be decrypted by anyone who can factor the product of these primes, the two primes must be large enough so that their product is extremely difficult to factor. A 400 digit integer fits the bill since factoring requires an extremely large amount of computer time.

Because the use of very large prime numbers would make our examples impractical as examples, we shall illustrate the RSA system using smaller primes. We will discuss at the end of this section how you can use *Mathematica* to generate large prime numbers.

Key Generation

The first step in key generation is to choose two distinct large prime numbers, p and q. From these, we produce the public key, which consists of the public modulus $n = p \cdot q$ and the public exponent e which is relatively prime to $\phi(n) = (p-1)(q-1)$. We also produce the private key, consisting of the public modulus n and the inverse of e modulo $(p-1)(q-1)$. Since e is unrelated to the primes p and q, it can be generated in a number of ways. For our implementation below, we will take e to be 13.

Here is a *Mathematica* function to handle key generation. The **generateKeys** function accepts as input two prime numbers. It returns a list of two lists where the sublists are the public and private keys. That is, it returns $\{\{n, e\}, \{n, e^{-1}\}\}$. Given the primes p and q, the procedure computes $n = p \cdot q$, $\phi(n) = (p-1)(q-1)$, and $d = e^{-1} \pmod{\phi(n)}$.

```
In[180]:= generateKeys[p_?PrimeQ, q_?PrimeQ] := Module[{n, phin, e, d},
            e = 13;
            n = p * q;
            phin = (p - 1) * (q - 1);
            d = PowerMod[e, -1, phin];
            {{n, e}, {n, d}}
          ]
```

In a practical RSA implementation, we would likely use some of the techniques discussed at the end of this section to incorporate into our **generateKeys** procedure the generation of the primes p and q, rather than passing them as arguments.

We generate keys using the prime numbers $p = 59$ and $q = 71$

```
In[181]:= keys = generateKeys[59, 71]
Out[181]= {{4189, 13}, {4189, 937}}
```

The public and private keys are:

```
In[182]:= publickey = keys[[1]]
Out[182]= {4189, 13}
In[183]:= privatekey = keys[[2]]
Out[183]= {4189, 937}
```

Encoding

Now that we have the keys, we turn to encoding the message. As described in the text, we encode the message in much the same way as for affine ciphers, except that we block groups of characters into single integers. The block length must be chosen so that, after conversion, the largest integer produced is less than the modulus n. Here, we have $n = 4198$ and the largest block that can be produced is 2626 for "ZZ".

We need to ensure that this part of the process is reversible. Consider the string "VA". This comprises one block. Since "V" has code 22 and "A" has code 1, it is tempting to code "VA" as 221. But when you go to convert this back to a string, it is impossible to tell if it was 22 and 1 indicating "VA" or if it was 2 and 21, which represents "BU". To avoid this, we code "A" as 01. Or, what amounts to the same thing, when we compose the block, we multiply the value of the first character by 100.

For a specific example, consider the message "SECRET MESSAGE". We can use our **stringToNums** function from above to get the numeric representation of each character.

```
In[184]:= messageString = stringToNums["SECRET MESSAGE"]
Out[184]= {19, 5, 3, 18, 5, 20, 0, 13, 5, 19, 19, 1, 7, 5}
```

You can see that the first pair should be encoded as 1905, the second as 0318, and so on. Note that the extra 0 is unnecessary in second block, since 0318 and 318 are numerically equal. We can obtain the desired results by multiplying the first number in each pair by 100 as follows.

```
In[185]:= messageCode =
           Table[messageString[[i - 1]] * 100 + messageString[[i]],
            {i, 2, Length[messageString], 2}]
Out[185]= {1905, 318, 520, 13, 519, 1901, 705}
```

Note that the final 2 in the list describing the table variable **i** indicates that the variable should be increased by 2 each iteration.

Encryption

The encryption algorithm will take as input this list of integers and the public key. Each message block m_i is transformed into a ciphertext block c_i with the function $C \equiv M^e \pmod{n}$.

```
In[186]:= RSA[{n_Integer, e_Integer}, msg : {__Integer}] := Module[{c},
            c = Map[PowerMod[#, e, n] &, msg]
           ]
```

Observe how the arguments for this function were defined. The function **RSA** accepts a list consisting of two integers named **n** and **e** and a list of integers called **msg**.

Our "SECRET MESSAGE" is encrypted as

```
In[187]:= cipherText = RSA[publickey, messageCode]
Out[187]= {723, 3360, 2306, 1979, 2695, 917, 1863}
```

Decryption is accomplished by applying the same algorithm with the private decryption key.

```
In[188]:= RSA[privatekey, cipherText]
Out[188]= {1905, 318, 520, 13, 519, 1901, 705}
```

Note that the result is identical to **messageCode** and it can be decoded into the message "SECRET MESSAGE".

Generating Large Primes

If you were to use small primes, as we did in the example, there would be no real security. Anyone could factor *n*, the product of the primes, and then could compute the decrypting key *d* from the encrypting key *e*.

Using *Mathematica*'s computational abilities, we can generate fairly large prime numbers for use in an RSA key. Remember that what is needed is a pair of prime numbers, each of about 200 digits. Moreover, they should be selected in an unpredictable fashion. To do this in *Mathematica*, we can use the RandomPrime function.

The first argument to RandomPrime is either an integer, in which case the function returns a prime up to that value, or a list of two integers, in which case the prime generated will be between them. The function can also accept a second argument in order to produce more than one at a time. For example, with second argument 2, the function will return a list of two primes.

Of course, the primes generated by RandomPrime are in fact pseudorandom, not truly random.

To produce two random primes with between 200 and 300 digits, we call RandomPrime as follows.

```
In[189]:= RandomPrime[{10^200, 10^300}, 2]
Out[189]= {457 470 868 081 910 306 859 462 238 743 895 755 314 347 261 556 410 903 ↘
    640 625 321 769 946 844 773 754 142 026 227 432 145 916 096 381 792 ↘
    696 409 400 992 053 626 995 507 142 873 626 500 111 933 890 008 195 ↘
    751 585 051 179 309 967 799 727 851 917 975 154 148 052 787 289 033 ↘
    177 677 770 723 780 447 683 644 745 273 832 584 852 454 770 176 179 ↘
    686 565 703 694 327 587 460 149 242 772 179 945 323 701,
   842 162 494 477 098 003 796 865 021 670 155 127 125 584 249 463 527 635 ↘
    849 111 696 959 608 420 595 155 514 529 336 562 896 667 053 489 876 ↘
    715 729 724 227 110 612 112 784 311 745 623 039 975 364 378 292 920 ↘
    705 556 809 814 274 749 416 848 059 270 839 233 189 581 662 359 185 ↘
    337 025 502 391 921 791 572 073 995 567 674 054 818 872 858 284 792 ↘
    885 548 490 389 975 624 702 221 425 127 071 640 653 459}
```

It is left to the reader to incorporate these ideas in improved versions of the **generateKeys** and **RSA** functions.

Solutions to Computer Projects and Computations and Explorations

Computer Projects 3

> Given a positive integer, find the Cantor expansion of this integer (see the preamble to Exercise 48 of Section 4.2).

Solution: Recall the definition of the Cantor expansion. Given an integer a, the Cantor expansion of a is

$$a = a_n\, n! + a_{n-1}(n-1)! + \cdots + a_2\, 2! + a_1\, 1!$$

Observe that every term except for $a_1\, 1!$ is divisible by 2. That is,

$$a_n\, n! + a_{n-1}(n-1)! + \cdots + a_2\, 2! + a_1\, 1! \,(\mathrm{mod}\, 2) = a_1$$

So set $a_1 = a \,(\mathrm{mod}\, 2)$. And let y_1 be the remainder with the 2 divided out. In other words, $y_1 = \frac{a-a_1}{2}$, or

$$y_1 = \frac{a_n\, n! + a_{n-1}(n-1)! + \cdots + a_2\, 2!}{2} = a_n\, \frac{n!}{2} + a_{n-1}\, \frac{(n-1)!}{2} + \cdots + a_3\, 3 + a_2$$

Now every term other than the last contains a factor of 3, so set $a_2 = y_1 \,(\mathrm{mod}\, 3)$ and let $y_2 = \frac{y_1-a_2}{3}$.

In general, $a_k = y_{k-1} \,(\mathrm{mod}\, k+1)$ and $y_k = \frac{y_{k-1}-a_k}{k+1}$. It is left to the reader to verify that this process produces the Cantor expansion of a.

The algorithm described above leads to the function below which accepts a positive integer as input and returns a list of integers $\{a_1, a_2, \ldots, a_n\}$.

```
In[190]:= cantorExpansion[n_Integer] := Module[{a, k = 1, y = n},
            Reap[
              While[y ≠ 0,
               a = Mod[y, k + 1];
               Sow[a];
               y = (y - a) / (k + 1);
               k++
              ]
             ][[2, 1]]
           ]

In[191]:= cantorExpansion[471]

Out[191]= {1, 1, 2, 4, 3}
```

Computer Projects 21

> Generate a shared key using the Diffie-Hellman key exchange protocol.

Solution: Recall from Section 4.6 of the text the Diffie-Hellman key exchange protocol.

(1) Alice and Bob agree on a prime number p and a primitive root a of p. For this example, we'll use a relatively small prime, one with between 6 and 8 digits.

```
In[192]:= dhPrime = RandomPrime[{10^6, 10^8}]
Out[192]= 24 720 211
```

For the primitive root, we'll use PrimitiveRoot to get the smallest primitive root of the prime.

```
In[193]:= dhRoot = PrimitiveRoot[dhPrime]
Out[193]= 2
```

(2) Alice chooses a secret integer k_1. Let's choose 421 since this is Computer Project 21 in Chapter 4. We need to compute $a^{k1} \pmod p$ and send the resulting value to Bob.

```
In[194]:= aliceSends = PowerMod[dhRoot, 421, dhPrime]
Out[194]= 18 566 175
```

Note that we use PowerMod so that the exponentiation is computed efficiently.

(3) Bob also chooses a secret integer k_2. From the perspective of Alice, we won't know what value of k_2 that Bob chooses, only the value of $a^{k2} \pmod p$. So we'll have *Mathematica* choose k_2 randomly in the computation.

```
In[195]:= bobSends =
            PowerMod[dhRoot, RandomInteger[{1, dhPrime}], dhPrime]
Out[195]= 14 065 885
```

The RandomInteger function has similar syntax to RandomPrime. Given a list of two integers as its argument, it returns a pseudorandom integer between the two integers. There are no necessary restrictions on the values of k_1 and k_2, however, $a^{k+p} \equiv a^k \pmod p$, so it is no loss to assume the integers lie between 1 and p.

(4) and (5) Alice computes $\left(a^{k2}\right)^{k1} \pmod p$ using the result that Bob transmitted and her k_1. Bob does the same using the value he got from Alice and his secret k_2.

```
In[196]:= sharedKey = PowerMod[bobSends, 421, dhPrime]
Out[196]= 2 470 046
```

At the conclusion, both Alice and Bob know this shared key, but no one else does.

Computations and Explorations 1

Determine whether $2^p - 1$ is prime for each of the primes not exceeding 100.

Solution: To solve this problem, we will write a *Mathematica* program that tests each prime p less than or equal to a given value to see whether $2^p - 1$ is a Mersenne prime. The function will output a list of those primes p for which $2^p - 1$ is prime.

```
In[197]:= checkMersenne[max_Integer] := Module[{p = 2},
            Reap[
              While[p ≤ max,
               If[PrimeQ[2^p - 1], Sow[p]];
               p = NextPrime[p]
              ]
             ][[2, 1]]
           ]
```

The primes p less than 100 such that $2^p - 1$ is prime are:

```
In[198]:= checkMersenne[100]
Out[198]= {2, 3, 5, 7, 13, 17, 19, 31, 61, 89}
```

It is of note that there is a better test, called the Lucas-Lehmer test, that is more efficient than PrimeQ for checking primality of numbers of the form $2^p - 1$, and can be implemented in *Mathematica*. For a complete description of that algorithm, consult Rosen's text on Number Theory.

Computations and Explorations 5

> Find as many primes of the form $n^2 + 1$ where n is a positive integer as you can. It is not known whether there are infinitely many such primes.

Solution: We write a *Mathematica* function that, given a maximum n, tests the integers of the given form.

```
In[199]:= ce5[max_Integer] := Module[{n},
            Reap[
              For[n = 1, n ≤ max, n++,
               If[PrimeQ[n^2 + 1], Sow[n^2 + 1]]
              ]
             ][[2, 1]]
           ]
```

To save space, we'll only compute up to a maximum of $n = 100$.

```
In[200]:= ce5[100]
Out[200]= {2, 5, 17, 37, 101, 197, 257, 401, 577, 677,
           1297, 1601, 2917, 3137, 4357, 5477, 7057, 8101, 8837}
```

Exercises

1. Use *Mathematica* to generate the list of the first 100 prime numbers larger than one million.
2. Use *Mathematica* to find the one's complement of an arbitrary integer (see the prelude to Exercise 34 of Section 4.2).

3. For which odd prime moduli is -1 a square? That is, for which prime numbers p does there exist an integer x such that $x^2 \equiv -1 \pmod{p}$?

4. Use *Mathematica* to determine which numbers are perfect squares modulo n for various values of the modulus n. For each perfect square s, determine how many square roots s has. That is, for how many values of x is $x^2 \equiv s \pmod{n}$. What conjectures can you make about the number of different square roots an integer has modulo n? (The *Mathematica* functions `PowerMod` and `Solve` may be of use.)

5. Use *Mathematica* to find the base 2 expansion of the 4th Fermat number $F_4 = 2^{2^4} + 1$. Do the following for several large integers n. Compute the time required to calculate the remainder modulo n of various bases b raised to the power F_4 (that is, to calculate $b^{F_4} \pmod{n}$) using two different methods. First, do the calculation by a straightforward exponentiation. Second, do it using the binary expansion of F_4 with repeated squaring and multiplications. Why do you think F_4 is a good choice for the public exponent in the RSA encryption scheme?

6. Modify the function **`generateKeys`** that we developed to produce the keys for the RSA system to incorporate the techniques for generating random large primes. Make your procedure take as an argument a "security" parameter which measures the number of digits in the primes.

7. Write *Mathematica* functions to encode and decode English sentences into lists of integers, appropriate for encryption with RSA. You may ignore punctuation and insist that all letters are uppercase. Your functions should accept as input the block size.

8. There are infinitely many primes of the form $4n + 1$ and infinitely many of the form $4n + 3$. Use *Mathematica* to determine for various values of x whether there are more primes of the form $4n + 1$ less than x than there are of the form $4n + 3$. What conjectures can you make from this evidence?

9. Develop a function for determining whether Mersenne numbers are prime using the Lucas-Lehmer test as described in number theory books, such as *Elementary Number Theory and its Applications* by K. Rosen. How many Mersenne numbers can you test for primality using *Mathematica*?

10. *Repunits* are integers with decimal expansions consisting entirely of 1s (e.g., 11, 111, 1111, etc.). Use *Mathematica* to factor repunits. How many prime repunits can you find? Explore the same question for repunits in different base expansions.

11. Compute the sequence of pseudorandom numbers generated by the linear congruential generator $x_{n+1} = (a \cdot x_n + c) \pmod{m}$ for various values of the multiplier a, the increment c, and the modulus m. For which values do you get a period of length m for the sequence that you generate? Formulate a conjecture.

12. The *Mathematica* function `DivisorSigma` implements the function defined, for all positive integers n, by: $\sigma_k(n)$ is the sum of the kth powers of the positive divisors of n, i.e., $\sigma_k(n) = \sum_{d|n} d^k$. For $k = 0$, $\sigma_0(n)$ is the number of positive divisors of n, which is sometimes also denoted $\tau(n)$. Use *Mathematica* to study the function σ_0. What conjectures can you make about it? For example, when is $\sigma_0(n)$ odd? Is there a formula for $\sigma_0(n)$? For which integers m does $\sigma_0(n) = m$ have a solution for some integer n? Is there a formula for $\sigma_0(m \cdot n)$ in terms of $\sigma_0(m)$ and $\sigma_0(n)$? (Note: $\sigma_0(n)$ is computed by **`DivisorSigma[0,n]`**.)

13. A sequence $a_1, a_2, a_3, \ldots$ is called *periodic* if there are positive integers N and p for which $a_n = a_{n+p}$ for all $n \geq N$. The least integer p for which this is true is called the *period* of the

sequence. The sequence is said to be *periodic modulo m*, for a positive integer *m*, if the sequence $a_1 \pmod m$, $a_2 \pmod m$, $a_3 \pmod m$ … is periodic. Use *Mathematica* to determine whether the Fibonacci sequence is periodic modulo *m* for various integers *m* and, if so, find the period. Can you, by examining enough different values of *m*, make any conjectures concerning the relationship between *m* and the period? Do the same thing for other sequences that you find interesting.

14. (Class project) The Data Encryption Standard (DES) specifies a widely used algorithm for private key cryptography. Find a description of this algorithm (for example, in *Cryptography, Theory and Practice* by Douglas Stinson). Implement the DES in *Mathematica*.