4 Number Theory and Cryptography

Introduction

Maple includes numerous capabilities for exploring number theory. In this chapter, we will see how to use Maple'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 Maple 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 Maple 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 Maple can be used to compute addition and multiplication tables.

Quotient, Remainder, and Divisibility

Maple's commands **iquo** and **irem** compute the quotient and remainder, respectively, obtained when you divide two integers. For example, consider 99 divided by 13.

```
> iquo(99, 13)
```

$$7 \tag{4.1}$$

```
> irem(99, 13)
```

$$8 \tag{4.2}$$

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

These commands both accept a variable as an optional third argument. In this case, the **iquo** command will still return the quotient but will assign the remainder to the name, while the **irem** command will return the remainder and assign the value of the quotient to the name. Putting the name in right single quotes ensures that if the name already stores a value, it will be overwritten. Omitting the single quotes will result in an error if the name has already been assigned a value.

```
> iquo(99, 13, 'remainderName')
```

$$7 \tag{4.3}$$

```
> remainderName
```

$$8 \tag{4.4}$$

Checking Divisibility

To test whether one integer divides another, you check to see if the remainder is 0 or not. For example, the following shows that 3 | 132.

```
> irem(132, 3)
```

$$0 \qquad (4.5)$$

Since the question of whether one integer divides another is fundamental to our study of number theory, we create a procedure to test this for us. The **IsDivisor** procedure below accepts two integers as arguments. It returns true if the first argument divides the second and false otherwise. The procedure body is only one line—it computes the remainder and compares it to 0.

```
IsDivisor := proc(a::integer, b::integer)
   return evalb(irem(b,a) = 0);
end proc:
```

```
> IsDivisor(3, 132)
```

$$true \qquad (4.6)$$

```
> IsDivisor(13, 99)
```

$$false \qquad (4.7)$$

The mod Operator

The textbook uses the notation $a \bmod m$ to represent the remainder when a is divided by m. Maple's **mod** operator works in the same way.

```
> 99 mod 13
```

$$8 \qquad (4.8)$$

Recall from the division algorithm that the remainder must always be positive, even when the dividend is negative. Maple's **mod** function respects that convention by default.

```
> −27 mod 5
```

$$3 \qquad (4.9)$$

However, there may be 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 124 mod 24,

```
> 142 mod 24
```

$$22 \qquad (4.10)$$

we see that the time will be the same 142 hours from now as 22 hours from now. However, it is 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, that is, 9:00 AM. You can see that this congruence is somewhat more convenient.

The **mods** command (the "s" is for symmetric) returns the integer closest to 0 that is congruent to the value in its first argument modulo the second argument.

```
> mods(142, 24)
                    -2                                    (4.11)
```

The **modp** command returns the "positive" representation.

```
> modp(142, 24)
                    22                                    (4.12)
```

The default behavior of the **mod** operator is to apply the **modp** command to the operands. However, this can be overridden by executing the assignment **'mod' := mods;**, in which case **mod** will act as the symmetric modulus. Likewise, **'mod' := modp;** will revert to the default. Because it is possible to modify its behavior, **mod** is ambiguous and thus we will typically use the explicit **modp** command in procedures. This way, there is no possibility that a reassignment of **'mod'** could wreak havoc on our programs.

Congruences

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

```
> modp(3 + 4 * 9^2, 5)
                     2                                    (4.13)
```

The first argument can also be an equation. Recall from Theorem 3 in Section 4.1 that $a \equiv b \pmod{m}$ if and only if $a \bmod m = b \bmod m$. If you enter an equation as the first argument to **modp**, Maple will evaluate both sides of the equation modulo the value given in the second argument. For example, consider the congruence $428 \equiv 530 \pmod{17}$. In Maple, you would enter the following:

```
> modp(428 = 530, 17)
                    3 = 3                                 (4.14)
```

Note that the result is the equation $428 \bmod 17 = 530 \bmod 17$. By applying the **evalb** command, we obtain a truth value.

```
> evalb(modp(428 = 530, 17))
                   true                                   (4.15)
> evalb(modp(289 = 311, 17))
                   false                                  (4.16)
```

Solving Congruences

Maple can solve congruences with the **msolve** command. This command has two required arguments. The first is an equation or set of equations representing the congruences to be solved. The second argument is the modulus. As an example, consider Exercise 17a from Section 4.1 of the textbook. Under the assumption that $a \equiv 4 \pmod{13}$, we need to solve $c \equiv 9\ a \pmod{13}$. We can solve this with Maple as follows:

```
> msolve({a = 4, c = 9 a}, 13)
                  {a = 4, c = 10}                         (4.17)
```

If there are no solutions, then **msolve** will return NULL, so that no result is displayed.

```
> msolve(n^2 = 3, 4)
```

On the other hand, if the solution is indeterminate, then a family of solutions may be returned.

```
> msolve(3^i = 4, 7)
      {i = 4 + 6 _Z1}                                          (4.18)
```

The symbol *_Z1* indicates that any integer may be substituted to obtain a value for *i* that satisfies the congruence. For example, substituting 5 for *_Z1* and then substituting the result into $3^i = 4$ yields

```
> evalb(modp(3^(4+6·5) = 4, 7))
      true                                                     (4.19)
```

If you prefer a particular name instead of symbols such as *_Z1*, you can provide a name or set of names as an optional second argument to **msolve** as illustrated below.

```
> msolve(3^i = 4, C, 7)
      {i = 4 + 6 C}                                            (4.20)
```

Arithmetic Modulo m

In this section, we 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 Maple return 5.

The usual style of writing arithmetic operators in between the operands is referred to as infix notation. In Maple, we create operators that can be used in infix style by using neutral operators. The name of a neutral operator must begin with an ampersand (&) and be followed either by a valid Maple name composed of letters and numbers or by one or more allowable special characters, which include symbols such as + and *. The two forms cannot be mixed, which means that if the name of the neutral operator includes a special character, then it cannot include letters or numbers. We will use the names &+ and &* for our modular addition and multiplication operators.

You define a neutral operator in the same way as you normally define a procedure or functional operator, with some small differences. First, when defining the operator, the name must be enclosed in left single quotes. Second, the function or procedure should have only one or two arguments. When there is one argument, the operator will function as a unary operator, like negation. When two arguments are allowed, the operator will function like a binary operator such as addition.

We define addition and multiplication modulo 11 as follows. The neutral operator names are assigned to functional operators which accept two arguments, **a** and **b**, and apply **modp** to their sum or product, respectively.

```
> '&+' := (a, b) → modp(a + b, 11)
      &+ := (a, b) ↦ modp(a + b, 11)                           (4.21)

> '&*' := (a, b) → modp(a b, 11)
      &* := (a, b) ↦ modp(b a, 11)                             (4.22)
```

This allows us to perform arithmetic modulo 11 with infix notation. (Note that these operators will not respect the usual order of operations, so parentheses are needed.)

```
> 7 &+ (9 &* 2)
```

$$3 \tag{4.23}$$

Addition and Multiplication Tables

We conclude Section 4.1 by producing addition and multiplication tables.

We will represent the tables with matrices whose entries represent the sums or products. In each matrix, the first row and first column should correspond to the value 0, so that the $(1,\ 1)$ entry corresponds to $0 + 0 \pmod{m}$, the $(1,\ 2)$ entry to $0 + 1 \pmod{m}$, and so on. In general, the $(i,\ j)$ entry should correspond to $(i - 1) + (j - 1) \pmod{m}$.

Recall from Section 2.6 of this manual that you can define a matrix by specifying a size along with a procedure or function that accepts two arguments indicating the row and column position in the matrix and outputs the entry in that position. As an example with modulus 5,

```
> Matrix(5, (i,j) → modp((i − 1) + (j − 1), 5))
```

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 0 \\ 2 & 3 & 4 & 0 & 1 \\ 3 & 4 & 0 & 1 & 2 \\ 4 & 0 & 1 & 2 & 3 \end{bmatrix} \tag{4.24}$$

```
> Matrix(5, (i,j) → modp((i − 1)(j − 1), 5))
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 2 & 4 & 1 & 3 \\ 0 & 3 & 1 & 4 & 2 \\ 0 & 4 & 3 & 2 & 1 \end{bmatrix} \tag{4.25}$$

4.2 Integer Representations and Algorithms

In this section, we will see how Maple can be used to explore representations of integers in various bases and algorithms for computing with integers. We begin by looking at Maple's built-in commands for converting between bases. Then, we focus our attention on binary representations of integers and the **Bits** package. Finally, we see how to implement algorithms for addition and multiplication on binary representations. In this section, we restrict our attention to positive integers. Many of the commands discussed here can be applied to negative integers as well, but we will not discuss their behavior in that case.

Base Conversion

Maple provides support for converting from one base representation to another via the **convert** command. The **convert** command has two required arguments. The first argument is an expression that will be converted. The second required argument is a "form" that specifies what the expression is converted into. This command is extremely general and can be used to convert a variety of Maple objects into other objects. As a first example, the command below converts the list provided as the first argument into a set.

```
> convert([1, 2, 3], set)
```

$$\{1, 2, 3\} \quad \textbf{(4.26)}$$

There are over 100 different valid forms available in Maple, and users can create additional forms in much the same way as types are created.

Maple includes forms for **binary**, **octal**, and **hexadecimal** (abbreviated **hex**). Using the **convert** command with a positive integer as the first argument and one of these forms as the second argument will produce the expected output. Compare the following to Examples 4, 5, and 6 in the text:

```
> convert(12 345, octal)
```

$$30071 \quad \textbf{(4.27)}$$

```
> convert(177 130, hex)
```

$$\text{'2B3EA'} \quad \textbf{(4.28)}$$

```
> convert(241, binary)
```

$$1111\,0001 \quad \textbf{(4.29)}$$

To convert from a base other than 10 to base 10, you use the **decimal** form together with a third argument indicating the base being converted from.

```
> convert(30071, decimal, octal)
```

$$12\,345 \quad \textbf{(4.30)}$$

```
> convert(1111 0001, decimal, binary)
```

$$241 \quad \textbf{(4.31)}$$

For bases larger than 10 but not more than 36, the characters "A" through "Z" (lowercase is also allowed) represent digits with values 10 or larger. When letters are used as part of the representation, the entire representation must be enclosed in quotation marks.

```
> convert("2B3EA", decimal, hex)
```

$$177\,130 \quad \textbf{(4.32)}$$

The third argument can also be given as an integer representing the base, which is particularly useful for converting from bases other than binary, octal, and hexadecimal.

```
> convert(1111 0001, decimal, 2)
```

$$241 \quad \textbf{(4.33)}$$

```
> convert(12, decimal, 3)
```

$$5 \quad \textbf{(4.34)}$$

The last example showed how to use **convert** to go from a base 3 representation to a decimal representation. With the exceptions of binary, octal, and hexadecimal, which were described above, going in the other direction and converting from a decimal representation to an arbitrary base requires a bit more explanation.

Consider converting the decimal number 194 to base 5. Working by hand, you can use Algorithm 1 and determine that $194_{10} = 1234_5 \wedge 1234_5 = 194$. To use Maple to obtain this representation, we use the **convert** command with the **base** form and a third argument representing the desired base.

```
> convert(194, base, 5)
```

$$[4, 3, 2, 1] \qquad \textbf{(4.35)}$$

Note that this form does not return 1234. Instead, it returns a list of the digits in "reverse" order, that is, with the least significant digit (that is, the "one's" place) first.

This is the case even for the common bases, such as **octal**.

```
> convert(12 345, base, 8)
```

$$[1, 7, 0, 0, 3] \qquad \textbf{(4.36)}$$

By converting a decimal number to base 10, you can obtain the list of the digits.

```
> convert(12 345, base, 10)
```

$$[5, 4, 3, 2, 1] \qquad \textbf{(4.37)}$$

The **base** form can be used to convert between any two bases. To do this, the first argument must be the list of digits in the order with least significant first. The second argument is the **base** keyword. The third argument is the original base, and the fourth argument is the target base. For example, to find the base three expansion of $(123)_5$ you would enter the following command:

```
> convert([3, 2, 1], base, 5, 3)
```

$$[2, 0, 1, 1] \qquad \textbf{(4.38)}$$

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

Binary and the Bits Package

We will now focus on binary representations. The **Bits** package provides a selection of commands that are especially suited to working with binary representations. In Chapter 2 of this manual, we used the **Bits** package to compute bitwise **and** and **or**. Here, we will make more extensive use of this package.

Split and Join

In the **Bits** package, the **Split** command is used to turn an integer into its binary representation as a list of 0s and 1s. Note that once again the digits are listed in reverse order, that is, with the least significant digit first.

```
> with(Bits) :

> Split(241)
```

$$[1, 0, 0, 0, 1, 1, 1, 1] \qquad \textbf{(4.39)}$$

The output above indicates that $241 = (1111\,0001)_2$.

Split can accept the option **bits=n**, where **n** is a positive integer. This specifies the number of bits to include in the result. If **n** is smaller than the number of bits needed to represent the integer, the bits in higher position are ignored. If **n** is larger than the minimum required number of bits, then 0s are added.

> *Split*(241, *bits* = 20)
[1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] **(4.40)**

The reverse operation is **Join**, which accepts a list of digits in reverse order and returns the base ten representation of the integer.

> *Join*([1, 0, 0, 0, 1, 1, 1, 1])
241 **(4.41)**

The GetBits Command

Related to **Split** is the more flexible **GetBits** command. This command has two required arguments. The first is an integer. The second is a location, or a range, or a sequence of locations or ranges that you want to extract. Note that for **GetBits**, nonnegative locations correspond to the exponent on 2 in the binary expansion. That is, the least significant digit is considered to have location 0, the coefficient of 2^1 has location 1, etc. Note that this is different from the digit's position in the list produced by **Split**. Negative locations can be used, with -1 referring to the most significant digit. Note that **GetBits** returns a sequence, so it must be enclosed in brackets if you need a list.

> *GetBits*(241, 0 .. − 1)
1, 0, 0, 0, 1, 1, 1, 1 **(4.42)**

> *GetBits*(241, 0 ..3)
1, 0, 0, 0 **(4.43)**

> *GetBits*(241, 0, 4 .. − 1)
1, 1, 1, 1, 1 **(4.44)**

Note in the last example, the command requested the location **0** followed by the range **4..-1**. This selected all of the 1 digits from the binary representation.

The example below uses the range **-1..0** to output the digits in the usual order.

> *GetBits*(241, −1 ..0)
1, 1, 1, 1, 0, 0, 0, 1 **(4.45)**

You can also pass the equation **output=number** as an optional argument. This causes **GetBits** to output the result as a decimal number.

> *GetBits*(241, 1 ..4)
0, 0, 0, 1 **(4.46)**

> *GetBits*(241, 1 ..4, *output* = *number*)
8 **(4.47)**

Bitwise Operations

Finally, **Bits** contains several commands for performing bitwise logical operations: **Not**, **And**, **Or**, **Xor**, **Nand**, **Nor**, **Iff**, and **Implies**. Each of these can be applied to two integers (excepting **Not**) and returns the integer obtained as the result of applying the bitwise operation to the binary representations of the integers.

For example, consider $43 = (10\,1011)_2$ and $44 = (10\,1100)_2$.

```
> Split(43)
```

$$[1, 1, 0, 1, 0, 1] \tag{4.48}$$

```
> Split(44)
```

$$[0, 0, 1, 1, 0, 1] \tag{4.49}$$

Applying **and** to each pair of corresponding digits produces 0,0,0,1,0,1.

```
> zip(And, Split(43), Split(44))
```

$$[0, 0, 0, 1, 0, 1] \tag{4.50}$$

This corresponds to the integer $(10\,1000)_2 = 40$.

```
> Join([0, 0, 0, 1, 0, 1])
```

$$40 \tag{4.51}$$

This is the same result as you get from applying **And** to 43 and 44.

```
> And(43, 44)
```

$$40 \tag{4.52}$$

Binary Addition

We now implement Algorithm 2 from Section 4.2, addition of integers. Our procedure will accept two binary representations (lists of 0s and 1s with the least significant digit first). The first task for our procedure 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 (stored as n) and then add as many 0s to the list as are necessary to make both lists that length.

Next, we initialize a sum list S to a list of all 0s. The sum S must have size $n + 1$ to allow for a carry to surpass the lengths of the two input values. Once these initial tasks are completed, we follow Algorithm 2.

```
Addition := proc(a::list({0,1}), b::list({0,1}))
   local n, A, B, S, c, j, d;
   n := max(nops(a),nops(b));
   A := [op(a), 0$(n-nops(a))];
   B := [op(b), 0$(n-nops(b))];
   S := [0$n+1];
   c := 0;
   for j from 1 to n do
      d := floor((A[j]+B[j]+c)/2);
```

```
        S[j] := A[j] + B[j] + c - 2*d;
        c := d;
    end do;
    S[n+1] := c;
    return S;
end proc:
```

Adding $10 = (1010)_2$ and $58 = (11\ 1010)_2$ with our procedure produces:

```
> Addition([0, 1, 0, 1], [0, 1, 0, 1, 1, 1])
```

$$[0, 0, 1, 0, 0, 0, 1] \quad \textbf{(4.53)}$$

```
> Join(%)
```

$$68 \quad \textbf{(4.54)}$$

Binary Multiplication

Finally, we implement a multiplication algorithm, presented as Algorithm 3 in Section 4.2. Once again, our procedure 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 follows. To shift the list **[1,1,1,1]** by 5 places, we must prepend 5 zeros on front of the list. (Remember, our binary representations have least significant digit first, which is why 0s are added to the front of the list instead of the back.) We do this by creating a new list that begins with 5 zeros and is followed by the elements of the original list.

```
> shiftExample := [1, 1, 1, 1]
```

$$shiftExample := [1, 1, 1, 1] \quad \textbf{(4.55)}$$

```
> [0 $ 5, op(shiftExample)]
```

$$[0, 0, 0, 0, 0, 1, 1, 1, 1] \quad \textbf{(4.56)}$$

Note the use of the **$** operator to form the sequence of 0 repeated five times.

We will store the partial products as a table of lists. Recall from Section 2.3 of this manual that we can create a table by assigning **table()** to a name. We can then use the selection operation (brackets) to both assign entries to indices and to retrieve entries.

The product p will be initialized to [0], a binary representation of 0. The addition in the final for loop will be performed by the **Addition** procedure we created above.

Here is our implementation of Algorithm 3.

```
Multiplication := proc(a::list({0,1}), b::list({0,1}))
    local j, C, p;
    C := table();
    for j from 1 to nops(b) do
        if b[j] = 1 then
            C[j] := [ 0 $ (j-1), op(a) ];
```

```
        else
          C[j] := [0];
        end if;
      end do;
      p := [0];
      for j from 1 to nops(b) do
        p := Addition(p,C[j]);
      end do;
      return p;
    end proc:
```

We test our procedure using Example 10 from Section 4.2.

> *Multiplication*([0, 1, 1], [1, 0, 1])

$$[0, 1, 1, 1, 1, 0] \tag{4.57}$$

4.3 Primes and Greatest Common Divisors

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

Primes

We will first introduce some of Maple's commands for testing whether a number is prime and for finding primes.

Testing for Primality

The **isprime** command accepts a single argument, an integer to be tested, and returns true or false.

> *isprime*(5)

$$true \tag{4.58}$$

> *isprime*(10)

$$false \tag{4.59}$$

> *isprime*$\left(2^{13}-1\right)$

$$true \tag{4.60}$$

Unlike the trial division algorithm discussed in the book, which checks all possible divisors to see if a number is prime or composite, **isprime** 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. As the help page asserts, there is no known example of an integer for which **isprime** is incorrect and any such example must be exceptionally large. Therefore, **isprime** is reliable.

Listing Primes

The command **ithprime** accepts as input a positive integer i and computes the ith prime number.

> *ithprime*(1)

$$2 \tag{4.61}$$

```
> ithprime(2)
                                  3                                  (4.62)

> seq(ithprime(i), i = 1..20)
        2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71   (4.63)

> ithprime(100000)
                                1299709                              (4.64)
```

For small prime numbers, **ithprime** simply looks up the result in an internal table, while for larger arguments, it operates recursively.

Maple also provides the commands **nextprime** and **prevprime**. Both commands accept an integer as their single argument. The **nextprime** command returns the smallest prime larger than the input value, and **prevprime** returns the largest prime smaller than the input. For example, to find the first prime number larger than 1000, we enter the following:

```
> nextprime(1000)
                                  1009                               (4.65)
```

Similarly, the prime number before that one is

```
> prevprime(%)
                                   997                               (4.66)
```

Both **nextprime** and **prevprime** are based on the **isprime** command.

Inspecting Procedure Definitions

We can see that **nextprime** relies on **isprime** by looking at its definition. To do this, we set the **interface** variable **verboseproc** equal to 2 and then call **eval** on the procedure name.

```
> interface(verboseproc = 2):

> eval(nextprime)
    proc(n)
      option Copyright (c) 1990 by the University of Waterloo. All rights reserved.;
      local i;
      if type(n, 'integer') then
        if n < 2 then
          2
        else
          i := n + if(n :: even, 1, 2);
          while not isprime(i) do i := i + if(irem(i, 6) = 1, 4, 2) end do;
          i
        end if
      elif type(n, 'numeric') then
        error "argument must be an integer, but received %1", n
      else
        'procname(args)'
      end if
    end proc                                                         (4.67)
```

The **verboseproc** variable controls how much detail is shown about procedures. A value of 2 forces printing the full body for all procedures. The default value of 1 prints the body for user-defined procedures only.

Prime Factorization

To compute the prime factorization of an integer, we can use the Maple command **ifactor**.

```
> ifactor(100)
```

$$(2)^2(5)^2 \qquad \textbf{(4.68)}$$

```
> ifactor(123 456 789)
```

$$(3)^2(3803)(3607) \qquad \textbf{(4.69)}$$

```
> ifactor(-987 654 321)
```

$$-(3)^2(17)^2(379\,721) \qquad \textbf{(4.70)}$$

The letter "i" in **ifactor** refers to integer factorization. The **factor** command is used for factoring polynomials.

The **expand** command can be used to reverse the process.

```
> expand((4.70))
```

$$-987\,654\,321 \qquad \textbf{(4.71)}$$

Note that **ifactor** can accept optional arguments that allow you to specify the method Maple uses to factor the integer. A discussion of these methods is beyond the scope of this manual. However, one method, the "**easy**" method is worth mentioning. The example below illustrates the effect of the **easy** method.

```
> ifactor(236 914 830 635 411 777 378 758 175 934 586 404 476 822)
```

$$(2)(197)(509)(32\,129\,861)^2(10\,459\,723)^3 \qquad \textbf{(4.72)}$$

```
> ifactor(236 914 830 635 411 777 378 758 175 934 586 404 476 822, easy)
```

$$(2)(197)(509)\ \_c37\_1 \qquad \textbf{(4.73)}$$

Without the "**easy**" method specified, **ifactor** factors the given integer into its complete prime factorization. With the keyword **easy**, only the "easy" factors are produced, with the symbol *_c37_1* indicating that the remaining factor has 37 digits. This gives you a way to have Maple only perform the quick parts of factorizations, which can help ensure that your procedures run quickly during development. Then, when you are ready to let the procedure take all the time it needs, you can remove the **easy** keyword.

The **ifactors** command is different from the **ifactor** command. Instead of producing an algebraic expression of the form $sp_1^{e_1}p_2^{e_2}\cdots p_n^{e_n}$, **ifactors** produces a list of the form $[s, [[p_1, e_1], [p_2, e_2], .., [p_n, e_m]]]$, where the p_i are the prime factors, the e_i their multiplicities, and s is the sign of the integer, represented as a positive or negative 1. Compare the output below to the results from **ifactor** at the start of this section.

```
> ifactors(100)
```
$$[1, [[2, 2], [5, 2]]] \quad \textbf{(4.74)}$$

```
> ifactors(123 456 789)
```
$$[1, [[3, 2], [3607, 1], [3803, 1]]] \quad \textbf{(4.75)}$$

```
> ifactors(−987 654 321)
```
$$[-1, [[3, 2], [17, 2], [379\,721, 1]]] \quad \textbf{(4.76)}$$

The format returned by **ifactors** can be easier to use in programs.

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 section, we will use Maple's graphing capabilities to graph the number of primes not exceeding x.

Recall from Section 3.3 of this manual that we can graph specific points by using the **plot** command applied to two lists where the first list is the list of x-values and the second list is the list of y-values. We will consider the integers from 1 to 1000, so our first list is obtained as follows (the output has been suppressed).

```
> xList := [$1 ..1000] :
```

To find the number of primes not exceeding x, we use the command **pi** found in the **NumberTheory** package. The function $\pi(x)$ is the standard notation for the number of primes less than or equal to x. To calculate the number of primes less than or equal to 1000, for example, we enter the following:

```
> NumberTheory[pi](1000)
```
$$168 \quad \textbf{(4.77)}$$

Note that the number π, the ratio of the circumference of a circle to its diameter, is denoted **Pi** in Maple.

We obtain the list of values of $\pi(x)$ associated to the values of **xList** by

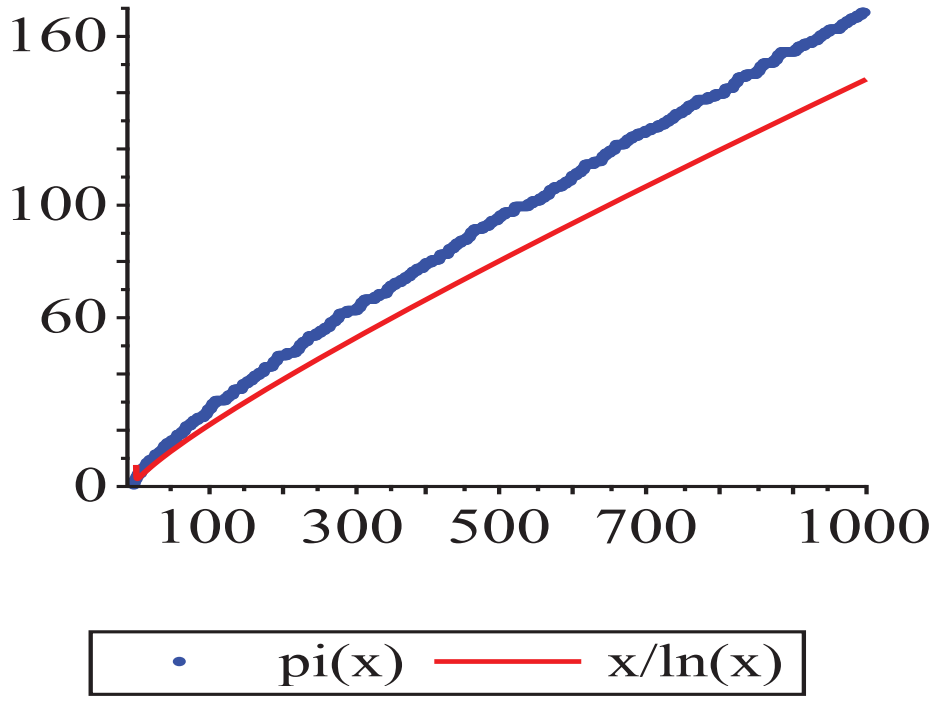
```
> piList := [seq(NumberTheory[pi](x), x = 1 ..1000)] :
```

We are going to graphically compare the values of $\pi(x)$ to the function $\frac{x}{\ln(x)}$. We will define two plot objects and then combine them with the **display** command. Refer to Chapter 3 of this manual, particularly Section 3.3 and the solution to Computer Project 10, for detailed information about the commands used here.

```
> piPlot := plot(xList, piList, color = blue, view = [1 ..1000, 0 ..170],
    style = point, symbol = solidcircle, symbolsize = 7, legend = "pi(x)") :
```

```
> xlnxPlot := plot(x/ln(x), x = 1..1000, color = red, view = [1..1000, 0..25],
      legend = "x/ln(x)"):

> plots[display](piPlot, xlnxPlot)
```



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 Divisors and Least Common Multiples

Maple provides commands **igcd** and **ilcm** 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 **igcd** command to them.

```
> igcd(6, 9)
      3                                                    (4.78)
```

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 the integers. For example, 3 divides 6, 9, and 12, so

```
> igcd(6, 9, 12)
      3                                                    (4.79)
```

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

```
> ilcm(6, 9)
```

$$18 \tag{4.80}$$

```
> ilcm(12, 18, 33)
```

$$396 \tag{4.81}$$

Note that Maple also includes commands called **gcd** and **lcm**. These commands are more general than **igcd** and **ilcm**, and can find the greatest common divisor and least common multiple of polynomials as well as integers. The **igcd** and **ilcm** commands, however, are optimized for integer inputs and should be the commands you use when working with integers.

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.

```
> igcd(10, 21)
```

$$1 \tag{4.82}$$

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

The following procedure accepts two integers as input and returns true if they are relatively prime and false otherwise.

```
AreRelPrime := proc(a::integer, b::integer)
   if igcd(a,b) = 1 then
      return true;
   else
      return false;
   end if;
end proc:
```

```
> AreRelPrime(3, 6)
```

$$\mathit{false} \tag{4.83}$$

```
> AreRelPrime(22, 15)
```

$$\mathit{true} \tag{4.84}$$

Pairwise Relatively Prime

Recall that a list of integers $a_1, a_2, .., a_n$ is 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.

Note that the **igcd** command can be applied to more than two integers, but only determines the largest divisor common to all of the values. For example, 4, 6, and 9 are not pairwise relatively prime, but 1 is their greatest common divisor.

```
> igcd(4, 6, 9)
```

$$1 \tag{4.85}$$

We can write a procedure to test whether or not a list of integers is pairwise relatively prime. We use two **for** loops with variables i and j. The first loop, the i loop, runs from 1 to $n-1$ (i is not allowed to be equal to n because the inner loop variable will always be strictly greater than i). The inner loop has j run from $i+1$ to n so that j is always greater than i, so as to avoid testing each pair twice. Within the two loops, we test the greatest common divisor of the entries in the input list with indices i and j. If the gcd is ever not 1, the procedure immediately returns false. If all the pairs pass the test, then after both loops are terminated, true is returned.

```
ArePairwisePrime := proc(A::list(integer))
   local n, i, j;
   n := nops(A);
   for i from 1 to n-1 do
      for j from i+1 to n do
         if gcd(A[i],A[j]) <> 1 then
            return false;
         end if;
      end do;
   end do;
   return true;
end proc:
```

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

> *ArePairwisePrime*([14, 39, 55])
>
> *true* **(4.86)**

However, 42, 165, and 182 are not pairwise relatively prime, although their common gcd is 1.

> *igcd*(42, 165, 182)
>
> 1 **(4.87)**

> *ArePairwisePrime*([42, 165, 182])
>
> *false* **(4.88)**

The Extended Euclidean Algorithm

While **igcd** 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 $sa + tb$ 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. Following the statement of the theorem, the extended Euclidean algorithm is described, which produces not only the greatest common divisor but also the integers s and t.

In Maple, the command **igcdex** is an implementation of the extended Euclidean algorithm for integers. This command accepts two integers and two optional names. When executed, Maple returns the greatest common divisor of the two integers. If the optional names have been included, then the Bézout coefficients are stored in them. As an example, consider 252 and 198, the values used in Example 17.

```
> igcdex(252, 198, 's', 't')
                    18                                    (4.89)

> s; t
                     4
                    -5                                    (4.90)
```

The results above indicate that $gcd(252, 198) = 18 = 4 \cdot 252 - 5 \cdot 198$. We enclose the names **s** and **t** in single right quotes to ensure that we pass their names rather than any previously assigned values. If the quotes were not present and one of the names had previously been assigned a value, an error would result.

4.4 Solving Congruences

In this section, we will see how Maple 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 Maple 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, we saw that the **igcdex** command can be used to obtain the Bézout coefficients.

Finding Inverses with igcdex

For example, to find the inverse of 264 modulo 3185, we need to find s so that $264\,s + 3\,185\,t = 1$ (provided that 264 and 3185 are relatively prime).

```
> igcdex(264, 3185, 's', 't')
                     1                                    (4.91)
```

Since the statement returned 1, we know that 264 and 3185 are relatively prime.

igcdex assigns the coefficient of the first integer to the first name and the second number to the second name.

```
> s
                    374                                   (4.92)

> t
                   -31                                    (4.93)
```

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

```
> 374 · 264 mod 3185
                     1                                    (4.94)
```

Finding Inverses with ^(-1)

Maple actually 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 $3^{-1} = 1/3$. Maple interprets an exponent of -1, within the context of modular arithmetic, as the modular inverse. For example, we can obtain the inverse of 264 modulo 3185 as follows.

```
> 264^(-1) mod 3185;
```

$$374 \qquad \textbf{(4.95)}$$

In 2-D input mode, as above, the exponent of -1 is obtained by typing a caret followed by -1, and then using the right-arrow key to exit the exponent. In 1-D input mode, parentheses around -1 are required. Also note that if the integer and the modulus are not relatively prime, no inverse exists and an error is generated.

```
> 4^(-1) mod 10;
```

Error, the modular inverse does not exist

Solving Congruences

We saw in Section 4.1 of this manual the **msolve** command for solving congruences. We can use this command to solve linear congruences of the form $4\ x \equiv 3 \pmod{11}$ as follows.

```
> msolve(4*x = 3, 11);
```

$$\{x = 9\} \qquad \textbf{(4.96)}$$

The first argument to **msolve** is the congruence expressed with an equals sign and the second is the modulus. Maple returns a set whose elements express the solution to the congruence. If there is no solution, Maple returns nothing (technically, the command returns **NULL**, which results in no output being displayed).

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

```
> msolve(4*x = 1, 10);
```

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

```
> msolve(3*x = 9, 12);
```

$$\{x = 3\}, \{x = 7\}, \{x = 11\} \qquad \textbf{(4.97)}$$

The reader should refer back to Section 4.1 of this manual for information about solving systems of congruences with the same modulus.

The Chinese Remainder Theorem

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

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

The first approach, used in Example 5 of Section 4.4, is based on the proof of the Chinese remainder theorem. The second approach is the technique of back substitution described in Example 6. Maple's command **chrem** is an efficient implementation of back substitution.

The **chrem** command accepts two lists as its arguments. The first argument is the list $[a_1, a_2, \ldots, a_n]$ and the second is the list of moduli $[m_1, m_2, \ldots, m_n]$. The result is the smallest positive integer that satisfies all of the congruences. As an example, we solve the congruences

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

```
> chrem([2, 4, 6, 10], [3, 5, 7, 11])
                          1154                                    (4.98)
```

Creating Our Own Procedure

In the remainder of this section we will provide an implementation of the method 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 Maple's **chrem** command, implementing the algorithm can help you to better understand the proof of the theorem.

Our procedure, which we call **CRTheorem**, will accept as input 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 **ArePairwisePrime** from Section 4.3 of this manual to check that the moduli are pairwise relatively prime.

Once the preliminary tests are complete, the procedure sets **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 procedure and the text.) Recall that the Maple command **mul** computes the product of the values obtained by evaluating the first argument at each element in the range given in the second argument.

The procedure then needs to compute the M_k and y_k. To do this, we create empty tables **M** and **y** to store the values. Once the empty tables are initialized, the procedure enters a for loop in which the

values for **M** and **y** are calculated. The values for **M** are calculated by the formula $M_k = \dfrac{P}{m_k}$. For **y**, we use the fact that the y_k are the inverses of M_k modulo m_k, that is, $y_k = M_k^{-1} \pmod{m_k}$.

Finally, we compute the result $x = a_1M_1y_1 + a_2M_2y_2 + \cdots + a_nM_ny_n$ using the **add** command and return the result modulo **P**. Here is the procedure.

```
CRTheorem := proc(a::list(integer),m::list(posint))
   local P, M, y, i, x;
   if not nops(a) = nops(m) then
      error "Lists must be the same length.";
   end if;
   if not ArePairwisePrime(m) then
      error "Moduli must be pairwise relatively prime.";
   end if;
   P := mul(m[i],i=1..nops(m));
   M := table();
   y := table();
   for i from 1 to nops(m) do
      M[i] := P/m[i];
      y[i] := M[i]^(-1) mod m[i];
   end do;
   x := add(a[i]*M[i]*y[i],i=1..nops(m));
   return x mod P;
end proc:
```

Note that our procedure produces the same result as **chrem** did above.

```
> CRTheorem([2,4,6,10],[3,5,7,11])
                    1154                                    (4.99)
```

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 procedure to find pseudoprimes. Our procedure 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 first test the congruence. If the congruence holds, then we use **isprime** to check whether the number is prime or composite. If it is composite, then it is added to the list of pseudoprimes.

```
FindPseudoprimes := proc(b::posint, max::posint)
   local PList, n;
   PList := [];
   for n from 3 to max by 2 do
      if (b &^ (n-1) mod n) = 1 then
         if not isprime(n) then
            PList := [op(PList),n];
```

```
            end if;
        end if;
    end do;
    return PList;
end proc:
```

Note that we used **&^** in the calculation of the congruence instead of **^**. The syntax **&^** is the "inert" version of the exponent operator. When we use **^**, the integer exponentiation is performed first, which can result in an extremely large number. With the inert **&^** operator, Maple performs the 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.

> *FindPseudoprimes*(2, 100 000)

[341, 561, 645, 1105, 1387, 1729, 1905, 2047, 2465, 2701, 2821
3277, 4033, 4369, 4371, 4681, 5461, 6601, 7957, 8321, 8481,
8911, 10 261, 10585, 11 305, 12801, 13 741, 13747, 13 981, 14491,
15 709, 15841, 16 705, 18705, 18 721, 19951, 23 001, 23377, 25 761,
29 341, 30121, 30 889, 31417, 31 609, 31621, 33 153, 34945, 35 333,
39 865, 41041, 41 665, 42799, 46 657, 49141, 49 981, 52633, 55 245,
57 421, 60701, 60 787, 62745, 63 973, 65077, 65 281, 68101, 72 885,
74 665, 75361, 80 581, 83333, 83 665, 85489, 87 249, 88357, 88 561,
90 751, 91001, 93 961] **(4.100)**

Primitive Roots and Discrete Logarithms

Maple includes several commands for computing primitive roots and discrete logarithms. The commands we will discuss here are found in the **NumberTheory** package.

> *with*(*NumberTheory*) :

Primitive Roots

Maple provides a command, **PrimitiveRoot**, that computes primitive roots. The basic form of this command accepts one argument, the modulus, and returns the smallest primitive root. For example, the smallest primitive root of 11 is 2.

> *PrimitiveRoot*(11)

2 **(4.101)**

The **PrimitiveRoot** can also accept an option. To find the smallest primitive root of the modulus that is greater than a value, use the option **greaterthan**. For example, the next smallest primitive roots of 11 are

> *PrimitiveRoot*(11, *greaterthan* = 2)

6 **(4.102)**

> *PrimitiveRoot*(11, *greaterthan* = 6)

7 **(4.103)**

> *PrimitiveRoot*(11, *greaterthan* = 7)

8 **(4.104)**

> *PrimitiveRoot*(11, *greaterthan* = 8)

Error, (in NumberTheory:-PrimitiveRoot) there does not exist a primitive root modulo 11 greater than 8

Observe that the command returns an error when there are no larger primitive roots. This makes it fairly easy to write a procedure that returns a list of all the primitive roots of a number. We initialize an empty list and set a variable **x** to 0. We then create a **do** loop that finds the next primitive root greater than **x**. The variable **x** is updated to this new value and the value is added to the list of primitive roots. It is usually a bad idea to create a loop without any control like **while** or **if**, since this is structurally an infinite loop. However, we know that the **PrimitiveRoot** command will eventually raise an error. We can prevent the error raised by **PrimitiveRoot** from causing our procedure to crash by enclosing the error-prone code in a try–catch structure. The basic syntax is to follow the keyword **try** with a sequence of statements, which are executed. Those statements are followed by **catch:**, noting that the colon is required. If an error occurs during execution of the statements between the **try** and the **catch:**, then the statements following the **catch:** are executed. If there is no error, then those statements are skipped. Here, our response to an error is to return the list of primitive roots that we found.

```
AllPrimRoots := proc(n::posint)
   local L, x;
   uses NumberTheory;
   L := [];
   x := 0;
   try
      do
         x := PrimitiveRoot(n,greaterthan=x);
         L := [op(L),x];
      end do;
   catch:
      return L;
   end try;
end proc:
```

> *AllPrimRoots*(11)

[2, 6, 7, 8] **(4.105)**

Generally speaking, it is better to write computer code that is robust, that is to say, code that "handles" errors to produce meaningful output rather than just crashing. The try–catch structure is a commonly used tool for creating robust programs, and there are other elements of the syntax allowing for more detailed error handling, including different catch statements for different kinds of errors. However, it should be stated that creating code that depends on an error being raised is not ideal, compared to code that first checks values to be sure they will not produce an error. In this case, Euler's totient function could be used to determine the number of primitive roots and control the loop. The interested reader is encouraged to research Euler's totient function.

Note that Maple's **PrimitiveRoot** command applies to nonprime moduli as well. Maple is using 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.

Discrete Logarithms

Maple provides the command **ModularLog** for computing discrete logarithms. The **ModularLog** command requires three arguments: the value a, the base b, and the modulus n. It solves the congruence $b^y \equiv a \pmod{n}$. Thus, to find the discrete logarithm of 3 modulo 11 to the base 2, that is, to solve the congruence $2^y \equiv 3 \pmod{11}$, you would enter the following:

```
> ModularLog(3, 2, 11)
                                  8                                  (4.106)
```

The **mlog** command can also accept two options. One of the options can be used to specify a solution method. The possible methods are beyond the scope of this manual and will not be discussed.

The other option is to specify the output. The default output is the smallest nonnegative solution. If you provide the option **output=[result,char]**, then the output will be a pair of values indicating that all possible values of y solving the congruence are congruent to the **result** modulo the **char** (short for characteristic). For example,

```
> ModularLog(3, 2, 11, output = [result, char])
                                 8, 10                               (4.107)
```

indicates that 8 is the smallest nonnegative solution to $2^y \equiv 3 \pmod{11}$, but that any value of y such that $y \equiv 8 \pmod{10}$ is also a solution.

In other words, any element of the set solves the congruence. We see below the values of y that result from assigning k to the integers between -10 and 10.

```
> ySet := {seq(8 + 10 k, k = -10 ..10)}
    ySet := {-92, -82, -72, -62, -52, -42, -32, -22, -12, -2,
       8, 18, 28, 38, 48, 58, 68, 78, 88, 98, 108}                    (4.108)
```

Computing $2^y \bmod 11$ for these values confirms that each is a solution to $2^y \equiv 3 \pmod{11}$.

```
> seq(2^y mod 11, y in ySet)
   3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3      (4.109)
```

Exploring the Structure of Primitive Roots

Let p be a prime. Recall from the text that an integer r is a primitive root modulo p if every integer between 1 and $p - 1$, inclusive, can be obtained as a power of r modulo p.

Example 12 in Section 4.4 of the text shows that 2 is a primitive root modulo 11 by computing powers of 2 up to 2^{10} and seeing that these generate all the integers from 1 through 10. On the other hand, 3 is not a primitive root modulo 11 because the powers of 3 produce only 3, 9, 5, 4, and 1.

To help better understand primitive roots, we will write a procedure that displays, for each positive integer r less than p, all the different powers of that integer. Note that if some power of r is congruent to 1, then the next higher power will be congruent to r, and thus any higher powers of r will be redundant. This means that as we generate the powers of r, obtaining 1 indicates that we can stop. It is left to the reader to verify the converse: if two different powers are congruent, then there is a power congruent to 1. More precisely, if p is prime, $j < i$, and $r^i \equiv r^j \pmod{p}$, then there is a k such that $r^k \equiv 1 \pmod{p}$.

Our procedure, **DisplayPowers**, takes as input a prime number (note that **prime** is a Maple type). Using a for loop, it steps through each positive integer r less than the prime. Within the for loop, a while loop calculates successive powers of r and adds them to a list until 1 is obtained. Then, the value of r and the list of powers is printed before moving on to the next value of r.

```
DisplayPowers := proc(p::prime)
   local r, x, L;
   for r from 1 to p-1 do
      L := [r];
      x := r;
      while x <> 1 do
         x := x * r mod p;
         L := [op(L),x];
      end do;
      print(r,L);
   end do;
end proc:
```

```
> DisplayPowers(11)
   1, [1]
   2, [2, 4, 8, 5, 10, 9, 7, 3, 6, 1]
   3, [3, 9, 5, 4, 1]
   4, [4, 5, 9, 3, 1]
   5, [5, 3, 4, 9, 1]
   6, [6, 3, 7, 9, 10, 5, 8, 4, 2, 1]
   7, [7, 5, 2, 3, 10, 4, 6, 9, 8, 1]
   8, [8, 9, 6, 4, 10, 3, 2, 5, 7, 1]
   9, [9, 4, 3, 5, 1]
   10, [10, 1]                                                    (4.110)
```

From the above, you can see that 2, 6, 7, and 8 are all primitive roots of 11.

4.5 Applications of Congruences

In this section, we will see how Maple 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 procedure 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 table with indices "ID" and "GPA". Here are three example students.

```
> student1 := table(["ID" = 7319, "GPA" = 3.21])
```

$$student1 := table(["GPA" = 3.21, "ID" = 7319]) \quad \textbf{(4.111)}$$

```
> student2 := table(["ID" = 2908, "GPA" = 2.89])
```

$$student2 := table(["GPA" = 2.89, "ID" = 2908]) \quad \textbf{(4.112)}$$

```
> student3 := table(["ID" = 6578, "GPA" = 3.42])
```

$$student3 := table(["GPA" = 3.42, "ID" = 6578]) \quad \textbf{(4.113)}$$

Recall that the information in a table can be accessed by enclosing the index in brackets. For instance, to obtain the GPA of **student1**, we issue the following command.

```
> student1["GPA"]
```

$$3.21 \quad \textbf{(4.114)}$$

Our student records are going to be stored in an **Array**. In Maple, a list is an immutable data structure, which means that when you alter the information stored in it (e.g., assign a new value to a position), a new list is created that is a modified copy of the old list. This makes lists inefficient, particularly with regard to memory usage. Unlike a list, a Maple **Array** is a mutable data structure, which means that there is only one in memory, regardless of the number of changes you make.

Because the school is small, it will suffice to allocate space for 57 records in the school's database and so we create an **Array** with 57 entries all initialized to 0. There are a variety of ways to construct arrays, but the simplest is to apply **Array** to a range. The default behavior is to fill the entries in the array with 0s.

```
> studentRecords := Array(1 ..57)
```

$$studentRecords := \begin{bmatrix} 1\ ..57\ Array \\ Data\ Type:\ anything \\ Storage:\ rectangular \\ Order:\ Fortran\_order \end{bmatrix} \quad \textbf{(4.115)}$$

Very small arrays will be displayed as a list or matrix. For arrays that would consume more space, a summary is displayed. In the Context Panel, you can select "Browse" to open a window showing all of the entries.

In order to store a student record in the array (which represents the school's database), we need to apply a hashing function to the unique student ID. The hashing function we will 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** array run from 1 to 57 while the values of k mod 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.

```
> calculateHash := id::integer → modp(id, 57) + 1:
```

For example,

```
> calculateHash(student1["ID"])
                              24                                    (4.116)
```

indicates that **student1**'s record should be stored in location 24. The notation for assigning a value to a position in an array is the same as for a list.

```
> studentRecords[24] := student1
                  studentRecords_24 := student1                     (4.117)
```

Note that accessing location 24 returns the table **student1**.

```
> studentRecords[24]
                              student1                              (4.118)
```

To access the ID and GPA stored in location 24, we use a second pair of brackets with the indices "ID" or "GPA".

```
> studentRecords[24]["ID"]
                              7319                                  (4.119)

> studentRecords[24]["GPA"]
                              3.21                                  (4.120)
```

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

```
> calculateHash(student2["ID"])
                              2                                     (4.121)

> studentRecords[2] := student2
                  studentRecords_2 := student2                      (4.122)
```

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

```
> calculateHash(student3["ID"])
                              24                                    (4.123)
```

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

```
> evalb(studentRecords[25] = 0)
                              true                                  (4.124)
```

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.

```
> studentRecords[25] := student3
```

$$studentRecords_{25} := student3 \tag{4.125}$$

Printing Records

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

```
> studentRecords
```

$$\begin{bmatrix} 1..57\ Array \\ Data\ Type:\ anything \\ Storage:\ rectangular \\ Order:\ Fortran\_order \end{bmatrix} \tag{4.126}$$

This is not very informative. You can use the "Browse" tool from the Context Panel or apply **op**, but even those options only display the name of the table in each location, not the data.

```
> op(studentRecords)
```

$$1..57,\ \{2 = student2, 24 = student1, 25 = student3\},\ datatype = anything,\ storage = rectangular,\ order = Fortran\_order \tag{4.127}$$

We need to write a procedure to print out the data. To do this, we loop through the entries of the array and, for those that are nonzero, print the index and the data from the table stored in that position. Recall that the **entries** command applied to a table returns the data stored in the table.

```
PrintRecords := proc()
   local i;
   global studentRecords;
   for i from 1 to 57 do
      if studentRecords[i] <> 0 then
         print(i,entries(studentRecords[i]));
      end if;
   end do;
end proc:
```

```
> PrintRecords()
```

$$2,\ [2.89],\ [2908]$$
$$24,\ [3.21],\ [7319]$$
$$25,\ [3.42],\ [6578] \tag{4.128}$$

Note that we chose not to include the database as a parameter, but instead described the procedure in relation to the **studentRecords** array that we began above. This can result in a significant improvement in performance, especially when the array of records is long, particularly when it comes time to write procedures that modify the array, because the database does not have to be passed as an

argument to the procedure and then returned from it. The disadvantage, of course, is that in order to use a different name for the database, we have to revise the procedure.

In order to use **studentRecords**, it must be declared in the procedure as a global variable. This is done immediately after the local variables are declared with the keyword **global**. Without this declaration, Maple will assume that we forgot to list it as a local variable and will treat the **studentRecords** name inside the procedure as different from the **studentRecords** list we created outside the procedure.

Storing New Records

Now, we write a procedure **Store** to automate the process of storing information in the array. **Store** will accept two arguments, the ID and GPA of a student, and will add that student's record to the **studentRecords** array.

The first step in implementing **Store** will be to assign to a local variable, which we call **newrecord**, the table representing the student record. Then, **Store** needs to determine the location in the **studentRecords** array in which the record will be stored. In particular, it will need to avoid collision. To do this, we use something similar to the linear probing function defined in the text. Beginning with $i = 0$, we 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. Otherwise, we increment i and continue looking. Once we have found an open position, we just assign our **newrecord** to that position. We give **return NULL;** as the final command so that the procedure does not display anything when the record is successfully stored.

Here is the completed **Store** procedure.

```
Store := proc(id::integer,gpa::float)
   local hash, i, newrecord;
   global studentRecords;
   newrecord := table(["ID"=id,"GPA"=gpa]);
   for i from 0 to 56 do
      hash := calculateHash(id+i);
      if studentRecords[hash] = 0 then
         break;
      end if;
   end do;
   studentRecords[hash] := newrecord;
   return NULL;
end proc:
```

We now add a few records.

```
> Store(2216, 1.98)
```

```
> Store(1325, 3.14)
```

```
> Store(7061, 3.51)
```

Look again at **studentRecords** with **op**.

```
> op(studentRecords)
```

$$1..57,\ \{2 = student2, 15 = newrecord, 24 = student1, 25 = student3,\ 51 = newrecord, 52 = newrecord\}\ datatype = anything,\ storage = rectangular,\ order = Fortran\_order \quad \textbf{(4.129)}$$

Note that the three records we just added all appear as "newrecord" in the list. This is because **newrecord** was the name we used in the **Store** procedure. The **PrintRecords** procedure shows us, however, that despite having the same names, they store the correct information.

```
> PrintRecords()
```

2, [2.89], [2908]
15, [3.14], [1325]
24, [3.21], [7319]
25, [3.42], [6578]
51, [1.98], [2216]
52, [3.51], [7061] **(4.130)**

Because **newrecord** was declared as local within **Store**, Maple considers each one to be a distinct table, even though they have the same name.

Retrieving Records

We now have procedures for storing a student record in our database and for printing all of the records. 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 **Retrieve** procedure will accept a student ID number as its input and return the table storing the students record. Most of the work will take place within the same for loop as was in the **Store** procedure. We first test to make sure the location we are looking at is nonzero. If the location is 0, that tells us that the entry does not exist and the procedure will return **FAIL**.

Assuming the location is not 0, we check to see if the ID of the record in that position is the ID we are looking for. If so, we return the table by applying **eval** to the entry in **studentRecords**. (Without the **eval**, the procedure would return the name of the table rather than the table itself.) If the ID is not the one we are 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.

```
Retrieve := proc(id::integer)
   local hash, i;
   global studentRecords;
   for i from 0 to 56 do
      hash := calculateHash(id + i);
      if studentRecords[hash] = 0 then
         return FAIL;
      end if;
      if studentRecords[hash]["ID"] = id then
```

```
        return eval(studentRecords[hash]);
      end if;
    end do;
    return FAIL;
end proc:
```

> *Retrieve* (1325)

table (["*GPA*" = 3.14, "*ID*" = 1325]) **(4.131)**

> *Retrieve* (7061)

table (["*GPA*" = 3.51, "*ID*" = 7061]) **(4.132)**

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 truly 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 $x_{n+1} = (ax_n + c) \bmod m$. 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 the result of the **time** command by 1000. We apply the **floor** command to be certain that we obtain an integer.

> floor(*time*[*real*]( ) · 1000)

96 865 119 **(4.133)**

Invoking the **real** option causes the **time** command to return the real time elapsed since the Maple kernel was started, rather than the amount of CPU time that the kernel has used.

We will write two procedures 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 procedure **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. Thus, 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. Therefore, our modulus will be 8999. We 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 the time as described above.

The procedure is straightforward. Worthy of note is the use of the **from**...**to**...**do** loop without **for**. The purpose of the loop in the procedure is to repeat the same action a number of times. Since the action does not depend on the value of a loop variable, Maple allows the variable, and the **for** keyword, to be omitted. In fact, you can also omit "**from 1**" and Maple will assume 1 as the starting value. Neither of these options result in any significant performance difference, but they illustrate the flexibility of Maple's control structures.

```
randomIDs := proc(n::posint)
   local S, m, a, c, x;
   S := NULL;
   m := 8999;
   a := 57;
   c := 328;
   x := modp(floor(time[real]()*1000),m);
   from 1 to n do
      x := modp(a*x+c,m);
      S := S,x+1000;
   end do;
   return S;
end proc:
```

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

```
> someIDs := [randomIDs(10)]
```

$$someIDs := [3874, 3164, 7689, 4643, 2002, 4448, 8885, 9822, 9237, 2889] \quad \textbf{(4.134)}$$

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 divide the random integer by $2^{31} - 2$ and multiply by 4.

```
randomGPAs := proc(n::posint)
   local S, m, a, x, gpa;
   S := NULL;
   m := 2^31 - 1;
   a := 7^5;
   x := modp(floor(time[real]()*1000),m);
   from 1 to n do
      x := modp(a*x,m);
      gpa := convert((x/(m-1))*4,float,3);
      S := S,gpa;
   end do;
   return S;
end proc:
```

```
> someGPAs := [randomGPAs(10)]
```

$$someGPAs := [1.09, 0.538, 2.39, 0.933, 3.86, 1.97, 1.24, 1.23, 1.11, 1.20] \quad \textbf{(4.135)}$$

Note the use of **convert**. The expression **(x/(m-1))*4** is being converted into a floating point number with a precision of 3 significant digits.

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

```
> for i from 1 to 10 do
    Store(someIDs[i], someGPAs[i]);
  end do :
```

```
> PrintRecords ()
    2, [2.89], [2908]
    3, [1.97], [4448]
    4, [1.11], [9237]
    8, [3.86], [2002]
    15, [3.14], [1325]
    19, [1.23], [9822]
    24, [3.21], [7319]
    25, [3.42], [6578]
    27, [0.933], [4643]
    30, [0.538], [3164]
    40, [1.20], [2889]
    51, [1.98], [2216]
    52, [3.51], [7061]
    53, [2.39], [7689]
    54, [1.24], [8885]
    56, [1.09], [3874]                                    (4.136)
```

Check Digits

We conclude this section with a procedure 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

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

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

Our **checkISBN** procedure 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

```
> isbnExample := "0073383090"
    isbnExample := "0073383090"                           (4.137)
```

Remember that, in Maple, you can use the selection operation on a string in the same way as for a list. Therefore, the third character is obtained as follows:

```
> isbnExample[3]
    "7"                                                   (4.138)
```

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

```
> parse(isbnExample[3])
```

$$7 \tag{4.139}$$

Our procedure will compute the sum indicated by the formula above using the **add** command. Recall that the first argument to **add** 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.

```
checkISBN := proc(isbn::string)
   local i, check;
   check := modp(add(i*parse(isbn[i]),i=1..9),11);
   if check = 10 then
      return evalb(isbn[10] = "X");
   else
      return evalb(parse(isbn[10]) = check);
   end if;
end proc:
```

```
> checkISBN(isbnExample)
```

$$true \tag{4.140}$$

```
> checkISBN("084 930 149X")
```

$$false \tag{4.141}$$

```
> checkISBN("232 150 031X")
```

$$true \tag{4.142}$$

4.6 Cryptography

In this section, we will see how Maple 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 Commands for Working with Strings

Maple's **StringTools** package contains several commands that will be useful to us.

```
> with(StringTools):
```

First, the **UpperCase** command makes all of the letters in its input string upper case.

```
> UpperCase ("The quick brown fox")
    "THE QUICK BROWN FOX"                                          (4.143)
```

The **UpperCase** 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 second command we will use is the **Explode** command and its inverse **Implode**. The **Explode** command takes a string and returns a list of characters.

```
> Explode ("THE QUICK BROWN FOX")
    ["T", "H", "E", "", "Q", "U", "I", "C", "K", "",
      "B", "R", "O", "W", "N", "", "F", "O", "X"]                  (4.144)
```

The **Implode** command does the opposite. Given a list of strings, it joins them into one string.

```
> Implode (%)
    "THE QUICK BROWN FOX"                                          (4.145)
```

Mapping Characters to Integers

To represent the function that maps characters to integers, and its inverse, we will use two tables, **CharToNum** and **NumToChar**. In the **CharToNum** table, the space character and capital letters will serve as the indices with the corresponding integers the entries. The **NumToChar** table will be the reverse.

```
> CharToNum := table(["" = 0, "A" = 1, "B" = 2, "C" = 3, "D" = 4,
    "E" = 5, "F" = 6, "G" = 7, "H" = 8, "I" = 9, "J" = 10, "K" = 11,
    "L" = 12, "M" = 13, "N" = 14, "O" = 15, "P" = 16, "Q" = 17,
    "R" = 18, "S" = 19, "T" = 20, "U" = 21, "V" = 22, "W" = 23,
    "X" = 24, "Y" = 25, "Z" = 26]) :

> NumToChar := table([0 = "", 1 = "A", 2 = "B", 3 = "C", 4 = "D",
    5 = "E", 6 = "F", 7 = "G", 8 = "H", 9 = "I", 10 = "J", 11 = "K",
    12 = "L", 13 = "M", 14 = "N", 15 = "O", 16 = "P", 17 = "Q",
    18 = "R", 19 = "S", 20 = "T", 21 = "U", 22 = "V", 23 = "W",
    24 = "X", 25 = "Y", 26 = "Z"]) :
```

To compute the numeric value associate to a character, we use the **CharToNum** table.

```
> CharToNum["K"]
    11                                                             (4.146)
```

In the other direction, we get the character associated to a number using the **NumToChar** table.

```
> NumToChar[18]
    "R"                                                            (4.147)
```

Converting Between a String and a Numeric Representation
We now have the tools needed to encode a string as a list of numbers and a decode the numeric representation as a string.

In the **StringToNums** procedure, we will first apply **UpperCase** and **Explode** to produce a list of uppercase characters. We then use the **map** command to apply the **CharToNum** table to each character. When **map** is applied to a procedure (in this case a functional operator) and a list, it returns the list obtained by applying the procedure to each element of the list. In this case, we will use the functional operator **c -> CharToNum[c]** as **map**'s first argument. Note that we declare **CharToNum** as a global variable. This is not strictly necessary since Maple will infer that it is global from context, but it is a good programming habit to always declare global variables as such.

```
StringToNums := proc(s::string)
    local S;
    uses StringTools;
    global CharToNum;
    S := Explode(UpperCase(s));
    S := map(c -> CharToNum[c],S);
    return S;
end proc:
```

```
> StringToNums("The quick brown fox")
  [20, 8, 5, 0, 17, 21, 9, 3, 11, 0, 2, 18, 15, 23, 14, 0, 6, 15, 24]        (4.148)
```

The **NumsToString** procedure begins with a list of integers and returns the string.

```
NumsToString := proc(S::list)
    local s;
    global NumToChar;
    s := map(c -> NumToChar[c],S);
    s := Implode(s);
    return s;
end proc:
```

```
> NumsToString([8, 5, 12, 12, 15, 0, 23, 15, 18, 12, 4])
  "HELLO WORLD"                                                              (4.149)
```

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

Classical Cryptography

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

$$f(p) = (ap + b) \pmod{27},$$

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 a 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 numeric representation via **StringToNums**. Second, the function f is applied to each number. Third, the **NumsToString** procedure transforms the result back into a string. Our **AffineCipher** procedure accepts as input a string and values of a and b.

```
AffineCipher := proc(s::string, a::integer, b::integer)
   local S, T;
   S := StringToNums(s);
   T := map(p -> modp(a*p+b,27),S);
   return NumsToString(T);
end proc:
```

Note the use of **map** to apply the function $f(p) = (ap + b) \pmod{27}$ to each character.

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

```
> AffineCipher("The quick brown fox",5,3)
```

"VPACG URDCMLXJSCFXO" **(4.150)**

To decrypt the message, we use the same procedure. The discussion following Example 4 in Section 4.5 of the text indicates that decrypting amounts to solving $c \equiv (ap + b) \pmod{27}$ 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 $(a^{-1}, -a^{-1}b)$.

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

```
> 5^(-1) mod 27
```

11 **(4.151)**

Second, compute $-a^{-1}b$, being sure to include the negative.

```
> -5^(-1) · 3 mod 27
```

21 **(4.152)**

Thus, the decryption key is $(11, 21)$.

```
> AffineCipher((4.150),11,21)
```

"THE QUICK BROWN FOX" **(4.153)**

RSA Encryption

We will now see how to use Maple 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 300 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.

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 Maple 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 $n = pq$ 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 Maple procedure to handle key generation. The **GenerateKeys** procedure 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 $n = pq$, $\phi(n) = (p-1)(q-1)$, and $d = e^{-1} \pmod{\phi(n)}$.

```
GenerateKeys := proc(p::prime, q::prime)
   local n, phin, e, d;
   e := 13;
   n := p * q;
   phin := (p - 1) * (q - 1);
   d := modp(e^(-1),phin);
   return [[n,e],[n,d]];
end proc:
```

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 $q = 71$.

```
> keys := GenerateKeys(59, 71)
```

$$keys := [[4189, 13], [4189, 937]] \quad \textbf{(4.154)}$$

The public and private keys are

```
> publickey := keys[1]
```

$$publickey := [4189, 13] \quad \textbf{(4.155)}$$

```
> privatekey := keys[2]
```

$$privatekey := [4189, 937] \quad \textbf{(4.156)}$$

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 $n = 4189$ 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. However, 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** procedure from above to get the numeric representation of each character.

```
> messageString := StringToNums("SECRET MESSAGE")
```

$$messageString := [19, 5, 3, 18, 5, 20, 0, 13, 5, 19, 19, 1, 7, 5] \quad \textbf{(4.157)}$$

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 the second block, since 0318 and 318 are equivalent. We can obtain the desired results by multiplying the first number in each pair by 100 as follows.

```
> messageCode := []:

> for i from 2 to nops(messageString) by 2 do
    messageCode := [op(messageCode),
      100 · messageString[i − 1] + messageString[i]]:
  end do:

> messageCode
```

$$[1905, 318, 520, 13, 519, 1901, 705] \quad \textbf{(4.158)}$$

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 $C \equiv M^e \pmod{n}$.

```
RSA := proc(key::[posint,posint], msg::list(posint))
   local n, e, C;
   n := key[1];
   e := key[2];
   C := map(m -> modp(m &^ e,n),msg);
   return C;
end proc:
```

Our "SECRET MESSAGE" is encrypted as

```
> cipherText := RSA(publickey, messageCode)
```

$$cipherText := [723, 3360, 2306, 1979, 2695, 917, 1863] \quad \textbf{(4.159)}$$

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

```
> RSA(privatekey, cipherText)
```

$$[1905, 318, 520, 13, 519, 1901, 705] \quad \textbf{(4.160)}$$

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 Maple'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 300 digits. Moreover, they should be selected in an unpredictable fashion. To do this in Maple, we can use the **rand** command to produce a random 300 digit number. Then, we use the **nextprime** command to find the smallest prime number that exceeds our random number. This will guarantee that the prime number has at least 300 digits.

The **rand** command can be used with or without an argument. Without an argument, it returns a random 12 digit nonnegative integer. That is not nearly large enough for our purposes. The other way to use **rand** is to give a range of integers as the argument (or a single integer which is interpreted as the range from 0 to the given value). In our case, we want integers between 10^{299} and 10^{300}. In this form, **rand** does not return such an integer. Instead, its result is a procedure that produces integers in the specified range. Therefore, we need to assign a name like **bigInt** to the result of **rand** and then call **bigInt()** to produce the integers.

```
> bigInt := rand(10^299 ..10^300)
```

```
> a := bigInt()
```

$$a := 235\,726\,808\,767\,240\,131\,892\,387\,679\,230\,487\,763\,079\,953\,883\,726\,554\,914\,270\,335\,331\,988\,936\,067\,036\,004\,248\,743\,352\,686\,385\,930\,952\,790\,393\,226\,156\,791\,082\,950\,595\,300\,471\,690\,316\,314\,646\,561\,668\,134\,009\,554\,428\,603\,253\,983\,753\,534\,033\,365\,546\,308\,155\,005\,729\,686\,838\,034\,395\,625\,006\,588\,935\,807\,374\,308\,092\,774\,154\,589\,243\,063\,887\,435\,863\,555\,163\,865\,559\,297\,655\,953\,965\,900\,318\,854\,667\,516\,172\,768 \quad \textbf{(4.161)}$$

```
> b := bigInt()
```

$$b := 152\,437\,929\,765\,352\,918\,510\,221\,499\,252\,192\,939\,125\,747\,817\,161\,311\,103\,686\,087\,626\,346\,460\,386\,764\,591\,409\,816\,784\,447\,234\,447\,040\,237\,967\,851\,784\,887\,467\,280\,704\,542\,118\,084\,668\,842\,861\,067\,385\,555\,997\,426\,820\,063\,761\,992\,732\,944\,467\,932\,964\,476\,405\,481\,212\,187\,426\,959\,784\,692\,692\,674\,837\,781\,675\,579\,996\,667\,516\,040\,022\,110\,392\,777\,999\,797\,121\,227\,204\,804\,954\,864\,379\,556\,266\,233\,780\,403\,407\,631 \quad \textbf{(4.162)}$$

Now, we apply **nextprime** to **a** and **b** in order to produce the needed primes.

```
> p := nextprime(a)
```

$$p := 235\,726\,808\,767\,240\,131\,892\,387\,679\,230\,487\,763\,079\,953\,883\,726\,554\,914\,270\,335\,331\,988\,936\,067\,036\,004\,248\,743\,352\,686\,385\,930\,952\,790\,393\,226\,156\,791\,082\,950\,595\,300\,471\,690\,316\,314\,646\,561\,668\,134\,009\,554\,428\,603\,253\,983\,753\,534\,033\,365\,546\,308\,155\,005\,729\,686\,838\,034\,395\,625\,006\,588\,935\,807\,374\,308\,092\,774\,154\,589\,243\,063\,887\,435\,863\,555\,163\,865\,559\,297\,655\,953\,965\,900\,318\,854\,667\,516\,172\,901 \quad \textbf{(4.163)}$$

```
> q := nextprime(b)
```

$$q := 152\,437\,929\,765\,352\,918\,510\,221\,499\,252\,192\,939\,125\,747\,817\,161\,311\,103\,686\,087\,626\,346\,460\,386\,764\,591\,409\,816\,784\,447\,234\,447\,040\,237\,967\,851\,784\,887\,467\,280\,704\,542\,118\,084\,668\,842\,861\,067\,385\,555\,997\,426\,820\,063\,761\,992\,732\,944\,467\,932\,964\,476\,405\,481\,212\,187\,426\,959\,784\,692\,692\,674\,837\,781\,675\,579\,996\,667\,516\,040\,022\,110\,392\,777\,999\,797\,121\,227\,204\,804\,954\,864\,379\,556\,266\,233\,780\,403\,408\,163 \quad \textbf{(4.164)}$$

It is left to the reader to incorporate these ideas in improved versions of our **GenerateKeys** and **RSA** procedures.

Homomorphic Encryption

The text defines what it means for a cryptosystem to be homomorphic and demonstrates, in Example 11, that RSA is multiplicatively homomorphic. With our RSA encryption and decryption algorithms in place, we can make this fact a bit more concrete.

Suppose that, using the same keys as above, that we have encrypted and then stored the value 23 in the cloud.

```
> cloud := RSA(publickey, [23])[1]
```

$$cloud := 3655 \quad \textbf{(4.165)}$$

Note that we apply the selection operator because our **RSA** procedure was written to work on lists of values, and we are focused here on storing and manipulating a single value.

Having stored this value, suppose we later need to multiply it by 45. One option, of course, would be to retrieve and decrypt the value, perform the multiplication locally, and then encrypt and store the product. This, though, is rather inefficient. Particularly if this multiplication is but one of a vast number of operations we need to perform on stored data, running the computations on a very powerful remote machine can be desirable. That RSA is multiplicatively homomorphic means that we can, instead, encrypt the value 45 and have the multiplication performed on the cloud server.

```
> cloud := cloud · RSA(publickey, [45])[1]
```

$$cloud := 7\,894\,800 \quad \textbf{(4.166)}$$

After having the computations performed remotely, we now retrieve and decrypt the result.

```
> RSA(privatekey, [cloud])
```

[1035] **(4.167)**

Observe that this is identical to $23 \cdot 45$.

```
> 23 · 45
```

1035 **(4.168)**

One issue to be aware of when performing computations this way is that, just like the message code must be less than the modulus above, the result of the computation should be less than the modulus.

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 54 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,

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

Therefore, set $a_1 = a \pmod 2$, and let y_1 be the remainder with the 2 divided out. In other words, $y_1 = \dfrac{a - a_1}{2}$, 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 $a_2 = y_1 \pmod 3$ and let $y_2 = \dfrac{y_1 - a_2}{3}$.

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

The algorithm described above leads to the procedure below which accepts a positive integer as input and returns a list of integers $[a_1, a_2, .., a_n]$.

```
1 CantorExpression := proc(a::posint)
2   local A, n, y;
```

```
    A := [];
    n := 1;
    y := a;
    while y <> 0 do
       A := [op(A),modp(y,n+1)];
       y := (y-A[n])/(n+1);
       n := n+1;
       end do;
    return A;
end proc:
```

> *CantorExpression* (471)

$$[1, 1, 2, 4, 3] \tag{4.169}$$

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 use a relatively small prime.

> *DHprime* := *nextprime* (*rand* ())

$$DHprime := 395\,718\,860\,549 \tag{4.170}$$

For the primitive root, we use **PrimitiveRoot** to get the smallest primitive root.

> *DHroot* := *primroot* (*DHprime*)

$$DHroot := 3 \tag{4.171}$$

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

> *AliceSends* := *DHroot* &^ 421 **mod** *DHprime*

$$AliceSends := 287\,654\,735\,840 \tag{4.172}$$

Note that we are using the **&^** exponentiation operator so that Maple will use smarter and faster exponentiation algorithms.

(3) Bob also chooses a secret integer k_2 and sends the value to Alice. From the perspective of Alice, we do not know what value of k_2 that Bob chooses, only the value of $a^{k_2} \pmod p$. Thus, we have Maple choose k_2 randomly in the computation.

> *BobSends* := *DHroot* &^ *rand*( ) **mod** *DHprime*

$$BobSends := 346\,411\,045\,536 \tag{4.173}$$

(4) and (5) Alice computes $\left(a^{k_2}\right)^{k_1} \pmod p$ 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.

> *sharedKey* := *BobSends* &^ 421 **mod** *DHprime*

sharedKey := 318 885 707 608 **(4.174)**

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 Maple 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 procedure will output a list of those primes p for which $2^p - 1$ is prime.

```
CheckMersenne := proc(max::posint)
   local p, L;
   p := 2;
   L := [];
   while p <= max do
      if isprime(2^p-1) then
         L := [op(L),p];
      end if;
      p := nextprime(p);
   end do;
   return L;
end proc:
```

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

> *CheckMersenne*(100)

[2, 3, 5, 7, 13, 17, 19, 31, 61, 89] **(4.175)**

For another approach, consider the **IsMersenne** command from the **NumberTheory** package. This command is based on a table lookup. The command accepts one argument, either an integer or a list with one integer element. If you pass an integer n to the **IsMersenne** command, it will compute $2^n - 1$. If that value is prime, the command returns it. It returns false if $2^n - 1$ is composite. If the command cannot determine whether $2^n - 1$ is prime or not, it returns **FAIL**.

> *IsMersenne*(19)

true **(4.176)**

> *IsMersenne*(20)

false **(4.177)**

> *IsMersenne*(457 237 649 731)

FAIL **(4.178)**

The **IthMersenne** function will return the ith Mersenne prime, provided it has been found.

> *IthMersenne*(5)

13 **(4.179)**

```
> IthMersenne(100)
```

Error, (in NumberTheory:-IthMersenne) only 50 Mersenne primes are known

It is of note that there is a better test for checking whether a Mersenne number is prime, called the Lucas–Lehmer test, that is more efficient and can be implemented in Maple. 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 Maple procedure that, given a maximum n, tests the integers of the given form.

```
CE5 := proc(max::posint)
   local n, L;
   L := [];
   for n from 1 to max do
      if isprime(n^2+1) then
         L := [op(L),n^2+1];
      end if;
   end do;
   return L;
end proc:
```

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

```
> CE5(100)
```

$$[2, 5, 17, 37, 101, 197, 257, 401, 577, 677, 1297, 1601, 2917, 3137, 4357, 5477, 7057, 8101, 8837] \qquad \textbf{(4.180)}$$

Exercises

Exercise 1. Test which is faster for computing the greatest common divisor of a collection of integers, the **igcd** or **gcd** command.

Exercise 2. Use Maple to generate the list of the first 100 prime numbers larger than one million.

Exercise 3. Use Maple to find the one's complement of an arbitrary integer (see the prelude to Exercise 40 of Section 4.2).

Exercise 4. For which odd prime moduli are -1 a square? That is, for which prime numbers p does there exist an integer x such that $x^2 \equiv -1 \pmod{p}$?

Exercise 5. Use Maple 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 Maple functions **ModularSquareRoot** and **msolve** may be of use.)

Exercise 6. Use Maple to find the base 2 expansion of the 4th Fermat number $F_4 = 65\,537$. 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 (i.e., the time 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 squarings and multiplications. Why do you think F_4 is a good choice for the public exponent in the RSA encryption scheme?

Exercise 7. Modify the procedure **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.

Exercise 8. Write Maple routines 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 procedures should accept as input the block size.

Exercise 9. There are infinitely many primes of the form $4n + 1$ and infinitely many of the form $4n + 3$. Use Maple 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?

Exercise 10. Develop a procedure 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 Maple?

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

Exercise 12. Compute the sequence of pseudorandom numbers generated by the linear congruential generator $x_{n+1} = (ax_n + c) \pmod{m}$ 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 (period is defined in Exercise 14) for the sequence that you generate? Formulate a conjecture.

Exercise 13. The Maple command **tau** (in the **NumberTheory** package) implements the function defined, for all positive integers n, by: $\tau(n)$ is the number of positive divisors of n. Use Maple to study the function τ. What conjectures can you make about it? For example, when is $\tau(n)$ odd? Is there a formula for $\tau(n)$? For which integers m does the equation $\tau(n) = m$ have a solution for some integer n? Is there a formula for $\tau(mn)$ in terms of $\tau(m)$ and $\tau(n)$?

Exercise 14. 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 $N \leq n$. 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 $a_1 \pmod{m}, a_2 \pmod{m}, a_3 \pmod{m}, \ldots$ is periodic. Use Maple 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.

Exercise 15. Write a function to implement the Paillier cryptosystem, described in the preamble to Exercise 34 of Section 4.6 in the main text. Use your function to build a simple voting system with Maple. Your system should store the number of votes for each candidate as a list in which the entries are encrypted. When a user casts a vote, their vote should be encrypted and then added to the encrypted totals, taking advantage of the fact that the Paillier system is additively homomorphic. (Keep in mind that addition of plaintext is accomplished through multiplication of ciphertext.)

Exercise 16. (Class project) The Data Encryption Standard (DES) specifies a widely used algorithm for private key cryptography. Find a description of this algorithm (e.g, in *Cryptography, Theory and Practice* by Douglas Stinson). Implement the DES in Maple.