5 Induction and Recursion

Introduction

In this chapter we describe how Maple can be used to help you make conjectures and prove them with mathematical induction and strong induction. We will also look at several examples of using Maple to explore recursive definitions and to implement recursive algorithms. Recursion is an important tool in any computer programming language, and Maple is no exception. We conclude the section with an example of proving program correctness of a Maple procedure.

5.1 Mathematical Induction

In this section we will demonstrate how to use Maple both to discover propositions and to aid in the use of mathematical induction to verify them. We begin with two examples of how to use Maple to discover and prove a summation formula. We then consider a question of divisibility.

Summation Example 1

As our first example, we will explore a formula that you've already seen:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}.$$

This formula is the subject of Example 1 in Section 5.1 of the text. Here, we will proceed as if we did not already know the formula.

Listing and graphing to find the formula

Our first step is to discover the formula. To do this, we will have Maple compute the sums for a variety of values of n, using the **add** command.

The **add** command requires two arguments. The first argument is an expression representing the values to be added in terms of a variable. The second argument will be an equation with that variable on the left and a range indicating the bounds of the summation on the right side of the equation. The syntax is modeled on the summation symbol syntax. To compute

$$\sum_{i=a}^{b} f(i),$$

you enter `add(f(i),i=a..b)`.

In our situation, we want to add the first several positive integers. For example, the sum of the first ten positive integers is

```
> add(i,i=1..10);
```

$$55 \qquad \textbf{(5.1)}$$

To discover the formula for the sum of the first n positive integers, we will want several specific examples to analyze. To calculate a lot of examples at once, we'll use a name for the maximum value in the range: `add(i,i=1..n)`. Maple won't execute that command since `n` has no value. But we can make `add(i,i=1..n)` the first argument to **seq**, with `n` as the index to the sequence. This will produce the sums of the first n positive integers for different values of n.

```
> seq(add(i,i=1..n),n=1..50);
```

1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136, 153, 171, 190, 210, 231, 253, **(5.2)**

276, 300, 325, 351, 378, 406, 435, 465, 496, 528, 561, 595, 630, 666, 703, 741, 780, 820, 861, 903, 946, 990, 1035, 1081, 1128, 1176, 1225, 1275

Remember that we're working as if we do not already know the answer. Just looking at the data, you might notice a pattern, but if not, it's sometimes helpful to pair the value of n with the result. To do this, we only need to modify the **seq** command so that the first argument is the list whose first element is **n** and whose second is the **add** command.

```
> seq([n,add(i,i=1..n)],n=1..50);
```

[1, 1], [2, 3], [3, 6], [4, 10], [5, 15], [6, 21], [7, 28], [8, 36], [9, 45], [10, 55], [11, 66], [12, 78], [13, 91], [14, 105], [15, 120], [16, 136], [17, 153], [18, 171], [19, 190], [20, 210], [21, 231], [22, 253], [23, 276], [24, 300], [25, 325], [26, 351], [27, 378], [28, 406], [29, 435], [30, 465], [31, 496], [32, 528], [33, 561], [34, 595], [35, 630], [36, 666], [37, 703], [38, 741], [39, 780], [40, 820], [41, 861], [42, 903], [43, 946], [44, 990], [45, 1035], [46, 1081], [47, 1128], [48, 1176], [49, 1225], [50, 1275] **(5.3)**

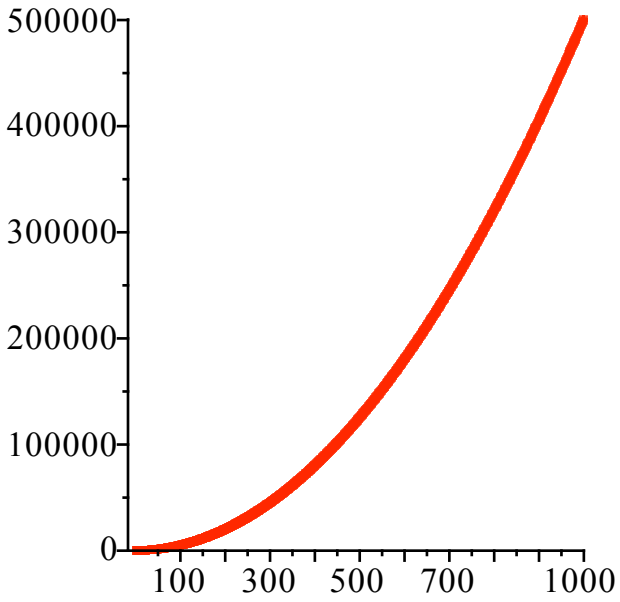
The entry [23, 276] indicates that the sum of the first 23 positive integers is 276.

This still may not be enough to get an idea of what the formula could be, in which case a graph of the data might be of use. The **plot** command is used to graph functions and data, and was first discussed in Section 2.3 of this manual. In this situation, we want to plot the points that the previous application of **seq** produced: (1, 1), (2, 3), (3, 6), (4, 10),.... To do this, we need two lists. One list must contain all of the "x" coordinates and the other list all the "y" coordinates. The lists should be the same size and need to match up. That is, the first element of the x list must correspond to the first element of the y list, the second element of the x list must match the second element of the y list, and so on. To get the y-coordinates, we will just repeat **(5.2)** and the x values are just the values of n. We will name the x-coordinate list **nList** and the y-coordinates will be **sumList**. Since Maple computes the sums quickly, we'll go out to a maximum of n = 1000. That way, we'll be sure to have a good idea of the shape of the graph.

```
> nList := [seq(n,n=1..1000)]:
> sumList := [seq(add(i,i=1..n),n=1..1000)]:
```

These lists will be the first two arguments to **plot**, with the x-values as the first argument and the y-values the second. These are the only required arguments when graphing data, but we will use several options. The **style=point** and **symbol=solidcircle** options will cause Maple to draw the graph as a series of dots. (The default is to "connect the dots" and draw the graph using straight line segments.) We will also increase the size of the dots slightly with the **symbolsize=10** option. We actually have enough data in this example that even with these options, the graph will look like a smooth line, but they are useful options to be aware of.

```
> plot(nList,sumList,style=point,symbol=solidcircle,symbolsize=
  10);
```



It is also worth mentioning the option **`view=[xmin..xmax,ymin..ymax]`** which specifies the region that is displayed in the graph. Without it, Maple chooses a view window.

The particular values in output **(5.3)** may not have been helpful at all in figuring out what kind of formula we were looking for. But this graph probably looks very familiar. It looks very much like the right half of a parabola, suggesting that the formula is quadratic. (Of course, it may be cubic or quartic or some other polynomial, but we'll start with the simplest possibility based on the graph.)

Finding the coefficients

Now that we have guessed the kind of formula, we can write it as $f(n) = an^2 + bn + c$. Determining the coefficients a, b, c is our next task. We will have Maple find them for us.

We already know a bunch of values for this function. Here are the first few again.

```
> seq([n,add(i,i=1..n)],n=1..10);
```

$[1, 1], [2, 3], [3, 6], [4, 10], [5, 15], [6, 21], [7, 28], [8, 36], [9, 45], [10, 55]$ **(5.4)**

This data tell us a lot of information about our formula. For instance, if we plug in $n = 2$, then $f(2) = 3$, meaning

$$3 = a \cdot 2^2 + b \cdot 2 + c = 4a + 2b + c.$$

For $n = 1$, we have

$$1 = a + b + c.$$

For $n = 3$:

$$6 = 9a + 3b + c.$$

Of course, Maple can produce these formulas for us. To do this, we'll use the **`eval`** command, which was introduced in Section 1.1. Remember that the **`eval`** command accepts two arguments. The first is an expression and the second is an equation specifying the substitution to be made. For example,

```
> eval(2*x+5,x=3);
```

11 **(5.5)**

evaluates the expression $2x + 5$ after substituting 3 for x. In our case, the first argument will be the

equation $f(n) = an^2 + bn + c$. Instead of $f(n)$, we need to use our data **sumList**. So our first argument will be **sumList[n] = a*n^2+b*n+c**. With the second argument **n=2**, say, Maple will replace the **n**'s with 2. It will then look up the second entry in **sumList** and simplify the right side of the equation for us.

```
> eval(sumList[n] = a*n^2 + b*n + c, n=2);
```

$$3 = 4a + 2b + c \tag{5.6}$$

We can create a list of such equations with the **seq** command.

```
> [seq(eval(sumList[n] = a*n^2 + b*n + c, n=N),N=1..10)];
```

$$[1 = a + b + c, 3 = 4a + 2b + c, 6 = 9a + 3b + c, 10 = 16a + 4b + c, 15 = 25a + 5b + c, 21 = 36a + 6b + c, 28 = 49a + 7b + c, 36 = 64a + 8b + c, 45 = 81a + 9b + c, 55 = 100a + 10b + c] \tag{5.7}$$

We now have a system of equations. In particular, we have 10 equations in three variables. You have probably seen systems of at least 2 and 3 equations in 2 or 3 variables in previous mathematics courses. You can have Maple solve the system of equations by applying the **solve** command to the list.

```
> solve((5.7));
```

$$\left\{a = \frac{1}{2}, b = \frac{1}{2}, c = 0\right\} \tag{5.8}$$

The first argument to **solve** can be a single equation or a set or list of equations. If a set or list of equations is provided, as we did here, Maple will solve the equations as a system of simultaneous equations (*i.e.*, it finds values for the variables that make all the equations true simultaneously). You can also provide an optional second argument: a name or a set or list of names. If provided, Maple will solve the equations for those variables only, treating any other names as if they were constants.

You may recall that to solve a system of equations in three unknowns, only three equations are required. In this situation, having more equations is useful. If we were wrong about the formula being quadratic but attempted to find coefficients with only three equations, Maple may still have found values for a, b, and c that satisfied the three equations we chose. With ten equations, if the actual formula were not quadratic, there is a greater chance that no values of a, b, and c would satisfy all ten equations. In that case, Maple would have returned nothing to indicate the absence of a solution and indicating that our guess about the kind of formula was incorrect.

Let's review what we've done so far. Our goal is to find a formula for the sum $\sum_{i=1}^{n} i$. We used Maple to compute a bunch of values of this sum and then graphed n versus the sum up to n. This graph suggested a quadratic formula, *i.e.*, one of the form $an^2 + bn + c$ for some values of a, b, and c. We then used Maple's **solve** command to determine that $a = b = \frac{1}{2}$ and $c = 0$. In other words, we've found the formula $\frac{1}{2}n^2 + \frac{1}{2}n$, which, of course, is the same as $\frac{n(n+1)}{2}$. Although we have found a formula, we have not yet *proven* anything, we've only made a conjecture.

The induction proof

To prove that our formula is correct, we use mathematical induction. First, let's make our formula into a functional operator.

```
> sumF := n -> (1/2)*n^2 + (1/2)*n;
```

$$sumF := n \to \frac{1}{2} n^2 + \frac{1}{2} n \tag{5.9}$$

To complete the basis step of the induction, we need to see that the formula agrees with the sum for $n = 1$.

```
> add(i,i=1..1);
```

$$1 \tag{5.10}$$

```
> sumF(1);
```

$$1 \tag{5.11}$$

They are equal and the basis step is verified.

For the inductive step, we assume that the formula is correct for k and need to demonstrate that it is true for $k + 1$. In Example 1, this was done by starting with the sum $1 + 2 + \cdots + k + (k + 1)$ and applying the inductive hypothesis to the first k terms to obtain $\frac{k(k+1)}{2} + (k + 1)$. Then algebra is used to turn that expression into the formula evaluated at $k + 1$. With Maple, we can just check whether the expressions are the same.

The sum of the first $k + 1$ terms is $1 + 2 + \cdots + k + (k + 1) = f(k) + (k + 1)$.

```
> sumF(k) + (k+1);
```

$$\frac{1}{2} k^2 + \frac{3}{2} k + 1 \tag{5.12}$$

That used the inductive hypothesis. We need to see if this is the same as $f(k + 1)$.

```
> sumF(k+1);
```

$$\frac{1}{2} (k + 1)^2 + \frac{1}{2} k + \frac{1}{2} \tag{5.13}$$

To check whether they are equal, we can form the equation obtained by equating the two expressions, apply **simplify** to the equation to have Maple simplify both sides as much as it can, and then use **evalb** to find out if the equation is an identity. Note that without **simplify**, Maple will not return the correct result. All **evalb** does in this case is to check to see if the two expressions are verbatim the same, it does not automatically do any algebra to check. The application of **simplify** causes Maple to do the symbolic algebra that allows **evalb** to recognize the truth of the equation.

```
> evalb(simplify(sumF(k) + (k+1) = sumF(k+1)));
```

$$true \tag{5.14}$$

This indicates that the inductive step is verified, and hence the formula is correct.

Summation Example 2

As a second example of using Maple to find and prove summation formulae, consider the sum $\sum_{i=1}^{n} i^2 (i + 1)!$. For this example, we won't go through the process of computing values and graphing as we did above. That is a very valuable process, and we strongly recommend that you go

through it yourself with a few examples. But Maple includes a command that will give us the result.

Using sum to determine the formula

We used the **add** command for calculating the numeric sum of a sequence of numbers. Maple also has a **sum** command, which is used for symbolic summation. To calculate $\sum_{i=1}^{10} i^2 (i+1)!$, we use **add** as we did in the previous example.

```
> add(i^2*(i+1)!,i=1..10);
```

$$4311014402 \qquad \textbf{(5.15)}$$

(Note the use of the exclamation mark for the computation of factorial. You can also use the command **factorial**, if you prefer.)

The **add** command is designed specifically for numeric computations like the above. For symbolic calculations, as in $\sum_{i=1}^{n} i^2 (i+1)!$, the **sum** command is used.

```
> sum(i^2*(i+1)!,i=1..n);
```

$$(n-1)\,(n+2)! + 2 \qquad \textbf{(5.16)}$$

The syntax of the two commands is nearly identical. The first argument is the expression to be summed in terms of an index variable. The second argument is an equation that sets the index variable equal to a range. The difference is that the **sum** command allows the range to be symbolic, for example **1..n** or **0..infinity**. Note that the **sum** command could be used for the numeric calculation as well, but **add** is optimized for that purpose.

Similarly, Maple includes **mul** for computing numeric products and **product** for computing symbolic products. These commands have the same syntax as **add** and **sum**.

The induction proof

Now that the **sum** command has determined the formula

$$\sum_{i=1}^{n} i^2 (i+1)! = (n-1)(n+2)! + 2,$$

we will use Maple to help us prove it. Even though we can be very confident that Maple has given us a correct formula, applying a Maple command is not the same as a proof.

As before, we'll define a functional operator based on the formula.

```
> sumF2 := n -> (n-1)*(n+2)! + 2;
```

$$sumF2 := n \rightarrow (n-1)\,(n+2)! + 2 \qquad \textbf{(5.17)}$$

For the basis step of the induction, we need to see that the formula holds for $n = 1$. The value of the sum for $n = 1$ is

```
> 1^2*(1+1);
```

$$2 \qquad \textbf{(5.18)}$$

And the formula applied to $n = 1$ is

```
> sumF2(1);
```

$$2 \qquad \textbf{(5.19)}$$

The basis step is verified.

For the inductive step, we assume that the formula is correct for k. That is, we assume that

$$\sum_{i=1}^{k} i^2 (i+1)! = (k-1)(k+2)! + 2.$$

We need to show that the formula works for $k+1$. Now,

$$\sum_{i=1}^{k+1} i^2 (i+1)! = \sum_{i=1}^{k} i^2 (i+1)! + (k+1)^2 ((k+1)+1)!.$$

Using the inductive hypothesis that the formula is correct for k, this is

```
> sumF2(k) + (k+1)^2*((k+1)+1)!;
```

$$(k-1)\,(k+2)! + 2 + (k+1)^2\,(k+2)! \qquad \textbf{(5.20)}$$

We need to check to see if this is the same as the formula applied to $k+1$.

```
> sumF2(k+1);
```

$$k\,(k+3)! + 2 \qquad \textbf{(5.21)}$$

```
> evalb(simplify(sumF2(k) + (k+1)^2*((k+1)+1)!=sumF2(k+1)));
```

$$true \qquad \textbf{(5.22)}$$

This completes the induction and proves the correctness of the formula.

Divisibility

For a final example, we'll see how Maple can be used to help prove results about divisibility. In Example 8 from Section 5.1 of the text, it was shown that $n^3 - n$ is divisible by 3 for all positive integers n. Exercise 33 asks you to prove that $n^5 - n$ is divisible by 5 for all positive integers. These two facts suggest that perhaps $n^7 - n$ is divisible by 7 for all positive integers n. Let's see how Maple can help us prove this fact.

We begin by creating a functional operator to represent the expression $n^7 - n$.

```
> divExpr := n -> n^7 - n;
```

$$divExpr := n \to n^7 - n \qquad \textbf{(5.23)}$$

The basis case is $n = 1$. For $n = 1$, the expression is

```
> divExpr(0);
```

$$0 \qquad \textbf{(5.24)}$$

which is divisible by 7, so the basis case holds.

The inductive hypothesis is that $k^7 - k$ is divisible by 7 for a positive integer k. We need to show that $(k+1)^7 - (k+1)$ is divisible by 7. To do this, we'll use the following fact: if $n|a$ and $n|b - a$ then $n|b$. (The reader should verify this statement, which is equivalent to Theorem 1, part (i), of Section 4.1.)

By assumption, 7 divides $k^7 - k$. We will have Maple compute the difference.

```
> divExpr(k+1)-divExpr(k);
```

$$(k+1)^7 - 1 - k^7 \qquad \textbf{(5.25)}$$

```
> simplify(%);
```

$$7k^6 + 21k^5 + 35k^4 + 35k^3 + 21k^2 + 7k \qquad \textbf{(5.26)}$$

You can see that the coefficients are all multiples of 7 and thus the expression is divisible by 7. We can confirm this with Maple by checking that the result of that expression modulo 7 is 0.

```
> % mod 7;
```

$$0 \qquad (5.27)$$

Thus, the inductive step is verified and hence $n^7 - n$ is divisible by 7 for all positive integers n.

5.2 Strong Induction and Well-Ordering

In this section, we'll see one way that Maple can be used to support a proof by strong induction. In particular, we will consider the class of problems illustrated in Example 4: prove that every amount of postage of 12 cents or more can be formed using just 4-cent and 5-cent stamps. The second solution to that example will form the model for this discussion. (See also Exercises 3 through 8 as other examples of problems of this kind.)

The basis step of the induction argument requires several propositions to be verified. Using the notation in the text, the propositions $P(b), P(b+1), \ldots, P(b+j)$ must all be demonstrated for some integer b and a positive integer j. Maple can be useful in these situations because it can often verify the basis cases for you. This is particularly useful when j is large.

Showing that every amount of postage over 12 cents can be formed using 4 and 5 cent stamps requires 4 basis cases: $P(12), P(13), P(14), P(15)$. We begin by showing how to use Maple to demonstrate $P(12)$, that postage of 12 cents can be formed using 4 and 5 cent stamps. While you may object that it is obvious that $12 = 4 \cdot 3$, our ultimate goal is to generalize our code to encompass the entire class of postage problems.

Making postage

To verify $P(12)$, we must find nonnegative integers a and b, representing the number of stamps of the two denominations, such that $12 = 4a + 5b$. The most straightforward way of finding a and b is to test all the possibilities. We know that a and b must both be nonnegative.

Also, the maximum possible values of a and b are $\left\lfloor \frac{12}{4} \right\rfloor$ and $\left\lfloor \frac{12}{5} \right\rfloor$, respectively. (Recall that $\lfloor x \rfloor$ is the notation used to represent the floor of x, that is, the largest integer less than or equal to x.) To see why this is so, consider b: $12 = 4a + 5b \geq 5b$ since $a \geq 0$. That is, $5b \leq 12$ and so $b \leq \frac{12}{5}$. And since b must be an integer, we have $b \leq \left\lfloor \frac{12}{5} \right\rfloor$. Likewise for a.

Since a and b must be between 0 and the floor of 12 divided by 4 or 5, respectively, we can use a pair of nested for loops to check all the possible values. Within the loops, we only need to check to see if $4a + 5b = 12$. If so, we'll print the pair $[a, b]$.

```
> for a from 0 to floor(12/4) do
    for b from 0 to floor(12/5) do
      if 4*a + 5*b = 12 then
        print([a,b]);
      end if;
    end do;
  end do;
```

$$[3, 0] \qquad (5.28)$$

We can very easily generalize this by replacing the target value and the stamp denominations with variables. The following procedure implements this generalization. It accepts the two denominations and the target value as input and returns a list whose elements represent the number of each kind of stamp required. In the event that the procedure fails to form the desired amount of

postage with the given stamps, it returns **FAIL**.

```
> MakePostage := proc(A::posint,B::posint,postage::posint)
    local a, b;
    for a from 0 to floor(postage/A) do
      for b from 0 to floor(postage/B) do
        if A*a + B*b = postage then
          return [a,b];
        end if;
      end do;
    end do;
    return FAIL;
  end proc:
```

Applying **MakePostage** with stamps 4 and 5 and postage 13, tells us that it requires two 4-cent stamps and one 5 cent stamps to make 13 cents postage.

```
> MakePostage(4,5,13);
```

$$[2, 1] \tag{5.29}$$

On the other hand, it is not possible to make 11 cents postage with 4 and 5 cent stamps.

```
> MakePostage(4,5,11);
```

$$FAIL \tag{5.30}$$

Automating the basis step

The **MakePostage** procedure finds the number of stamps of each denomination needed to produce the desired postage. As such, it verifies individual basis step propositions. With a simple loop, we can verify all of the basis cases. Recall that the basis step was to verify that postage can be made for 12, 13, 14, and 15 cents.

```
> for p from 12 to 15 do
    MakePostage(4,5,p);
  end do;
```

$$[3, 0]$$
$$[2, 1]$$
$$[1, 2]$$
$$[0, 3] \tag{5.31}$$

We will make a procedure for this in a moment, but first observe that the number of propositions in the basis step is equal to the smaller denomination stamp. The reason for this is in the proof of the inductive step. You should review the second solution to Example 4 in the text. The key point is that to make postage of $k + 1$ cents, the proof relies on the inductive assumption that you can make postage of $k + 1 - 4 = k - 3$ cents. This requires that $k - 3 \geq 12$, that is, $k \geq 15$.

Generically, if a is the smaller of the stamps and x is the minimum postage that we claim can be made ($x = 12$ in the example), then the inductive step requires $k + 1 - a \geq x$. Which is to say $k \geq x + (a - 1)$. Thus $P(x), P(x + 1), \ldots, P(x + (a - 1))$ must form the basis step.

This is useful to us in the following way: given the values of the stamps and the minimum value of postage, we can automate the verification of the appropriate basis cases.

```
> PostageBasis := proc(A::posint,B::posint,minpost::posint)
    local small, postList, post, R;
    small := min(A,B);
    postList := [];
```

```
    for post from minpost to (minpost + small - 1) do
      R := MakePostage(A,B,post);
      if R <> FAIL then
        postList := [op(postList),post=R];
      else
        return false;
      end if;
    end do;
    return postList;
  end proc:
```

We apply it to the example of using 4 and 5 cent stamps to make postage of at least 12 cents.

```
> PostageBasis(4,5,12);
```

$$[12 = [3, 0], 13 = [2, 1], 14 = [1, 2], 15 = [0, 3]] \quad \textbf{(5.32)}$$

The **PostageBasis** procedure above accepts as input the denominations of the two stamps, and the minimum value such that all postage values equal to or greater than that minimum can be made. The procedure uses the **min** command to set **small** equal to the lesser of the two denominations. The **postList** variable is used to store the information that shows how to make the various amounts of postage. This list will store equations of the form *postage* = [*a*, *b*] which indicate that the specified amount of postage can be made with *a* stamps of value *A* and *b* of *B*.

The for loop considers each of the basis cases (using the observation above). For each amount of postage, we use **MakePostage** to determine if it is possible to form that postage from the stamp values. If so, **MakePostage** returns a pair indicating how the desired postage is made and this is added to the **postList**.

If **MakePostage** ever returns **FAIL**, that indicates that the particular basis case cannot be verified. In this case, **PostageBasis** returns false, indicating that the basis step cannot be completed. For instance, the following indicates that it is false that every amount of postage of 12 cents or larger can be obtained using 4 and 6 cent stamps.

```
> PostageBasis(4,6,12);
```

$$\textit{false} \quad \textbf{(5.33)}$$

5.3 Recursive Definitions and Structural Induction

In this section we will show how functions and sets can be defined recursively in Maple.

A Simple Recursive Function

First we'll consider the recursively defined function from Example 1 in Section 5.3 of the text. This function is defined by $f(0) = 3$ and $f(n+1) = 2 \cdot f(n) + 3$.

In order to represent this function in Maple, the first thing we must do is transform the recursive part of the definition into an equation for $f(n)$ instead of $f(n+1)$. We have to decrease all of the arguments in the recursive part of the definition by 1: $f(n) = 2f(n-1) + 3$. The formula $f(n+1) = 2f(n) + 3$ is perhaps more expressive in that the $n+1$ suggests that the definition is about the "next" value of the function. But Maple cannot interpret $n+1$ as a parameter in a function definition.

It is also important to point out that changing the recursive formula also changes the domain over which it is valid. The formula $f(n+1) = 2f(n) + 3$ applies for all $n \geq 0$, while $f(n) = 2f(n-1) + 3$ applies for all $n \geq 1$. This does not affect the value of $f(n)$ for any n, however.

In Section 2.3 of this manual, we saw three ways to represent functions in Maple: as a procedure, a functional operator, or as a table. Technically, all three of these representations could be used to represent a recursively defined function. However, a table representation would be less natural and much less convenient for implementing the recursion than the other options. We will not use tables in this section.

The most natural way to create a recursively defined function in Maple is with a functional operator. Recall that the definition of a functional operator takes the following form: the name of the operator followed by the `:=` assignment operator, then the arguments to the function followed by the `->` arrow operator, and finally the formula defining the function. For example, the function $f(x) = 3x^2 - 2x + 4$ would be defined by the following.

```
> functionEx := x -> 3*x^2 - 2*x + 4;
```

$$functionEx := x \to 3x^2 - 2x + 4 \qquad \textbf{(5.34)}$$

Defining a functional operator recursively is essentially the same. Just as in the mathematical formula $f(n) = 2f(n-1) + 3$, the formula contains reference to the function name.

```
> f := n -> 2*f(n-1) + 3;
```

$$f := n \to 2f(n-1) + 3 \qquad \textbf{(5.35)}$$

We also must define the basis step. To declare $f(0) = 3$, you make the assignment

```
> f(0) := 3;
```

$$f(0) := 3 \qquad \textbf{(5.36)}$$

(Note that the basis values must be declared after the recursive formula has been assigned.)

Now that the recursive definition and the basis step have been assigned, you can use `f` like any other function.

```
> f(10);
```

$$6141 \qquad \textbf{(5.37)}$$

And you can compute the values of f from 1 to 9 using `seq`.

```
> seq(f(n),n=1..9);
```

$$9, 21, 45, 93, 189, 381, 765, 1533, 3069 \qquad \textbf{(5.38)}$$

Remember tables

It's worth understanding a bit of what Maple is doing when you define and evaluate a recursively defined function. Any time you evaluate a procedure (including a functional operator), Maple automatically checks the procedure's remember table (if one exists) before executing any commands. The remember table for a procedure is a table used to keep track of the output for certain input values. So if you've already executed a procedure on a particular input and stored it in the remember table, Maple can return the previous result rather than recomputing it.

When you define the recursive formula for a function, Maple stores the formula as the definition of the functional operator. When you then enter the statement `f(0) := 3;`, Maple interprets that as a command to add an entry to `f`'s remember table. In fact, you can look at the remember table of any procedure you write by applying `op` and `eval` to the name of the procedure.

```
> op(eval(f));
```

$$n, operator, arrow, table([0 = 3]) \qquad \textbf{(5.39)}$$

The last entry in this list is the remember table which stores the pair `0=3` to indicate that the value of the procedure applied to 0 is 3.

Note that if you define the basis value first and then enter the recursive definition, the functional operator definition wipes out the basis value. This is a good feature to have, because if you define a procedure and then later redefine it to something else, you don't want the new function reporting values from the old function's remember table.

If you apply **f** to 0, Maple sees that 0 is an index in the table and returns the value. If you apply **f** to a different value, say 2, Maple sees that 2 is not in the table and so it applies the formula. The formula says that $f(2) = 2 \cdot f(1) + 3$. When Maple tries to evaluate this, it recognizes that it needs to find $f(1)$. Again, it checks the remember table and, not finding 1 in the table, it applies the formula $f(1) = 2 \cdot f(0) + 3$. Since $f(0)$ is in the remember table, Maple looks up $f(0)$, which allows it to compute $f(1) = 9$ and then $f(2) = 21$.

You may be wondering why output **(5.39)** indicates that **f**'s remember table only stores the value for $f(0)$, even though we've computed other values. The reason is that Maple doesn't automatically store values in a remember table. Most of the time, you don't execute procedures on the same input multiple times, so it would be a waste of memory to store a remember table for every procedure. You have to explicitly tell Maple to create a remember table and store values in it.

For a functional operator, we've seen that you explicitly store a value in the remember table with the syntax **f(x) := v**. Unfortunately, there is no way to have a functional operator automatically store values that it computes. Even though we computed $f(10)$ (which required the recursive calculations $f(9), f(8), f(7), \ldots f(1)$), we did not explicitly tell Maple to store any of those values and so it did not add them to the remember table. Procedures, on the other hand, do give you the option to have Maple automatically populate the remember table with all the values they compute. We will see this in the next example.

You may occasionally want to clear a procedure or function's remember table. To do this, you apply the **forget** command to the name of the function or procedure, *e.g.*, **forget(f)**.

A Second Recursive Function

Now let's consider the function F defined by the basis values $F(0) = 1$ and $F(1) = 1$ and the recursive formula $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$. This is the function whose values are the Fibonacci sequence.

We could define this function as a functional operator as we did in the previous subsection. However, modeling the function as a procedure will allow us to instruct Maple to automatically create a remember table. This will make the procedure much more efficient.

To model the function F as a procedure, we define a procedure **F** that accepts a positive integer as input. The only statement in the procedure is the computation defined by the recursive formula. We instruct Maple to automatically construct a remember table by issuing the option **remember** as shown below.

```
> F := proc(n::nonnegint)
    option remember;
    F(n-1) + F(n-2);
  end proc:
```

We declare the basis values in the same way as for functional operators.

```
> F(0) := 1;
```

$$F(0) := 1 \qquad \textbf{(5.40)}$$

```
> F(1) := 1;
```

$$F(1) := 1 \tag{5.41}$$

The procedure **F** is now completely defined and produces correct output.

```
> seq(F(n),n=0..10);
```

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 \tag{5.42}$$

Also note that, unlike the functional operator above, **F** has added the results of its calculations to its remember table.

```
> op(eval(F));
```

$$n::nonnegint, remember, table([0 = 1, 1 = 1, 2 = 2, 3 = 3, 4 = 5, 5 = 8, 6 = 13, 7 = 21, 8 = 34, 9 = 55, 10 = 89]) \tag{5.43}$$

Comparing Complexity

The difference in time complexity between a recursive procedure that builds a remember table and one that doesn't is very significant, perhaps much more than you might think. Let's create two new procedures, both of which model the function $g(n) = 2 \cdot g(n-1) + 3 \cdot g(n-2)$ with $g(0) = 5$ and $g(1) = 2$. The procedure **gR** will include the remember option while **gF** will be "forgetful" and not build a remember table.

Here are the two procedures.

```
> gR := proc(n::nonnegint)
    option remember;
    2*gR(n-1) + 3*gR(n-2);
  end proc:
```

```
> gR(0) := 5;
```

$$gR(0) := 5 \tag{5.44}$$

```
> gR(1) := 2;
```

$$gR(1) := 2 \tag{5.45}$$

```
> gF := proc(n::nonnegint)
    2*gF(n-1) + 3*gF(n-2);
  end proc:
```

```
> gF(0) := 5;
```

$$gF(0) := 5 \tag{5.46}$$

```
> gF(1) := 2;
```

$$gF(1) := 2 \tag{5.47}$$

In order to track what these procedures do, and particularly what they do differently, we're going to trace them both. The **trace** command accepts as its arguments a sequence of procedure names. It has the effect of "turning on" tracing (also called debugging) for the listed procedures. In particular, when tracing is on and you execute a procedure, Maple will display any results of assignments made within the procedure and calls to other procedures. Let's turn on tracing for both **gR** and **gF**.

```
> trace(gR,gF);
```

$$gR, gF \tag{5.48}$$

Tracing the procedure with a remember table

Now we'll execute **gR** on 5.

```
> gR(5);
{--> enter gR, args = 5
```

```
{--> enter gR, args = 4
{--> enter gR, args = 3
{--> enter gR, args = 2
value remembered (in gR): gR(1) -> 2
value remembered (in gR): gR(0) -> 5
                                19
<-- exit gR (now in gR) = 19}
value remembered (in gR): gR(1) -> 2
                                44
<-- exit gR (now in gR) = 44}
value remembered (in gR): gR(2) -> 19
                               145
<-- exit gR (now in gR) = 145}
value remembered (in gR): gR(3) -> 44
                               422
<-- exit gR (now at top level) = 422}
                               422
```

(5.49)

Let's analyze what happened. The first line of green output tells us that the `gR` procedure was entered (or called or executed) with argument 5. That's because we entered the command `gR(5)`.

When `gR` was executed on 5, its only command is the computation `2*gR(4) + 3*gR(3)`. When Maple sees `gR(4)`, it immediately calls the `gR` procedure on the argument 4. (`gR(3)` has to wait until Maple resolves `gR(4)`.) This is the second line of green.

Maple is now executing `gR(4)`. The same thing happens. Maple comes to the statement `2*gR(3)+3*gR(2)` and immediately calls `gR` on 3. This is what `trace` is reporting on the third line.

Again, executing `gR(3)` Maple finds that it must compute `2*gR(2) + 3*gR(1)`. So it executes `gR` with argument 2. This is the fourth line of output above.

While executing `gR(2)`, Maple must compute `2*gR(1) + 3*gR(0)`. It looks at `gR(1)` and since `gR`'s remember table has an entry for 1, it can look up that value instead of executing the procedure. Maple can then continue on with the rest of the formula and it finds that it can also use the remember table for `gR(0)`. Lines 5 and 6 in green above are reporting the use of the remember table. And now Maple has resolved the formula into the expression `2*(2) + 3*5`, so it computes this and `gR(2)` returns 19. The trace displays the return value 19 in blue, and the green line after that tells us that `gR(2)` is being exited with return value 19. Also, while the trace does not report this, 19 is added to the remember table. This is because we defined `gR` with the `remember` option.

All this time, Maple has been keeping track of where it is in all of these calls to `gR`. It knows that `gR(2)` was called when it was trying to compute `2*gR(2) + 3*gR(1)` within `gR(3)`. When `gR(2)` returns 19, it now knows that the expression is `2*19+3*gR(1)`. Now it needs to know `gR(1)`, but this is in the remember table. So it is able to compute the value 44 and exits `gR(3)`. In the trace above, the reference to the remember table is the green line above the blue 44, the blue 44 indicates that 44 was returned by the procedure call, and the green line after that is the report that `gR` was exited with return value 44.

Once `gR(3)` returns 44, Maple continues "backing out of the recursion." `gR(3)` was called in the computation of `gR(4)`, specifically the formula `2*gR(3)+3*gR(2)`. Since `gR(3)` just returned

44, Maple can turn this expression into `2*44+3*gR(2)`. And now it needs to determine the value of `gR(2)`. Thanks to the remember option, `gR(2)` was added to the remember table. The trace tells us this two lines below the blue 44. Maple replaces `gR(2)` with the remembered value of 19 and computes that `gR(4)` is 145. It returns this value and exits the `gR(4)` computation.

Now we're back up to `gR(5)` and the computation `2*gR(4) + 3*gR(3)`. `gR(4)` was just returned and again Maple can look up the value for `gR(3)`. So `gR(5)` returns 422 and exits. Since this was the "top level", the computation is finished.

Tracing the forgetful procedure
Contrast the above to what happens when we execute the forgetful `gF` on 5.

```
> gF(5);
{--> enter gF, args = 5
{--> enter gF, args = 4
{--> enter gF, args = 3
{--> enter gF, args = 2
value remembered (in gF): gF(1) -> 2
value remembered (in gF): gF(0) -> 5
                                19
<-- exit gF (now in gF) = 19}
value remembered (in gF): gF(1) -> 2
                                44
<-- exit gF (now in gF) = 44}
{--> enter gF, args = 2
value remembered (in gF): gF(1) -> 2
value remembered (in gF): gF(0) -> 5
                                19
<-- exit gF (now in gF) = 19}
                               145
<-- exit gF (now in gF) = 145}
{--> enter gF, args = 3
{--> enter gF, args = 2
value remembered (in gF): gF(1) -> 2
value remembered (in gF): gF(0) -> 5
                                19
<-- exit gF (now in gF) = 19}
value remembered (in gF): gF(1) -> 2
                                44
<-- exit gF (now in gF) = 44}
                               422
<-- exit gF (now at top level) = 422}
                               422                              (5.50)
```

The trace begins in the same way as before. We asked Maple to compute `gF(5)`. For this it needs to execute `gF(4)`, which requires `gF(3)` which uses `gF(2)`.

When Maple gets to `gF(2)`, it again uses the remember table and looks up the values for `gF(1)` and `gF(0)`. It calculates that `gF(2)` is 19, so it returns that value and the trace reports that it exits that application of `gF`.

Maple then tracks up back to the execution of `gF(3)` and the expression `2*gF(2)+3*gF(1)`. It

just returned the value of `gF(2)` and it looks up `gF(1)`. So it returns 44 and exits `gF(3)`.

The first eleven lines of the output from the trace are the same as for `gR`. But now something different happens. Once Maple exists `gF(3)`, it's back to the execution of `gF(4)` and the formula `2*gF(3)+3*gF(2)`. It's just returned 44 for `gF(3)`, so the expression has been partially resolved and stands as `2*44+3*gF(2)`. When `gR` was at this stage, it was able to look up the value of `gF(2)` because it had automatically stored that in the remember table. But `gF` is not automatically recording output in its remember table. So to complete the computation of `gF(4)`, it has to once again call `gF` with argument 2.

After executing `gF(2)` and recomputing 19, `gF(4)` is able to complete and it returns 145.

This brings it back to `gF(5)`. Recall that `gF(5)` needed to compute `2*gF(4)+3*gF(3)`. It now knows `gF(4)`, but it must once again execute `gF(3)`, which requires another execution of `gF(2)`. `gF(2)` is able to look up values and returns 19 for the third time. Then `gF(3)`, armed with the return value from `gF(2)`, can look up `gF(1)` and return 44 for the second time. And that, finally, allows `gF(5)` to perform its computation and return the final value of 422.

As you can imagine, the difference between having the remember table and not is even more extreme for larger input values. With the remember table, once the recursion starts working its way back up the ladder, it remembers all the results from the lower values. Without the **remember** option, the chain of recursive calls has to keep recomputing the results from lower valued inputs.

A Recursive Function with Two Parameters

In Example 13 of Section 5.3 of the text, the sequence $a_{m,\,n}$ is defined. While it may seem natural to model a mathematical sequence like $a_{m,\,n}$ as an expression sequence, it is actually more accurate in Maple to model the sequence as a procedure, like we did with the function above.

We will define a function $A(m, n)$ that models the sequence $a_{m,\,n}$. The basis value is $A(0, 0) = 0$, and the recursion formula is

$$A(m,n) = \begin{cases} A(m-1,n)+1 & \textbf{if } n = 0 \textbf{ and } m > 0 \\ A(m,n-1)+n & \textbf{if } n > 0 \end{cases}.$$

As with the previous example, we will model this using a procedure `A` with the **remember** option.

```
> A := proc(m::nonnegint,n::nonnegint)
    option remember;
    if n=0 and m>0 then
      return A(m-1,n)+1;
    else
      return A(m,n-1)+n;
    end if;
  end proc:
```

```
> A(0,0) := 0;
```

$$A(0,0) := 0 \qquad \textbf{(5.51)}$$

Now we can compute some values of A.

```
> A(3,2);
```

$$6 \qquad \textbf{(5.52)}$$

```
> A(5,3);
```

(5.53)

Displaying values of A

To get a better idea of what the values of A are, it may be useful to display a table of them. In Maple, the easiest way to display a table of values is to create a matrix. In this case, we'll use the **Matrix** command with two arguments. The first argument will be the size of the matrix, say 10. This will produce a square matrix of dimension 10.

The second argument that we'll pass to Matrix will be an "initializer" which tells Maple what values to put in the entries of the matrix. In our case, we want the values of A in the matrix. Luckily, Maple allows the initializer for a matrix to be a procedure, so long as the procedure accepts pairs of positive integers as input. Maple evaluates the procedure at the index (location) to the matrix. That is, in order to determine the entry at location (i, j) in the matrix, Maple executes the procedure with input (i, j).

However, there is one complication. Specifically, the top left entry in a matrix is considered (1,1). It would be more natural to display $A(0, 0)$ as the top left entry.

To do this, we'll use a functional operator as the initializer for the matrix. The functional operator will take a pair **(i,j)**, the position of an entry in the matrix, and subtract one from **i** and **j** before passing the values to **A**. This means that when **Matrix** calls the functional operator on **(1,1)** to obtain the top left entry, it will receive $A(0, 0)$.

```
> Matrix(10,(i,j)->A(i-1,j-1));
```

$$\begin{bmatrix} 0 & 1 & 3 & 6 & 10 & 15 & 21 & 28 & 36 & 45 \\ 1 & 2 & 4 & 7 & 11 & 16 & 22 & 29 & 37 & 46 \\ 2 & 3 & 5 & 8 & 12 & 17 & 23 & 30 & 38 & 47 \\ 3 & 4 & 6 & 9 & 13 & 18 & 24 & 31 & 39 & 48 \\ 4 & 5 & 7 & 10 & 14 & 19 & 25 & 32 & 40 & 49 \\ 5 & 6 & 8 & 11 & 15 & 20 & 26 & 33 & 41 & 50 \\ 6 & 7 & 9 & 12 & 16 & 21 & 27 & 34 & 42 & 51 \\ 7 & 8 & 10 & 13 & 17 & 22 & 28 & 35 & 43 & 52 \\ 8 & 9 & 11 & 14 & 18 & 23 & 29 & 36 & 44 & 53 \\ 9 & 10 & 12 & 15 & 19 & 24 & 30 & 37 & 45 & 54 \end{bmatrix} \quad \textbf{(5.54)}$$

A Recursively Defined Set

In Example 5, the text describes how to recursively define a set. Here, we will consider a slightly more complicated example.

Let S be the subset of the integers defined by

Basis step: $4 \in S$ and $7 \in S$.

Recursive step: if $x \in S$ and $y \in S$, then $x + y \in S$.

(Note that this is the set of all postage that can be formed with 4 cent and 7 cent stamps.)

To model S in Maple, we will define a set that includes the elements called for in the basis step. For the recursive step, we will define a procedure that applies the recursion to the set.

The basis step requires that 4 and 7 are members of S. So we define S to be the set consisting of 4 and 7.

```
> S := {4,7};
```

$$S := \{4, 7\} \quad \textbf{(5.55)}$$

To implement the recursive step, we will create a procedure **recurseS**. This procedure will accept as input the current S and will return the set obtained after applying the recursive rule. For instance, in the first application of **recurseS**, the procedure needs to add 4 + 4, 4 + 7, 7 + 4, and 7 + 7. (The duplication obtained from commutativity will be automatically removed by Maple.)

We will use a pair of for loops of the form **for x in S do** and **for y in S do**. Recall that for sets and lists, the in clause in a for loop causes the index variable to be assigned to each element of the set in turn. By nesting these two loops, we ensure that every possible pair of x and y is added together.

Here is the procedure. (Remember that we cannot modify a parameter, and so the first command in the procedure is to copy S.)

```
> recurseS := proc(S::set)
    local x, y, T;
    T := S;
    for x in S do
      for y in S do
        T := T union {x + y};
      end do;
    end do;
    return T;
  end proc:
```

Now we apply this procedure to S.

```
> S := recurseS(S);
```

$$S := \{4, 7, 8, 11, 14\} \quad \textbf{(5.56)}$$

After a second iteration:

```
> S := recurseS(S);
```

$$S := \{4, 7, 8, 11, 12, 14, 15, 16, 18, 19, 21, 22, 25, 28\} \quad \textbf{(5.57)}$$

A third:

```
> S := recurseS(S);
```

$$S := \{4, 7, 8, 11, 12, 14, 15, 16, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 46, 47, 49, 50, 53, 56\} \quad \textbf{(5.58)}$$

A Set of Strings

As the final example in this section, we will look at how to generate sets of strings over a finite alphabet, as described in Definition 1 of Section 5.3.

The alphabet we will use is {"a", "b", "c", "d"}. We begin by assigning this to a name.

```
> Alphabet := {"a","b","c","d"};
```

$$Alphabet := \{\text{"a"}, \text{"b"}, \text{"c"}, \text{"d"}\} \quad \textbf{(5.59)}$$

According to Definition 1, the basis step is that our set of strings must contain the empty string. In Maple, the empty string is given by **""**. We will use the name **S2** for our set of strings, since S is used above.

```
> S2 := {""};
```

$$S2 := \{""\} \tag{5.60}$$

Note that this is not the same as the empty set. The empty set contains no elements. This set contains one element, which happens to be the empty string.

Like the previous example, we will create a procedure to build the set of strings. The recursive step in the definition tells us that we build the set by combining every element of **S2** with every letter in the alphabet. Again we will use nested loops. We will define the procedure to accept the current version of the set and the alphabet as arguments.

Recall from the Introduction that the **cat** command concatenates strings. Here is the procedure.

```
> buildStrings := proc(S::set,A::set)
    local T, w, x;
    T := S;
    for w in T do
      for x in A do
        T := T union {cat(w,x)};
      end do;
    end do;
    return T;
  end proc:
```

The first application of the recursion adds the alphabet to the set.

```
> S2 := buildStrings(S2,Alphabet);
```

$$S2 := \{"", "a", "b", "c", "d"\} \tag{5.61}$$

The second application adds all the two-character strings.

```
> S2 := buildStrings(S2,Alphabet);
```

$$S2 := \{"", "a", "aa", "ab", "ac", "ad", "b", "ba", "bb", "bc", "bd", "c", "ca", "cb", "cc", "cd", "d", "da", "db", "dc", "dd"\} \tag{5.62}$$

The third application includes the three-character strings.

```
> S2 := buildStrings(S2,Alphabet);
```

$$S2 := \{"", "a", "aa", "aaa", "aab", "aac", "aad", "ab", "aba", "abb", "abc", "abd", "ac", "aca", "acb", "acc", "acd", "ad", "ada", "adb", "adc", "add", "b", "ba", "baa", "bab", "bac", "bad", "bb", "bba", "bbb", "bbc", "bbd", "bc", "bca", "bcb", "bcc", "bcd", "bd", "bda", "bdb", "bdc", "bdd", "c", "ca", "caa", "cab", "cac", "cad", "cb", "cba", "cbb", "cbc", "cbd", "cc", "cca", "ccb", "ccc", "ccd", "cd", "cda", "cdb", "cdc", "cdd", "d", "da", "daa", "dab", "dac", "dad", "db", "dba", "dbb", "dbc", "dbd", "dc", "dca", "dcb", "dcc", "dcd", "dd", "dda", "ddb", "ddc", "ddd"\} \tag{5.63}$$

A general procedure

We can put this process all together in one procedure. Given a set of strings representing the alphabet and a positive integer indicating the number of iterations desired, the following procedure will return the set of strings obtained after the given number of iterations.

```
> AllStrings := proc(A::set,n::posint)
    local S, i;
    S := {""};
    for i from 1 to n do
      S := buildStrings(S,A);
```

```
    end do;
    return S;
  end proc:
```

Below, we apply this procedure to the alphabet consisting of the strings "ab" and "ba" (in discrete mathematics, an alphabet does not have to consist of single letters).

```
> AllStrings({"ab","ba"},4);
```

{"", "ab", "abab", "ababab", "ababba", "abba", "abbaab", "abbaba", "ba", "baab", "baabab", "baabba", "baba", "babaab", "bababa", "abababab", "ababbba", "abbbaab", "abbbaba", "abbaabab", "abbaabba", "abbabaab", "abbababa", "baababab", "baababba", "baabbaab", "baabbaba", "babaabab", "babaabba", "bababaab", "babababa"} **(5.64)**

5.4 Recursive Algorithms

In this section we will use Maple to implement several different recursive algorithms. First, we will look at two different recursive implementations of modular exponentiation and compare their performance. Then we will contrast a recursive approach to computing factorial with an iterative approach. And finally, we will provide an implementation of merge sort.

Modular Exponentiation

Example 4 of Section 5.4 of the text describes two recursive approaches to computing b^n **mod** m. Both of these use the initial condition b^0 **mod** $m = 1$. The first approach is based on the fact that b^n **mod** $m = \left(b \cdot \left(b^{n-1} \text{ mod } m\right) \text{ mod } m\right)$.

The second approach is based on the observation that for even exponents, we can compute via the formula b^n **mod** $m = \left(b^{n/2} \text{ mod } m\right)^2$ **mod** m. And if the exponent is odd, we can use the identity b^n **mod** $m = \left(\left(b^{\lfloor n/2 \rfloor} \text{ mod } m\right)^2 \text{ mod } m\right) \cdot (b \text{ mod } m)$ **mod** m.

Approach 1

First we will implement exponentiation based on the initial condition b^0 **mod** $m = 1$ and the formula b^n **mod** $m = \left(b \cdot \left(b^{n-1} \text{ mod } m\right) \text{ mod } m\right)$. The procedure **power1** will accept three arguments: the base b, the exponent n, and the modulus m.

If the exponent is 0, then regardless of the base or the modulus, the procedure returns 1. If the exponent is greater than 0, then it computes the product of the base with the procedure applied to the same base and modulus but the power decreased by 1.

```
> power1 := proc(b::integer,n::nonnegint,m::posint)
    if n=0 then
      return 1;
    else
      return modp(b * power1(b,n-1,m),m);
    end if;
  end proc:
```

Note that for this procedure we cannot use the remember table to store the basis case as we did in the previous section. This is because our basis case includes all possible values of b and m and the remember table only stores particular values.

Also note that we did not use the **remember** option in this procedure. The reason for this is that each iteration of the procedure only depends on one other call to the procedure. This is in contrast to the **gR** and **gF** procedures from the previous section. Those procedures called themselves twice in each iteration. As a result, those procedures made use of the same value multiple times. Our **power1** procedure will not, unless you execute the procedure on the same input values. When deciding whether or not to use the **remember** option, you must weigh its potential benefit for not repeating computation with the cost of storage requirements.

We can use **power1** to compute 3^6 **mod** 7 and compare the result to Maple's computation of the same expression.

```
> power1(3,6,7);
```

$$1 \qquad \textbf{(5.65)}$$

```
> 3^6 mod 7;
```

$$1 \qquad \textbf{(5.66)}$$

Approach 2

The second approach computes the power based on Algorithm 4 from Section 5.4. For exponent 0, it returns 1, just as before. If the exponent is even, then the algorithm uses the formula $b^n \bmod m = \left(b^{n/2} \bmod m\right)^2 \bmod m$, and for odd powers, it computes the power using the identity $b^n \bmod m = \left(\left(b^{\lfloor n/2 \rfloor} \bmod m\right)^2 \bmod m\right) \cdot (b \bmod m) \bmod m$.

Since there are three possibilities, 0, even, or odd, we'll use the **elif** clause as part of an **if** statement. Refer to the Introduction for details about **elif**. Note that **even** is a Maple type, so we can use the **type** command to form the condition.

Here is the implementation of Algorithm 4.

```
> power2 := proc(b::integer,n::nonnegint,m::posint)
    if n=0 then
      return 1;
    elif type(n,even) then
      return modp(power2(b,n/2,m)^2,m);
    else
      return modp(modp(power2(b,floor(n/2),m)^2,m)*b,m);
    end if;
  end proc:
```

We apply **power2** to 3^6 **mod** 7 as well.

```
> power2(3,6,7);
```

$$1 \qquad \textbf{(5.67)}$$

Comparing performance of the algorithms

Now that we've implemented these two algorithms, let's compare their performance on a variety of input values.

We fix the base 3 and the modulus 7 and consider the exponents from 900 to 1000. We start by forming the list of exponents.

```
> N := [seq(i,i=900..1000)]:
```

To compare the performance, we'll time the execution of each procedure on the exponents from 900 to 1000. We use a for loop to build two lists containing the amount of time required for each

procedure.

```
> times1 := []:
> times2 := []:
> for n from 900 to 1000 do
    st := time();
    power1(3,n,7);
    t := time() - st;
    times1 := [op(times1),t];
    st := time();
    power2(3,n,7);
    t := time() - st;
    times2 := [op(times2),t];
  end do:
```

We have suppressed the output in the statements above, but now **times1** and **times2** contain the running times for **power1** and **power2**, respectively. We can compare their maximum values using the **max** command.

```
> max(times1);
```

$$0.004 \qquad \textbf{(5.68)}$$

```
> max(times2);
```

$$0.001 \qquad \textbf{(5.69)}$$

We see that the worst performance of the **power1** procedure is much worse than the worst performance of **power2**.

Now let's graph the time results. We define **P1** and **P2** to be the plots for **times1** and **times2**.

```
> P1 := plot(N,times1,style=point,symbol=solidcircle,view=[900.
  .1000,0..(.003)],color=red,symbolsize=8,legend="power1");
```

$$P1 := PLOT(\ldots) \qquad \textbf{(5.70)}$$

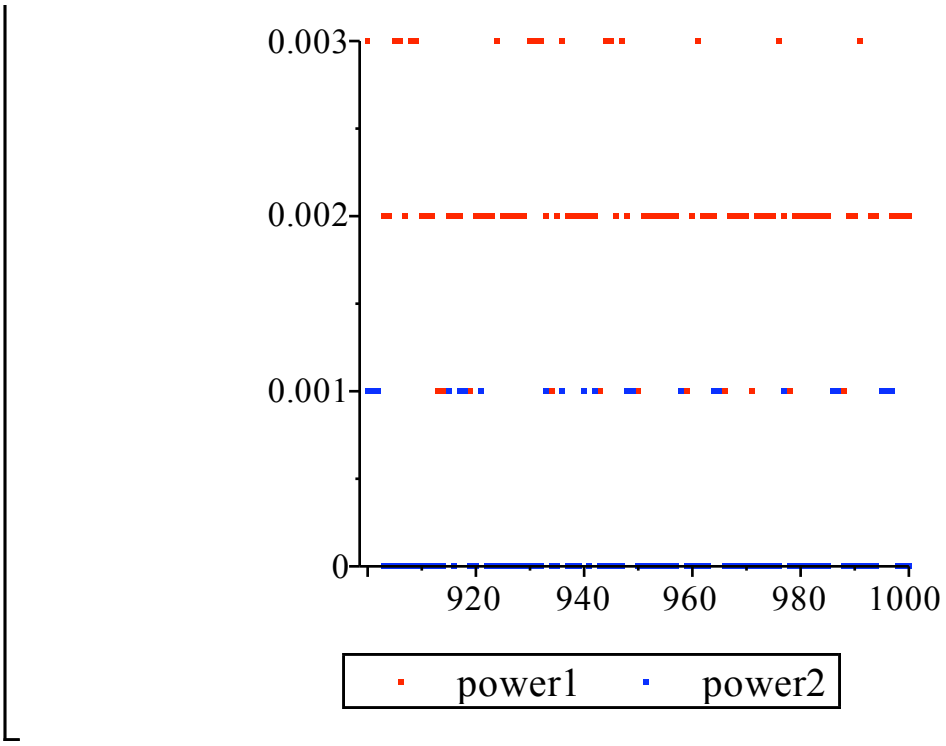
```
> P2 := plot(N,times2,style=point,symbol=solidcircle,view=[900.
  .1000,0..(.003)],color=blue,symbolsize=7,legend="power2");
```

$$P2 := PLOT(\ldots) \qquad \textbf{(5.71)}$$

We made the symbol size in the first plot slightly larger in order to be able to see them when the two plots overlap. The **legend** option defines a legend that will be displayed on the composite graph.

We display the plots using the **display** command in the **plots** package.

```
> plots[display](P1,P2);
```



This gives us a visual comparison of the time complexity of the two algorithms. You can see that the second approach significantly outperforms the first.

Recursion and Iteration

In this subsection we'll compare recursive and iterative procedures for computing factorial.

Recursive factorial

First we will implement Algorithm 1 from Section 5.4, a recursive algorithm for computing $n!$. This procedure accepts a nonnegative integer n as its input. If the input value is 0, the procedure returns 1. Otherwise, it multiplies n by the value of the procedure applied to $n - 1$.

```
> factorialR := proc(n::nonnegint)
    if n=0 then
      return 1;
    else
      return n * factorialR(n-1);
    end if;
  end proc:
```

We test this procedure on 10 and verify that it has the same result as Maple's built-in operator.

```
> factorialR(10);
```

$$3628800 \qquad \textbf{(5.72)}$$

```
> 10!;
```

$$3628800 \qquad \textbf{(5.73)}$$

Iterative factorial

We can implement factorial with an iterative algorithm as well. Our procedure will use a variable `f`, initialized to 1, to store the value of the factorial. It will compute using a for loop with a loop variable looping from 1 to n. Within the loop, `f` will be multiplied by the current value of the loop variable.

```
> factorialI := proc(n::nonnegint)
    local f, i;
    f := 1;
    for i from 1 to n do
      f := f * i;
    end do;
    return f;
  end proc:
```

We again check to make sure the result is correct on $n = 10$.

```
> factorialI(10);
```

$$3628800 \qquad \textbf{(5.74)}$$

Comparing recursion and iteration

Note that these two algorithms require exactly the same number of multiplications. From this point of view, their complexity is the same.

However, let's look at their performance. We consider values of n from 1 to 1500.

```
> M := [seq(i,i=1..1500)]:
```

We use the same approach as we did for **power1** and **power2** above to record and plot the time performance.

```
> timesR := []:
> timesI := []:
> for n from 1 to 1500 do
    st := time();
    factorialR(n);
    t := time() - st;
    timesR := [op(timesR),t];
    st := time();
    factorialI(n);
    t := time() - st;
    timesI := [op(timesI),t];
  end do:
```

We compute the maximum of all the values in the two lists in order to know how large to specify the maximum y-value in the view window for the graph.

```
> max(timesR,timesI);
```

$$0.042 \qquad \textbf{(5.75)}$$

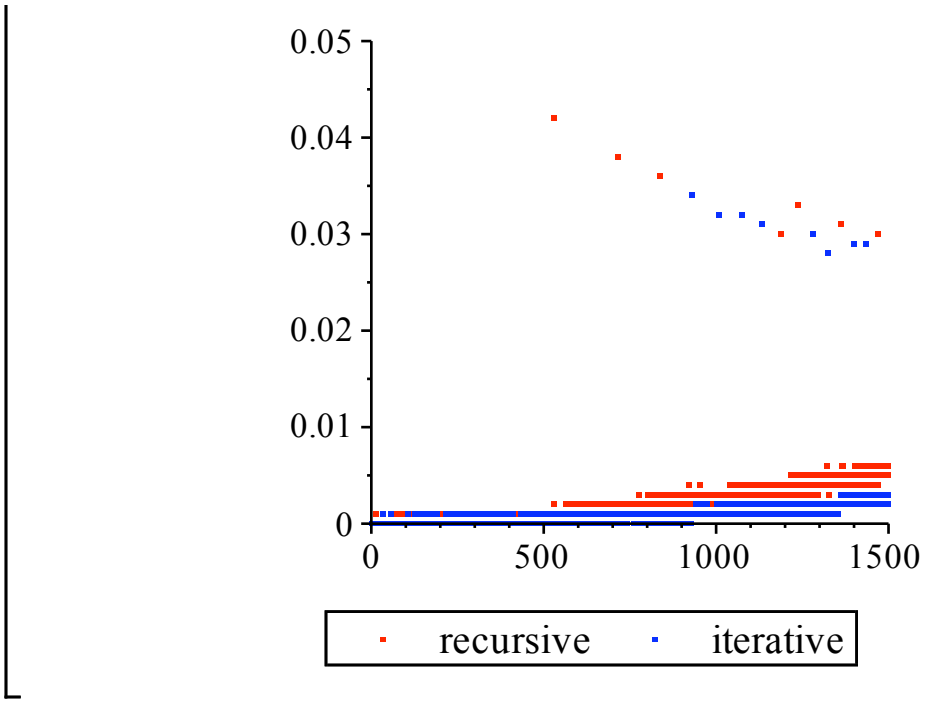
```
> PR := plot(M,timesR,style=point,symbol=solidcircle,view=[0.
  .1500,0..(.05)],color=red,symbolsize=8,legend="recursive");
```

$$PR := PLOT(\ldots) \qquad \textbf{(5.76)}$$

```
> PI := plot(M,timesI,style=point,symbol=solidcircle,view=[0.
  .1500,0..(.05)],color=blue,symbolsize=7,legend="iterative");
```

$$\Pi := PLOT(\ldots) \qquad \textbf{(5.77)}$$

```
> plots[display](PR,PI);
```



First, you might wonder about the outliers which appear to be values of *n* for which one procedure or the other performs particularly poorly. These are essentially "noise" resulting from other processes on your computer. They are also inconsistent; running the commands again will produce slightly different results.

However, it is clear that, despite the occasional peculiar value, the iterative procedure outperforms the recursive one for large values of *n*. This is in spite of the fact that the two algorithms involve the same number of multiplications. Let's consider other sources of potential differences in complexity.

The for loop in the iterative procedure includes an implicit comparison (`i` must be tested against `n` to determine if the loop continues) in contrast to the explicit comparison that the recursive procedure makes. So the two procedures are effectively equal in terms of the number of comparisons used, even though it does not appear so at first glance.

The iterative procedure involves two assignments (the explicit assignment of `f` and the implicit assignment of `i`) absent from the recursive procedure. But two assignments are *much* less costly than a procedure call. Recursive procedure calls require Maple to perform several operations in the background, both in order to execute the recursive call and to keep track of where Maple is in the chain of recursive calls. All of which takes time and memory that the iterative approach does not require.

It is important to keep in mind the cost of recursion. When an iterative algorithm is available, it may be more efficient. On the other hand, recursive algorithms are often more natural to use and can better reveal the mathematical concepts.

Also be aware that Maple, like most programming languages, imposes a limit on the number of recursive calls that can be made simultaneously. When the recursion becomes too "deep", Maple will generate an error. How many calls is too deep may vary based on your computer.

Merge Sort

We conclude this section by implementing merge sort. Note that Maple's `sort` command for

sorting lists is an implementation of merge sort.

The merge sort algorithm is described in Algorithm 9 of the text. The **mergesort** procedure will accept as its input a list of integers. Its main function is to split the single list it receives into two halves, unless the list contains only one element. The return value of **mergesort** is the result of applying **mergesort** to both halves of the list and then recombining them with **merge**.

The following implements Algorithm 9. Note that the **merge** procedure will be written next. For now, Maple simply accepts **merge** as a name.

```
> mergesort := proc(L::list(integer))
    local m, L1, L2;
    if nops(L) > 1 then
      m := floor(nops(L)/2);
      L1 := L[1..m];
      L2 := L[(m+1)..-1];
      return merge(mergesort(L1),mergesort(L2));
    else
      return L;
    end if;
  end proc:
```

Implementing merge

To complete the procedure, we need to define **merge**. The **merge** procedure accepts two lists, which it assumes are sorted, and returns a single list that contains all the elements of both of the inputs and is sorted. The procedure is described in Algorithm 10 of the text.

In implementing **merge**, the first step is to duplicate the input lists. This is because the lists are emptied of their elements as the merge proceeds, and arguments to a procedure cannot be modified. We also initialize the list that will be the result as the empty list.

The main work of **merge** is contained in a while loop conditioned on both lists being nonempty. Within this while loop there are two if statements. The first if statement will implement the instruction to "remove smaller of first elements of L_1 and L_2 from its list; put it at the right end of L" from Algorithm 10. This if statement will test whether the first element of L_1 is smaller than the first element of L_2. If so, then the first element is removed from L_1 and added to L. If not, in the else clause, the first element of L_2 is moved to L.

The second if statement will implement the explicit if statement found in Algorithm 10. The if condition will be that L_1 is empty, and if so the remainder of L_2 will be added to L and L_2 will be emptied. In an elif clause, L_2 will be tested.

Here is the implementation of **merge**.

```
> merge := proc(l1::list(integer),l2::list(integer))
    local L, L1, L2;
    L1 := l1;
    L2 := l2;
    L := [];
    while L1 <> [] and L2 <> [] do
      if L1[1] < L2[1] then
        L := [op(L),L1[1]];
        L1 := L1[2..-1];
```

```
      else
        L := [op(L),L2[1]];
        L2 := L2[2..-1];
      end if;
      if L1 = [] then
        L := [op(L),op(L2)];
        L2 := [];
      elif L2 = [] then
        L := [op(L),op(L1)];
        L1 := [];
      end if;
    end do;
    return L;
  end proc:
```

We apply **mergesort** to a list as follows.

```
> mergesort([7,4,1,5,2,3,6]);
```

$$[1, 2, 3, 4, 5, 6, 7] \qquad \textbf{(5.78)}$$

Tracing merge sort

Applying **trace** to **mergesort** and **merge** will allow us to see the steps that these procedures take. We apply them to a small list so that the output is not too excessive.

```
> trace(mergesort,merge):
> mergesort([3,1,2]);
{--> enter mergesort, args = [3, 1, 2]
```

$$m := 1$$

$$L1 := [3]$$

$$L2 := [1, 2]$$

```
{--> enter mergesort, args = [3]
<-- exit mergesort (now in mergesort) = [3]}
{--> enter mergesort, args = [1, 2]
```

$$m := 1$$

$$L1 := [1]$$

$$L2 := [2]$$

```
{--> enter mergesort, args = [1]
<-- exit mergesort (now in mergesort) = [1]}
{--> enter mergesort, args = [2]
<-- exit mergesort (now in mergesort) = [2]}
{--> enter merge, args = [1], [2]
```

$$L1 := [1]$$

$$L2 := [2]$$

$$L := [\,]$$

$$L := [1]$$

$$L1 := [\,]$$

$$L := [1, 2]$$

$$L2 := [\,]$$

```
<-- exit merge (now in mergesort) = [1, 2]}
<-- exit mergesort (now in mergesort) = [1, 2]}
{--> enter merge, args = [3], [1, 2]
```

$$L1 := [3]$$
$$L2 := [1, 2]$$
$$L := [\,]$$
$$L := [1]$$
$$L2 := [2]$$
$$L := [1, 2]$$
$$L2 := [\,]$$
$$L := [1, 2, 3]$$
$$L1 := [\,]$$

```
<-- exit merge (now in mergesort) = [1, 2, 3]}
<-- exit mergesort (now at top level) = [1, 2, 3]}
```

$$[1, 2, 3] \qquad \textbf{(5.79)}$$

We recommend reading through the result of the trace. Then try it with a larger example, say with 7 elements, to make sure that you understand how merge sort works. To turn tracing off, use the **untrace** command.

```
> untrace(mergesort,merge):
```

5.5 Program Correctness

In this section we will prove the correctness of the merge sort program that we implemented in the last section. This will require that we prove the correctness of **merge** as well as **mergesort**. We begin with **merge**.

merge

For convenience, we will repeat the definition of **merge**. Also, we've added comments to indicate that we've broken the procedure into three segments: S_1, S_2, and S_3.

```
> merge := proc(l1::list(integer),l2::list(integer))
    local L, L1, L2;
    # begin S1
     L1 := l1;
     L2 := l2;
     L := [];
    # end S1
    while L1 <> [] and L2 <> [] do
      # begin S2
       if L1[1] < L2[1] then
         L := [op(L),L1[1]];
         L1 := L1[2..-1];
       else
         L := [op(L),L2[1]];
         L2 := L2[2..-1];
       end if;
      # end S2
      # begin S3
       if L1 = [] then
         L := [op(L),op(L2)];
         L2 := [];
       elif L2 = [] then
         L := [op(L),op(L1)];
         L1 := [];
```

```
        end if;
      # end S3
    end do;
    return L;
  end proc:
```

Let p be the assertion that l_1 and l_2 (the inputs to the procedure) are ordered, nonempty, and disjoint lists of integers. Let q be the proposition that L (the output) is an ordered list and that $L = l_1 \cup l_2$ as sets (that is, the set of integers appearing in L is equal to the union of the set of integers appearing in l_1 and the set of integers in l_2). We claim that $p\{merge\}q$, that is, that **merge** is partially correct with respect to the initial condition p and the final assertion q.

Let q_1 be the proposition that $L_1 = l_1$ and $L_2 = l_2$ and L is the empty list. It is clear that $pS_1(p \wedge q_1)$.

Define the following propositional variables:

- r_1 is the proposition that L_1 is a sublist of l_1; that is, the set of elements appearing in L_1 is a subset of the set of elements appearing in l_1, and the order of the elements in L_1 is the same as their order in l_1. Note that it immediately follows that L_1 is ordered.
- r_2 is the proposition that L_2 is a sublist of l_2 (and thus L_2 is ordered).
- r_3 is the assertion that L is ordered.
- r_4 is the assertion that $L \cup L_1 \cup L_2 = l_1 \cup l_2$ as sets.
- r_5 is the proposition that all members of L are smaller than all members of L_1 and L_2. That is, $(\forall x \in L)\,(\forall y \in L_1 \cup L_2)\,(x < y)$.
- r is the proposition $r_1 \wedge r_2 \wedge r_3 \wedge r_4 \wedge r_5$.

We claim that $p \wedge q_1 \rightarrow r$. Assume $p \wedge q_1$. That is, l_1 and l_2 are ordered, nonempty, and disjoint lists of integers. Also, $L_1 = l_1$ and $L_2 = l_2$ and L is empty. Then r_1 and r_2 hold since a list is a sublist of itself. Proposition r_3 holds since L is empty and thus is ordered vacuously. That r_4 is true follows from substituting l_1 and l_2 and [ ] for L_1, L_2, and L. And r_5 is vacuous since L is empty. From $pS_1(p \wedge q_1)$ and $(p \wedge q_1) \rightarrow r$, we have pS_1r.

Next, we will show that r is a loop invariant for the loop **while** $(L_1 \neq [\,]$ **and** $L_2 \neq [\,])\ S_2; S_3$. Denote by c the condition $L_1 \neq [\,]$ **and** $L_2 \neq [\,]$. We must show that if r and c hold, then r is true after $S_2; S_3$ is executed. First we will show $(r \wedge c)S_2r$ and then that rS_3r.

To show that $(r \wedge c)S_2r$, assume $r \wedge c$. That is, L_1 is a sublist of l_1, L_2 is a sublist of l_2, L is ordered, $L \cup L_1 \cup L_2 = l_1 \cup l_2$, and all members of L are smaller than every member of L_1 and L_2. Also, L_1 and L_2 are nonempty. Then both L_1 and L_2 have first elements. Assume that the if condition of S_2 holds. That is, the first element of L_1 is smaller than the first element of L_2. Then the two commands in the then clause of S_2 are executed: the first element of L_1 is added to the end of L and L_1 has its first element removed.

We need to show that r holds following the execution of the then clause of S_2.

- r_1: the new L_1 is a sublist of the old L_1 since an element was removed meaning that $L_{1\,new} \subset L_{1\,old}$ as sets and the order of the remaining elements was not modified. Since $L_{1\,new}$ is a sublist of $L_{1\,old}$ which was a sublist of l_1, the new L_1 is a sublist of l_1.
- r_2: the new L_2 is identical is the old L_2 and thus remains a sublist of l_2.
- r_3: the old L was ordered, and, since we assume that r_5 held before execution of S_2, every element of L_{old} was smaller than every element of $L_{1\,old} \cup L_{2\,old}$. In particular, the first element of $L_{1\,old}$ was larger than all elements of L_{old}. And so L_{new} is ordered.
- r_4: The smallest element of L_1 was removed from L_1 and added to L. Thus $L_{new} \cup L_{1\,new} = L_{old} \cup L_{1\,old}$ and hence $L \cup L_1 \cup L_2 = l_1 \cup l_2$.
- r_5: We must show that all members of L_{new} are smaller than all members of both $L_{1\,new}$ and $L_{2\,new}$. Let x be an arbitrary element of L_{new}. Either x was a member of L_{old} or x was the first element of $L_{1\,old}$. If x was a member of L_{old} then the assumption that r_5 held before execution of S_2 guarantees that x is smaller than all elements of $L_{1\,new} \cup L_{2\,new}$. On the other hand, assume x was the first element of $L_{1\,old}$. Since $L_{1\,old}$ is a sublist of l_1, it is ordered and thus x was also the smallest element of $L_{1\,old}$ and hence is less than all elements of $L_{1\,new}$. Also, by the assumption that the if condition of S_2 evaluated true, x is smaller than the first (and smallest) element of $L_{2\,old} = L_{2\,new}$ as well. So x is smaller than all members of $L_{1\,new} \cup L_{2\,new}$.

The above shows that if the if condition of S_2 holds, then r holds after executing the then clause. In case the condition fails and the else clause executes, the proof is similar. We conclude that $(r \wedge c) S_2 r$.

Next we will show that $r S_3 r$. Assume r holds. Consider the case that L_1 is empty. Then L_2 is appended on the end of L and L_2 is set to the empty list.

- r_1: L_1 is, since we assume the if condition, empty and thus a sublist of l_1.
- r_2: the second statement in the then clause sets L_2 equal to the empty list, which is a sublist of l_2.
- r_3: we assume that L_{old} is ordered, that $L_{2\,old}$ is a sublist of l_2 and thus is ordered, and that all members of L_{old} are smaller than all members of $L_{2\,old}$. Thus, adding $L_{2\,old}$ to the end of L_{old} to produce L_{new} results in an ordered list.
- r_4: as sets, $L_{new} = L_{old} \cup L_{2\,old}$, so $L_{new} \cup L_{1\,new} \cup L_{2\,new} = \left(L_{old} \cup L_{2\,old}\right) \cup L_{1\,old} \cup \varnothing$. This is equal to $l_1 \cup l_2$ by the assumption that r_4 held before execution.
- r_5: after execution of S_3, both L_1 and L_2 are empty and assertion r_5 is true vacuously.

The case that L_2 is empty is similar. Thus, $r S_3 r$.

We have shown $(r \wedge c) S_2 r$ and $r S_3 r$, and thus $(r \wedge c) S_2 S_3 r$. Hence r is a loop invariant for the while loop. By the inference rule for while loops, we have that $r \{\textbf{while } c\ S_2 S_3\} (\neg c \wedge r)$.

Combining this result with the conclusion that $p S_1 r$ from the paragraph immediately following the definition of r, we have that $p \{merge\} (\neg c \wedge r)$.

We conclude by claiming $\neg c \wedge r \rightarrow q$. Recall that q was the final assertion that L is ordered and $L = l_1 \cup l_2$ as sets. Assume $\neg c \wedge r$. That L is ordered is the claim of r_3. By r_4, we have that, as sets, $L \cup L_1 \cup L_2 = l_1 \cup l_2$. But $\neg c$ implies that L_1 and L_2 are empty. Hence, $L = l_1 \cup l_2$. Thus q holds and we have completed the proof that $p\{merge\}q$ and hence **merge** is partially correct.

mergesort

Now we turn to the **mergesort** procedure. We repeat its definition below.

```
> mergesort := proc(L::list(integer))
    local m, L1, L2;
    if nops(L) > 1 then
      m := floor(nops(L)/2);
      L1 := L[1..m];
      L2 := L[(m+1)..-1];
      return merge(mergesort(L1),mergesort(L2));
    else
      return L;
    end if;
  end proc:
```

Let p be the assertion that L is a nonempty list of distinct integers, and let q be the assertion that the procedure returns a list which has the same elements as L and is ordered. Our claim is that $p\{mergesort\}q$. Since **mergesort** is recursive, our proof will be by strong induction on the length of the list L.

For the basis case, assume that L has only one element. Also assume p. Then the if condition of **mergesort** fails and the program terminates by returning L unmodified. But since L has only one element, it is trivially ordered. Thus, under the basis assumption that L has only one element, $p\{mergesort\}q$.

For the inductive case, we make the inductive assumption that for all $k \leq n$, if a list has length k then **mergesort** returns the list sorted. Assume L has $n + 1$ elements. Also assume p. Under these assumptions, the if condition is satisfied.

The first command in the then clause assigns $m = \left\lfloor \frac{n+1}{2} \right\rfloor$. Note that $m < n + 1$ and, since $n > 1$, $m > 0$. All of the inequalities are strict.

The next two commands assign L_1 to the list consisting of the first m elements in L and L_2 to the remainder. Note that since $0 < m < n + 1$, both of these lists are nonempty with at most n elements.

In the final statement of the if clause, **mergesort** is applied to L_1 and to L_2. Since these two lists both have length at most n, the inductive assumption implies that the results of **mergesort** on L_1 and L_2 are lists with the same elements and ordered. Since **merge** is partially correct, as shown in the previous subsection, the result of **merge** is an ordered list consisting of the elements of its input lists. Hence, the result of **mergesort** is an ordered list consisting of the same elements as L. That is, q holds.

This concludes the inductive step and we conclude $p\{mergesort\}q$ for all lengths of L. Hence, **mergesort** is partially correct.

Solutions to Computer Projects and Computations and Explorations

Computer Projects 2

Generate all well-formed formulae for expressions involving the variables x, y, and z and the operators $\{+, \cdot, /, -\}$ with n or fewer symbols.

Solution: This problem asks us to not only generate well-formed formulae, but to generate *all* such formulae subject to a limitation on the number of symbols.

To begin, we present a recursive definition of the set of well-formed formulae on the symbols.
Basis Step: x, y, and z are well-formed formulae.
Recursive Step: If F and G are well-formed formulae, then so are: $(-F)$, $(F+G)$, $(F-G)$, $(F \cdot G)$, and (F/G).
Note that we will fully parenthesize the well-formed formulae so as to avoid ambiguity, but parentheses will not be considered symbols.

Also note that we will implement the well-formed formulae as strings, not as algebraic expressions. The reason for this is that if we build algebraic expressions, Maple will perform unwanted simplification. For example, $-(-x)$ is a well-formed formula distinct from x, but if we enter $-(-x)$ as a Maple expression, it will be simplified to x.

We will approach this problem in two steps. First, we generate well-formed formulae using a procedure with sufficiently many applications of the recursive step to guarantee that every well-formed formula of length n or less is produced. Second, we will prune the well-formed formulae with greater than n symbols. This will leave us with all well-formed formulae involving at most n symbols.

Generating formulae
The first step is to generate well-formed formulae. For this, we will create a pair of procedures, similar to **AllStrings** and **buildStrings** from Section 5.3 of this manual.

The procedure **buildWFFs** will accept a single argument, a set of well-formed formulae. It will apply the recursive step to the existing set. The procedure first makes a copy of the input set, since arguments cannot be modified. Second, using a for loop over the input set, it applies unary negation. Then, with two for loops over the input set and a third nested for loop over the binary operations, the procedure adds the rest of the well-formed formulae.

```
> buildWFFs := proc(S::set)
    local T, f, g, o;
    T := S;
    for f in S do
      T := T union {cat("(-",f,")")};
    end do;
    for o in ["+","*","/","-"] do
      for f in S do
        for g in S do
          T := T union {cat("(",f,o,g,")")};
        end do;
      end do;
    end do;
    return T;
  end proc:
```

Let's confirm that this works as expected by applying it to the basis set {"x", "y", "z"}.

```
> buildWFFs({"x","y","z"});
```

{"(-x)", "(-y)", "(-z)", "(x*x)", "(x*y)", "(x*z)", "(x+x)", "(x+y)", "(x+z)", "(x-x)", **(5.80)**
"(x-y)", "(x-z)", "(x/x)", "(x/y)", "(x/z)", "(y*x)", "(y*y)", "(y*z)", "(y+x)",
"(y+y)", "(y+z)", "(y-x)", "(y-y)", "(y-z)", "(y/x)", "(y/y)", "(y/z)", "(z*x)",
"(z*y)", "(z*z)", "(z+x)", "(z+y)", "(z+z)", "(z-x)", "(z-y)", "(z-z)", "(z/x)",
"(z/y)", "(z/z)", "x", "y", "z"}

Note that the order is not the order in which the well-formed formulae are added, it is the order Maple imposes on the set.

The other component is the procedure that calls **buildWFFs**. This is nearly identical to **AllStrings**. **AllWFFs** accepts a positive integer **m** representing the number of applications of the recursive step that are to be performed. It initializes the set of formulae to the basis set and applies **buildWFFs** as many times as is called for.

```
> AllWFFs := proc(m::posint)
    local S, i;
    S := {"x","y","z"};
    for i from 1 to m do
      S := buildWFFs(S);
    end do;
    return S;
  end proc:
```

Now the question is: how many applications of the recursive step are needed to be sure that the result contains all well-formed formulae of length at most *n*? Clearly, 0 applications of **buildWFFs** are needed to obtain all formulae consisting of 1 symbol, as this is the basis step. Also, the formulae produced by an application of **buildWFFs** contain at least one symbol more than was present in the previous step (from the unary negation). So after $n - 1$ applications of **buildWFFs**, we are guaranteed to have all well-formed formulae with *n* symbols or fewer.

To illustrate, we will find all well-formed formulae of length at most 3. Apply **AllWFFs** to 2.

```
> AllWFF3 := AllWFFs(2):
```

We suppressed the output since the output would be lengthy.

```
> nops(AllWFF3);
```

7101 **(5.81)**

Here is the set of every 300th formula.

```
> AllWFF3[[seq(300*i+1,i=0..19)]];
```

{"(-(-x))", "((-y)-(x*x))", "((x*x)+(x-x))", "((x*z)*(y*x))", "((x+x)/(y-x))", **(5.82)**
"((x+z)-(z*x))", "((x-y)+(z-x))", "((x/x)*x)", "((x/z)*(x*x))", "((y*x)/(x-x))",
"((y*z)-(y*x))", "((y+y)+(y-x))", "((y-x)*(z*x))", "((y-y)/(z-x))", "((y/x)-x)",
"((y/z)-(x*x))", "((z*y)+(x-x))", "((z+x)*(y*x))", "((z+y)/(y-x))",
"((z-x)-(z*x))"}

Note that these involve up to seven symbols. Since we want the formulae with at most three symbols, we must remove from this set all the formulae with more than three. For this, we will need a procedure that calculates the number of symbols in a formula.

Pruning the set

To count the number of symbols in a formula, we can use the **length** command. If you apply **length** to a string, the command returns the number of characters in the string.

```
> length("abcde");
```

5 **(5.83)**

However, the number of symbols in a well-formed formula is not equal to its length, since parentheses are not considered symbols. So we will need to count the number of parentheses in the formula. To do this, we use the fact that we can use the **for i in S** form of a for loop with **S** a string. Then **i** will be successively assigned to each character. We can then see if **i** is "(" or ")" to count the number of parentheses. Here is the procedure.

```
> countSymbols := proc(WFF::string)
    local count, i;
    count := length(WFF);
    for i in WFF do
      if i="(" or i=")" then
        count := count - 1;
      end if;
    end do;
    return count;
  end proc:
```

For example, the number of symbols in "((z-x)-(z*x))" is:

```
> countSymbols("((z-x)-(z*x))");
```

7 **(5.84)**

In order to prune the set **AllWFF3** so that it contains only the formulae with 3 or fewer symbols, we'll use the **select** command. The **select** command can be used to find the subset of a given set consisting of those elements satisfying a given condition. **select** requires two arguments. The first is a boolean-valued function and the second will be a set. The result is the set of elements of the original set for which the function returned true. (Technically, the second argument can be any expression. Refer to the help page for more information.)

In our case, the boolean-valued function should return true if the well-formed formula has three or fewer symbols. We will test the result of **countSymbols** against 3 using a functional operator.

```
> AllWFF3:=select(f -> evalb(countSymbols(f) <= 3),AllWFF3):
```

This results in a much more manageable number of results.

```
> AllWFF3;
```

{"(-(-x))", "(-(-y))", "(-(-z))", "(-x)", "(-y)", "(-z)", "(x*x)", "(x*y)", "(x*z)", **(5.85)**
"(x+x)", "(x+y)", "(x+z)", "(x-x)", "(x-y)", "(x-z)", "(x/x)", "(x/y)", "(x/z)",
"(y*x)", "(y*y)", "(y*z)", "(y+x)", "(y+y)", "(y+z)", "(y-x)", "(y-y)", "(y-z)",
"(y/x)", "(y/y)", "(y/z)", "(z*x)", "(z*y)", "(z*z)", "(z+x)", "(z+y)", "(z+z)",
"(z-x)", "(z-y)", "(z-z)", "(z/x)", "(z/y)", "(z/z)", "x", "y", "z"}

Computations and Explorations 2

Determine which Fibonacci numbers are divisible by 5, which are divisible by 7, and which are divisible by 11. Prove that your conjectures are correct.

Solution: First we will generate some data to work with. We use the **fibonacci** command in the **combinat** package. This command applied to an integer n returns the nth Fibonacci number.

```
> with(combinat):
> fibList := [seq(fibonacci(i),i=1..50)]:
```

To answer the first part of the question, we want to know which Fibonacci numbers are divisible by 5. That is, we want to determine for which n is the nth Fibonacci divisible by 5. We will use the data above to construct a list consisting of those indices between 1 and 50 for which the corresponding Fibonacci number is divisible by 5.

```
> fib5 := [];
```

$$fib5 := [\,] \tag{5.86}$$

```
> for i from 1 to 50 do
    if fibList[i] mod 5 = 0 then
      fib5 := [op(fib5),i];
    end if;
  end do;
> fib5;
```

$$[5, 10, 15, 20, 25, 30, 35, 40, 45, 50] \tag{5.87}$$

This list suggests that the nth Fibonacci number is divisible by 5 when n is. To obtain more evidence, we'll design a procedure to look for counterexamples to the assertion: F_n is divisible by 5 if and only if n is divisible by 5.

Our procedure will accept a maximum index to check. For each n from 1 to this maximum index, it will use the **fibonacci** command to compute the nth Fibonacci number. If n is divisible by 5 and F_n is not, or if n is not divisible by 5 but F_n is, it will print a message indicating that it found a counterexample. At the conclusion, the procedure will print a message indicating that it has finished computing.

```
> testFib5 := proc(m::posint)
    local n, F;
    uses combinat;
    for n from 1 to m do
      F := fibonacci(n);
      if (n mod 5 = 0) and (F mod 5 <> 0) then
        print("n divisible by 5, Fn not:",n,F);
      elif (n mod 5 <> 0) and (F mod 5 = 0) then
        print("Fn divisible by 5, n not:",n,F);
      end if;
    end do;
    print("finished");
  end proc:
```

Running the procedure up to 500 provides some evidence that our conjecture is true.

```
> testFib5(500);
```

"finished" **(5.88)**

Proving the conjecture, as well as forming and proving conjectures for 7 and 11, is left to the reader.

Exercises

Exercise 1. Use Maple to find and prove formulas for the sum of the first k nth powers of positive integers for $n = 4, 5, 6, 7, 8, 9, 10$.

Exercise 2. For what positive integers k is $n^k - n$ divisible by k for all positive integers n?

Exercise 3. Use Maple to help you find and prove the formulas sought in Exercises 9, 10, and 11 of Section 5.1 of the text. Do not use the **sum** command to form your conjectures.

Exercise 4. Find integers a and d such that d divides $a^{n+1} + (a+1)^{2n-1}$ for all positive integers n. (Exercises 36 and 37 in Section 5.1 indicate that $a = 4$, $d = 21$ and $a = 11$, $d = 133$ are two such pairs.)

Exercise 5. Supplementary Exercises 4 and 5 suggest a more general conjecture. Use Maple's **sum** command to produce evidence for this conjecture.

Exercise 6. Use the **PostageBasis** algorithm from Section 5.2 of this manual to find the smallest n such that every amount of postage of n cents or more can be made from stamps worth 78 cents and \$5.95.

Exercise 7. Write a procedure that accepts two stamp denominations as input and returns the smallest n such that every amount of postage of n cents or more can be made from the given denominations, or returns **FAIL** if it cannot find such an n. Use your procedure to make a conjecture that describes for which pairs of denominations such an n exists and for which there is no such n.

Exercise 8. Write a procedure to recursively build a set from the following definition: basis step: 2 and 3 belong to the set; recursive step: if x and y are members, then xy is a member.

Exercise 9. Use the **time** command to compare the performance of **gR** and **gF** from Section 5.3. Graph the time performance for the two procedures. (Be sure to apply the forget command to **gR** prior to every execution so that the comparison is fair.)

Exercise 10. Write a procedure that accepts two stamp denominations and returns all amounts of postage that can be paid with up to n stamps.

Exercise 11. The solution provided for Computer Projects 2 is inefficient as a means of finding all well-formed formulae involving at most n symbols. This is because each iteration produces formulae with too many symbols. In particular, after n iterations, the resulting set includes formulae of up to $2^{n+1} - 1$ symbols (but not all such formulae). Use the **countSymbols** procedure to modify the approach taken in the solution to Computer Projects 2 so that, with each application of the recursive step, only symbols that include up to n symbols are included.

Exercise 12. Write a procedure to compute the number of partitions of a positive integer (see Exercise 47 in Section 5.3 of the text).

Exercise 13. Write a procedure to compute Ackerman's function (see the prelude to Exercise 48 in Section 5.3).

Exercise 14. Implement Algorithm 3 for computing $gcd(a, b)$ from Section 5.4 of the text.

Exercise 15. Implement Algorithm 5, the recursive linear search algorithm, from Section 5.4 of the text.

Exercise 16. Implement Algorithm 6, the recursive binary search algorithm, from Section 5.4 of the text.

Exercise 17. Compare the performance of your implementations of Algorithm 5 and Algorithm 6 as follows: for a variety of values of n, let L be the list of integers from 1 to n. Randomly choose one hundred integers between 1 and n and measure the average of the times taken for each algorithm to find the randomly chosen integers. Graph n versus the average times. (See Section 4.6 of this manual for a description of the `rand` command for generating random integers.)

Exercise 18. Create three procedures to compute Fibonacci numbers: an iterative procedure, a recursive procedure with the remember option, and a recursive procedure without the remember option. Base your procedures on Algorithms 7 and 8 in Section 5.4 of the text. Create a graph illustrating the time performance of the three procedures.

Exercise 19. Implement quick sort, described in the prelude to Exercise 50 in Section 5.4 of the text. Compare the performance with the merge sort implemented in this manual.

Exercise 20. Implement the algorithm described in Supplementary Exercise 44 for expressing a rational number as a sum of Egyptian fractions.

Exercise 21. Use Maple to study the McCarthy 91 function. (See the prelude to problem 45 in the Supplementary Exercises of Chapter 5.)