2 Basic Structures: Sets, Functions, Sequences, Sums, and Matrices

Introduction

Chapter 2 of the textbook covers mathematical objects that are fundamental to the study of discrete mathematics. We will see how Maple represents these objects and how they correspond to the fundamental data structures used in Maple.

In Sections 1 and 2 we will see that Maple's implementation of a set corresponds naturally to the mathematical concept. We will also see how to extend Maple's capabilities to include fuzzy sets. In Section 3 we will consider three distinct ways in which the concept of function can be represented in Maple and how these three approaches can be used in different circumstances. Section 4 will look at the idea of sequence as it is used in Maple, which is somewhat different from the mathematical meaning of sequence, and how Maple can be used to compute both finite and symbolic summations. In Section 5, we will use Maple to list positive rational numbers in a way that demonstrates the fact that the rationals are enumerable. And in Section 6, we will see how Maple can be used to study matrices.

2.1 Sets

Sets are fundamental to the description of almost all of the discrete objects that we will study. They are also fundamental to Maple. As such, Maple provides extensive support for both their representation and manipulation.

Set Basics

As is standard in mathematics, you can create a set in Maple using the roster method by listing the elements of the set separated by commas and enclosed in braces. The elements of the set can be any of the objects known to Maple. Typical examples are shown here.

```
> {1,2,3};
```

$$\{1, 2, 3\} \quad (2.1)$$

```
> {"a","b","c"};
```

$$\{\text{"a"}, \text{"b"}, \text{"c"}\} \quad (2.2)$$

```
> {{1,2},{1,3},{2,3}};
```

$$\{\{1, 2\}, \{1, 3\}, \{2, 3\}\} \quad (2.3)$$

```
> {};
```

$$\{\} \quad (2.4)$$

```
> {[1,2],[2,5],[3,11]};
```

$$\{[1, 2], [2, 5], [3, 11]\} \quad (2.5)$$

In the first two examples above, the sets contain the numbers 1, 2, and 3, and the characters a, b, and c, respectively. In the third example, the elements of the set are themselves sets. The fourth example is the empty set. And in the final example, the elements of the set are 2-element lists.

Note that Maple's idea of a set corresponds to the mathematical notion. In particular, there is no notion of "multiplicity" for set members, nor is the order of elements relevant. For example, consider the sets defined below.

```
> set1 := {1,2,3,1,2};
```

$$set1 := \{1, 2, 3\} \quad \textbf{(2.6)}$$

Observe that the duplicates in the input above are only included once in the set.

```
> set2 := {2,3,1};
```

$$set2 := \{1, 2, 3\} \quad \textbf{(2.7)}$$

The second example illustrates that order is irrelevant for sets. Maple sorted the input into numerical order in order to improve efficiency in computations (*e.g.*, if we have a large set that we want to search to see if a particular element is in the set or not, having the elements of the set in a particular order can make that search run much more quickly). Maple puts the elements of a set into a canonical order, but it understands that the order is irrelevant from a mathematical perspective. If order is important in a particular context or if repeated elements are allowed, you should use a list instead of a set.

To confirm that two sets A and B are equal, we use the **evalb** (evaluate boolean) command on the proposition $A = B$.

```
> evalb(set1 = set2);
```

$$true \quad \textbf{(2.8)}$$

Selection

A useful consequence of the fact that Maple stores sets in an order is that you can use the selection operator to access individual elements. To select an individual element from a set, you enclose the index of the element in brackets, as below.

```
> set3 := {"a","b","c","d","e","f"};
```

$$set3 := \{\text{"a"}, \text{"b"}, \text{"c"}, \text{"d"}, \text{"e"}, \text{"f"}\} \quad \textbf{(2.9)}$$

```
> set3[2];
```

$$\text{"b"} \quad \textbf{(2.10)}$$

Negative values count from the right, so the second to last entry (according to Maple's imposed order) is accessed as follows.

```
> set3[-2];
```

$$\text{"e"} \quad \textbf{(2.11)}$$

Multiple entries can be accessed by using a range instead of a single value.

```
> set3[3..5];
```

$$\{\text{"c"}, \text{"d"}, \text{"e"}\} \quad \textbf{(2.12)}$$

By putting a list within the selection brackets, it's possible to obtain any subset you wish. Note that the double brackets are required as the outer set is representing the selection operator and the inner set is enclosing the list of indices being accessed.

```
> set3[[1,3,5]];
```

$$\{\text{"a"}, \text{"c"}, \text{"e"}\} \quad \textbf{(2.13)}$$

The seq command

One of the most useful commands for constructing sets or lists is the **seq** (for sequence) command. For example, consider the set consisting of the squares of integers. The **seq** command requires two arguments. The first argument in this case will be the expression **i^2**, which indicates that the elements of the sequence will be the square of the value of the index variable **i**. The second argument will be **i=-10..10** which indicates that the index variable **i** will range from -10 to 10.

```
> seq1 := seq(i^2, i=-10..10);
```

$$seq1 := 100, 81, 64, 49, 36, 25, 16, 9, 4, 1, 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 \quad \textbf{(2.14)}$$

Note that the **seq** command has produced a sequence of values and that the values are listed in the order in which they were generated and with repetition. We make a set from these values by enclosing it in braces.

```
> set4 := {seq1};
```

$$set4 := \{0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100\} \quad (2.15)$$

```
> set5 := {seq(i^4, i=-10..10)};
```

$$set5 := \{0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10000\} \quad (2.16)$$

Membership, subset, and size

Perhaps the most basic question one can ask about a set is whether or not a particular object is or is not a member of a set. In Maple, you do this with the **in** operator. To check that $4 \in set4$ but $5 \notin set4$ we enter the following commands.

```
> evalb(4 in set4);
```

$$true \quad (2.17)$$

```
> evalb(5 in set4);
```

$$false \quad (2.18)$$

Note that we need to use **evalb** so that Maple will compute the truth value. Without it, Maple will just restate the proposition, as below.

```
> 6 in set4;
```

$$6 \in \{0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100\} \quad (2.19)$$

Maple provides the **subset** operator to check whether or not one set is a subset of another. For example,

```
> {1,2} subset {1,2,3};
```

$$true \quad (2.20)$$

```
> {} subset {1,2,3};
```

$$true \quad (2.21)$$

```
> {1,2,5} subset {1,2,3};
```

$$false \quad (2.22)$$

In the previous chapter, we made use of the **nops** command to determine the number of elements in a list. This command also calculates the size of a finite set.

```
> nops(set5);
```

$$11 \quad (2.23)$$

Power Sets

Maple has a built-in command to compute the power set of a finite set. The **powerset** command is part of the **combinat** combinatorics package and accepts a set as an argument and returns the power set of the given set.

Commands that are part of Maple packages can be used in one of two ways. The long form consists of the name of the package followed by the name of the command in brackets and then the arguments in parentheses: **package[command](arguments)**.

```
> combinat[powerset]({1,2,3});
```

$$\{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\} \quad (2.24)$$

In order to use the short form of the calling sequence, that is, to be able to omit the name of the

package, you must first use the **with** command to load the package.

```
> with(combinat);
```

[Chi, *bell*, binomial, *cartprod*, *character*, *choose*, *composition*, *conjpart*, *decodepart*, *encodepart*, *eulerian1*, *eulerian2*, *fibonacci*, *firstpart*, *graycode*, *inttovec*, *lastpart*, *multinomial*, *nextpart*, *numbcomb*, *numbcomp*, *numbpart*, *numbperm*, *partition*, *permute*, *powerset*, *prevpart*, *randcomb*, *randpart*, *randperm*, *setpartition*, stirling1, stirling2, *subsets*, *vectoint*] **(2.25)**

The result of this command is to list the commands that have been made available. Typically, you'll end **with** statements with colons to suppress this output. Note that you can choose to load only those commands from a package that you will actually be using by following the name of the package with the names of the commands separated by commas.

```
> with(combinat,powerset,subsets,cartprod);
```

[*powerset*, *subsets*, *cartprod*] **(2.26)**

However you choose to load the command, this makes it possible to call the command without the name of the package.

```
> powerset({"a","b"});
```

{{ }, {"a"}, {"b"}, {"a", "b"}} **(2.27)**

The subsets command

The textbook mentions that the size of the power set of a set is 2^n where n is the size of the original set. For even reasonably sized sets, their power set can be very large and can easily tax your computer's memory. For this reason, Maple provides a second command for computing with power sets called **subsets**, which is also in the **combinat** package. The result of applying **subsets** is not a list of sets. Instead, the **subsets** command returns a table. Tables will be discussed in detail in Section 3 of this chapter; for now, it is enough to see how to use the **subsets** command.

Consider the set consisting of the first ten positive integers.

```
> firstTen := {seq(1..10)};
```

firstTen := {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} **(2.28)**

(Note the alternate form of the **seq** command.)

Apply the **subsets** command and store the result as **subsetTen**.

```
> subsetTen := subsets(firstTen):
```

It is typical to suppress the output as we have done here. Those readers who are curious can issue the command with the output displayed, but do not expect to see any subsets listed in the output from **subsets**.

As we mentioned, the result of the **subsets** command is a table containing two objects. One of these objects is a procedure, which is executed with the syntax **subsetTen[nextvalue]();** . The first time this procedure is called, it will return the empty set.

```
> subsetTen[nextvalue]();
```

{ } **(2.29)**

Each subsequent time it is called, it returns the "next" subset of the given set.

```
> subsetTen[nextvalue]();
```

{1} **(2.30)**

```
> subsetTen[nextvalue]();
```

$$\{2\} \quad \textbf{(2.31)}$$

```
> subsetTen[nextvalue]();
```

$$\{3\} \quad \textbf{(2.32)}$$

```
> subsetTen[nextvalue]();
```

$$\{4\} \quad \textbf{(2.33)}$$

The other object in the **subsetTen** table is a boolean value **finished**. This value is accessed by

```
> subsetTen[finished];
```

$$\mathit{false} \quad \textbf{(2.34)}$$

Once the **subsetTen[nextvalue]** procedure has returned the "last" subset, which will always be the original set itself, this boolean is set to false. This provides a way to control a while loop. By setting the condition in the while loop to **not subsetTen[finished]**, the loop continues until all of the subsets have been considered.

Example using the subsets command

As an example of a practical use of this command, let's search for the subsets of the first five positive integers which have their own cardinality as a member, *i.e.*, those sets S such that $|S| \in S$. We'll do this by considering each subset in turn and checking whether its size, obtained using the **nops** command, is **in** the set.

Here's the procedure that will list all subsets of the first five positive integers whose cardinalities are members of themselves..

```
> selfSize := proc()
    local onetofive, pSet, S, n;
    onetofive := {seq(1..5)};
    pSet := combinat[subsets](onetofive);
    while not pSet[finished] do
      S := pSet[nextvalue]();
      if nops(S) in S then
        print(S);
      end if;
    end do;
  end proc:
```

After declaring local variables, we form the set of the integers from 1 to 5. Then we apply the **subsets** command to form **pSet**. Remember that this is not actually the power set, it is the table described above. Then we begin a while loop, which continues until **pSet[finished]** is true. Remember that **pSet[finished]** is false until the **pSet[nextvalue]** procedure produces the final subset. Inside the while loop, we print those sets which have their own size as a member.

Now let's execute the procedure.

```
> selfSize();
```

$$\{1\}$$
$$\{1, 2\}$$
$$\{2, 3\}$$
$$\{2, 4\}$$
$$\{2, 5\}$$
$$\{1, 2, 3\}$$

$$\{1, 3, 4\}$$
$$\{1, 3, 5\}$$
$$\{2, 3, 4\}$$
$$\{2, 3, 5\}$$
$$\{3, 4, 5\}$$
$$\{1, 2, 3, 4\}$$
$$\{1, 2, 4, 5\}$$
$$\{1, 3, 4, 5\}$$
$$\{2, 3, 4, 5\}$$
$$\{1, 2, 3, 4, 5\} \quad \textbf{(2.35)}$$

It's worth reiterating the purpose of the **subsets** command in contrast with the **powerset** command. For sets of any significant size, calculating the power set can be very taxing both on a computer's memory and with regards to time. In an example like **selfSize**, the **subsets** command avoids the need to store the entire powerset by generating and testing one subset at a time. Additionally, suppose that instead of listing all of the subsets with a given property, we only wanted to find an example of a set with that property, for instance, to find a counterexample. In those circumstances, we could return the example as soon as it was found and save the time it would have taken **powerset** to have calculated all of the subsets.

Cartesian Product

The **combinat** package also has a command for calculating the Cartesian product of sets called **cartprod**. This command is very similar to the **subsets** command except that the **nextvalue** procedure returns a list representing the "next" element in the Cartesian product of the given sets.

As a first example, recall that Example 17 from Section 2.1 of the text computes the Cartesian product of $\{1, 2\}$ and $\{a, b, c\}$ to be $\{(1, a), (1, b), (1, c), (2, a), (2, b), (2, c)\}$. To compute this product in Maple, we use the **cartprod** function. It accepts only one argument: a list whose members are the sets whose product is desired. We'll store the result as **Ex17** and, as before, we suppress the output for the result.

```
> Ex17 := cartprod([{1,2},{"a","b","c"}]):
```

We can now display the elements one at a time using the same kind of loop as we used in the **selfSize** procedure. We create a while loop that continues as long as **Ex17[finished]** is false and prints the result of **Ex17[nextvalue]()**.

```
> while not Ex17[finished] do
    print(Ex17[nextvalue]());
  end do;
```

$$[1, \text{"a"}]$$
$$[1, \text{"b"}]$$
$$[1, \text{"c"}]$$
$$[2, \text{"a"}]$$
$$[2, \text{"b"}]$$
$$[2, \text{"c"}] \quad \textbf{(2.36)}$$

Maple displays the entries in the Cartesian product as two-element lists.

The argument to **cartprod** can be a list of any number of sets. To compute the members of the Cartesian product of three sets, we include three sets in the list. Also note that this command accepts both sets and lists as the objects to be multiplied. Below, we expand our previous example and compute $\{1, 2\} \times \{a, b, c\} \times \{\pi, e\}$. We give the last set as a list instead of a set. (Note that we obtain π with the Maple constant **Pi**, and e is obtained by applying the exponential function **exp** with exponent 1.)

```
> cartesian3 := cartprod([{1,2},{"a","b","c"},[Pi,exp(1)]]):
> while not cartesian3[finished] do
    print(cartesian3[nextvalue]());
  end do;
```

$$[1, \text{"a"}, \pi]$$
$$[1, \text{"a"}, e]$$
$$[1, \text{"b"}, \pi]$$
$$[1, \text{"b"}, e]$$
$$[1, \text{"c"}, \pi]$$
$$[1, \text{"c"}, e]$$
$$[2, \text{"a"}, \pi]$$
$$[2, \text{"a"}, e]$$
$$[2, \text{"b"}, \pi]$$
$$[2, \text{"b"}, e]$$
$$[2, \text{"c"}, \pi]$$
$$[2, \text{"c"}, e] \quad \textbf{(2.37)}$$

Maple does not include a command analogous to **powerset** for computing the Cartesian product all at once as a single set of elements. The task of creating such a command is left to the reader.

2.2 Set Operations

In this section we will examine the commands Maple provides for computing set operations. Then we will use these commands and the concept of membership tables to see how Maple can be used to prove set identities. Finally, we see how we can use Maple to represent and manipulate fuzzy sets.

Basic Operations

Maple provides fairly intuitive commands related to the basic set operations of union, intersection, and set difference. The commands are named **union**, **intersection**, and **minus** and, like the logical connectives from the previous chapter, are infix operators. For example, consider the following sets.

```
> primes := {2,3,5,7,11,13};
```

$$primes := \{2, 3, 5, 7, 11, 13\} \quad \textbf{(2.38)}$$

```
> odds := {1,3,5,7,9,11,13};
```

$$odds := \{1, 3, 5, 7, 9, 11, 13\} \quad \textbf{(2.39)}$$

We compute their union and intersection as follows:

```
> primes union odds;
```

$$\{1, 2, 3, 5, 7, 9, 11, 13\} \quad \textbf{(2.40)}$$

```
> primes intersect odds;
```
$$\{3, 5, 7, 11, 13\} \quad \textbf{(2.41)}$$

The set difference is obtained by use of the **minus** operator. The following examples illustrate the fact that, unlike, union and intersection, set difference is not symmetric, *i.e.*, $A - B$ is generally not the same as $B - A$.

```
> primes minus odds;
```
$$\{2\} \quad \textbf{(2.42)}$$

```
> odds minus primes;
```
$$\{1, 9\} \quad \textbf{(2.43)}$$

Maple does *not* provide a complement command. Such a command would be ambiguous with regards to the universe that should be applied. Instead, you must compute complements with the minus operator. For example, the complement of the **primes** set in the universe consisting of the positive integers 1 to 13 is computed as follows.

```
> universe := {seq(1..13)};
```
$$universe := \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13\} \quad \textbf{(2.44)}$$

```
> universe minus primes;
```
$$\{1, 4, 6, 8, 9, 10, 12\} \quad \textbf{(2.45)}$$

The **union** operator is often used to build sets within procedures. As an example, consider the following procedure that creates the set of the squares of the first ten positive integers.

```
> tenSquares := proc()
    local S, i;
    S := {};
    for i from 1 to 10 do
      S := S union {i^2};
    end do;
    return S;
  end proc:
> squares := tenSquares();
```
$$squares := \{1, 4, 9, 16, 25, 36, 49, 64, 81, 100\} \quad \textbf{(2.46)}$$

After the declaration of local variables, we initialize **S**, the set being built, to the empty set. Inside the for loop, we produce the square of the loop index and add it to the set **S** by setting **S** equal to its union with the singleton containing the square of the index variable. While this is a very simple example, it illustrates a common technique. (Note: the set of squares could also have been created with the **seq** command.)

Finally, the **union** and **intersect** operators can also be used as commands and these commands can take any number of sets as arguments. For example, to compute the union of three sets, you can use either of the two methods below.

```
> primes union odds union squares;
```
$$\{1, 2, 3, 4, 5, 7, 9, 11, 13, 16, 25, 36, 49, 64, 81, 100\} \quad \textbf{(2.47)}$$

```
> `union`(primes,odds,squares);
```
$$\{1, 2, 3, 4, 5, 7, 9, 11, 13, 16, 25, 36, 49, 64, 81, 100\} \quad \textbf{(2.48)}$$

Note that the single left quotes are required in the second option. Generally, single left quotes are used to tell Maple that the string of characters they enclose is a name. In the first of the statements above, the **union** keyword is used to invoke the operator. In the second, a procedure whose name

is **union** is executed. The single left quotes are required in order to use a keyword as a name.

Set Identities and Membership Tables

The textbook discusses how membership tables can be used to prove set identities. We'll use the idea of membership tables to have Maple prove set identities.

A membership table is very similar to a truth table. In a membership table, each row corresponds to a possible element in the universe. We use 1 and 0 to indicate that the element corresponding to that row is or is not in the set.

An Illustration of the approach with an example

Let's look at a specific example in detail in order to get an idea of how we can use Maple to automate the construction of membership tables. Consider the De Morgan's law $\overline{A \cup B} = \overline{A} \cap \overline{B}$. We begin the table by considering all possible combinations of 1s and 0s for A and B and add columns for the two sides of the identity. (Ordinarily, when doing this by hand you would add columns for the intermediary steps as well.)

row number	A	B	$\overline{A \cup B}$	$\overline{A} \cap \overline{B}$
1	1	1		
2	1	0		
3	0	1		
4	0	0		

We can determine the values for the last two columns as follows. Let the universe be the set consisting of the row numbers $\{1, 2, 3, 4\}$. Now form sets A and B as follows: a number in the universe of row numbers is in A if there is a 1 in A's column in that row. Thus $A = \{1, 2\}$ because rows 1 and 2 have 1s in A's column. Likewise, we form $B = \{1, 3\}$.

```
> rows := {1,2,3,4};
```

$$rows := \{1, 2, 3, 4\} \quad \textbf{(2.49)}$$

```
> setA := {1,2};
```

$$setA := \{1, 2\} \quad \textbf{(2.50)}$$

```
> setB := {1,3};
```

$$setB := \{1, 3\} \quad \textbf{(2.51)}$$

Next, compute both sides of the identity $\overline{A \cup B} = \overline{A} \cap \overline{B}$. Remember that we must use set differences to obtain the complements.

```
> rows minus (setA union setB);
```

$$\{4\} \quad \textbf{(2.52)}$$

This indicates that row 4 is the only row with a 1 in the column for $\overline{A \cup B}$.

```
> (rows minus setA) intersect (rows minus setB);
```

$$\{4\} \quad \textbf{(2.53)}$$

This tells us that row 4 is also the only row with a 1 in the column for $\overline{A} \cap \overline{B}$. Since the two sets are equal, the two columns must be identical.

The above indicates the approach that we will be using. First, compute the initial entries in the rows of the membership table; each row corresponds to a different assignment of 1s and 0s. Second, construct sets whose entries are the row numbers corresponding to 1s in the table. And finally, compute both sides of the identity. If the resulting sets are equal, then we have confirmed the identity.

Revising the GetVars procedure

Much of what we do here will be very similar to how we created the **AreEquivalent** procedure in Section 1.3 of this manual. First, let's create expressions representing the two sides of the identity from Example 14 of the text. We'll use the name **U** for the universe.

```
> Ex14L := U minus (A union (B intersect C));
```

$$Ex14L := U \setminus (A \cup B \cap C) \tag{2.54}$$

```
> Ex14R:=((U minus C) union (U minus B)) intersect (U minus A);
```

$$Ex14R := (U \setminus A) \cap ((U \setminus B) \cup (U \setminus C)) \tag{2.55}$$

```
> Ex14 := Ex14L = Ex14R;
```

$$Ex14 := U \setminus (A \cup B \cap C) = (U \setminus A) \cap ((U \setminus B) \cup (U \setminus C)) \tag{2.56}$$

Now we revive the **GetVars** procedure from Section 1.3.

```
> GetVars := proc(exp)
    local L, i, j;
    L := [exp];
    i := 1;
    while i <= nops(L) do
      if type(L[i],name) then
        i := i + 1;
      else
        L := subsop(i=op(L[i]),L);
      end if;
    end do;
    L := {op(L)} minus {U};
    return [op(L)];
  end proc:
> Ex14Vars := GetVars(Ex14);
```

$$Ex14Vars := [A, B, C] \tag{2.57}$$

Note that this is identical to the procedure we created in Section 1.3 with one small change. In the next to last line we convert **L** into a set and remove the name **U** from it. In these procedures, we will always consider **U** to be the name of the universe. The last line turns the set of variables back into a list. This isn't necessary, strictly speaking, but it is more natural to consider the variables stored in a list, and therefore with order.

Producing the rows of the table

In Section 1.3, we created a procedure called **nextTA**. This procedure was responsible for producing the truth value assignments for the variables. In other words, it produced the rows of the truth table. Look again at the membership table above. Observe that the rows correspond to the members of the Cartesian product

$$\{0, 1\} \times \{0, 1\} = \{(0, 0), (0, 1), (1, 0), (1, 1)\}.$$

In fact, the definition of the Cartesian product is exactly suited to what we need. The rows of the table are all the possible choices of 0s and 1s for the variables. The Cartesian product of $\{0, 1\}$ with itself is the collection of all possible tuples with each entry in the tuple equal to 0 or to 1.

We can use the following code to produce the Cartesian product with 3 variables.

```
> CartesianMembership := cartprod([seq({0,1},i=1..3)]):
```

Note the use of **seq** to create three copies of the set {0, 1}.

```
> while not CartesianMembership[finished] do
    CartesianMembership[nextvalue]();
  end do;
```

$$[0, 0, 0]$$
$$[0, 0, 1]$$
$$[0, 1, 0]$$
$$[0, 1, 1]$$
$$[1, 0, 0]$$
$$[1, 0, 1]$$
$$[1, 1, 0]$$
$$[1, 1, 1] \quad \textbf{(2.58)}$$

You can see that the results are identical to the first three columns of Table 2 of Section 2.2 in the textbook.

Building sets to correspond to the table rows

We need to build sets whose entries are determined by the rows of the membership table (*i.e.*, by the elements of a Cartesian product of {0, 1} as above). The sets, corresponding to what we called **setA** and **setB** in the example at the start of this subsection, will be stored in a list. That is, we'll create a list of sets. These sets are identified with the variables in the identity to be checked as follows: the set in position *i* in the list of sets corresponds to the variable in position *i* in the list that results from **GetVars**.

Begin by initializing a list of the right size (the number of variables) whose entries are the empty set. We use the **seq** command again to create multiple copies.

```
> MTableSets := [seq({},i=1..3)];
```

$$MTableSets := [\{\}, \{\}, \{\}] \quad \textbf{(2.59)}$$

Note that we can access and modify the lists as usual. For instance, to add 5 to the second set:

```
> MTableSets[2] := MTableSets[2] union {5};
```

$$MTableSets_2 := \{5\} \quad \textbf{(2.60)}$$

```
> MTableSets;
```

$$[\{\}, \{5\}, \{\}] \quad \textbf{(2.61)}$$

Let's re-initialize this so we can use it below.

```
> MTableSets := [seq({},i=1..3)];
```

$$MTableSets := [\{\}, \{\}, \{\}] \quad \textbf{(2.62)}$$

We re-execute the **cartprod** command from above in order to reset it. We also need an index variable that we initialize to 0.

```
> CartesianMembership := cartprod([seq({0,1},i=1..3)]):
> MTrownum := 0;
```

$$MTrownum := 0 \quad \textbf{(2.63)}$$

Now we use the standard Cartesian product while loop, but instead of just calculating and displaying the 3-tuple, we will use an inner loop to consider each entry (*i.e.*, variable) in turn. If the

value is 1, we add the current value of **MTrownum** to the corresponding set. In this example, we'll have the loop print out the index, the tuple from the Cartesian product, and the current state of **MTableSets**.

```
> while not CartesianMembership[finished] do
    MTrownum := MTrownum + 1:
    currentTuple := CartesianMembership[nextvalue]():
    for varI from 1 to 3 do
      if currentTuple[varI] = 1 then
        MTableSets[varI] := MTableSets[varI] union {MTrownum}:
      end if:
    end do:
    print(MTrownum,currentTuple,MTableSets);
  end do:
```

$$1, [0, 0, 0], [\{\}, \{\}, \{\}]$$
$$2, [0, 0, 1], [\{\}, \{\}, \{2\}]$$
$$3, [0, 1, 0], [\{\}, \{3\}, \{2\}]$$
$$4, [0, 1, 1], [\{\}, \{3, 4\}, \{2, 4\}]$$
$$5, [1, 0, 0], [\{5\}, \{3, 4\}, \{2, 4\}]$$
$$6, [1, 0, 1], [\{5, 6\}, \{3, 4\}, \{2, 4, 6\}]$$
$$7, [1, 1, 0], [\{5, 6, 7\}, \{3, 4, 7\}, \{2, 4, 6\}]$$
$$8, [1, 1, 1], [\{5, 6, 7, 8\}, \{3, 4, 7, 8\}, \{2, 4, 6, 8\}] \quad \textbf{(2.64)}$$

We also need the universe represented.

```
> Ex14U := {seq(i,i=1..MTrownum)};
```

$$Ex14U := \{1, 2, 3, 4, 5, 6, 7, 8\} \quad \textbf{(2.65)}$$

Evaluating the identity for each row

Once the list of sets is built up, all that remains is to evaluate the identity with these sets in place of the names. We do this with the **eval** and **zip** commands (recall that we previously used this technique in Section 1.3 of this manual).

First we use **zip** to create equations that identify the variables with the corresponding sets.

```
> Ex14eqns := zip((a,b) -> a=b, Ex14Vars, MTableSets);
```

$$Ex14eqns := [A = \{5, 6, 7, 8\}, B = \{3, 4, 7, 8\}, C = \{2, 4, 6, 8\}] \quad \textbf{(2.66)}$$

We add the equation **U=Ex14U**.

```
> Ex14eqns := [op(Ex14eqns),U=Ex14U];
```

$$Ex14eqns := [A = \{5, 6, 7, 8\}, B = \{3, 4, 7, 8\}, C = \{2, 4, 6, 8\}, U = \{1, 2, 3, 4, 5, 6, 7, 8\}] \quad \textbf{(2.67)}$$

And then we apply **eval** to perform the substitution and apply **evalb** to obtain a truth value.

```
> eval(Ex14,Ex14eqns);
```

$$\{1, 2, 3\} = \{1, 2, 3\} \quad \textbf{(2.68)}$$

```
> evalb((2.68));
```

$$true \quad \textbf{(2.69)}$$

The procedure

Finally, we combine it all into a single procedure.

```
> MemberTable := proc(identity)
    local vars, numvars, cartesian, setList, rowNum, curTuple,
  vI, Universe, eqns, U;
    vars := GetVars(identity);
    numvars := nops(vars);
    cartesian := combinat[cartprod]([seq({0,1},i=1..numvars)]);
    setList := [seq({},i=1..numvars)];
    rowNum := 0;
    while not cartesian[finished] do
      rowNum := rowNum + 1;
      curTuple := cartesian[nextvalue]();
      for vI from 1 to numvars do
        if curTuple[vI] = 1 then
           setList[vI] := setList[vI] union {rowNum};
        end if;
      end do;
    end do;
    Universe := {seq(i,i=1..rowNum)};
    eqns := zip((a,b) -> a=b, vars, setList);
    eqns := [op(eqns),U=Universe];
    return evalb(eval(identity,eqns));
  end proc:
```

We can now prove: $(A - B) - C = (A - C) - (B - C)$.

```
> MemberTable((A minus B) minus C = (A minus C) minus (B minus
  C));
```

$$true \qquad \textbf{(2.70)}$$

However, $\overline{A \cup B} \neq \overline{A} \cup \overline{B}$.

```
> MemberTable(U minus (A union B) = (U minus A) union (U minus
  B));
```

$$false \qquad \textbf{(2.71)}$$

Computer Representation of Fuzzy Sets

The textbook describes a way to represent sets as bit strings in order to efficiently store and compute with them. Here, we will explore this idea further in order to see how we can represent fuzzy sets in Maple. Fuzzy sets are described in the preamble to Exercise 63 in Section 2.2.

Two representations of fuzzy sets

In a fuzzy set, every element has an associated degree of membership, which is a real number between 0 and 1. We'll represent fuzzy sets in two different ways.

The first way we can represent a fuzzy set in Maple is to combine the element with the degree of membership as a two-element list. For example, if the elements of our fuzzy set are the letters "a", "b", and "e", where "a" has degree of membership 0.3, "b" has degree 0.7, and "e" has degree 0.1, then we would represent the set as:

```
> fuzzyR := {["a",0.3],["b",0.7],["e",0.1]};
```

$$fuzzyR := \{["a", 0.3], ["b", 0.7], ["e", 0.1]\} \qquad \textbf{(2.72)}$$

We'll refer to this as the "roster representation."

The second approach is to use a "fuzzy-bit string" in essentially the same way as described in the text. First we need to specify the universe and impose an order on it. Let's say the universe consists of the letters "a" through "g" ordered alphabetically. We represent the universe in Maple as

a list so that the order we impose is preserved.

```
> fuzzyU := ["a","b","c","d","e","f","g"];
```

$$fuzzyU := [\text{"a"}, \text{"b"}, \text{"c"}, \text{"d"}, \text{"e"}, \text{"f"}, \text{"g"}] \quad \textbf{(2.73)}$$

Then the fuzzy-bit string for the set **fuzzyR** will be the list of the degrees of membership of each element of the universe with 0 indicating non-membership.

```
> fuzzyBitS := [0.3,0.7,0,0,0.1,0,0];
```

$$fuzzyBitS := [0.3, 0.7, 0, 0, 0.1, 0, 0] \quad \textbf{(2.74)}$$

Converting from bit string to roster representation

Converting from a fuzzy-bit string to the roster representation is fairly straightforward. Use a for loop with index running from 1 to the number of elements in the universe. For each index, if the entry in the fuzzy-bit string is non-zero, then we add to the roster the pair consisting of the element from the universe and the degree of membership.

```
> BitToRoster := proc(bitstring,universe)
    local S, i;
    S := {};
    for i from 1 to nops(universe) do
      if bitstring[i] <> 0 then
         S := S union {[universe[i],bitstring[i]]};
      end if;
    end do;
    return S;
  end proc:
> BitToRoster(fuzzyBitS,fuzzyU);
```

$$\{[\text{"a"}, 0.3], [\text{"b"}, 0.7], [\text{"e"}, 0.1]\} \quad \textbf{(2.75)}$$

Converting from roster to bit string representation

In the other direction, we'll initialize a bit string to the 0-string. Then we consider each member of the roster representation in turn, using the **for object in Set do** form of a for loop. For every member of the set, we will need to determine the position of the set member in the universe in order to change the correct bit in the fuzzy-bit string.

To do this, we'll make use of the **member** command. Like the **in** operator, **member** will return true or false depending on whether or not the first argument is a member of the list (or set) given as the second argument. For example,

```
> member(3,{1,2,3,4,5});
```

$$true \quad \textbf{(2.76)}$$

```
> member(7,{1,2,3,4,5});
```

$$false \quad \textbf{(2.77)}$$

member also accepts a third, optional, argument which must be an unevaluated name. If the object is in fact a member of the set or list, then the position of the object is assigned to the name. (If the object is repeated, the location of the first occurrence is stored in the name.) We surround the name with right single quotes to prevent evaluation. Without the single quotes, the name could evaluate to a value that was previously assigned and Maple would not be able to assign to the name.

```
> member("d",["a","f","s","g","d","q"],'pos');
```

$$true \quad \textbf{(2.78)}$$

```
> pos;
```

(2.79)

```
> 5                                                                    (2.79)
```

Now we can write the `RosterToBit` procedure.

```
> RosterToBit := proc(roster,universe)
    local B, i, e, pos;
    B := [seq(0,i=1..nops(universe))];
    for e in roster do
      if member(e[1],universe,'pos') then
        B[pos] := e[2];
      else
        error "Roster contained a member not in the universe.";
      end if;
    end do;
    return B;
  end proc:
> RosterToBit(fuzzyR,fuzzyU);
                          [0.3, 0.7, 0, 0, 0.1, 0, 0]                (2.80)
```

Observe that we surrounded the modification of `B` in an if statement to ensure that the roster does not contain any members not in the given universe.

2.3 Functions

In this section we we will see three different ways to represent functions in Maple and explore a variety of the concepts described in the text relative to these different representations.

Procedures

In this manual we have already seen several examples of procedures. In some ways, a procedure, or more broadly any computer program, is the ultimate generalization of a mathematical function. As an example, consider the `GetVars` procedure. This procedure assigns to each valid input (a Maple expression) a unique output (a list of the names appearing in the expression). Setting A equal to the set consisting of all possible Maple expressions and B equal to all possible lists of valid names, `SetVars` satisfies the definition of being a function from A to B.

We discussed procedures in some depth in the introductory chapter. Here, we will discuss the concepts of domain and codomain as they relate to programs via the computer programming concept of type.

In Example 5 of Section 2.3, the text gives examples from Java and C++ showing how domain and codomain are specified in those programming languages. The procedure below illustrates how this is done in Maple.

```
> Floor1 := proc(x::float)::int;
    return floor(x);
  end proc:
```

The body of our `Floor1` procedure is merely a call to Maple's internal `floor` command. But the example illustrates how you can specify the domain (*i.e.*, the type of a parameter) of a procedure and the codomain (*i.e.*, the return type). Note that in Maple, unlike some languages such as Java and C++, such declarations are entirely optional.

Declaring the return type of a procedure is done by following the right parenthesis that ends the list of parameters with two colons and a valid Maple type and then a semicolon. Typically, this has no effect on the actual operation of the procedure and is more informational. It is possible to have

Maple enforce the return type by executing the command `kernelopts(assertlevel) := 2;`. More information can be found on the `proc` help page. In this manual, we will typically not declare return types for procedures.

To declare the type of parameters, you follow the name of the parameter by two colons and a valid Maple type. When the procedure is applied, before any of the code in the body of the procedure is executed, Maple checks the input against the declared type. If the input does not match, then an error is raised.

```
> Floor1("hello");
Error, invalid input: Floor1 expects its 1st argument, x, to
be of type float, but received hello
```

This is useful because it helps to ensure that the procedure is never applied to invalid input, which may have undesirable consequences. For example, consider the procedure below.

```
> loopy := proc(n::posint)
    local m;
    m := n;
    while m <> 0 do
      m := m - 1;
    end do;
  end proc:
> loopy(-5);
Error, invalid input: loopy expects its 1st argument, n, to
be of type posint, but received -5
```

Without the parameter declared as a positive integer, applying `loopy` to -5 would have resulted in an infinite loop.

Some common types are: float, integer, posint, nonnegint, set, and list. A complete list can be found on the help page for type. Also, Maple provides a method for easily creating new types via the structured type syntax. For example, our `Floor` procedure has a slight problem, as the following illustrates.

```
> Floor1(5.);
```

$$5 \tag{2.81}$$

```
> Floor1(5);
Error, invalid input: Floor1 expects its 1st argument, x, to
be of type float, but received 5
```

Maple distinguishes integers like 5 from floats like `5.`. We can make our procedure accept either floats or integers as follows.

```
> Floor2 := proc(x::{float,integer})
    return floor(x);
  end proc:
> Floor2(5.), Floor2(5);
```

$$5, 5 \tag{2.82}$$

Enclosing two or more types in braces indicates that any of the types are acceptable.

It is also useful to be able to specify that a procedure should accept a set or list. Let's rewrite the `Floor` procedure once again so that it accepts a list and applies Maple's `floor` function to each of the members of the list.

```
> Floor3 := proc(L::list)
    return map(floor,L);
  end proc:
```

```
> Floor3([12.5,13.9,-2.5]);
```

$$[12, 13, -3] \quad \textbf{(2.83)}$$

Note that we've used the **map** command to apply the **floor** function to each member of the input list **L**. The **map** command has a variety of syntax options, but this is one of the more common.

We can also be more specific and specify that the list must contain floats and integers. We do this by following the list keyword with parentheses and indicating the types that are allowed in the list.

```
> Floor4 := proc(L::list({float,integer}))
    return map(floor,L);
  end proc:
> Floor4([47.298,3,13.7]);
```

$$[47, 3, 13] \quad \textbf{(2.84)}$$

```
> Floor4([5.6,3,22/7,6.8]);
Error, invalid input: Floor4 expects its 1st argument, L, to
be of type list({float, integer}), but received [5.6, 3,
22/7, 6.8]
> Floor4(3.2);
Error, invalid input: Floor4 expects its 1st argument, L, to
be of type list({float, integer}), but received 3.2
```

The last examples failed because **22/7** is neither a float nor an integer and 3.2 is not a list. Maple provides the numeric type as a useful catch-all for numeric objects. Also, we can make our procedure accept either single values or lists as follows.

```
> Floor := proc(v::{numeric,list(numeric)})
    if type(v,numeric) then
      return floor(v);
    else
      return map(floor,v);
    end if;
  end proc:
> Floor([5.6,3,22/7,6.8]);
```

$$[5, 3, 3, 6] \quad \textbf{(2.85)}$$

```
> Floor(3.2);
```

$$3 \quad \textbf{(2.86)}$$

Functional Operators

Next we'll look at functional operators as a way to represent functions in Maple. This is the most natural representation for functions defined by a formula.

To represent the function defined by the formula $f(x) = x^2$, we enter the following command.

```
> f := x -> x^2;
```

$$f := x \rightarrow x^2 \quad \textbf{(2.87)}$$

There are three basic components to defining a functional operator. First, the name, which in this case is **f**. The name is followed by the assignment operator. Second is the variable or variables. (When specifying multiple variables, they must be enclosed in parentheses.) Following the variables, you enter an arrow composed of a hyphen and then a greater-than symbol. And third is the formula that provides the result.

You apply the functional operator just as you would expect.

```
> f(3);
```

$$9 \tag{2.88}$$

Note that it is not possible to specify the type of the arguments to a functional operator.

Composition of functions

We will now use functional operators to briefly explore composition and graphs of functions. You can combine functions algebraically in the natural way. Maple will return the formula for a functional operator when you apply it to an unassigned name.

```
> g := x -> x + 1;
```

$$g := x \to x + 1 \tag{2.89}$$

```
> (f+g)(x);
```

$$x^2 + x + 1 \tag{2.90}$$

```
> (f/g)(t);
```

$$\frac{t^2}{t+1} \tag{2.91}$$

The parentheses around **f+g** and **f/g** are necessary because the application of a procedure is of higher precedence than the arithmetic operators. Contrast the above with **f/g(t)**, which results in the name **f** divided by the formula for **g**.

```
> f/g(t);
```

$$\frac{f}{t+1} \tag{2.92}$$

For composition, Maple provides the **@** operator.

```
> (f @ g)(x);
```

$$(x+1)^2 \tag{2.93}$$

```
> (g @ f)(x);
```

$$x^2 + 1 \tag{2.94}$$

Composition can also be applied to procedures, both those defined by you and any built into Maple. For example, the following defines the function that computes the square of the floor of a number.

```
> squareFloor := f @ floor;
```

$$squareFloor := f@\mathrm{floor} \tag{2.95}$$

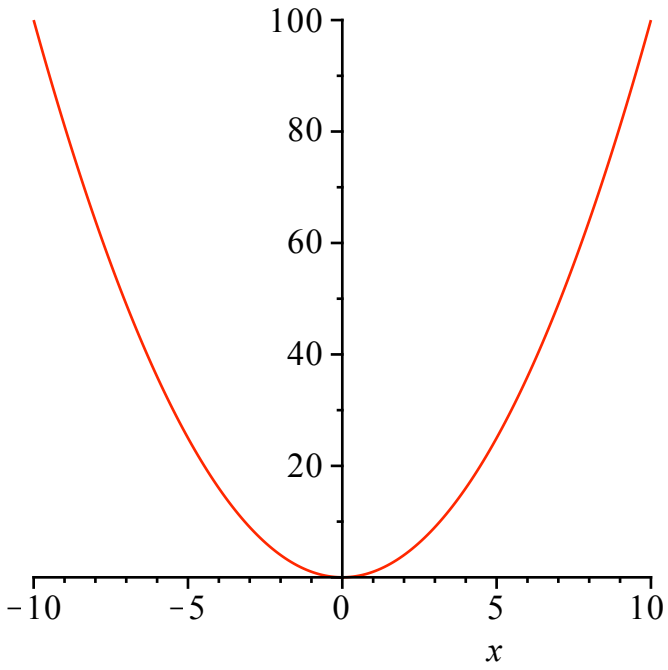
```
> squareFloor(3.2);
```

$$9 \tag{2.96}$$

Plotting graphs of functions

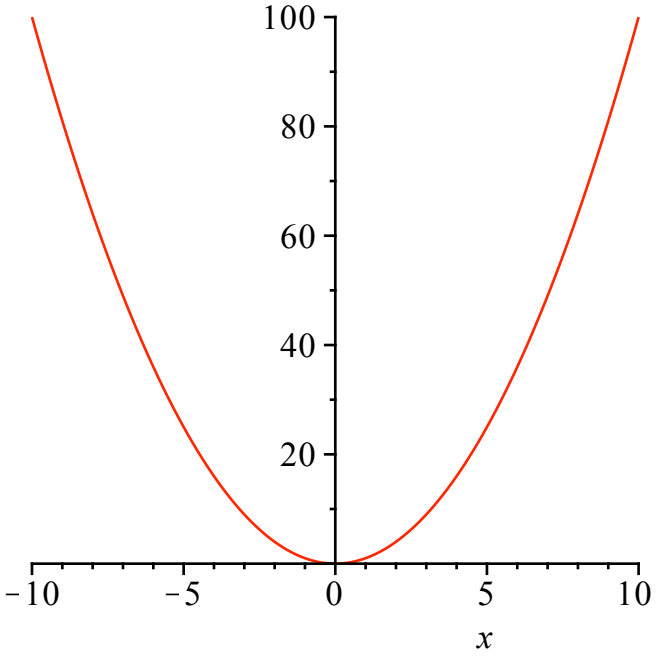
You can have Maple draw the graph of a function by using the **plot** command. The most basic syntax requires only two arguments: the function to be graphed in terms of an independent variable and the variable. The first argument can be given as an expression as follows.

```
> plot(x^2,x);
```

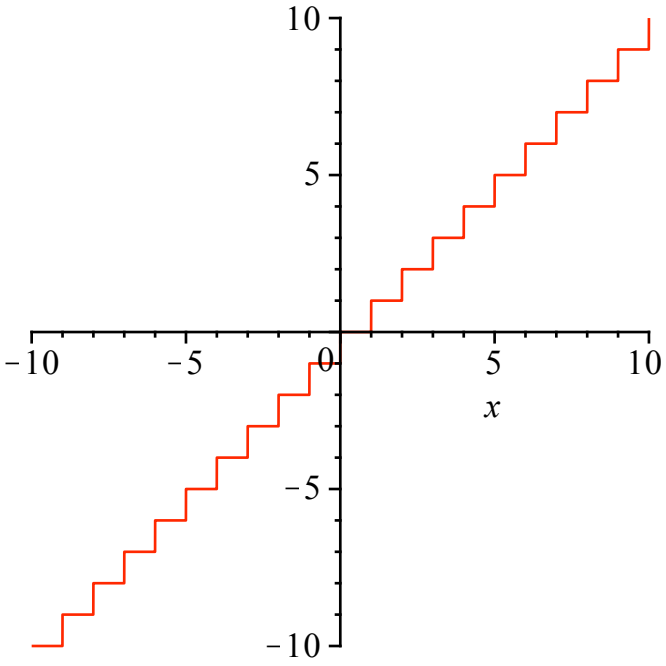


The first argument can also be given as a functional operator or even a procedure as the following two examples illustrate. The essential requirement is that the first argument must evaluate to a numeric value whenever the independent variable is assigned a value.

```
> plot(f(x),x);
```

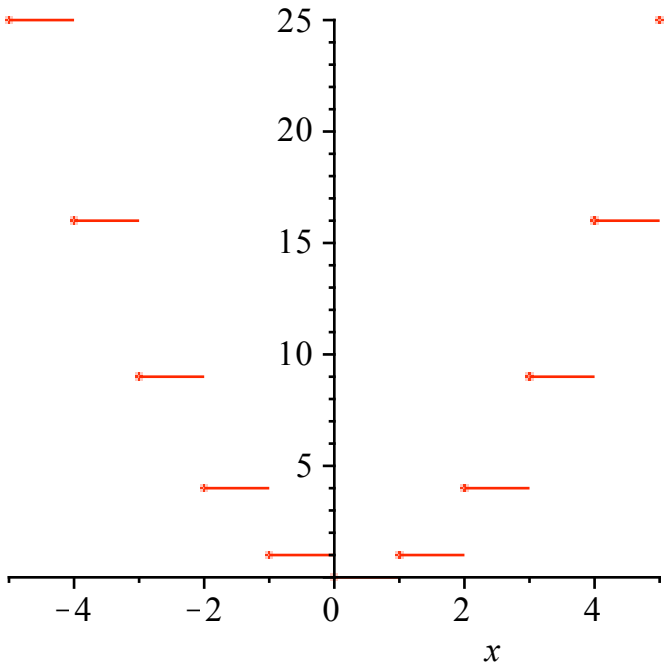



```
> plot(floor(x),x);
```



In the last example above, the vertical lines are artifacts of how Maple draws graphs. Basically, Maple is computing the value of the function at a large number of x-values between -10 and 10. It then "connects the dots." The vertical lines appear when Maple connects the points from either side of the jumps. We can eliminate them with the **discont=true** option to tell Maple that the graph has discontinuities. We can also specify the range of x-values we want to display by entering the equation **x=min..max** as the second argument. In the next graph, we use those two options to display a graph of the square of the floor function.

```
> plot(squareFloor(x),x=-5..5,discont=true);
```



Tables

For finite domains, a table can be used to represent a function. To define a table, use the **table** command. For example, suppose f is the function whose domain is the set of students in a class and that maps each student to their grade on an exam. Let f be defined by $f(Ann) = 83$, $f(Bob) = 79$, $f(Carla) = 91$, and $f(Dave) = 72$. We model f as a table named **exams** by applying the **table** command to the list whose entries are equations of the form **a=b** to represent $f(a) = b$.

```
> exams := table(["Ann"=83,"Bob"=79,"Carla"=91,"Dave"=72]);
```

$$exams := table([\text{"Dave"} = 72, \text{"Carla"} = 91, \text{"Bob"} = 79, \text{"Ann"} = 83]) \quad (2.97)$$

Once the table is defined, you can obtain the value $f(Carla)$ with the bracket selection operation.

```
> exams["Carla"];
```

$$91 \quad (2.98)$$

You can also use selection together with assignment to modify values or to add entries to the table.

```
> exams["Ann"] := 84;
```

$$exams_{\text{"Ann"}} := 84 \quad (2.99)$$

```
> exams["Ernie"] := 86;
```

$$exams_{\text{"Ernie"}} := 86 \quad (2.100)$$

To see the table definition, you apply the **op** command. You can obtain the list of equations by applying **op** twice.

```
> exams;
```

$$exams \quad (2.101)$$

```
> op(exams);
```

$$table([\text{"Ernie"} = 86, \text{"Dave"} = 72, \text{"Carla"} = 91, \text{"Bob"} = 79, \text{"Ann"} = 84]) \quad (2.102)$$

```
> op(op(exams));
```

$$[\text{"Ernie"} = 86, \text{"Dave"} = 72, \text{"Carla"} = 91, \text{"Bob"} = 79, \text{"Ann"} = 84] \quad (2.103)$$

Domain and range

Since tables are finite, we can write procedures to check various properties. First, we'll find the domain (technically, the domain of definition) and range of a function defined as a table. For these procedures, we use the commands **indices** and **entries**. In the **exams** table, the students' names (Ann, Bob, etc.) are the *indices* or *keys* of the table and the scores (84, 79, etc.) are the *entries* or *values* of the table.

You would probably expect **indices(exams);** to return the list or set of students' names and **entries(exams)** to return the list of scores. Observe what actually is returned.

```
> indices(exams);
```

$$[\text{"Ernie"}], [\text{"Dave"}], [\text{"Carla"}], [\text{"Bob"}], [\text{"Ann"}] \quad (2.104)$$

```
> entries(exams);
```

$$[86], [72], [91], [79], [84] \quad (2.105)$$

The **indices** command returns a sequence of lists where each list contains an index of the table. The reason for this is to allow for very complicated indices which may even be sequences of values. For example, we can define the following table.

```
> M := table():
> M[1,1] := 1:
> M[1,2] := 0:
> M[2,1] := 0:
```

```
> M[2,2] := 1:
> op(M);
```

$$table([(1,2)=0,(2,2)=1,(2,1)=0,(1,1)=1]) \quad \textbf{(2.106)}$$

```
> indices(M);
```

$$[1,2],[2,2],[2,1],[1,1] \quad \textbf{(2.107)}$$

If the indices command had returned a sequence of the indices, it would have appeared that the indices were 1 and 2 repeated several times:

```
> (2,1),(1,1),(2,2),(1,2);
```

$$2, 1, 1, 1, 2, 2, 1, 2 \quad \textbf{(2.108)}$$

(Note that we defined **M** to be the empty table and then added entries to it. This is a very common way to define a table. Also, readers with some experience with matrices will note that **M** is a representation of the identity matrix of dimension 2.)

In cases where you are sure that there is no need for **indices** to return the indices in lists, you can use the **'nolist'** symbol. The same is true for the **entries** command.

```
> indices(exams,'nolist');
```

"Ernie", "Dave", "Carla", "Bob", "Ann" **(2.109)**

```
> entries(exams,'nolist');
```

$$86, 72, 91, 79, 84 \quad \textbf{(2.110)}$$

Note that while Maple chooses the order in which to report the indices and entries and the user has no control over that order, the order is consistent between the two results. That is, the entry that is listed first is the entry corresponding to the index that is listed first, the second entry corresponds to the second index, and so on.

The discussion above allows us to easily write procedures to compute the domain and range of a function represented by a table.

```
> FindDomain := proc(T::table)
    return {indices(T,'nolist')};
  end proc:
> FindRange := proc(T::table)
    return {entries(T,'nolist')};
  end proc:
> FindDomain(exams);
```

{"Ann", "Bob", "Carla", "Dave", "Ernie"} **(2.111)**

```
> FindRange(exams);
```

$$\{72, 79, 84, 86, 91\} \quad \textbf{(2.112)}$$

Injective and surjective

Let's create a few more examples. Then we will write procedures to check for injectivity and surjectivity. The examples below correspond to the functions $f_1(x) = x^2, f_2(x) = x^3$, and $f_3(x) = |x|$ on the domain D = $\{-5,-4,...,5\}$.

```
> f1 := table([seq(x=x^2,x=-5..5)]);
```

$$f1 := table([0=0, 1=1, 2=4, 3=9, 4=16, 5=25, -5=25, -4=16, -3=9, -2=4, -1=1]) \quad \textbf{(2.113)}$$

```
> f2 := table([seq(x=x^3,x=-5..5)]);
```

$$f2 := table([0=0, 1=1, 2=8, 3=27, 4=64, 5=125, -5=-125, -4=-64, -3= \quad \textbf{(2.114)}$$

```
    -27, -2 = -8, -1 = -1])
> f3 := table([seq(x=abs(x),x=-5..5)]);
```

$$f3 := table([0 = 0, 1 = 1, 2 = 2, 3 = 3, 4 = 4, 5 = 5, -5 = 5, -4 = 4, -3 = 3, -2 = 2, -1 = 1]) \quad (2.115)$$

We can check to see if a table is surjective for a specified codomain by comparing the codomain to the range.

```
> IsOnto := proc(T::table,codomain::set)
    return evalb(FindRange(T) = codomain);
  end proc:
> IsOnto(f1,{0,1,2,3,4,5});
```

$$false \quad (2.116)$$

```
> IsOnto(f3,{0,1,2,3,4,5});
```

$$true \quad (2.117)$$

We can check for injectivity by making sure that no entry value is repeated. The easiest way to do this is to check that the number of values in the result of **FindRange** is the same as the number in the domain returned by **FindDomain**.

```
> IsOnetoOne := proc(T::table)
    return evalb(nops(FindDomain(T))=nops(FindRange(T)));
  end proc:
> IsOnetoOne(f1);
```

$$false \quad (2.118)$$

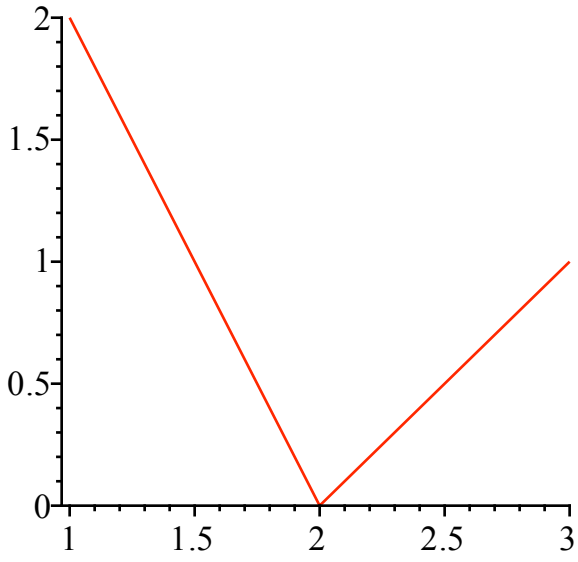
```
> IsOnetoOne(f2);
```

$$true \quad (2.119)$$

Graphing a function from a table

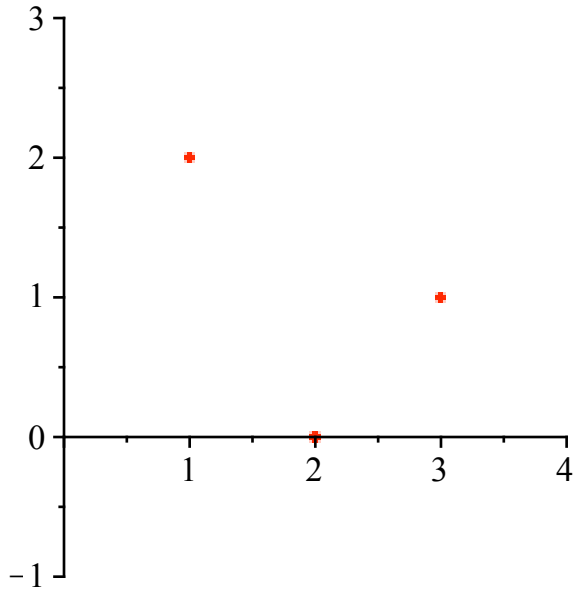
Finally, let's see how we can graph a function defined by a table. We'll use an alternate form of the **plot** command which accepts two arguments. The first argument will be the list of x-values and the second will the list list of y-values. For example,

```
> plot([1,2,3],[2,0,1]);
```



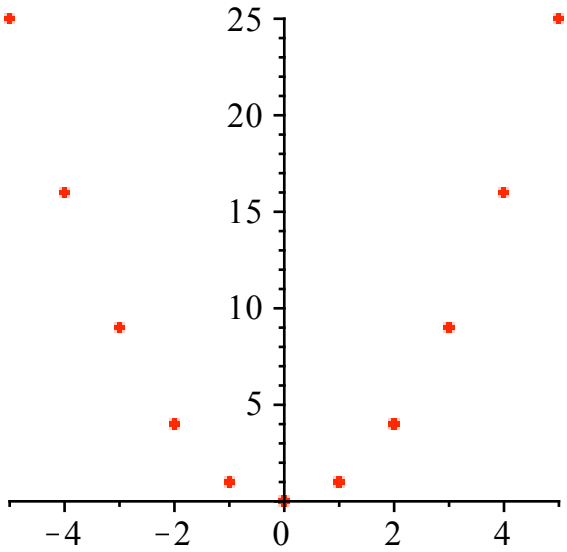
Note that Maple connected the points $(1, 2)$, $(2, 0)$, and $(3, 1)$ to draw the graph. We can use the **style=point** option to have Maple draw points instead of connecting the dots. Also, the options **symbol=solidcircle** and **symbolsize=15** will cause the points to be drawn as solid circles 15 points in diameter. Finally, **view=[0..4,-1..3]** will make the bounds of the graph 0 to 4 on the *x* axis and -1 to 3 on the *y*.

```
> plot([1,2,3],[2,0,1],style=point,symbol=solidcircle,
  symbolsize=15,view=[0..4,-1..3]);
```



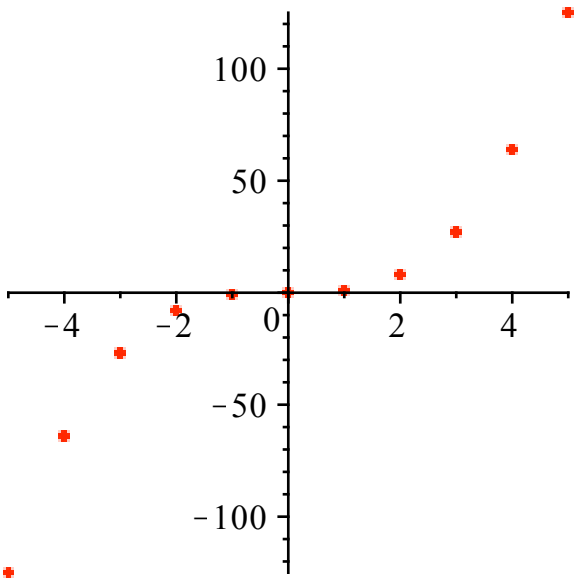
Graphing a function defined by a table can be done in the same way, using the **indices** and **entries** commands to create lists of the *x* and *y* coordinates of the desired points. Remember that Maple orders the indices and entries so that they are consistent, *i.e.*, the *x* and *y* values will correspond.

```
> plot([indices(f1,'nolist')],[entries(f1,'nolist')],style=
  point,symbol=solidcircle,symbolsize=15,view=[-5..5,0..25]);
```

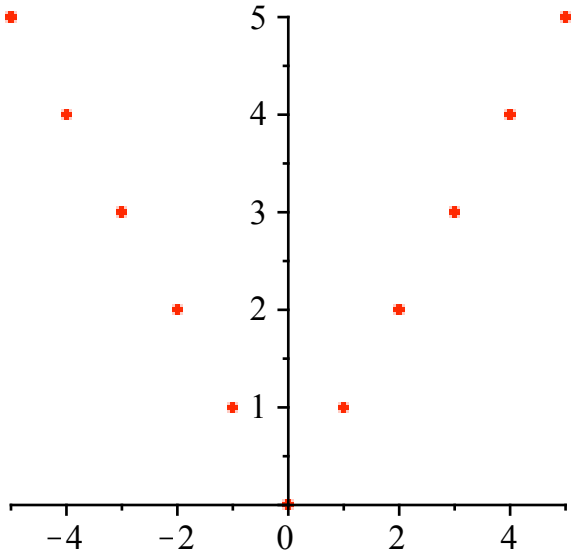



```
> plot([indices(f2,'nolist')],[entries(f2,'nolist')],style=
```

```
point,symbol=solidcircle,symbolsize=15,view=[-5..5,-125..125]
);
```




```
> plot([indices(f3,'nolist')],[entries(f3,'nolist')],style=
point,symbol=solidcircle,symbolsize=15,view=[-5..5,0..5]);
```



Some Important Functions

We've already seen that Maple has a built-in floor function, **floor**. It also includes **ceil** for computing the ceiling of a real number.

```
> floor(2.7);
```

$$2 \qquad (2.120)$$

```
> ceil(2.7);
```

$$3 \qquad (2.121)$$

Maple contains some additional related functions. The **round** command rounds a number to the nearest integer. The **trunc** command truncates the number, removing any fractional part,

producing the next nearest integer towards 0. And the **frac** command returns the fractional part of the number.

```
> round(2.7);
```

$$3 \quad \textbf{(2.122)}$$

```
> trunc(2.7);
```

$$2 \quad \textbf{(2.123)}$$

```
> trunc(-2.7);
```

$$-2 \quad \textbf{(2.124)}$$

```
> frac(2.7);
```

$$0.7 \quad \textbf{(2.125)}$$

The text also discusses the factorial function. In Maple, you compute the factorial of a number by entering the number followed by the exclamation point. You can also use the **factorial** command.

```
> 6!;
```

$$720 \quad \textbf{(2.126)}$$

```
> factorial(6);
```

$$720 \quad \textbf{(2.127)}$$

2.4 Sequences and Summations

In this section we will see how Maple can be used to create and manipulate sequences, and in particular, we will see a way to use Maple to generate the terms of a recurrence sequence. We will also look at summations and see how Maple's symbolic computation abilities can be used to explore both finite and infinite series.

Sequences are fundamental to Maple. In Maple, an expression sequence is any ordered collection of valid expressions separated by commas. For example,

```
> aSequence := 1,"a",x,Pi,3*x^2+5,{"a","b","c"};
```

$$aSequence := 1, \text{"a"}, x, \pi, 3x^2 + 5, \{\text{"a"}, \text{"b"}, \text{"c"}\} \quad \textbf{(2.128)}$$

is an expression sequence (or just sequence). Note that both sets and lists are formed by wrapping an expression sequence in the appropriate symbols and procedures are called on particular values by passing the procedure a sequence of arguments in parentheses.

Elements of a sequence can be accessed in the same was as lists and sets, with the selection operation, as follows.

```
> aSequence[3];
```

$$x \quad \textbf{(2.129)}$$

The **nops** command cannot be used on a sequence in order to determine its length. To find the number of elements in a sequence, you must first convert the sequence to a list and then apply **nops**. Likewise, **op** does not work correctly for sequences.

```
> nops(aSequence);
Error, invalid input: nops expects 1 argument, but received 6
> nops([aSequence]);
```

$$6 \quad \textbf{(2.130)}$$

```
> op(aSequence);
Error, invalid input: op expects 1 or 2 arguments, but
```

received 6
Many commands commonly used with lists and sets will produce errors or incorrect results when applied to a sequence.

The "empty sequence" is represented by the name **NULL**. Often in procedures, you will initialize a variable to **NULL** in order to build up a sequence of values. You may also have a procedure return **NULL** in order to cause the procedure to exit without output.

There are three main tools for creating a sequence in Maple: the comma operator, the **seq** command, and the **$** operator.

Building Sequences: Comma Operator

The comma operator is used to join two expressions or expression sequences into a sequence. The comma operator is used when forming a sequence by listing the elements, as in the following.

```
> sequence2 := 1,2,3,4;
```

$$sequence2 := 1, 2, 3, 4 \quad \textbf{(2.131)}$$

It is also used to combine two existing sequences or a sequence and a single expression. Note that the original sequences are not modified unless you reassign the name to the result.

```
> aSequence, sequence2;
```

$$1, \text{"a"}, x, \pi, 3x^2 + 5, \{\text{"a"}, \text{"b"}, \text{"c"}\}, 1, 2, 3, 4 \quad \textbf{(2.132)}$$

```
> sequence2 := sequence2,15;
```

$$sequence2 := 1, 2, 3, 4, 15 \quad \textbf{(2.133)}$$

This application of the comma operator is often used in procedures to build a sequence one term at a time. For example, the following procedure builds the sequence consisting of the first $n + 1$ terms of the geometric progression a, ar, ar^2, ar^3, …, ar^n.

```
> GeometricSeq := proc(a,r,n)
    local S, i;
    S := NULL;
    for i from 0 to n do
      S := S,a*r^i;
    end do;
    return S;
  end proc:
> GeometricSeq(3,4,10);
```

$$3, 12, 48, 192, 768, 3072, 12288, 49152, 196608, 786432, 3145728 \quad \textbf{(2.134)}$$

The output sequence is initialized to the **NULL** value. At each step in the for loop, the next term in the sequence is added to the existing sequence **S** with the comma operator.

Building Sequences: seq Command

We've already seen several examples of the **seq** command. We'll briefly summarize some of the ways it can be called.

The most common way to call **seq** is demonstrated in the following example, which recreates the geometric sequence produced above.

```
> seq(3*4^i,i=0..10);
```

$$3, 12, 48, 192, 768, 3072, 12288, 49152, 196608, 786432, 3145728 \quad \textbf{(2.135)}$$

The first argument is an expression which may involve an index variable, in this case **i**. The second argument is of the form **i=m..n**. This indicates that the index variable should range from

m to **n**.

A third argument can be added to control the step, *i.e.*, the amount by which the index variable is incremented. For example, the command below will produce every other term of the geometric sequence from above.

```
> seq(3*4^i,i=0..10,2);
```

$$3,\ 48,\ 768,\ 12288,\ 196608,\ 3145728 \qquad \textbf{(2.136)}$$

The bounds of the range for the index and the step do not necessarily need to be integers. For example,

```
> seq(i,i=2.3..5.6,.25);
```

$$2.3,\ 2.55,\ 2.80,\ 3.05,\ 3.30,\ 3.55,\ 3.80,\ 4.05,\ 4.30,\ 4.55,\ 4.80,\ 5.05,\ 5.30,\ 5.55 \qquad \textbf{(2.137)}$$

There is also an abbreviated form allowing you to omit the first argument and the index variable. For example, to obtain the first ten positive even integers, you can issue the following command.

```
> seq(2..20,2);
```

$$2,\ 4,\ 6,\ 8,\ 10,\ 12,\ 14,\ 16,\ 18,\ 20 \qquad \textbf{(2.138)}$$

The other main use of **seq** is to apply the expression to each element of a set or list. This is illustrated in the command below which finds the squares of the first six prime numbers.

```
> seq(i^2,i={2,3,5,7,11,13});
```

$$4,\ 9,\ 25,\ 49,\ 121,\ 169 \qquad \textbf{(2.139)}$$

Note that for sets, the order of the elements in the sequence is determined by the order that Maple imposes on the set. For example, if we rearrange the primes in the example above, the output will be the same.

```
> seq(i^2,i={2,5,7,13,11,3});
```

$$4,\ 9,\ 25,\ 49,\ 121,\ 169 \qquad \textbf{(2.140)}$$

To impose a particular order, use a list instead.

```
> seq(i^2,i=[2,5,7,13,11,3]);
```

$$4,\ 25,\ 49,\ 169,\ 121,\ 9 \qquad \textbf{(2.141)}$$

As an alternative to **i=**, you may use the word **in** in place of the equals sign.

```
> seq(i^2,i in [2,5,7,13,11,3]);
```

$$4,\ 25,\ 49,\ 169,\ 121,\ 9 \qquad \textbf{(2.142)}$$

While **seq** is most commonly used in conjunction with lists and sets, any expression can be used in place of the list. For example, the following computes the sequence consisting of the squares of the terms in the given algebraic expression.

```
> seq(i^2,i=3*x^5+2*x^4-x^3+7*x^2-8*x+9);
```

$$9x^{10},\ 4x^{8},\ x^{6},\ 49x^{4},\ 64x^{2},\ 81 \qquad \textbf{(2.143)}$$

In fact, **seq** can be used with any expression to which **op** can be applied.

Building Sequences: $ Operator

The **$** operator is an alternative to the **seq** command, though it is somewhat more limited. The **$** operator is a binary operator, like **+** or *****. Its left operand is the expression in terms of an index variable and the right operand is the equation that specifies the range for the variable. We can produce the geometric sequence from above as follows.

```
> 3*4^i $ i=0..10;
```

3, 12, 48, 192, 768, 3072, 12288, 49152, 196608, 786432, 3145728 **(2.144)**

Note that this is the same as `seq(3*4^i,i=0..10)`.

It is sometimes necessary to enclose the expression on the left and the variable on the right in single right quotes to prevent early evaluation. This is the case when the index has previously been assigned a value.

```
> anIndex := 20;
```

$$anIndex := 20 \quad \textbf{(2.145)}$$

```
> anIndex*2+3 $ anIndex=1..5;
Error, invalid input: `$` expects its 2nd argument, range, to
be of type {numeric, algebraic, name = literal .. literal,
name = algebraic .. algebraic}, but received 20 = 1 .. 5
```

The error occurs because Maple evaluates the expressions on both sides of the **$** operator before applying the operator to them. Since **anIndex** stored a value, the right hand side became **20=1..5**, causing an error. We fix this by enclosing the name of the index variable on the right in single quotes so that it is not evaluated. We also must enclose the entire expression on the left in single quotes so that it is not evaluated to 43.

```
> 'anIndex*2+3' $ 'anIndex'=1..5;
```

5, 7, 9, 11, 13 **(2.146)**

The single quotes were not necessary for previous examples because **i** had not been assigned a value. Also note that this issue is not a concern for **seq**.

```
> seq(anIndex*2+3,anIndex=1..5);
```

5, 7, 9, 11, 13 **(2.147)**

The **$** operator is particularly useful in the following two situations. Like **seq**, the left operand and the index variable can be omitted to produce the specified range.

```
> $ 5..11;
```

5, 6, 7, 8, 9, 10, 11 **(2.148)**

Also, if the right operand is an integer rather than a range, **$** will produce that many copies of the right operand. For example, to create a sequence of 11 copies of the string "a", we give "a" as the left hand operand and 11 as the right operand to **$**.

```
> "a"$11;
```

"a", "a", "a", "a", "a", "a", "a", "a", "a", "a", "a" **(2.149)**

Doing this with the **seq** command requires the following.

```
> seq("a",i=1..11);
```

"a", "a", "a", "a", "a", "a", "a", "a", "a", "a", "a" **(2.150)**

Recurrence Relations

Next, we will see how we can use Maple to explore sequences that arise from recurrence relations. We will go into much more depth, especially in regards to Maple's functions related to solving recurrence relations, in Chapter 8. Here, we will only explore how we can have Maple compute terms of sequences defined by recurrence relations.

As an example, consider the Fibonacci sequence, which has recurrence relation $f_n = f_{n-1} + f_{n-2}$ and initial conditions $a_1 = 0$ and $a_2 = 1$. (Note that the text uses 0 as the first index for a sequence,

but Maple uses 1 as the first index for sequences and lists.) To produce this sequence in Maple, we can use a functional operator to represent the recurrence relation as follows.

```
> Fib := n -> Fib(n-1) + Fib(n-2);
```

$$Fib := n \rightarrow Fib(n-1) + Fib(n-2) \quad \textbf{(2.151)}$$

Next, we set the initial values as follows.

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

$$Fib(1) := 1 \quad \textbf{(2.152)}$$

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

$$Fib(2) := 1 \quad \textbf{(2.153)}$$

Now, Maple will compute values of the sequence.

```
> Fib(7);
```

$$13 \quad \textbf{(2.154)}$$

To display the sequence, use the **seq** command.

```
> seq(Fib(n),n=1..20);
```

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765 \quad \textbf{(2.155)}$$

While the above approach for calculating recurrence relations is convenient and intuitive, it does not make available all of the facilities for improving efficiency that are available using the **proc** command. We can get Maple to calculate these values more efficiently by using the **remember** option. This option requires Maple to "remember" any values for the procedure that it has already computed by storing them in a table.

```
> Fib2 := proc(n::posint)
    option remember;
    if n <= 2 then
      return 1;
    else
      return Fib2(n-1) + Fib2(n-2);
    end if;
  end proc:
```

This procedure encompasses both the initial conditions (when $n \leq 2$) and the recurrence formula. The remember option causes Maple to store the results of the procedure when it is called so that if it is called again with the same input, the result can be looked up in the "remember table" rather than recomputed.

To illustrate the difference in performance, let's see how long it takes to compute the 1000th Fibonacci number. We use the standard approach to timing procedures.

```
> st := time(): Fib2(1000): time() - st;
```

$$0.003 \quad \textbf{(2.156)}$$

The **time** command returns the total CPU time used in the current Maple session. So the above works by setting **st** (for start time) equal to the amount of CPU time used before executing the procedure being timed, then executing the procedure, and then computing the difference of the amount of CPU time used with the start time.

The output above shows the amount of time, in seconds, used to find the thousandth Fibonacci number using **Fib2**. Note that if we repeat the computation,

```
> st := time():  Fib2(1000): time() - st;
```

$$0. \quad \textbf{(2.157)}$$

the total time take drops to nothing, or at least very close to it. This is because Maple doesn't need to compute the value again. You can cause Maple to reset the remember table with the **forget** command.

In comparison, consider the **Fib** functional operator applied to 30.

```
> st := time(): Fib(30): time() - st;
```

$$0.735 \qquad (2.158)$$

Note that the purely recursive implementation Fib cannot be used to compute the thousandth Fibonacci number. In fact, to compute the 1000th Fibonacci number, **Fib** would need to be invoked approximately

```
> Fib2(999);
```

$$26863810024485359386146727202142923967616609318986952340123175997617981\ 70024788168933836965448335656419182785616144335631297667364221035032\ 46348504103776803673341511728991697231970827639856157644500784741746\ 26 \qquad (2.159)$$

times in order to handle all the recursive sub-calls that are made. (The reader is encouraged to prove this fact.)

Even at a billion calls per second, this would require

```
> (2.159)/1000000000.;
```

$$2.686381002\ 10^{199} \qquad (2.160)$$

seconds, or

```
> (2.160)/(60*60*24*365);
```

$$8.518458276\ 10^{191} \qquad (2.161)$$

years to complete.

Summations

Finally, we will see how Maple can be used to compute with summations, both numerically for finite sums and symbolically for infinite sums.

To add a finite sequence of values, we use the **add** command. This command is very similar to the **seq** command, though with somewhat fewer options. It requires two arguments. The first argument must be an expression in terms of an index variable such as **i**. The second argument can be either an equation of the form **i=m..n**, indicating the range of values for the index variable, or it can be of the form **i=x** or **i in x** where **x** is a list, set, or other such object. The forms **i=x** and **i in x** are equivalent.

For example, to compute the sum of the squares of the first ten positive integers, $\sum_{i=1}^{10} i^2$, we enter the following.

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

$$385 \qquad (2.162)$$

And to compute the sum of the members of a list:

```
> add(i,i in [1,2,4,6,9,11,14]);
```

$$47 \qquad (2.163)$$

The **mul** command is used with the same syntax as **add** to compute products of sequences.

The **sum** command is used for symbolic summation. The first argument to the **sum** command is the same as for **add**, an expression in terms of an index variable. The second argument is of the form **i=m..n**. The main difference is that for **add**, **m** and **n** must be numbers, while for **sum**, they can be unassigned names or even **infinity**.

As an example, we have Maple compute the sum of the squares of the first n positive integers, $\sum_{k=1}^{n} k^2$.

```
> sum(k^2,k=1..n);
```

$$\frac{1}{3}(n+1)^3 - \frac{1}{2}(n+1)^2 + \frac{1}{6}n + \frac{1}{6} \qquad \textbf{(2.164)}$$

We can also compute the sum of the terms with even index up to 2 n in a geometric series, *i.e.*, $\sum_{k=0}^{n} ar^{2k}$.

```
> sum(a*r^(2*k),k=0..n);
```

$$\frac{a\left(r^2\right)^{n+1}}{r^2-1} - \frac{a}{r^2-1} \qquad \textbf{(2.165)}$$

Finally, $\sum_{k=1}^{\infty} kx^{k-1}$ is computed by

```
> sum(k*x^(k-1),k=1..infinity);
```

$$\frac{1}{(x-1)^2} \qquad \textbf{(2.166)}$$

You can confirm that these results match the formulas given in Table 2 of Section 2.4.

2.5 Cardinality of Sets

In this section we will explore the countability of the positive rational numbers. In Example 4 of Section 2.5 of the text, it is shown that the positive rationals are countable by describing how to list them all. Here, we will use Maple to implement this listing algorithm. We will also consider the following two questions. First, given a positive rational number, what is its position in the list? Second, given a positive integer, what fraction is located at that position within the list?

We'll begin by reviewing the the description in Example 4. The first element of the list is the rational number $\frac{1}{1}$. Then we list the positive rationals $\frac{p}{q}$ such that $p+q=3$. Then come the rationals with $p+q=4$, excluding $\frac{2}{2}$ which is already in the list, being equivalent to $\frac{1}{1}$. This continues for each n: we list the fractions $\frac{p}{q}$ such that $p+q=n$, excluding those equivalent to fractions already in the list.

In our procedure, we'll refer to n as the stage, so that in stage 5, for example, we're listing the

fractions $\frac{p}{q}$ such that $p + q = 5$. The stage n will range from 2 up to some maximum value. This maximum value of n will be the parameter to the procedure. We'll implement this as a for loop with index variable n.

Within each stage, *i.e.*, within the for loop, we need to generate the rational numbers $\frac{p}{q}$ and add them to the list, provided they are not already in it. We can rewrite $p + q = n$ as $p = n - q$. By allowing q to range from 1 to $n - 1$ and calculating p, we will produce all the potential rationals in stage n. The **in** operator, discussed in Section 2.1 in relation to sets, applies to lists as well. So, for each q from 1 to $n - 1$, we will form the fraction $\frac{p}{q}$ (with $p = n - q$), use **in** to test whether this is already in our list of positive rationals, and, if not, add it to the list.

Here is the complete procedure.

```
> ListRationals := proc(max::posint)
    local L, n, p, q;
    L := [];
    for n from 2 to max do
      for q from 1 to n-1 do
        p := n-q;
        if not(p/q in L) then
          L := [op(L),p/q];
        end if;
      end do;
    end do;
    return L;
  end proc:
```

Applying this procedure to 6, we obtain the list through stage 6.

```
> ListRationals(6);
```

$$\left[1, 2, \frac{1}{2}, 3, \frac{1}{3}, 4, \frac{3}{2}, \frac{2}{3}, \frac{1}{4}, 5, \frac{1}{5}\right] \qquad \textbf{(2.167)}$$

Finding the Position Given a Positive Rational

Suppose we want to determine the position of a particular fraction within the list. Take for example $\frac{29}{35}$. Since $29 + 35 = 64$, we know that this fraction would first appear in stage 64. So we compute the list up to stage 64.

```
> RatsTo64 := ListRationals(64):
```

We suppress the output because this is a long list:

```
> nops(RatsTo64);
```

$$1259 \qquad \textbf{(2.168)}$$

Now we work backwards from the end of the list until we find the desired fraction. A simple loop will help with this. Recall that the **by** clause in a for loop allows us to specify how much the index variable is changed each time, so **by -1** causes the for loop to step backwards by 1 each iteration. Once we find the location of the desired fraction, we display the location and **break** the loop.

```
> for i from 1259 to 1 by -1 do
    if RatsTo64[i] = 29/35 then
```

```
      print(i);
      break;
    end if;
  end do:
```

$$1245 \qquad \textbf{(2.169)}$$

We can make this process into a procedure. Given a fraction, the **numer** and **denom** commands will extract the numerator and denominator, respectively.

```
> numer(29/35);
```

$$29 \qquad \textbf{(2.170)}$$

```
> denom(29/35);
```

$$35 \qquad \textbf{(2.171)}$$

So our procedure can accept a rational number (type **rational**) as input with an additional check to make sure the input is positive. We'll sum the results of **numer** and **denom** to determine the stage. Within the for loop, we'll use a **return** statement instead of **print** and **break**.

```
> LocateRational := proc(r::rational)
    local stage, L, i;
    if r <= 0 then
      error "Input value must be positive.";
    end if;
    stage := numer(r) + denom(r);
    L := ListRationals(stage);
    for i from nops(L) to 1 by -1 do
      if L[i] = r then
        return i;
      end if;
    end do;
  end proc:
> LocateRational(75/197);
```

$$22566 \qquad \textbf{(2.172)}$$

Finding the Rational In a Given Position

On the other hand, suppose we want to know which fraction is at a particular position. For instance, say we want to know which is the hundredth fraction listed. If we knew which stage of the process would yield a list of at least 100 rational numbers, we could just generate the list up to that stage. We can guess and check until we found a stage that produced a long enough list.

Putting a lower bound on the number of stages

We can guide our guesses a bit, however. Remember that at stage 2, the process generates 1 fraction. At stage 3, it generates 2 fractions. At stage 4, it generates 3 fractions, although one of them is discarded because it is a repeat. At stage k, the process generates $k-1$ rational numbers, some of which may be discarded as repeats. So we know that, after stage n is complete, the number of rational numbers in our list contains *at most* $\sum_{k=2}^{n} k-1$ rational numbers. We can use the **sum** command discussed in the previous section to find a formula for this summation.

```
> sum(k-1,k=2..n);
```

$$\frac{1}{2}(n+1)^2 - \frac{3}{2}n - \frac{1}{2} \qquad \textbf{(2.173)}$$

Applying **factor** will give us a more convenient formula.

```
> factor(%);
```

$$\frac{1}{2}\, n\,(n-1) \tag{2.174}$$

In other words, the number of rational numbers in the list produced by **ListRationals** at the conclusion of stage n is at most $\frac{n(n-1)}{2}$. Define $F(n)$ to be the number of positive rational numbers produced by the **ListRationals** algorithm at the conclusion of stage n. Alternately, $F(n)$ is the number of distinct positive rational numbers $\frac{p}{q}$ such that $p+q \le n$. Then we have determined that

$$F(n) \le \frac{n(n-1)}{2}.$$

Now we return to the question of how many stages we need to compute in order to find the 100th rational number. We can now restate this as follows: find n such that $F(n) \geqslant 100$. Combining our inequalities, we have that $\frac{n(n-1)}{2} \geqslant 100$. Maple's **solve** command will solve the equation for us. (Note that since an approximation is sufficient, we will enter the 100 as **100.** so that Maple will solve using floating-point arithmetic.)

```
> solve(n*(n-1)/2 = 100.);
```

$$14.65097170,\ -13.65097170 \tag{2.175}$$

This indicates that a stage of 14 *is not enough*. But it gives us a place to start guessing.

```
> RatsTo15 := ListRationals(15):
> nops(RatsTo15);
```

$$71 \tag{2.176}$$

```
> RatsTo17 := ListRationals(17):
> nops(RatsTo17);
```

$$95 \tag{2.177}$$

```
> RatsTo18 := ListRationals(18):
> nops(RatsTo18);
```

$$101 \tag{2.178}$$

```
> RatsTo18[100];
```

$$\frac{5}{13} \tag{2.179}$$

How tight is the bound?

We just saw how the formula $\frac{n(n-1)}{2}$ is an upper bound for $F(n)$, the number of positive rationals listed by the end of stage n. We'll conclude this section by exploring how good of a bound this is. In Section 2.3, we saw how to use the **plot** command to graph points. Let's use that technique to graph the upper bound up to $n = 100$.

To graph points, we need two lists for the x and y values to be graphed. The x values will be the values of n. We'll create the list by using the **seq** command.

```
> xValues := [seq(n,n=2..100)]:
```

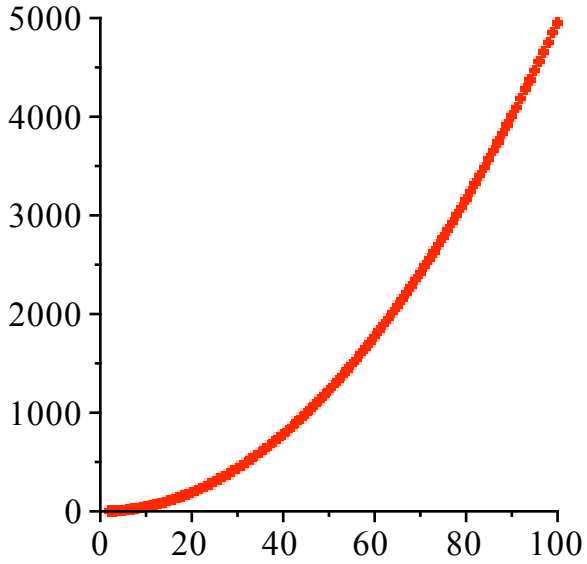
For the y-values, we use the **seq** command with the formula for the upper bound:

$\frac{n(n-1)}{2}$.

```
> boundValues := [seq(n*(n-1)/2,n=2..100)]:
```

Now we can plot the bound using the **plot** command and the options described in Section 2.3.

```
> plot(xValues,boundValues,style=point,symbol=solidcircle,
  symbolsize=15,view=[0..100,0..5000]);
```

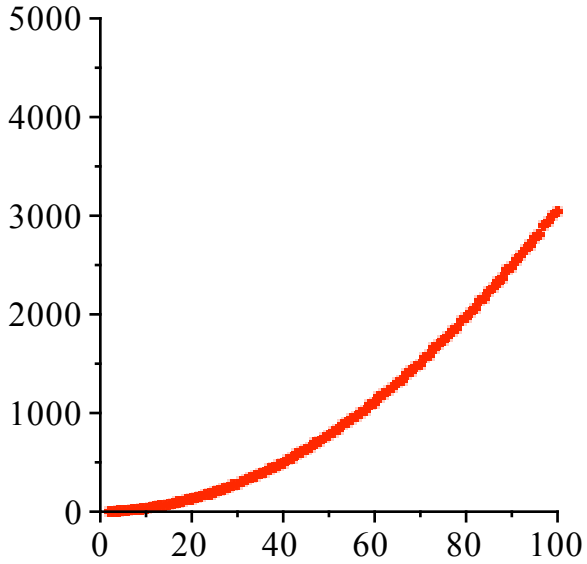


To find the actual values of $F(n)$, we need the size of the list returned by **ListRationals** applied to n. In other words, we apply **nops** to the result of **ListRationals(n)**. Again, we use **seq** to form the list of these counts.

```
> actualValues := [seq(nops(ListRationals(n)),n=2..100)]:
```

Again, we plot.

```
> plot(xValues,actualValues,style=point,symbol=solidcircle,
  symbolsize=15,view=[0..100,0..5000]);
```

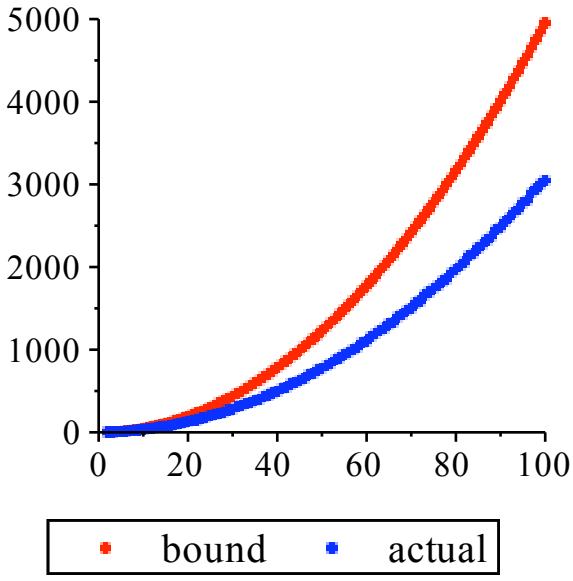


Since we used the same **view** for both graphs, you can immediately see that the value of $F(n)$ is much smaller than the upper bound. We can make the comparison easier by overlaying the graphs. We do this as follows. First, we assign the graphs to names. (We'll also change the color of the graph of the $F(n)$ data to blue.)

```
> boundPlot:= plot(xValues,boundValues,style=point,symbol=
  solidcircle,symbolsize=15,view=[0..100,0..5000],legend=
  "bound"):
> actualPlot := plot(xValues,actualValues,style=point,symbol=
  solidcircle,symbolsize=15,view=[0..100,0..5000],color=blue,
  legend="actual"):
```

Then we have Maple draw the two plots together using the **display** command in the **plots** package.

```
> plots[display](boundPlot,actualPlot);
```



You will explore $F(n)$ and the upper bound $\frac{n(n-1)}{2}$ further in the exercises.

2.6 Matrices

Maple provides extensive support for calculating with matrices. We'll begin this section by describing a variety of ways to construct matrices in Maple. Then we'll consider matrix arithmetic and operations on zero-one matrices.

Constructing Matrices

Matrices are constructed using the **Matrix** command. This command can be used in several different forms and with a large variety of options, only some of which we will discuss here. For complete information, refer to the Maple help page.

Specifying entries by listing the rows

The simplest way to construct a matrix in Maple is by representing each row as a list.

```
> m1 := Matrix([[1,2,3],[4,5,6]]);
```

$$m1 := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad \textbf{(2.180)}$$

In the above example, we passed one argument to the **Matrix** command: a list of lists where the inner lists are the rows of the matrix. This list of lists is referred to as the matrix initializer.

You can also explicitly set the size of the matrix by giving the number of rows and columns as arguments to the **Matrix** command.

```
> m2 := Matrix(2,3,[[1,2,3],[4,5,6]]);
```

$$m2 := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \tag{2.181}$$

Note that if the specified dimension is smaller than what is indicated by the initializer list, an error is generated.

```
> m3 := Matrix(2,2,[[1,2,3],[4,5,6]]);
Error, (in Matrix) initializer defines more columns (3) than
column dimension parameter specifies (2)
```

But if the specified dimension is larger, Maple will create the matrix of the desired size and fill the rest of the entries with 0s.

```
> m4 := Matrix(3,4,[[1,2,3],[4,5,6]]);
```

$$m4 := \begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & 5 & 6 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.182}$$

Also, if only one dimension is given, Maple assumes that a square matrix of that size is desired.

```
> m5 := Matrix(4,[[1,2,3],[4,5,6]]);
```

$$m5 := \begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & 5 & 6 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.183}$$

You can have Maple pad the matrix with a different value by using the optional **fill=value** argument. Below, we create a square matrix of dimension 3 whose entries are all 5.

```
> m6 := Matrix(3,fill=5);
```

$$m6 := \begin{bmatrix} 5 & 5 & 5 \\ 5 & 5 & 5 \\ 5 & 5 & 5 \end{bmatrix} \tag{2.184}$$

The initializer does not have to be a list of lists as in the previous examples. For instance, you can use another matrix, as in the example below where we expand the **m6** matrix.

```
> m7 := Matrix(3,4,m6,fill=2);
```

$$m7 := \begin{bmatrix} 5 & 5 & 5 & 2 \\ 5 & 5 & 5 & 2 \\ 5 & 5 & 5 & 2 \end{bmatrix} \tag{2.185}$$

Modifying entries
Once a matrix has been created, its entries can be altered by assigning the new value to the specified location, with the square bracket selection notation used to indicate the desired location.

For example, to change the lower left entry of matrix *m6* to 6, you would enter the following command.

```
> m6[3,1] := 6;
```

$$m6_{3,1} := 6 \tag{2.186}$$

Note that Maple reports the assignment. To see that it has happened, we have to explicitly command Maple to display the entire matrix.

```
> m6;
```

$$\begin{bmatrix} 5 & 5 & 5 \\ 5 & 5 & 5 \\ 6 & 5 & 5 \end{bmatrix} \tag{2.187}$$

Copying matrices
Using a matrix as the initializer for **Matrix** is commonly used to make a copy of a matrix. Consider the following sequence of commands.

```
> m7copy := m7;
```

$$m7copy := \begin{bmatrix} 5 & 5 & 5 & 2 \\ 5 & 5 & 5 & 2 \\ 5 & 5 & 5 & 2 \end{bmatrix} \tag{2.188}$$

```
> m7[1,2] := 11;
```

$$m7_{1,2} := 11 \tag{2.189}$$

```
> m7;
```

$$\begin{bmatrix} 5 & 11 & 5 & 2 \\ 5 & 5 & 5 & 2 \\ 5 & 5 & 5 & 2 \end{bmatrix} \tag{2.190}$$

```
> m7copy;
```

$$\begin{bmatrix} 5 & 11 & 5 & 2 \\ 5 & 5 & 5 & 2 \\ 5 & 5 & 5 & 2 \end{bmatrix} \tag{2.191}$$

Observe that the modification we made to the **m7** matrix was also made in the **m7copy** matrix. This is because the assignment **m7copy := m7;** did not create a new copy of the matrix to store in **m7copy**. Instead, that assignment made both names refer to the same matrix. The assignment **m7[1,2] := 11;** modified the row 1, column 2 entry of the unique matrix that both names refer to. In computer science, this is called a "reference type," meaning that the name does not store the object, it stores a reference to the object. Assigning one name to another makes a copy of the reference, but both references refer to the same underlying object. Both matrices and tables are reference types in Maple.

To make a true copy of a matrix, you use the **Matrix** command with the matrix you want to copy as the initializer.

```
> m7realcopy := Matrix(m7);
```

$$m7realcopy := \begin{bmatrix} 5 & 11 & 5 & 2 \\ 5 & 5 & 5 & 2 \\ 5 & 5 & 5 & 2 \end{bmatrix} \quad \textbf{(2.192)}$$

```
> m7[2,4] := 23;
```

$$m7_{2,4} := 23 \quad \textbf{(2.193)}$$

```
> m7, m7copy, m7realcopy;
```

$$\begin{bmatrix} 5 & 11 & 5 & 2 \\ 5 & 5 & 5 & 23 \\ 5 & 5 & 5 & 2 \end{bmatrix}, \begin{bmatrix} 5 & 11 & 5 & 2 \\ 5 & 5 & 5 & 23 \\ 5 & 5 & 5 & 2 \end{bmatrix}, \begin{bmatrix} 5 & 11 & 5 & 2 \\ 5 & 5 & 5 & 2 \\ 5 & 5 & 5 & 2 \end{bmatrix} \quad \textbf{(2.194)}$$

While the modification of **m7** altered both **m7** and **m7copy**, **m7realcopy** was unchanged.

Other ways to initialize matrices

Another common way to create a matrix is by specifying the values with a single list rather than a list of lists. In this case, you must provide the dimension of the matrix, whereas it is optional if the initializer is a list of lists.

```
> m8 := Matrix(2,3,[1,2,3,4,5,6]);
```

$$m8 := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad \textbf{(2.195)}$$

You can also specify a matrix with a table. Note the use of parentheses around the table indices. Also, in this case, the row and column dimensions are required.

```
> m9table := table([(1,2)=5,(1,3)=6,(2,1)=-2]);
```

$$m9table := table([(1,2) = 5, (1,3) = 6, (2,1) = -2]) \quad \textbf{(2.196)}$$

```
> m9 := Matrix(2,3,m9table);
```

$$m9 := \begin{bmatrix} 0 & 5 & 6 \\ -2 & 0 & 0 \end{bmatrix} \quad \textbf{(2.197)}$$

Finally, you can initialize the matrix with a procedure. The procedure must accept two integers as arguments. The entries of the matrix are obtained by evaluating the procedure at the row and column number. Below we provide an example using a functional operator to make the entries in the matrix equal to the sum of the row and column numbers and then construct the 4 × 4 matrix from it.

```
> m10F := (i,j) -> i + j;
```

$$m10F := (i,j) \rightarrow i + j \quad \textbf{(2.198)}$$

```
> m10 := Matrix(4,m10F);
```

(2.199)

$$m10 := \begin{bmatrix} 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \\ 5 & 6 & 7 & 8 \end{bmatrix} \quad (2.199)$$

Matrix Arithmetic

The textbook defines addition and multiplication of matrices. Maple implements these operations on matrices in a fairly intuitive way. To add two matrices, you use the + operator, as you would expect.

```
> m11 := Matrix([[1,2,3],[4,5,6]]);
```

$$m11 := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad (2.200)$$

```
> m12 := Matrix([[-2,3,-1],[1,5,2]]);
```

$$m12 := \begin{bmatrix} -2 & 3 & -1 \\ 1 & 5 & 2 \end{bmatrix} \quad (2.201)$$

```
> m11 + m12;
```

$$\begin{bmatrix} -1 & 5 & 2 \\ 5 & 10 & 8 \end{bmatrix} \quad (2.202)$$

Maple's syntax for multiplying a matrix by a scalar is also intuitive.

```
> 3*m12;
```

$$\begin{bmatrix} -6 & 9 & -3 \\ 3 & 15 & 6 \end{bmatrix} \quad (2.203)$$

This produces the matrix whose entries are three times the entries of **m12**.

Matrix multiplication is computed with the . (dot) operator instead of an asterisk. This is to emphasize that matrix multiplication is not commutative.

```
> m13 := Matrix([[3,6,11,1],[-2,5,2,0],[4,8,9,-3]]);
```

$$m13 := \begin{bmatrix} 3 & 6 & 11 & 1 \\ -2 & 5 & 2 & 0 \\ 4 & 8 & 9 & -3 \end{bmatrix} \quad (2.204)$$

```
> m14 := Matrix([[2,5],[1,-2],[3,7],[-1,0]]);
```

$$m14 := \begin{bmatrix} 2 & 5 \\ 1 & -2 \\ 3 & 7 \\ -1 & 0 \end{bmatrix} \quad (2.205)$$

```
> m13 . m14;
```

(2.206)

$$\begin{bmatrix} 44 & 80 \\ 7 & -6 \\ 46 & 67 \end{bmatrix} \tag{2.206}$$

Transposes, Powers, and Equality of Matrices

Positive powers of matrices in Maple work exactly as you would expect, with the caret symbol, provided, of course, that the matrix is square.

```
> m15 := Matrix([[1,2,3],[4,5,6],[7,8,9]]);
```

$$m15 := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \tag{2.207}$$

```
> m15^5;
```

$$\begin{bmatrix} 121824 & 149688 & 177552 \\ 275886 & 338985 & 402084 \\ 429948 & 528282 & 626616 \end{bmatrix} \tag{2.208}$$

The transpose of a matrix is most easily computed with the ^+ operator, as follows.

```
> m14^+;
```

$$\begin{bmatrix} 2 & 1 & 3 & -1 \\ 5 & -2 & 7 & 0 \end{bmatrix} \tag{2.209}$$

Alternately, you can use the **Transpose** command, but this requires the **LinearAlgebra** package.

```
> LinearAlgebra[Transpose](m14);
```

$$\begin{bmatrix} 2 & 1 & 3 & -1 \\ 5 & -2 & 7 & 0 \end{bmatrix} \tag{2.210}$$

Finally, we need to make a note about equality of matrices. For matrices, = does not test that two matrices are equal in the usual sense. For example, observe that the commands below do not yield the expected result.

```
> m16 := Matrix([[1,2],[3,4]]);
```

$$m16 := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \tag{2.211}$$

```
> m17 := Matrix([[1,2],[3,4]]);
```

$$m17 := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \tag{2.212}$$

```
> evalb(m16=m17);
```

$$false \tag{2.213}$$

In fact, what = tests for in the case of matrices is whether or not two names are referring to the same matrix object. This is another case in which being a reference type makes things a bit different.

To test for equality of matrices in the mathematical sense, it is necessary to use the **Equal** command in the **LinearAlgebra** package.

```
> LinearAlgebra[Equal](m16,m17);
```

$$true \tag{2.214}$$

Zero-One Matrices

With Maple, we can create and manipulate zero-one matrices as well. In particular, we'll consider how to compute the meet, join, and Boolean product of zero-one matrices. Unlike matrix addition and multiplication, Maple does not have built-in commands for these computations, so we'll need to create our own procedures.

Introducing Bits package commands

Calculation of meet, join, and the Boolean product require the use of the **and** and **or** bit operations. In the previous chapter we created our own **AND** procedure as a way to explore some fundamental programming constructs. In this section, we'll instead make use of Maple's **Bits** package. This package provides several commands related to performing operations on bits and bit strings. We'll only make use of two commands: **And** and **Or**. First we load the package.

```
> with(Bits):
```

The **And** and **Or** commands take two arguments and return the bit-wise $\wedge$ or $\vee$, respectively.

```
> And(1,1);
```

$$1 \tag{2.215}$$

```
> And(0,1);
```

$$0 \tag{2.216}$$

```
> Or(0,1);
```

$$1 \tag{2.217}$$

```
> Or(0,0);
```

$$0 \tag{2.218}$$

(Note: the arguments to **And** and **Or** are not restricted to 0 and 1. They can be any integers and Maple will perform the operation on the bit strings that represent the integers. We will not explore that further here as it is not necessary for the task at hand.)

A type for zero-one matrices

To create a zero-one matrix, we can use the **Matrix** command as usual.

```
> zo1 := Matrix([[1,0,1],[1,1,0],[0,1,0]]);
```

$$zo1 := \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{2.219}$$

As has been mentioned before, it is always a good idea to do type-checking in procedures. This is particularly important for the next procedures that we will write. Since we'll be using the **And** and **Or** commands from **Bits**, and these commands will *not* produce errors if their inputs are integers other than 0 and 1, it is important for our procedures to make sure that their input are zero-one matrices. Otherwise, the procedures may execute on 'bad' inputs and produce nonsense output.

In Maple, a matrix has type **Matrix**, but in this case, we'll want to be more specific. We'll use the following variant type: **'Matrix'(type)**. This allows us to specify the type of the entries that

are allowed in the matrix. For example, `'Matrix'(float)` indicates that the entries in the matrix must all be floating point numbers. Since we want to check for zero-one matrices, we'll use `{0,1}` as the type of entry allowed in the matrix. Remember that braces in a structured type indicate than any of the options inside the braces are acceptable. (Note: the single right quotes delay evaluation so that Maple doesn't consider the word `Matrix` to be the matrix construction command.)

Observe that `zo1` is a zero-one matrix, but `m17` is not.

```
> type(zo1,'Matrix'({0,1}));
```

true **(2.220)**

```
> type(m17,'Matrix'({0,1}));
```

false **(2.221)**

Implementing Join

With those preliminaries completed, we write the join procedure. This procedure will accept two arguments, both of which must be zero-one matrices. The procedure will also need to check the sizes of the matrices. To do this, we use the `RowDimension` and `ColumnDimension` commands from the `LinearAlgebra` package. These commands accept only one argument, the matrix, and return the number of rows or columns, respectively. If the dimensions do not match, an error will be generated.

After confirming that the two matrices are the same size, the procedure will create a new matrix that is the same size as the input matrices. By omitting any initialization information, Maple will automatically fill this matrix with 0s. The procedure will then consider each position in the matrix using two nested for loops to loop through the rows and the columns. For each position, it computes the bitwise **or** of the entries in the original matrices and sets the corresponding entry in the result matrix `R`.

Here is the procedure. Note that we make use of the `uses` statement. This indicates what packages the procedure relies on and ensures that those packages have been loaded, so we can use the short forms of commands.

```
> BoolJoin := proc(A::'Matrix'({0,1}),B::'Matrix'({0,1}))
    local numrows, numcols, R, r, c;
    uses LinearAlgebra, Bits;
    numrows := RowDimension(A);
    numcols := ColumnDimension(A);
    if numrows <> RowDimension(B) or
       numcols <> ColumnDimension(B) then
      error "Input matrices must be of the same size.";
    end if;
    R := Matrix(numrows,numcols);
    for r from 1 to numrows do
      for c from 1 to numcols do
        R[r,c] := Or(A[r,c],B[r,c]);
      end do;
    end do;
    return R;
  end proc:
```

Below we apply this procedure to two example matrices.

```
> zo1;
```

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{2.222}$$

```
> zo2 := Matrix([[1,0,0],[1,1,1],[0,0,0]]);
```

$$zo2 := \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \tag{2.223}$$

```
> BoolJoin(zo1,zo2);
```

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{2.224}$$

We leave the creation of **BoolMeet** to the reader.

Implementing Boolean product

We conclude by implementing the Boolean product. Recall two key points from Definition 9 in Section 2.6. First, the size of the product of an $m \times k$ matrix and an $k \times n$ is $m \times n$ and the product is undefined if the number of columns of the first matrix does not match the number of rows in the second. Second, the (i, j) entry of the product is given by the formula

$$c_{ij} = (a_{i1} \wedge b_{1j}) \vee (a_{i2} \wedge b_{2j}) \vee \cdots \vee (a_{ik} \wedge b_{kj}).$$

Our Boolean product procedure, **BProduct**, needs to begin by using **RowDimension** and **ColumnDimension** to find the values for m, k, and n and to raise an error if the number of columns of the first matrix does not match the number of rows of the second. Like **BoolJoin**, we create the result matrix **C** to be of the appropriate size and allow Maple to fill all the entries with 0.

The main work of the procedure is to loop over all the entries of the result matrix and calculate the appropriate value. We use two nested for loops with index variables **i** and **j** representing the rows and columns of the result matrix. Inside these for loops, we need to implement the formula for c_{ij}.

It will be helpful to consider a specific example:

$$(1 \wedge 0) \vee (0 \wedge 0) \vee (0 \wedge 1) \vee (1 \wedge 1) \vee (0 \wedge 1).$$

Note that **And** and **Or** can accept only two arguments, so we cannot use a single **Or** applied to a sequence of **And**s, as you might hope. Instead, we'll approach this in the following way. First, compute $1 \wedge 0$, the first **and**, and store the result as **c**.

```
> c := And(1,0);
```

$$c := 0 \tag{2.225}$$

Then, update c to be the result of applying **or** to it and the result of the next **and** term.

```
> c := Or(c,And(0,0));
```

$$c := 0 \tag{2.226}$$

And then repeat with each successive **and**.

```
> c := Or(c,And(0,1));
```

$$c := 0 \tag{2.227}$$

```
> c := Or(c,And(1,1));
```

$$c := 1 \tag{2.228}$$

```
> c := Or(c,And(0,1));
```

$$c := 1 \tag{2.229}$$

In terms of the generic formula, we initialize $c = (a_{i1} \wedge b_{1j})$. Then we begin a loop with index, say p, from 2 through k. At each step in the loop, $c = c \vee (a_{ip} \wedge b_{pj})$.

Here is the implementation of `BProduct`.

```
> BoolProduct := proc(A::'Matrix'({0,1}),B::'Matrix'({0,1}))
    local m, k, n, C, i, j, c, p;
    uses LinearAlgebra, Bits;
    m := RowDimension(A);
    k := ColumnDimension(A);
    if k <> RowDimension(B) then
      error "Dimension mismatch.";
    end if;
    n := ColumnDimension(B);
    C := Matrix(m,n);
    for i from 1 to m do
      for j from 1 to n do
        c := And(A[i,1],B[1,j]);
        for p from 2 to k do
          c := Or(c,And(A[i,p],B[p,j]));
        end do;
        C[i,j] := c;
      end do;
    end do;
    return C;
  end proc:
```

We test this procedure on the matrices from Example 8 in the textbook.

```
> Ex8A := Matrix([[1,0],[0,1],[1,0]]);
```

$$Ex8A := \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{2.230}$$

```
> Ex8B := Matrix([[1,1,0],[0,1,1]]);
```

$$Ex8B := \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \tag{2.231}$$

```
> BoolProduct(Ex8A,Ex8B);
```

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} \tag{2.232}$$

Solutions to Computer Projects and Computations and Explorations

Computer Projects 3

Given fuzzy sets A and B, find $\overline{A}$, $A \cup B$, and $A \cap B$ (see preamble to Exercise 63 of Section 2.2).

Solution: We will compute the union and leave complement and intersection to the reader. Recall, from Exercise 64, that the union of fuzzy sets is the fuzzy set in which the degree of membership of an element is the maximum of the degrees of membership of that element in the given sets.

Recall, from the final subsection of Section 2.2 in this manual, that we developed two possible representations of fuzzy sets and the procedures **BitToRoster** and **RosterToBit** to convert between them. We'll design our procedure to accept the roster representation as input and return a roster representation of the union, since this representation is the most natural for humans to interact with. But in implementing the union, it is more natural to work with the fuzzy bit string representation of the sets.

Our **FuzzyUnion** procedure will accept as input two fuzzy sets in the roster representation. It proceeds as follows.

1. Determine the effective universe for the two sets: (a) initialize **U** to **{}**; (b) loop over each element of the first set and add the name of that element to **U**; (c) do the same with the second set. Then **U** will contain the name of all elements appearing in the two sets.
2. Use **RosterToBit** to convert both sets to their fuzzy bit representations.
3. Use the **zip** command with the **max** function on the two fuzzy bit strings obtained from **RosterToBit** — **zip(max,A,B)** produces the list whose elements are the maximums of the corresponding entries in **A** and **B**.
4. Use **BitToRoster** on the result to obtain the roster representation.

Here is the implementation.

```
> FuzzyUnion := proc(A,B)
    local U, e, Abits, Bbits, Cbits, C;
    U := {};
    for e in A do
      U := U union {e[1]};
    end do;
    for e in B do
      U := U union {e[1]};
    end do;
    Abits := RosterToBit(A,U);
    Bbits := RosterToBit(B,U);
    Cbits := zip(max,Abits,Bbits);
    C := BitToRoster(Cbits,U);
    return C;
  end proc:
```

As an example, we will compute the union of the fuzzy sets defined below.

```
> fuzzyA := {["a",0.1],["b",0.3],["c",0.7]};
```

$$fuzzyA := \{["a", 0.1], ["b", 0.3], ["c", 0.7]\} \quad \textbf{(2.233)}$$

```
> fuzzyB := {["a",0.5],["b",0.1],["d",0.2]};
```

$$fuzzyB := \{["a", 0.5], ["b", 0.1], ["d", 0.2]\} \quad \textbf{(2.234)}$$

```
> FuzzyUnion(fuzzyA,fuzzyB);
```

$$\{["a", 0.5], ["b", 0.3], ["c", 0.7], ["d", 0.2]\} \quad \textbf{(2.235)}$$

Procedures for computing intersection and complement are similar and are left as an exercise.

Computer Projects 9

Given a square matrix, determine whether it is symmetric.

Solution: We will create a procedure, **IsSymmetric**, that tests a matrix to see if it is symmetric. Recall that a matrix is symmetric when it is equal to its transpose. So we just need to use the **Equal** command to compare the matrix with the result of applying the **Transpose** command. We will use the **Matrix** type in the argument to have Maple ensure that only matrices are allowed as arguments.

```
> IsSymmetric := proc(M::Matrix)
   LinearAlgebra[Equal](M,LinearAlgebra[Transpose](M));
  end proc:
```

```
> symmetricMatrix := Matrix([[1,2,3],[2,4,5],[3,5,6]]);
```

$$symmetricMatrix := \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix} \quad \textbf{(2.236)}$$

```
> IsSymmetric(symmetricMatrix);
```

$$true \quad \textbf{(2.237)}$$

```
> notSymmetricMatrix := Matrix([[1,2,3],[4,5,6],[7,8,9]]);
```

$$notSymmetricMatrix := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \textbf{(2.238)}$$

```
> IsSymmetric(notSymmetricMatrix);
```

$$false \quad \textbf{(2.239)}$$

Computations and Explorations 2

Given a finite set, list all elements of its power set.

Solution: The **powerset** and **subsets** commands were described in Section 2.1 above. We'll write a procedure independent of these built-in commands in order to see how such commands might be created.

Recall, from Section 2.2 of the text, that sets may be represented by bit strings. In particular, given a set, say $\{a, b, c, d, e\}$ for example, a subset may be represented by a string of 0s and 1s provided an order has been imposed on the set. For example, the string 0, 1, 1, 0, 0 corresponds to the subset $\{b, c\}$. (Refer to the textbook for a complete explanation.)

In terms of subsets, the bit string representation indicates that, for a given set, there is a one-to-one correspondence between subsets and bit strings. This means that we can solve the problem of listing all subsets of a given set by producing all corresponding bit strings.

To create the bit strings, we'll follow the approach used in the procedure **NextTA** from Section 1.3 of this manual. Given any bit string, the next string is obtained by working left to right: if a bit is 1, then it gets changed to a 0. When you encounter a 0 bit, it is changed to a 1 and you stop the process. For example, suppose the current string is

1, 1, 1, 0, 0, 1, 0.

You begin on the left changing the first three 1s to 0s. Then the fourth bit from the left is 0, so this is changed to a 1 and the process stops. The new bit string is

0, 0, 0, 1, 0, 1, 0.

Here is the **NextBitS** (next bit string) procedure. It accepts a bit string and implements the process described above to produce the next bit string. Note that we enforce the type of the input to this procedure using the structured type **list({0,1})** which indicates that the argument to **BitS** must be a list whose elements are 0s and 1s.

```
> NextBitS := proc(BitS::list({0,1}))
    local newBitS, i;
    newBitS := BitS;
    for i from 1 to nops(newBitS) do
      if newBitS[i] = 1 then
        newBitS[i] := 0;
      else
        newBitS[i] := 1;
        return newBitS;
      end if;
    end do;
    return NULL;
  end proc:
> NextBitS([1,1,1,0,0,1,0]);
                          [0, 0, 0, 1, 0, 1, 0]                    (2.240)
```

Next we'll need a way to convert a bit string into a subset of a given set. We can do this using a simplified version of **BitToRoster**. Note that this procedure relies on the fact that Maple imposes an order on sets and that this order is consistent.

```
> BitToSubset := proc(BitS::list({0,1}),S::set)
    local subS, i;
    subS := {};
    for i from 1 to nops(BitS) do
      if BitS[i] = 1 then
        subS := subS union {S[i]};
      end if;
    end do;
    return subS;
  end proc:
> BitToSubset([0,1,1,0,0],{"a","b","c","d","e"});
                              {"b", "c"}                           (2.241)
```

Finally, we combine these two procedures to produce the subsets of a given set.

```
> Subsets := proc(S::set)
    local BitS;
    BitS := [0 $ nops(S)];
    while BitS <> NULL do
      print(BitToSubset(BitS,S));
      BitS := NextBitS(BitS);
    end do;
    return NULL;
  end proc:
```

Recall that **0 $ nops(S)** produces a sequence of **nops(S)** 0s.

We apply our procedure to $\{a, b, c\}$ to confirm that it is functioning properly.

```
> Subsets({"a","b","c"});
                                  { }
```

{"a"}
{"b"}
{"a", "b"}
{"c"}
{"a", "c"}
{"b", "c"}
{"a", "b", "c"} **(2.242)**

Exercises

Exercise 1. Write a procedure **AreDisjoint** that accepts two sets as arguments and returns true if the sets are disjoint and false otherwise.

Exercise 2. Write a procedure, **Cartesian**, to compute the Cartesian product of two sets as a single set.

Exercise 3. Write procedures **FuzzyIntersection** and **FuzzyComplement** to complete Computer Project 3.

Exercise 4. Write procedures for computing the complement, union, intersection, difference, and sum for multisets. Represent a multiset as a set of pairs **[a,m]** where **m** is the multiplicity of the element **a**. (Refer to the preamble to Exercise 61 in Section 2.2 for information about multisets.)

Exercise 5. Write procedures to compute the image of a finite set under a function. Create one procedure for functions defined as a procedure or a functional operator and a second procedure for functions defined via tables.

Exercise 6. Write a procedure to find the inverse of a function defined by a table.

Exercise 7. Write a procedure to find the composition of functions defined by tables.

Exercise 8. Use computation to discover what the largest value of n is for which $n!$ has fewer than 1000 digits. (Hint: the **length** command applied to an integer will return the number of digits of the integer.)

Exercise 9. Write a procedure **ArithmeticSeq**, similar to **GeometricSeq** from above, that produces an arithmetic sequence.

Exercise 10. Find the first 20 terms of the sequences defined by the recurrence relations below

a) $a_n = 2\,a_{n-1} + 3\,a_{n-2}$, with $a_1 = 1$ and $a_2 = 0$.

b) $a_n = a_{n-1} + na_{n-2} + n^2 a_{n-3}$, with $a_1 = 1$, $a_2 = 1$, and $a_3 = 3$.

c) $a_n = a_{n-1} \cdot a_{n-2} + 1$, with $a_1 = a_2 = 1$.

Exercise 11. The Lucas numbers satisfy the recurrence $L_n = L_{n-1} + L_{n-2}$ and the initial conditions $L_1 = 2$ and $L_2 = 1$. Use Maple to gain evidence for conjectures about the divisibility of Lucas numbers by different integer divisors.

Exercise 12. Write a procedure to find the first n Ulam numbers and use the procedure to find as

many Ulam numbers as you can. (Ulam numbers are defined in Exercise 28 of the Supplemental Exercises for Chapter 2.)

Exercise 13. Use Maple to find formulae for the sum of the nth powers of the first k positive integers for n up to 10.

Exercise 14. The calculation of `actualValues` above is very inefficient, because Maple must calculate the entire list of rational numbers for each value from 2 to 100. Create a new procedure, `listActuals`, that accepts a maximum stage as input and returns the list whose entries are the values of $F(n)$. You can do this by modifying `ListRationals` so that at the completion of each stage, the size of `L`, i.e., the value $F(n)$, is recorded in a list.

Exercise 15. Find a value R so that the graph of $R \cdot \dfrac{n(n-1)}{2}$ is just above the graph of $F(n)$. Use your `listActuals` procedure to expand the data and refine the value of R.

Exercise 16. Use Maple to find the hundredth positive rational number in the list generated by `ListRationals`. What about the thousandth? Ten thousandth? (If you completed it, the result of the previous exercise can be helpful.)

Exercise 17. Write a procedure `BoolMeet` implementing the Boolean meet operation on zero-one matrices.