

Preface

This book is intended to supplement Ken Rosen's *Discrete Mathematics and Its Applications, Eighth Edition*, published by McGraw-Hill. It was developed with Maple 2018, created by Waterloo Maple Inc. This is intended to be a guide as you explore concepts in discrete mathematics and to provide you with tools you can use to investigate further on your own. This text can significantly enhance a traditional course in discrete mathematics in several ways. First, it makes a plethora of examples readily available that you can interact with easily. Second, it makes the notion of algorithm, which is central in discrete mathematics, concrete by giving you the opportunity to actually implement algorithms rather than only analyze them in the abstract. Finally, and most significantly, it provides you greater freedom to make conjectures and experiment without getting bogged down in repetitive calculation.

The focus of this manual is on Maple code and does not attempt to explain discrete mathematics. It is expected that you are taking, or have taken, a course in discrete mathematics. Ideally, you have access to Ken Rosen's *Discrete Mathematics and Its Applications*. It is not assumed that you have any prior experience with Maple. The introductory chapter that follows this preface is designed to introduce you to Maple. Likewise, it is not assumed that you have any experience with computer programming languages. Part of the Introduction is devoted to the basic concepts and techniques of computer programming. Subsequent chapters gradually introduce increasingly sophisticated programming ideas. While this is not a textbook on computer programming, you will likely find yourself fairly comfortable with the basics of programming by the end.

With the exception of the Introduction, the structure of this book follows that of *Discrete Mathematics and Its Applications*. For each section of each chapter in that text, this manual contains a corresponding section describing Maple commands and providing Maple procedures that are used to explore the mathematics topics in that section. Each chapter also contains solutions to some of the Computer Projects and Computations and Explorations exercises found at the end of the chapter of *Discrete Mathematics and Its Applications*. You will also find a number of exercises at the conclusion of each chapter designed to suggest additional questions that you can explore using Maple.

This manual strikes a balance between describing existing Maple commands and creating new procedures that extend Maple's capabilities for exploring discrete mathematics. For example, Maple does not currently include the capability of calculating with pseudographs. Therefore, in Chapter 10, in addition to describing Maple's capabilities for modeling graphs, we also write procedures relating to pseudographs. Some readers may not be interested in the detailed descriptions of how new procedures and programs like these are created. However, even if you are not interested in the programming aspect, the procedures we create are still available to you to explore those topics.

Much has happened since the first edition of this text was written. Rosen's text has undergone five revisions and there have been 18 major releases of Maple (it was at Release 4 when the first edition was published). As a result, this manual has undergone extensive revision, and has also been substantially expanded and reorganized. At the same time, this manual retains the spirit and goals of the original *Exploring Discrete Mathematics with Maple*. We therefore reproduce the preface of the original book below.

Changes in the New Edition

The previous version of this manual was written for the seventh edition of Ken Rosen's *Discrete Mathematics and Its Applications* and was developed with Maple Release 15. The current version includes significant revisions and updates to reflect the revisions in the eighth edition of the textbook and the improvements in the 2018 version of Maple. Some of the most notable revisions include:

- Exposition and programming examples have been improved, with a focus on simplifying and clarifying both to help you more easily understand connections to the mathematics content. Moreover, the Explore function is illustrated at multiple places in the text to illustrate how Maple can be used to easily create interactive elements for exploring concepts and making conjectures.
- Additional examples have been added to reflect new content in the eighth edition of *Discrete Mathematics and Its Applications*, including solving the n -Queens problem via satisfiability, implementing the naive string matching algorithm, and illustrating homomorphic encryption.
- Maple commands that have been improved or added to the system since the last version of the manual have been incorporated. In some cases, the improvements to the built-in functions have made it no longer necessary to develop functions within this manual to fill gaps in Maple's capabilities. For example, Maple's built-in function for finding graph isomorphisms can now handle directed graphs, making the procedure provided in the previous version of this manual unnecessary and making room for procedures for visualizing the isomorphisms instead. Several "from scratch" procedures duplicating built-in capabilities remain when doing so illustrates important mathematics content or programming techniques.
- Data structures introduced since the last version of the manual, including the DataFrame object and MultiSet structure, are discussed in relation to appropriate content. In particular, the built-in function for producing truth tables produces a DataFrame by default, and the MultiSet structure can be used to represent both multi- and fuzzy-sets. Generally speaking, however, preference is given to more fundamental data structures more universally present across computer algebra system and programming languages.
- Deprecated commands and packages have been replaced by their newer equivalents; most notably the numtheory package has been replaced by NumberTheory.

Acknowledgments

I am deeply grateful to Ken Rosen for having trusted me with this work and for his wisdom and guidance. I am indebted to the authors of the original *Exploring Discrete Mathematics with Maple* for providing an excellent foundation on which to build.

I also wish to thank Nora Devlin, the Product Developer at McGraw-Hill Higher Education for *Discrete Mathematics and Its Applications*, eighth edition, for her patience and confidence.

Thanks also to those who have provided feedback on the previous version.

I am grateful to Martin Erickson for his mentorship. To Daniel Baack, Jason Beckfield, Elizabeth Davis-Berg, Julie Minbiolet, Christopher Shaw, Michael Welsh, and Heather Minges Wols for their constant support and encouragement. Finally, I am always grateful to my parents for all they have done.

Daniel R. Jordan

djordan@colum.edu

Preface to the First Edition

This book is a supplement to Ken Rosen's text *Discrete Mathematics and Its Applications, Third Edition*, published by McGraw-Hill. It is unique as an ancillary to a discrete mathematics text in that its entire focus is on the computational aspects of the subject. This focus has allowed us to cover extensively and comprehensively how computations in the different areas of discrete mathematics can be performed, as well as how results of these computations can be used in explorations. This book provides a new perspective and a set of tools for exploring concepts in discrete mathematics, complementing the traditional aspects of an introductory course. We hope the users of this book will enjoy working with it as much as the authors have enjoyed putting this book together.

This book was written by a team of people, including Stan Devitt, one of the principle authors of the Maple system and Eithne Murray who has developed code for certain Maple packages. Two other authors, Troy Vasiga and James McCarron, have mastered discrete mathematics and Maple through their studies at the University of Waterloo, a key center of discrete mathematics research and the birthplace of Waterloo Maple Inc.

To effectively use this book, a student should be taking, or have taken, a course in discrete mathematics. For maximum effectiveness, the text used should be Ken Rosen's *Discrete Mathematics and Its Applications*, although this volume will be useful even if this is not the case. We assume that the student has access to Maple, Release 3 or later. We have included material based on Maple shareware and on Release 4 with explicit indication of where this is done. (Where to obtain Maple shareware is described in the Introduction.) We do not assume that the student has previously used Maple. In fact, working through the book can teach students Maple while they are learning discrete mathematics. Of course, the level of sophistication of students with respect to programming will determine their ability to write their own Maple routines. We make peripheral use of calculus in this book. Although all places where calculus is used can be omitted, students who have studied calculus will find this material of interest.

This volume contains a great deal of Maple code, much based on existing Maple functions. But substantial extensions to Maple can be found throughout the book; new Maple routines have been added in key places, extending the capabilities of what is currently part of Maple. An excellent example is new Maple code for displaying trees, providing functionality not currently part of the network package of Maple. All the Maple code in this book is available over the Internet; see the Introduction for details.

This volume contains an Introduction and 10 chapters. The Introduction describes the philosophy and contents of the chapters and provides an introduction to the use of Maple, both for computation and for programming. This chapter is especially important to students who have not used Maple before. (More material on programming with Maple is found throughout the text, especially in Chapters 1 and 2.) Chapters 1 to 10 correspond to the respective chapters of *Discrete Mathematics and Its Applications*. Each chapter contains a discussion of how to use Maple to carry out computation on the subject of that chapter. Each chapter also contains a discussion of the Computations and Explorations found at the end of the corresponding chapter of *Discrete Mathematics and Its Applications*, along with a set of exercises and projects designed for further work.

Users of this book are encouraged to provide feedback, either via the postal service or the Internet. We expect that students and faculty members using this book will develop material that they want to share with others. Consult the Introduction for details about how to download Maple software

associated with this book and for information about how to upload your own Maple code and worksheets.

Acknowledgments

Thanks go to the staff of the College Division of McGraw-Hill for providing us with the flexibility and support to put together something new and innovative. In particular, thanks go to Jack Shira, Senior Sponsoring Editor, and Maggie Rogers, Senior Associate Editor, for their strong interest, enthusiasm, and frequent inquiries into the status of this volume, and to Denise Schanck, Publisher, for her overall support. Thanks also goes to the production department of McGraw-Hill for their able work.

We would also like to express thanks to the staff of Waterloo Maple Inc. for their support of this project. In particular, we would like to thank Benton Leong and Ha Quang Le for their suggestions. Furthermore, we offer our appreciation to Charlie Colbourn of the University of Waterloo for helping bring this working team together as well as for his contributions as one of the authors of Maple's networks package which is heavily used in parts of this book.

As always, one of the authors, Ken Rosen, would like to thank his management at AT&T Bell Laboratories, including Steve Nurenberg, Ashok Kuthyar, Hrair Aldermishian, and Jim Day, for providing the environment and the resources that have made this book possible. Another author, Troy Vasiga, would like to thank his wife for her encouragement and support during the preparation of this book.

Introduction

Modern mathematical computation software, such as Maple, allow us to carry out complicated computations quickly and easily. As a supplement to traditional exercises solved by hand, having computational tools available while learning discrete mathematics provides a new dimension to the learning experience. Specifically, Maple supports an enquiry and experimental approach to learning. This book is designed to connect the traditional approach to learning discrete mathematics with this experimental approach.

Using computational software, students can experiment directly with many objects that are important in discrete mathematics. These include sets, large integers, combinatorial objects, graphs, and trees. Furthermore, by using interactive computational software to do this, students can explore these examples more thoroughly, fostering a deeper understanding of concepts, applications, and problem-solving techniques.

This supplement has two main goals. The first is to help students learn how to carry out computations in discrete mathematics using Maple. The second is to be a guide and a model as students discover mathematics with the use of computational tools.

This book is intended for use by any student of discrete mathematics. No previous familiarity with Maple is required. Likewise, we do not assume any previous experience with computer programming. The fundamentals of Maple and the basic concepts of computer programming will be thoroughly explained as they are needed.

Structure of This Manual

This supplement begins with a brief introduction to Maple, its capabilities, and its use. The material in this introductory chapter explains the philosophy behind working with Maple, how to use Maple

to carry out computations, and its basic structure. This introduction continues by explaining the basic concepts and syntax for programming with Maple. This will provide those who are new to Maple and programming languages the background they will need in the rest of the book.

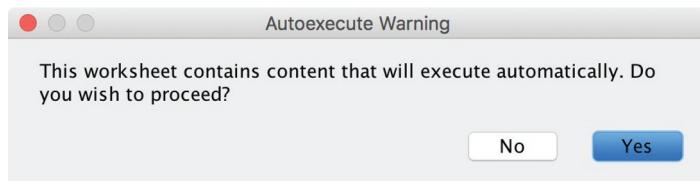
Following the introduction, the main body of this book contains 13 chapters. Each chapter parallels a chapter of *Discrete Mathematics and Its Applications, Eighth Edition*, by Kenneth H. Rosen (henceforth referred to as the text or the textbook). Each chapter includes comprehensive coverage explaining how Maple can be used to explore the topics of the corresponding chapter of the text. This includes a discussion of relevant Maple commands, many new procedures written expressly for this book, and examples illustrating how to use Maple to explore topics in the text.

Additionally, we discuss selected *Computer Projects* and *Computations and Explorations* from the corresponding chapter of the text. We provide guidance, partial solutions, or complete solutions to these exercises. A similar philosophy governs the inclusion of these solutions as does the inclusion of answers to selected exercises in the back of most mathematics textbooks. You should attempt the problem on your own first. The solutions in this manual are intended to be referred to: after you have succeeded in solving a problem to see a (potentially) different approach, when you have stopped making progress on your own and need a slight boost to continue, or when you are trying to solve a similar problem.

Finally, each chapter concludes with a set of additional questions for you to explore. Some of these are straightforward computational exercises, while others are more accurately described as projects requiring substantial additional work, including programming.

The chapters of this manual are available in two formats: as a PDF document and as a Maple Worksheet. The PDF format contains all of the text and Maple commands and other information that you need. The Maple Worksheet version of the chapter includes additional features, specifically active Maple code and links to Maple help pages. It is recommended that you primarily work with the Maple Worksheet version of this manual, and use the PDF version for when you do not have access to Maple.

When you first open the Maple Worksheet version of a chapter, a dialog box should pop up telling you that the worksheet contains content that will automatically execute (see image below). It asks whether you want to proceed. We recommend that you choose yes.



This way, the vital commands within the chapter, those that define variables and procedures, are executed for you right when you open the document. If you do not allow the automatic execution to happen, you may run into errors if you try to execute commands that require other statements to have been executed first. Of course, some groups of commands must be executed in order to produce the correct output, so if you encounter errors, you may need to backtrack and execute commands in order.

The main benefit of the Maple Worksheet version of this book is that it is interactive. That is, you can execute the Maple commands demonstrated in the chapter. Even better, you can modify the

example commands so that you can experiment right within the body of the chapter. Additionally, you have immediate access to Maple's help system. Within the text of this manual, Maple commands appear in red and are underlined indicating that clicking on them will open the corresponding Maple help page.

This book has been designed to help students achieve the main goals of a course in discrete mathematics. These goals, as described in the preface of the textbook, are the mastery of mathematical thinking, combinatorial analysis, discrete structures, algorithmic thinking, and applications and modeling. This supplement demonstrates how to use the interactive computational environment of Maple to enhance and accelerate the achievement of these goals.

Interactive Maple

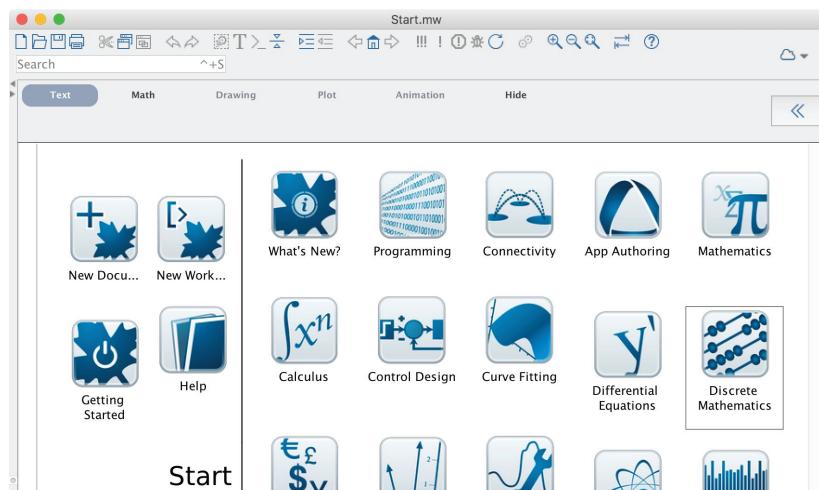
Exploring discrete mathematics with Maple is like exploring a mathematical topic with an expert assistant at your side. As you investigate a topic, you should always be asking questions. In many cases, the answer to your question can be found by experimenting. Maple, your highly trained mathematical assistant, can often carry out these directed experiments quickly and accurately, often with only a few simple instructions.

By hand, the magnitude and quantity of work required to investigate even one reasonable test case may be prohibitive. By delegating the details to Maple, your efforts can be much more focused on choosing the right mathematical questions and on interpreting results. Moreover, with a system such as Maple, the types of objects you are investigating, and tools for manipulating them, already exist as part of the basic infrastructure provided by the system. This includes sets, lists, variables, polynomials, graphs, arbitrarily large integers, rational numbers, and most important, support for exact and fast computations.

The use of Maple is merely a means to the end of achieving the goals of a course in discrete mathematics. As with any tool, to use it effectively you must have some basic understanding of the tool and its capabilities. In this section, we introduce Maple by working through a sample interactive session.

Starting Maple

A new Maple session begins when you start the Maple software. When you start Maple, you generally will see the Maple startup worksheet, which will look something like the window shown below.



The main portion of the startup worksheet displays icons referring to various topics. The left side of the window consists of four useful icons. The lower two are links to a worksheet that itself links to a variety of resources that can help you get started using Maple and a link to the Maple help system.

The top two icons on the left-hand side of the window are the new document options. Selecting one of those icons will create either a new Document or Worksheet. The difference between these is discussed below. To open an existing file, perhaps a file that you created or one of the chapters of this manual, you can select **Open** from the **File** menu or click on the open icon, which is typically the second icon in the toolbar along the top of the Maple window. A standard file selection dialog will open that allows you to select the file you want.

Documents versus Worksheets

This Introduction and, in fact, all the chapters of this manual were created in Document mode. In Document mode, you interact with Maple in similar ways to how you edit a document in a word processing program. You can type text, change the font, insert images, and complete other typical word processing tasks. However, you also have a powerful mathematical engine at your fingertips.

A file in Worksheet mode is more focused on executing Maple commands. When you start a Maple file in Worksheet mode, you immediately see a command prompt. The goal of this manual is to help you explore and learn discrete mathematics, so the execution of Maple commands is the focus. For this reason, it is likely to be more natural to interact with Maple in Worksheet mode. From the start-up worksheet, you can click on “New Worksheet” or select “Worksheet Mode” from the **New** submenu of the main **File** menu. Note that Document mode is the default, so if you click on the new document icon (the first icon in the toolbar at the top of the Maple window), you will be presented with a blank file in Document mode.

However, the difference between Worksheet mode and Document mode is mostly a matter of focus and style. Text blocks can be added in Worksheet mode and command prompts can be added in Document mode. In the toolbar at the top of the Maple window you should see icons like the ones shown below.

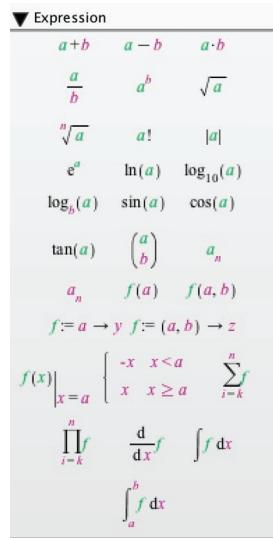


These icons should be about a third of the way along the toolbar, which begins with icons for a new file, open, save, and print. If you do not see them, click on the **View** menu and make sure that a checkmark appears next to **Toolbar**. When you are working in Maple, whether in a Worksheet or a Document, clicking on the capital T icon (or selecting **Text** from the **Insert** menu) will insert a text block immediately after your cursor. Clicking on the icon that looks like a greater than symbol followed by an underline (or selecting one of the **Execution Group** options in the **Insert** menu) will create a command prompt.

The first of the four icons displayed above inserts a Code Edit Region, which is useful when you are writing procedures, as opposed to executing an individual command. In a Code Edit Region, Maple will highlight syntax errors and warn you about other possible issues with your code as you type. In addition, a fixed-width font is used which makes careful checking of syntax easier. The procedures in this manual are entered in Code Edit Regions. The final icon is for entering a new Document Block. These, like the Document mode, are used for mixing text with results of computations to produce the kinds of polished documents and final reports you might see in a business context.

Maple Notation Versus 2-D Math Mode

Along the left side of the main window you may see palettes. If they are not already visible, they can be controlled by the **Expand Dock** or **Collapse Dock** options in the **Palettes** submenu of the **View** menu. Shown below is the Expression palette.



Palettes can be used to insert mathematical formulae and symbols within text blocks. They can also be used to make expressions in 2-D math mode. By default, for both Documents and Worksheets, math is input and output in 2-D math notation. The 2-D notation allows you to use, for example, the integral symbol as part of an expression that you have Maple evaluate. With 2-D notation, you can also access context menus by right-clicking on an expression to instruct Maple to perform certain operations on the expression.

These are nice features, but it is not required to use them. Everything you can do in Maple can be accomplished with plain keyboard input, and all of the commands demonstrated in this manual are presented in the form obtained by simple typing. Indeed, that is how all the commands were entered during the creation of the manual. There are several reasons we chose to use this approach for command entry, but the primary reason is that it is much easier to explain what to do when you just need to type what you see.

However, when working in 2-D input mode, Maple will automatically do some formatting as you type. You will most often notice this when typing exponents (obtained by a caret, or Shift+6) and arrows (obtained from a hyphen followed by a greater-than). For example, consider the input below.

```
> f := x → x3 + 4 · x - 7  
f := x ↪ x3 + 4x - 7
```

(0.1)

Contrast that with the following in 1-D input mode.

```
> f := x -> x3 + 4*x - 7;  
f := x ↪ x3 + 4x - 7
```

(0.2)

Do not worry about what these commands do right now. The point is that their output is exactly the same. And, in fact, they are entered in nearly the same way. The only significant differences are the semicolon and the fact that when entering the command in 2-D mode, you need to press the right

arrow key after entering the 3 to exit the exponent. We also added spaces to make the 1-D input a bit more readable, but they are unnecessary.

In this manual, input is almost always given in 2-D mode, but entered entirely from the keyboard. On those few occasions where Maple's automatic transformation of 2-D input obscures how to enter the command, it is provided in 1-D mode. That said, you can have Maple translate from 2-D mode to 1-D mode for you. Right-click on a 2-D input and in the pop-up menu, select the **2-D Math** menu, then the **Convert To** submenu and then choose **1-D Math Input**. If you do that to the 2-D command above, you will get a bit more than you bargained for, as the conversion will also reveal what the arrow is a shorthand for.

In traditional Maple notation, complete commands end in semicolons (`;`) or colons (`:`). In 2-D input mode, these are not usually required, unless you are entering a sequence of commands in the same input. More will be said about the use of semicolons and colons below, but know that including a semicolon at the end of a command will not produce an error so you can safely include one if you are unsure.

Executing Commands

To execute a command, make sure that the cursor is somewhere on the line containing the command and press the Enter or Return key to execute the command and display the result. It is time to execute your first Maple command. Let us start simple and add two plus three. To do this, make sure your cursor is on a command line and type **2+3** and then press Enter or Return.

```
> 2 + 3  
5
```

(0.3)

Here are a few more commands. Try entering them on your computer.

```
> add (i2, i = 1 ..10)  
385
```

(0.4)

```
> int ((x - 1)3, x)  

$$\frac{(x - 1)^4}{4}$$

```

(0.5)

```
> expand (%)  

$$\frac{1}{4}x^4 - x^3 + \frac{3}{2}x^2 - x + \frac{1}{4}$$

```

(0.6)

The percent symbol (`%`) is used to refer to the result of the most recently executed statement. The percent does not always refer to the value immediately above it, as commands can be executed out of order.

In addition, the line numbers that automatically appear next to the results can be used in commands to refer to specific results. To use a result, it is not enough to type (0.5) in a statement. Instead, you must select **Label...** from the **Insert** menu and enter the number of the label in the dialog box that opens. You can also use the shortcut Control+L (or Command+L on a Mac) to open the dialog. If equation numbers are not appearing when you execute commands, you should turn them

on. In the **Tools** menu, select **Options** (or **Preferences** from the **Maple** menu on a Mac). On the **Display** tab, make sure that the **Show Equation Labels** box is checked.

Try entering the following command.

```
> subs(x = 3, (0.5))  
4
```

(0.7)

A First Encounter with Maple

As already indicated, working with Maple is like working with an expert mathematical assistant. This requires a subtle change in the way you think about a problem. When working on an exercise by hand, your attention is focused on the details and quite often you can lose sight of the “big picture.” Maple takes care of the details for you and frees you to focus on deciding what needs to be done next. This is not to say that the details are not important nor does it imply that you should forgo learning how to solve the problems by hand.

Much of discrete mathematics is about understanding the relationships between objects or sets of objects and using mathematical models to capture some property of these objects. Understanding these relationships often requires that you view either the objects or the associated mathematical model in different ways.

Maple allows you to manipulate the mathematical models almost casually. For example, the polynomial $(x + (x + z)y)^3$ can be entered into Maple as:

```
> (x + (x + z) · y)3  
(x + (x + z) y)3
```

(0.8)

The result of executing this statement is displayed immediately. In this case, Maple simply echoes the polynomial as no special computations were requested.

The power of having a computational tool like Maple is that a wide range of standard operations become immediately available. For example, you can expand, differentiate, and integrate just by telling Maple to do so. Suppose you decided that it would be useful to see the full expansion of the polynomial above. All you need to do is issue the appropriate command to Maple. In this case, the command you would want is the **expand** command, which tells Maple to expand the polynomial.

```
> expand(%)  
x3y3 + 3x2y3z + 3xy3z2 + y3z3 + 3x3y2 + 6x2y2z + 3xy2z2 +  
3x3y + 3x2yz + x3
```

(0.9)

(Recall that Maple uses the percent sign (%) to refer to the output from the previous command.)

Perhaps you decide it would be useful to look at this as a polynomial in the variable x , with the variables y and z placed in the coefficients of x . Then, you would use the **collect** command.

```
> collect(% , x)  
(y3 + 3y2 + 3y + 1)x3 + (3y3z + 6y2z + 3yz)x2 +  
(3y3z2 + 3y2z2)x + y3z3
```

(0.10)

To return to a factored form, simply ask Maple to **factor** the previous result.

```
> factor(%)
(yx + yz + x)3 (0.11)
```

We used several commands without explanation in the above. Rest assured that in the body of this manual we will always provide detailed explanations of the usage and syntax of new commands when we first encounter them. The purpose of the last several paragraphs was not to introduce the commands, but to illustrate how easy it is to quickly move between different representations of the same object. Having these kinds of routine tasks performed quickly and accurately means that you are freer to experiment and explore.

A second very important benefit is that the particular computations that you choose to have Maple execute are performed accurately. Thus, the results you get from your experiments are much more likely to be feedback on the model you had chosen rather than nonsense arising from arithmetic errors.

Finally, the sheer computational power of Maple allows you to run much more extensive experiments and many more of them. This can be important when trying to establish or identify a relationship between a mathematical model and a collection of discrete objects.

It is worth making some comments about terminology and syntax. First, in this manual, we will use the term *command* to refer to **expand**, **collect**, **factor**, and the like. Maple's help documents refer to them as commands or functions, but we will avoid the use of function because of its mathematical meaning. Maple commands will appear in the red Maple notation font and will be underlined indicating that it links to the Maple help documents. On the other hand, programs that we write will be referred to as procedures.

Second, when you use a command on one or more objects, the objects are referred to as arguments. To execute a command, you type its name followed by a pair of parentheses. Inside the parentheses you list the arguments, separated by commas.

```
> max(9, 2, 12, 14, 7, 11)
14 (0.12)
```

Even commands that do not need any arguments require the parentheses. For example, the **time** command returns the total amount of computer time that the current Maple session has used.

```
> time()
3.894 (0.13)
```

The Basics

This section and the next are devoted to introducing you to the most essential Maple commands and concepts that will be used throughout this manual. Some of this material will be repeated, often in more depth, in the first few chapters when the topics arise naturally in conjunction with the content of the textbook. This section is focused on basic commands and the next focuses on programming.

Help

The most important command is the help command. Maple includes extensive documentation on all of its commands, including examples of how the command is used. There are two primary ways to access Maple's help documents. First, you can select **Maple Help** from the **Help** menu or click on the circled question mark in the toolbar. The Maple Help window will open and from there you can browse the table of contents or search for the topic or command you are interested in.

The more typical way to access Maple's help pages is by entering a question mark (?) on a command line. For example, if you need to know the command for computing the square root of a number, you could enter the following.

> ?square root

The Help window will open to the help page for the **sqrt** command, which computes the principle square root of a number or an algebraic expression. Note that you do not need to know the name of the command you are looking for help on. Following the ? with "square root" finds the **sqrt** command. You should try to make your query as simple as possible, though. Often, taking a guess at the name of the command works well. Remember that when commands are discussed in this manual, they appear in red and underlined. These are links to the Maple help documents and clicking on them will open the relevant help page.

Each help page on a Maple command provides several examples of how to use that command. Open the **sqrt** help page now and take a look. If you wish, you can open help pages as interactive worksheets. To do this, from the help system's **View** menu, select **Open Page as Worksheet**, or click on the next-to-last icon on the help system toolbar.

Arithmetic

Maple uses the typical notation for arithmetic. For addition and subtraction, Maple uses + and - just as you would expect. The - symbol is used for negation as well. Multiplication and division are performed with * and /, and ^ is used for exponentiation. Note that in 2-D input mode, typing the caret or slash will move the cursor to an exponent or denominator position and you will need to use the right-arrow key to input the next part of the expression.

Maple obeys the usual order of precedence for arithmetic operators, and parentheses serve as grouping symbols. However, brackets, braces, and angle brackets all have different meanings in Maple and cannot be used as grouping symbols in arithmetic expressions. Therefore, to compute the expression

$$7 + 2 \cdot \left[5 - \left(\frac{2}{3}\pi \right)^2 \right],$$

you would enter the following, using **Pi** for π and parentheses in place of the brackets.

$$\begin{aligned} &> 7 + 2 \cdot \left(5 - \left(\frac{2}{3}\text{Pi} \right)^2 \right) \\ &17 - \frac{8\pi^2}{9} \end{aligned} \tag{0.14}$$

In 1-D mode, this looks like:

$$> 7+2*(5-(2/3*\text{Pi})^2); \\ 17 - \frac{8\pi^2}{9} \quad (0.15)$$

Notice that Maple automatically simplifies the expression, but as an exact value in terms of π . If you prefer a decimal approximation, you can use the **evalf** command (for evaluate floating-point). This command takes one argument, an expression, and evaluates it with floating-point arithmetic.

$$> \text{evalf}(\%) \\ 8.227018307 \quad (0.16)$$

By default, **evalf** computes with 10 significant figures. If you prefer more or fewer significant digits, you can specify the number of digits to use as shown below.

$$> \text{evalf}[3]((0.14)) \\ 8.23 \quad (0.17)$$

$$> \text{evalf}[15]((0.14)) \\ 8.22701831014281 \quad (0.18)$$

Recall that the **%** symbol is used to refer to the previous computation and references to specific output lines can be inserted by clicking on the **Label** item in the **Insert** menu or with the shortcut **Ctrl+L** (**Command+L** on a Mac).

Maple considers integers, fractions, and floating-point numbers to be different and it works with them differently. In expression (0.14), we used only integers and the constant **Pi**. If we had used a floating-point number, Maple would have computed with floating-point arithmetic.

$$> \frac{2}{3} + \frac{3.1}{2} \\ 2.216666667 \quad (0.19)$$

The presence of 3.1 caused Maple to evaluate the entire expression with floating-point arithmetic. You can use this fact to cause Maple to evaluate with floating-point arithmetic even when only integers are involved. Mathematically, there is no difference between 3 and 3.0. Maple treats them differently, however.

$$> \frac{3}{5} \\ \frac{3}{5} \quad (0.20)$$

$$> \frac{3.0}{5} \\ 0.6000000000 \quad (0.21)$$

In fact, the trailing 0 is not required.

$$> \frac{3}{5} \\ 0.6000000000 \quad (0.22)$$

This discussion illustrates the concept of a *type*. A computer can be much more efficient if it knows what kinds of things it will be working with. If the computer knows that one object is going to be a floating-point number while another is going to be a string, it will allocate memory differently for the two objects, for instance.

Types also allows programs such as Maple and programming languages to make use of operator overloading. This means that the symbol $+$ means one thing when applied to two integers, something else when applied to floating-point numbers, and something completely different when applied to matrices. The concept of type is what makes it possible for Maple to figure out which version of $+$ is called for at the time. Maple recognizes over 200 different predefined types and users are free to create more. We will see much more of types as we go forward.

Names, Assignment, and Equality

In mathematics, we talk about variables as symbols that stand in for something else. In Maple, this role is filled by *names*. The simplest definition of a name in Maple is that a name must begin with a letter and may be followed by letters, digits, or the underscore character. The following are all valid Maple names: **evalf, name, x, a15, B_5x_**.

Names can be used as a variable in an algebraic expression as in the following.

$$> 3x^2 + 5x - 7 \\ 3x^2 + 5x - 7 \quad (0.23)$$

Names can also be used to store particular values using the assignment operator. The assignment operator consists of a colon followed by an equals sign ($:=$). To assign a value to a name, you begin with the name, followed by the assignment operator and then the expression that you want stored in the name. For example, to assign the value 12 to the name **twelve**, you type the statement below.

$$> twelve := 12 \\ twelve := 12 \quad (0.24)$$

When a value or other object has been assigned to a name, then any time that name appears in a statement, it is “resolved” to the expression stored in it.

$$> twelve + 5x \\ 12 + 5x \quad (0.25)$$

When Maple encounters an assignment statement, it first evaluates the right-hand side of the statement and then makes the assignment. You can use this fact to modify values as follows.

$$> twelve := 2 * twelve + 1 \\ twelve := 25 \quad (0.26)$$

In the statement above, the right-hand side is evaluated first, meaning that the **twelve** on the right is resolved to its “old” value of 12. Maple then computes $2 \cdot 12 + 1 = 25$ and assigns the value 25 to the name **twelve**, overwriting the value stored earlier.

In Maple parlance, **twelve** is referred to an assigned name, as opposed to an unassigned name. An assigned name, that is a name that has been assigned a value, can be used as an unassigned name by enclosing it in right single quotes. For example,

```
> 2 · 'twelve' + 5x
2 twelve + 5x
```

(0.27)

Right single quotes are used by Maple to delay evaluation of whatever expression they enclose. One important use of this is to unassign a name, as shown below.

```
> twelve := 'twelve'
twelve := twelve
```

(0.28)

After this statement, **twelve** is no longer assigned a value.

```
> twelve
twelve
```

(0.29)

It is important to note that Maple distinguishes between the right and left single quote.

Practically any expression can be assigned to a name, not just numbers. For example, we can assign the algebraic expression $2y + 5x$ to the name **f**.

```
> f := 2y + 5x
f := 2y + 5x
```

(0.30)

Now, every time **f** appears in a statement, it is resolved to this expression.

```
> sqrt(f)
sqrt(2y + 5x)
```

(0.31)

Even an equation can be assigned to a name.

```
> eqn := F =  $\frac{9}{5}C + 32$ 
eqn := F =  $\frac{9}{5}C + 32$ 
```

(0.32)

Observe in the last example the different uses of the assignment operator and the equals sign. Some programming languages use the equals sign for assignment, but Maple reserves the equals sign for mathematical equality. The previous statement assigns the name **eqn** to the mathematical equation $F = \frac{9}{5}C + 32$. Since Maple understands this to be an equation, we can, for instance, solve

it. Given an equation and a name appearing in the equation, the **solve** command solves the equation for the given name. Therefore, we can solve for C as follows.

$$\begin{aligned} > \text{solve}(eqn, C) \\ -\frac{160}{9} + \frac{5F}{9} \end{aligned} \tag{0.33}$$

When you execute the statement above, Maple first resolves **eqn** to the equation $F = \frac{9}{5}C + 32$. It also attempts to resolve **C**, but **C** is an unassigned name. (If **C** were not unassigned, an error would result.) Once the arguments have been evaluated, Maple applies the **solve** command to them.

Basic Types

We mentioned above that Maple recognizes many different types of objects. In this subsection, we will discuss some of the most fundamental types. We will not go into all the details of what it means to be a type in Maple. Our goal in this section is to introduce you to the kinds of objects you will be using and the use of the **type** command.

We have already seen numeric types, such as integers, fractions, and floating-point numbers. Maple has a host of names for numeric types, such as **integer**, **fraction**, **float**, **posint**, **realcons**, and **imaginary**, to name a few.

You can test whether a Maple expression is of a specific type by using the **type** command. This command requires two arguments. The first is the expression you want to test and the second is the type. For example, to see whether or not 2 is a positive integer, a fraction, and a float, you enter the following statements.

$$\begin{aligned} > \text{type}(2, \text{posint}) \\ \text{true} \end{aligned} \tag{0.34}$$

$$\begin{aligned} > \text{type}(2, \text{fraction}) \\ \text{false} \end{aligned} \tag{0.35}$$

$$\begin{aligned} > \text{type}(2, \text{float}) \\ \text{false} \end{aligned} \tag{0.36}$$

In addition to numeric types, Maple has a **string** type for strings of characters. You form a string by enclosing any sequence of characters within a pair of double quotes. For example, Einstein wrote:

$$\begin{aligned} > \text{quotation} := \text{“Pure mathematics is, in its way, the poetry of logical ideas.”} \\ \text{quotation} := \text{“Pure mathematics is, in its way, the poetry of logical ideas.”} \end{aligned} \tag{0.37}$$

Strings may be combined with the concatenation operator, **||**, or with the command **cat**, as demonstrated below.

$$\begin{aligned} > \text{cat}(\text{quotation}, \text{“- Einstein”}) \\ \text{“Pure mathematics is, in its way, the poetry of logical ideas. - Einstein”} \end{aligned} \tag{0.38}$$

$$\begin{aligned} > \text{“abc”} \text{||} \text{“def”} \\ \text{“abcdef”} \end{aligned} \tag{0.39}$$

You can ask Maple what the type of an object is with the **whattype** command. Since many objects satisfy the definitions of multiple types, this command returns the object's basic type.

```
> whattype(2)
integer
```

(0.40)

We will see why types are important in the next section when we discuss basic programming concepts.

Expression Sequences

An expression sequence, or simply sequence, is the fundamental Maple data structure. An expression sequence is formed using commas to separate expressions. For example, the following assigns the expression sequence 7, 8, 9, 10, 11 to the name **S**.

```
> S := 7, 8, 9, 10, 11
S := 7, 8, 9, 10, 11
```

(0.41)

To lengthen a sequence, you use a comma.

```
> S := S, 12
S := 7, 8, 9, 10, 11, 12
```

(0.42)

Selection

The *selection operation* is used to access members of a sequence and subsequences. The first element of the sequence can be obtained by typing the name assigned to the sequence followed by a pair of brackets enclosing the number 1.

```
> S[1]
7
```

(0.43)

You would use 2 to obtain the second element, 3 for the third, and so on.

```
> S[2]
8
```

(0.44)

```
> S[3]
9
```

(0.45)

You may also use negative integers to count from the right. Therefore, -1 refers to the last element of the sequence, -2 to the next to last, and so on.

```
> S[-1]
12
```

(0.46)

```
> S[-2]
11
```

(0.47)

Any expression can be placed inside the brackets, provided it evaluates to an integer that is not 0 and does not exceed the bounds of the sequence (for example, 7 or higher and -7 or lower, in this case).

```
> S[2 · 3 - 22]  
8
```

(0.48)

Ranges

A **range** is a Maple type consisting of two expressions connected by two periods. A common use of a range is in conjunction with the selection operation to extract a subsequence from a sequence. For example, to extract the subsequence consisting of the third through the fifth elements of S , you use the range **3..5** within the brackets.

```
> S[3 ..5]  
9, 10, 11
```

(0.49)

```
> S[-5 .. -3]  
8, 9, 10
```

(0.50)

Note that the left side of the range must be less than or equal to the right side. However, either or both sides may be omitted. If both are omitted, it is interpreted as the entire sequence.

```
> S[ ..]  
7, 8, 9, 10, 11, 12
```

(0.51)

If only one side is given, it is interpreted either as the sequence from the given location onwards,

```
> S[3 ..]  
9, 10, 11, 12
```

(0.52)

or as the sequence up to the given location.

```
> S[ ..5]  
7, 8, 9, 10, 11
```

(0.53)

The **seq** Command and Final Comments about Sequences

The **seq** command is often used to create sequences using a formula. Here we only discuss the most typical way to use **seq**. The command will be discussed more thoroughly in Section 2.4 of this manual.

You should first choose a name, which is referred to as the index variable. The letter **i** is a typical choice. In the simplest form, **seq** requires two arguments. The first argument is any expression, typically one that involves the index variable in its computation. The second argument is an equation with the index variable on the left-hand side and a range on the right-hand side, for example, **i=3..7**. The result of executing the **seq** command is the sequence obtained by evaluating the first argument at each value of the index variable determined by the range in the second argument. For example, we can obtain the squares of the integers between 5 and 11 as follows.

```
> seq (i2, i = 5 ..11)  
25, 36, 49, 64, 81, 100, 121
```

(0.54)

Expression sequences are an important Maple data structure and form the basis for several types including lists and sets, to be discussed below. However, you should be aware that sequences generally cannot be given as the argument to a command or procedure. To understand why, suppose that **ACommand** were a command that accepted only one argument. If you try to execute this command with a sequence, such as **S**, as the argument, Maple first resolves the expression sequence.

```
> ACommand(S)
ACommand(7, 8, 9, 10, 11, 12) (0.55)
```

It looks to Maple like you were passing six arguments to **ACommand** instead of one. That would generate an error if **ACommand** were actually a command. In order to pass a collection of values to a procedure or command, they must be enclosed in a list or a set.

Finally, the empty expression sequence is referred to as **NULL**. A command that results in **NULL** displays no output.

```
> NULL
```

Lists

In Maple, a list is an ordered sequence of expressions. The name of the type in Maple is **list**. You create a list by enclosing an expression sequence, that is, values separated by commas, in brackets.

```
> L := [6, 7, 8, 9, 10, 11]
L := [6, 7, 8, 9, 10, 11] (0.56)
```

Note that the **seq** command can be used to create a list by enclosing it in brackets. Maple computes the sequence and then forms the list based on that sequence.

```
> L2 := [seq(4*i + 3, i = 1 .. 10)]
L2 := [7, 11, 15, 19, 23, 27, 31, 35, 39, 43] (0.57)
```

At first glance, it may appear that the only difference between a list and the expression sequence that defines it is the brackets. Conceptually, the actual difference is that a list can be thought of as a single object. Therefore, unlike a sequence, a list can be passed as an argument to a procedure or command.

Selection

Selection works essentially the same with lists as with sequences. To access a single element of the list, you follow the name of the list with a pair of brackets containing the integer indicating the position of the element. Remember that the first element is 1 and that negative numbers count from the end.

```
> L2[3]
15 (0.58)
```

```
> L2[-2]
39 (0.59)
```

As with sequences, you can select a range as well. The difference is that when used with a list, the result will be a list.

```
> L2[3 .. -2]  
[15, 19, 23, 27, 31, 35, 39] (0.60)
```

The **op** Command

It is sometimes necessary to extract the underlying sequence from a list. This can be done with the **op** (short for operands) command. The most basic form of the **op** command takes one argument, which can be any expression. For lists, this will return the sequence of elements in the list.

```
> op(L2)  
7, 11, 15, 19, 23, 27, 31, 35, 39, 43 (0.61)
```

The **op** command can also accept either an integer or a range as its first argument with the list as the second. In this case, it behaves similar to selection, except **op** returns a sequence while selection returns a list.

```
> op(5, L2)  
23 (0.62)
```

```
> op(..5, L2)  
7, 11, 15, 19, 23 (0.63)
```

With objects other than lists, **op** has a slightly different behavior. For example, for polynomials, **op** will extract the terms.

```
> op(3x^2 - 5x + 7)  
3x^2, -5x, 7 (0.64)
```

We will discuss other uses of **op** in later chapters as they are needed.

Related to **op** is the **nops** (number of operands) command. For lists, **nops** returns the number of elements in the list.

```
> nops(L)  
6 (0.65)
```

```
> nops(L2)  
10 (0.66)
```

We saw that extending a sequence is just a matter of using a comma to continue the sequence. Adding elements to a list is a bit more complicated. Suppose you want to add 47 to the end of **L2**. To do this, you use **op** to extract the sequence of elements from **L2**. Then, add 47 to the sequence with a comma. Turn the extended sequence back into a list by surrounding it with brackets. Finally, reassign the result to the name **L2**.

```
> L2 := [op(L2), 47]  
L2 := [7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47] (0.67)
```

Adding to the beginning of the list is done in essentially the same way.

```
> L2 := [3, op(L2)]  
L2 := [3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47] (0.68)
```

If you want to insert an element in the middle of a list, say after the 5th member of **L2**, you would use **op** with ranges in order to separate the front and back ends of the existing list as follows.

```
> L2 := [op(..5, L2), 21, op(6.., L2)]  
L2 := [3, 7, 11, 15, 19, 21, 23, 27, 31, 35, 39, 43, 47] (0.69)
```

The map Command

We discuss one last command related to lists: the **map** command. This command requires two arguments. The first argument is the name of a command or procedure. The second argument is a list. (Technically, the second argument can be any expression, but we will typically use **map** with a list as the second argument.) The result is the list obtained by applying the command to each element of the list. For example, the statement below produces the list of square roots of the given list.

```
> map(sqrt, [3, 6, 9])  
[sqrt(3), sqrt(6), 3] (0.70)
```

Note that when **map** is applied to a name that has been assigned to a list, the named list is not modified.

```
> map(sqrt, L)  
[sqrt(6), sqrt(7), 2*sqrt(2), 3, sqrt(10), sqrt(11)] (0.71)
```

```
> L  
[6, 7, 8, 9, 10, 11] (0.72)
```

If you want the original list to be updated, you should reassign it.

```
> L := map(sqrt, L)  
L := [sqrt(6), sqrt(7), 2*sqrt(2), 3, sqrt(10), sqrt(11)] (0.73)
```

```
> L  
[sqrt(6), sqrt(7), 2*sqrt(2), 3, sqrt(10), sqrt(11)] (0.74)
```

Note that it is typical for Maple commands to *not* modify their arguments.

For commands that require more than one argument, **map** can accept optional arguments following the list that are then passed to the command. There are also variants such as **map2** and **zip**. The use of additional arguments and these other commands will be discussed as they are needed.

Sets

In mathematics, a set is a collection of objects that is unordered and without repetition. Maple's **set** type models the mathematical notion. To form a set, you enclose a sequence in braces.

```
> {1,2,3}  
{1,2,3} (0.75)
```

```
> S := {seq (i^2, i = 1 .. 10)}  
S := {1, 4, 9, 16, 25, 36, 49, 64, 81, 100} (0.76)
```

Note that repeated elements in a set are automatically removed by Maple.

```
> {1,2,3,2,1}  
{1,2,3} (0.77)
```

This is a useful feature that we will make use of quite often to avoid redundancy.

The **op** command works on sets in the same way as lists to return the sequence that underlies the set, and the **nops** command applied to a set returns the number of elements.

```
> op(S)  
1, 4, 9, 16, 25, 36, 49, 64, 81, 100 (0.78)
```

```
> nops(S)  
10 (0.79)
```

Note that, while sets are technically unordered, Maple actually imposes an order on them. This is done to improve efficiency. Rest assured that the implementation of sets and the commands related to them is done in such a way as to ensure that they behave as mathematical sets should. However, selection works with sets in the same way as lists, as does the **op** command with an integer or range as the first argument.

```
> S[3 .. 7]  
{9, 16, 25, 36, 49} (0.80)
```

```
> op(7, S)  
49 (0.81)
```

Sets will be explored in more detail in Chapter 2.

Printing

There are three main ways to have Maple display the result of a computation. First, of course, is to execute a command that displays a result.

```
> 2 + 3  
5 (0.82)
```

On the other hand, sometimes you may wish for a result to not be displayed. In this case, you end the statement with a colon instead of a semicolon.

```
> 2 + 3 : 
```

The second way to get Maple to display information is the **print** command. This is often used within a procedure to display the results of intermediate calculations before the final result is computed and displayed. The **print** command accepts as its input a sequence of expressions and displays them on a line. For example,

```
> print(L)
```

```
[\sqrt{6}, \sqrt{7}, 2\sqrt{2}, 3, \sqrt{10}, \sqrt{11}]
```

(0.83)

```
> print(S, L2)
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100} , [3, 7, 11, 15, 19, 21, 23, 27, 31, 35, 39, 43, 47] (0.84)
```

The third way to have Maple display output is **printf**. (There are three related commands that work in similar ways with slightly different purposes, but we will only discuss **printf**.)

The purpose of **printf** is to print expressions using a particular format (hence the f) that you specify. The first argument to the command is a string that details the format in which the information is to be displayed. The remaining arguments are the information to be displayed. Here is an example of using **printf** to display a number and its square with each number displayed with room for at least 5 digits.

```
> printf("The square of %5a is %5a.\n", 7, 7^2)
```

```
The square of 7 is 49.
```

The **%** symbol is used to indicate the beginning of the formatting specification **%5a**. The 5, which is optional, specifies that the width of that field is to be at least 5 characters. This is a useful option for displaying a table or otherwise ensuring that displayed values are aligned. Finally, the **a**, for anything, tells Maple to display the corresponding object in whatever format it normally would. Other letters can be used that are specific to different types of objects to display, such as integers and strings. The **\n** at the end of the formatting string indicates that a new line should be inserted at that point. Following the formatting string, are the two values **7** and **7^2**. Notice that the first value is put in place of the first formatting specification (the first **%5a**) and the second goes in place of the second.

The **printf** command is very flexible with a great many options. The interested reader should refer to the Maple help page for further information. In this manual, we will use **printf** rarely and we will not discuss it further here.

Programming Preliminaries

This section is intended for those readers who have little or no previous exposure to programming. We will endeavor to provide you with enough information to get you started so that you can work productively with Maple. For further information, you are encouraged to consult the Maple manuals, which will provide you with further examples of the use of Maple's programming facilities.

All programming languages provide a few basic means for the construction of algorithms. On the most basic level, a computer program is a sequence of instructions that the computer executes one after the other. Programs become more sophisticated when you start changing the flow of execution. Maple provides the same sort of mechanisms for flow control as is found in traditional programming

languages such as C. While the syntax varies from one computer language to the next, there are two primary kinds of control structures used: branching and iteration.

Branching

We will first discuss the concept of branching and its implementation in Maple. Branching is a mechanism that allows you to choose between statements based on conditions that can only be determined during a program's execution. This is also called a selection or conditional statement.

if-then

As an example, suppose that you want to display a message based on whether a particular value is positive. First, we assign a value to the name **z**.

```
> z := 5  
z := 5
```

(0.85)

The following code will display the message “That’s positive” if the value stored in **z** is greater than zero.

```
> if z > 0 then  
    print("That's positive")  
end if  
"That's positive"
```

(0.86)

Note that, in order to begin a new line within a single command prompt, you press Shift+Enter or Shift+Return. The line breaks and extra spaces are not required, in fact Maple ignores them, but they often make programs easier to read and understand.

Typically, the condition in an if statement depends on a value input to a program or some intermediary calculation or a value that changes during the execution of a procedure. In these examples, think about the name **z** as storing some value that varies based on some other computations. For instance, **z** could be the value of some function at a particular point. Then, the value of **z** would depend on which point was chosen.

Let us dissect the code above. The if statement begins with the keyword **if**. This is immediately followed by a conditional expression, that is, an expression that Maple can evaluate to true or false. Conditional expressions may include expressions that include relational operators ($<$, \leq , $=$, $>$, \geq , \neq), logical operators (**and**, **or**, **not**), or procedures and commands that return logical values (**true**, **false**, or **FAIL**). We will see many examples of conditional expressions in Chapter 1. Note that the nonstrict inequalities and not-equals are automatically transformed to the symbols \leq , \geq , and \neq in 2-D input mode.

Following the conditional expression, you must include the keyword **then**. Following the **then** keyword is a statement sequence, one or more statements that are to be executed in the event that the conditional expression is true. In our example above, the **then** keyword was followed by the statement sequence consisting of a single statement, the **print** command.

Finally, the phrase **end if** is used to indicate the end of the conditional statement.

When you execute the statement above, Maple evaluates the conditional expression $z > 0$. Since this is a true statement (because **z** happened to be 5), Maple executes the **print** command in the

statement sequence following the **then** keyword. If the conditional expression had been found to be false, then Maple would not execute the **print** command.

Below is another example, in which the conditional statement is false.

```
> if z ≥ 10 then  
    print("That has at least two digits")  
end if
```

Notice that in this case, nothing is displayed. Once Maple determines that the condition is false, it skips past the statement sequence following the **then** keyword.

else

Often, you will want to take one action if a condition is true and a different action if a condition is false. The **else** keyword allows you to extend a conditional statement to contain a second statement sequence to be executed if the condition is false.

```
> if z < 0 then  
    print("It's negative")  
else  
    print("It's not negative")  
end if  
"It's not negative" (0.87)
```

The **then** keyword separates the conditional expression from the statement sequence that is executed when the condition is true. The **else** keyword indicates the beginning of the statement sequence that is to be executed when the condition is not true.

In the example, Maple first tests to see if **z < 0**. Since this is false, Maple skips to the **else** clause and executes the second **print** command.

elif

You can also extend the if statement to a multiway branching structure for when there are more than two options. To do this, you use the **elif** keyword to introduce additional conditions that may be true when the initial **if** condition fails.

```
> if z ≥ 10 then  
    print("That has at least two digits")  
elif z > 0 then  
    print("It's positive")  
elif z ≥ 0 then  
    print("It's zero")  
else  
    print("It's negative")  
end if  
"It's positive" (0.88)
```

The statement above works as follows. First, Maple checks the condition **z >= 10**. If this were true, it would execute the first **print** statement. However, since it is false, Maple moves on to the

first **elif** condition, **$z > 0$** . This condition is true, so Maple executes the **print**("It's positive.") command.

Look at the next condition: **$z \geq 0$** . This condition is also true, but Maple does not execute the corresponding **print** command. In an **if-elif-else** structure, once the first conditional expression is found to be true, the statement sequence following that condition is executed and any conditions that follow the first are skipped. Likewise, the **else** statement sequence is only executed in case all the conditional expressions were false. In other words, only one of the statement sequences is ever executed.

That is why we can say that the number is 0 if the test **$z \geq 0$** is true. This condition is only checked if all the previous conditions failed. Thus, if the **$z \geq 0$** test is evaluated true, we know that **$z > 0$** was false, and hence **z** is 0.

Iteration

The previous subsection showed how to use branching in Maple to execute different blocks of code depending on whether or not a specified condition was met. In this subsection, we look at ways to repeat a block of code. Iteration is the mechanism for doing a given task repeatedly and is typically accomplished by a loop structure.

for Loops

The most common type of iteration is the **for** loop. The most basic kind of for loop executes a statement for each integer in a particular range. The example below computes the squares of the integers from 3 to 5.

```
> for i from 3 to 5 do  
    i^2  
end do  
9  
16  
25
```

(0.89)

The statement begins with the **for** keyword, indicating the type of loop. After the **for** keyword is a variable name, called the loop variable. The letter **i** is a typical choice. Then, the **from** keyword is followed by the initial value of the loop variable. The **to** keyword is followed by the maximum value for the loop. After that, the **do** keyword precedes the body of the loop, which is terminated by the **end do** phrase.

The statement(s) in between **do** and **end do** form the body of the loop. That is, those are the statements that are executed repeatedly. When Maple executes this loop, here is what happens. First, Maple assigns the starting value, specified by the **from** clause, to the loop variable **i**. Then, the statement sequence is executed and the loop variable is squared. Once the statement sequence is completed, Maple increments the loop variable by 1 and checks to see whether its new value exceeds the value specified by the **to** clause. If not, the statement sequence is executed again with the new value of the loop variable before incrementing it again. Once the loop variable exceeds the maximum value, the loop terminates.

In other words, **i** is assigned to **3**, and **3^2** is computed. Then **i** is incremented to **4** and **4^2** is computed. Then **i** is incremented to **5** and **5^2** is computed. Then **i** is incremented to **6**, which exceeds the maximum so the loop ends.

There is actually an extra step that we did not mention. Immediately after assigning the starting value to the loop variable, the loop variable is tested against the maximum value. This means that it is possible to create loops that never execute their statement sequence.

```
> for i from 7 to 2 do
   print("Execute")
end do
```

Maple also includes the option for a **by** clause if you would like to increment the loop variable by a value other than 1.

```
> for i from 3 by 2 to 11 do
    $i^2$ 
end do
9
25
49
81
121
```

(0.90)

In this example, the **by 2** clause indicates that the loop variable is incremented by 2. The value 2 is referred to as the *step*.

The **by** clause also provides a way to loop from high to low by using a negative step.

```
> for i from 10 by -1 to 5 do
    $i^2$ 
end do
100
81
64
49
36
25
```

(0.91)

Any of the clauses shown above can be omitted, under certain circumstances. If the **from** clause is omitted, Maple will assume the starting value is 1. If the **to** clause is omitted, the loop will execute forever, unless interrupted by another command. (You should take great care to avoid so-called infinite loops unless it is what you intend, and even then, save your work before executing the loop!) If the loop variable is not actually needed as part of the statement sequence, then the **for** clause can be omitted. For example, the following prints “Hello world” three times.

```
> to 3 do
   print("Hello world")
end do
“Hello world”
“Hello world”
“Hello world”
```

(0.92)

while Loops

Somewhat more general than a **for** loop is the **while** loop, which is another method for iteration available in Maple. In a while loop, execution continues as long as a conditional statement remains true.

```
> i := 2 :  
  while i < 107 do  
    print(i);  
    i := i2 + 2  
  end do :  
  2  
  6  
  38  
  1446  
  2090918
```

(0.93)

Note that loops, like other statements, can be ended with a colon. For loops, this suppresses the automatic printing of all the statements in the loop. Then, you can use the **print** command to print only the information that you want displayed.

Look at the previous example carefully. First, we assigned the value 2 to the name **i**. This initialization step is done automatically for us in a for loop but was made explicit in this while loop. The **while** keyword indicates that the loop is a while loop and is immediately followed by the conditional statement that controls the loop. This can be any condition you want. The condition is followed by the **do** keyword. Between the **do** keyword and the **end do** phrase is the statement sequence. The statement sequence is executed repeatedly until the condition is false.

In our example, there are two statements in the statement sequence. First, the current value of **i** is printed. Then, the value of **i** is changed to the result of squaring it and adding 2. This continues as long as the value of **i** is less than 10^7 . Note that the first semicolon is required in order to separate the two statements in the statement sequence.

It is very important in a while loop to be sure that the body of the loop will eventually have the effect of making the controlling condition false. Think about what would happen if the second command in the body of the previous loop had been **i := i - 2**. In that case, **i** would have started out equal to 2. Then, it would become 0, then -2, then -4, then -6, etc, and it would never exceed 10^7 , so it would never cease.

Mixing for and while

The for loops and while loops we have demonstrated in this section are the most common kinds of loops. However, Maple has a feature that allows you to combine the for and while loop semantics into a single loop construction. Here is an example of how this is done.

```
> for i from 3 to 44 while i2 < 50 do  
    i, i2  
  end do  
  3, 9  
  4, 16
```

5, 25
6, 36
7, 49

(0.94)

In fact, in Maple, there is only one kind of loop, and the **for** clause and **while** clause are considered optional statements that control the looping behavior in specific ways.

Looping Over a List or Set

There is one additional clause that can appear in a loop, the **in** clause. Given a list or a set, the **in** clause allows you to define a loop that executes once for each element of the list or set. In the example below, the loop squares each element of the list.

```
> for i in [2, 5, 6, 11, 8] do
    printf("The square of %2a is %3a.\n", i, i^2)
  end do
  The square of 2 is 4.
  The square of 5 is 25.
  The square of 6 is 36.
  The square of 11 is 121.
  The square of 8 is 64.
```

The **for** clause indicates that the name of the loop variable is **i**. The **in** clause specifies that the loop variable should be assigned to each element of the given list in turn. The **while** clause can also appear in conjunction with the **in** clause. However, none of **from**, **by**, or **to** can appear when **in** is used.

Premature Loop Exit

Sometimes it is necessary to terminate a loop prematurely. This may be in order to prevent an error or because the logic of a particular problem dictates that it must. In these cases, the **break** keyword is used to transfer control out of a loop. Consider the example below. Note that **even** and **odd** are considered types in Maple and so can be used as the second argument to the **type** command. (The reader is encouraged to research the Collatz conjecture, which forms the basis of this example.)

```
> n := 27:
  count := 0:
  while n ≠ 1 do
    if type(n, odd) then
      n := 3n + 1
    else
      n :=  $\frac{n}{2}$ 
    end if;
    count := count + 1;
    if count > 50 then
      break
    end if
  end do:
  count
```

51

(0.95)

In the above, the **break** statement is executed if the number of iterations of the loop exceeds a specified threshold. This example is meant to illustrate how you can use **break** to place limits on the number of iterations in a while loop.

Related to **break** is the **next** command. Instead of terminating a loop entirely, the **next** command causes the rest of the statements in the body of the loop to be skipped, but the loop continues. In a for loop, this means that it moves on to the next value of the loop variable. The example below computes the value of the rational expression $\frac{x^2 + 3}{x - 1}$ for the integers between -3 and 3 . A **next** statement is used to avoid a division by zero error.

```
> for x from -3 to 3 do
  if x = 1 then
    next
  end if;
   $x, \frac{x^2 + 3}{x - 1}$ 
end do
-3, -3
-2, -7/3
-1, -2
0, -3
2, 7
3, 6
```

(0.96)

Procedures

A Maple procedure is very much like a function in mathematics. It is an object that is capable of receiving data as input and producing output.

A Maple procedure is created using the **proc** keyword. Ordinarily, procedures are assigned to a name. Consider the simple example below. Observe that we have entered the procedure in a Code Edit Region. You can add a Code Edit Region to your worksheet by selecting **Code Edit Region** from the **Insert** menu or by clicking on the icon shown below in the toolbar.



The Code Edit Regions in this manual are all set to autoexecute, so if you allow Maple to execute the autoexecutable code when you open the file, those procedures will all be prepared for you to experiment with. To manually execute code in a Code Edit Region, you can right-click on it and use the pop-up menu item Execute Code, or you can click anywhere in the region and type either Control+E or Control+= in Windows or Unix, or type Command+E or Command+= on a Mac.

```
1 MySum := proc(a, b)
2   a + b;
3 end proc;
```

MySum := proc(a, b) a + b end proc

This creates a procedure and assigns it to the name **MySum**. Note that assignment of procedures to names via the `:=` assignment operator is identical to assignment of any other Maple object. Also note that the output of this assignment is merely a repetition of the procedure definition. In the future, we will terminate procedure definitions with a colon rather than a semicolon as it is not necessary to see the procedure definition repeated.

Following the assignment operator, the procedure definition begins with the keyword **proc**. Immediately following the **proc** keyword is a matched pair of parentheses enclosing a sequence of names. These names are called the parameters to the procedure. When the procedure is called, for instance as below,

```
> MySum(2, 3)  
5
```

(0.97)

the arguments **2** and **3** are assigned to the parameters **a** and **b**. Parameters are names used in the definition of the procedure to hold the place of input values, while arguments are the particular input values used in a particular execution of a procedure. (Note that this distinction is sometimes blurred and some programming languages use different terminology.) It is possible to define a procedure that requires no parameters, but even in this case the parentheses are required both in the procedure definition and when executing the procedure.

Following the parameter declaration is the statement sequence. This is the body of the procedure, consisting of all the commands that the procedure needs to perform to compute its output value. In this example only one command is used, the two parameters are added. Note that the output of a procedure is the result of the last statement that is executed. In the example above, 5 is output by the procedure because the final (and only) statement is the sum of **a** and **b**.

Declaring Parameter Types

Within the parameter declaration, it is common, and very useful, to declare the types of the parameters. Consider the following procedure.

```
1 newMySum := proc(a : :integer, b : :integer)  
2     a + b;  
3 end proc;
```

In this **newMySum** procedure, the double colons followed by the Maple type name **integer** tells Maple that we expect the parameters **a** and **b** to be integers. If we try to execute this procedure with noninteger arguments, Maple will prevent the statement sequence from being executed and will report an error.

```
> newMySum(3, 2.5)
```

Error, invalid input: newMySum expects its 2nd argument, b, to be of type integer, but received 2.5

Declaring the types of parameters is good programming practice and very useful, and we will often include parameter types in the procedures we create in this manual.

Return Statements and Global and Local Variables

Consider the procedure below.

```

1 Alg1 := proc (a :: numeric)
2   global w;
3   local x;
4   if a < 1 then
5     x := 1;
6   else
7     x := 2;
8   end if ;
9   w := x + 10;
10  return w;
11 end proc;

```

This procedure introduces a few more concepts. The first line assigns the procedure to the name **Alg1** and specifies that it takes one **numeric** argument (**numeric** is a broadly defined type for integers, rational numbers, and floating-point values).

Note the use of the **return** command in the final line of the procedure body. We have mentioned that the output of a procedure is, by default, the final computed value. The keyword **return** can be used, as it is here, to make explicit what is being output. It can also be used to cause a procedure to stop execution and immediately output a particular result. In this manual, we will usually use **return** statements, even when they are not required, so that it is clear what the output of a procedure is.

The second and third lines in **Alg1** use the **global** and **local** keywords followed by variable names. In any procedure, all of the names used in the procedure fall into one of three kinds: parameters, global variables, and local variables.

A variable is called local when it is only used within the procedure. Local variables exist only within the procedure and have no meaning outside of it.

To see what this means, we will assign the value 4 to the name **x**.

```

> x := 4
x := 4

```

(0.98)

If we apply the **Alg1** procedure to a value smaller than 1, within the body of the procedure, **x** will be assigned the value 1.

```

> Alg1(0.5)
11

```

(0.99)

However, if we check the value of **x**,

```

> x
4

```

(0.100)

it has remained 4. This is because the **x** inside the procedure is declared local. You can think about the local **x** in the procedure as different than and isolated from the **x** that stores 4.

Global variables are the opposite. A variable is global when it is accessible and has the same value both inside and outside the procedure. In our example, the name **w** is declared global. If you look at

its value before and after the execution of the procedure, you will see that, unlike **x**, the value of **w** is changed.

```
> w := 76  
w := 76
```

(0.101)

```
> Alg1(3)  
12
```

(0.102)

```
> w  
12
```

(0.103)

Use of global variables in procedures is generally discouraged in most programming languages. There are a variety of reasons to avoid global variables, not least of which is that they can cause unpredictable results, especially in larger projects. In this manual, we will have cause to use global variables on occasion, but generally we will declare variables to be local.

Parameters to a Maple procedure are considered local automatically, in the sense that previous values of those names are not affected by the execution of the procedure. In fact, parameters are subject to the additional restriction that they cannot normally be modified during execution. In particular, they cannot be assigned to. For example, the procedure below causes an error to be generated when we execute it. It also illustrates how you declare more than one variable.

```
1 Alg2 := proc (a :: numeric)  
2   local x, y;  
3   if a > 0 then  
4     x := sqrt(a);  
5     y := a^2;  
6   else  
7     x := sqrt(-a);  
8     y := -a^2;  
9   end if;  
10  a := x + y;  
11  return (a);  
12 end proc:
```

```
> Alg2(5)
```

Error, (in Alg2) illegal use of a formal parameter

The “illegal use” described in the error message is the assignment **a := x + y;**

A Final Example

We give one final example of a procedure.

```
1 Alg3 := proc (a :: numeric)  
2   # This procedure does nothing  
3   if a > 0 then  
4     return NULL;  
5   else
```

```

6      return FAIL;
7  end if ;
8      a := sqrt(a);
9  end proc:
```

First, note the line that begins with a pound symbol (also called hash or sharp or number symbol). This is Maple's syntax for commenting code. Anything following a # in a line is ignored by Maple. You can use comments to provide explanation, within your procedure's code, of what it does and how it works. Commenting can also be useful in debugging procedures because it allows you to temporarily deactivate lines of code without deleting them. In this manual, explanation of code will generally be given within the exposition rather than as comments within the code itself.

Second, this example has two **return** statements. In the case that the argument is positive, the procedure returns the value **NULL**. Recall from earlier that **NULL** is Maple's name for the empty sequence. Returning **NULL** is how you can cause a Maple procedure to have no output.

> *Alg3(5)*

The other **return** statement returns the value **FAIL**. It is typical to have a procedure return **FAIL** to indicate that the procedure is unable to compute the desired output.

> *Alg3(-2)*

FAIL (0.104)

Finally, notice that the two **return** statements block the final statement, **a := sqrt(a)**, from ever being encountered. Despite the statement being illegal (it assigns to a parameter), errors were not generated because the illegal assignment is never reached.

Functional Operators

We conclude this section with a brief discussion of functional operators. In Maple, a functional operator is a particular kind of procedure. Functional operators are often used to model simple mathematical functions.

The following defines a functional operator that models the function

$$f(x, y) = 2x^2 + 5y^2 + 3xy.$$

> *f* := $(x, y) \rightarrow 2x^2 + 3xy + 5y^2$
f := $(x, y) \mapsto 2x^2 + 3yx + 5y^2$

(0.105)

Let us look at the example above piece by piece. First is the name **f** followed by the assignment operator indicating that we are assigning the functional operator to the name **f**. Following the assignment operator is the list of parameters enclosed in parentheses. In the example above, there are two parameters, **x** and **y**. (If a functional operator only requires one parameter, then the parentheses are optional.) After the parameter list is the “arrow,” typed as a hyphen followed by a greater than sign. The definition concludes with the expression, which may use other Maple commands, that defines the operator.

Applying a functional operator is the same as applying a procedure. To calculate $f(2, -3)$, you enter **f(2,-3)**.

> *f*(2, -3)

35

(0.106)

Functional operators are particularly useful in conjunction with other commands that expect Maple commands or procedures as one of the arguments. For example, with the **map** command. Recall that the first argument of the **map** command must be a procedure. If a list is given as the second argument, then **map** returns the list obtained by applying the procedure to each element of the list. The following demonstrates how to use a functional operator together with **map** in order to square all of the elements of a list.

> *square* := *x* → x^2

square := *x* ↦ x^2

(0.107)

> *map*(*square*, [-7, -3, -1, 0, 2, 3, 5])

[49, 9, 1, 0, 4, 9, 25]

(0.108)

The functional operator **square** could also have been created using **proc**, but for such a simple procedure, the functional operator definition is easier. Functional operators also allow you to include the definition of the procedure as the argument. For instance, to cube the elements of a list, you could do the following.

> *map*($x \rightarrow x^3$, [-7, -3, -1, 0, 2, 3, 5])

[-343, -27, -1, 0, 8, 27, 125]

(0.109)

Functional operators will be discussed in more detail in Chapter 2.

System Architecture and Packages

This section briefly explains the overall structure of the Maple system. It is intended to help you better understand how Maple works and why some things work the way that they do.

Maple uses an innovative system architecture to achieve ambitious design goals. The Maple kernel implements the basic interpreter of the Maple programming language, the interface with the host computer's operating system, and certain time-critical services.

However, nearly all of Maple's mathematical power dwells in the extensive Maple library. Consisting of thousands of lines of code, the Maple library is written in the Maple programming language itself. The advantages of this design include: portability, extensibility, and openness.

Portability refers to the ability to quickly and easily modify a program to run on a different computer system or a different operating system. With Maple, only the relatively small kernel needs to be ported between systems while nothing needs to be done to the library.

Extensibility is the ability of users, like yourself, to add capabilities and features to Maple. You can even redefine existing Maple library routines to extend or modify their behavior. You will see extensibility throughout this manual as we guide you through the process of writing procedures for exploring discrete mathematics that Maple does not already include.

Openness, in Maple, means that you can examine the source code for many of the library functions, thereby gaining greater understanding of the algorithms used by Maple.

Because of its size, much of the Maple library is organized into packages. A package is a collection of Maple library routines that offer related functionality. Since these are not normally loaded when you start Maple, you must request the services localized in each package by telling Maple explicitly that you want to load them. For this purpose, Maple provides the **with** command.

For example, the **combinat** package provides routines related to combinatorics. To use procedures from the **combinat** package, you would first load the package by typing

```
> with(combinat)
[Chi, bell, binomial, cartprod, character, choose, composition, conjpart,
 decodepart, encodepart, eulerian1, eulerian2, fibonacci, firstcomb,
 firstpart, firstperm, graycode, inttovector, lastcomb, lastpart, lastperm,
 multinomial, nextcomb, nextpart, nextperm, numbcomb, numbcomp,
 numbpart, numbperm, partition, permute, powerset, prevcomb,
 prevpart, prevperm, randcomb, randpart, randperm, rankcomb,
 rankperm, setpartition, stirling1, stirling2, subsets, unrankcomb,
 unrankperm, vectoint] (0.110)
```

All the names of the newly loaded library routines are listed. Of course, you can suppress this list by terminating the statement with a colon.

You can access commands in a package without loading the whole package by using the long form of the command name. For example, Maple includes a command, **Mean**, for computing the arithmetic mean, or average, of a list of numbers. This command is located within the **Statistics** package. We can use it as shown below.

```
> Statistics[Mean]([1, 2, 3, 4, 5, 6, 7])
4. (0.111)
```

In the long form, the name of the package is followed by the name of the command in brackets. Alternately, for most packages used in this manual, you can use the **:-** operator.

```
> Statistics:-Mean([1, 2, 3, 4, 5, 6, 7])
4. (0.112)
```

In order to use the short form of the command name, that is, just **Mean**, you would first have to load the package.

```
> with(Statistics):
> Mean([5, 7, 12, 21])
11.250000000000000 (0.113)
```

Within a procedure definition, you have the option of specifying packages that the procedure requires by means of the **uses** keyword, as in the example below.

```

1 Avg3 := proc (a::numeric, b::numeric, c::numeric)
2   local m;
3   uses Statistics;
4   m := Mean([a,b,c]);
5   return m;
6 end proc;

```

> *Avg3(9,2,11)*
 7.333333333333333

(0.114)

The purpose of the **uses** statement is to ensure that commands used in the procedure are available. This way, if you were to try to execute this procedure without first loading the **Statistics** package via the **with** command, Maple would still be able to execute it. In this manual, when writing procedures, we will either use the long form of the name or include a **uses** statement in the procedure definition.

Maple Versions

The procedures and examples in this manual were developed and tested with Maple 2018.

On-Line Material

The files for this manual, including both the PDF and Maple Worksheet versions of all chapters, are available at the website for the eighth edition of *Discrete Mathematics and Its Applications* by Kenneth Rosen: www.mhhe.com/rosen. This site includes many other kinds of supplementary material for students and instructors.

The website includes one file in particular to be aware of: RosenMaplePackages.mla. This file is a repository for Maple packages that include many of the commands developed in this manual. There are two ways in which you can make these packages available.

The recommended way is to install RosenMaplePackages.mla in a library directory. First, execute the following statement (shown here without output)

> *libname*

The name **libname** is a system variable that specifies where on your computer Maple looks for library files, such as those that define packages. When you execute the statement, Maple will display the current value of the variable, which should be one or more directories on your computer.

Choose one of the directories listed and copy RosenMaplePackages.mla into that directory.

Then, restart the Maple kernel by executing the **restart** command. Note that the execution group below has been set to be nonexecutable, so that if you have Maple execute the entire worksheet, it will not automatically restart the kernel. Either enter it on a new input line in your worksheet, or make it executable by right clicking to open the pop-up menu and select **Executable Math**.

> *restart*

The alternative, in case you are not able to copy files into the Maple directories, is to add a directory to **libname**. First, copy RosenMaplePackages.mla into a directory on your computer.

Suppose that the full path of the directory in which you copied RosenMaplePackages.mla is:

```
/Users/danieljordan/DiscreteMath/
```

Then, you would execute the following statement, replacing the directory shown with your directory.

```
> libname := libname, "/Users/danieljordan/DiscreteMath/"
```

As with the restart command, this has been made nonexecutable, since your directory will be different.

Note that if you must use this method, you will need to repeat it every time you start a new Maple session. The first approach needs to be done only once, however.

If you are not sure what the correct directory is, you can execute the command below. This will open a file selection dialog window; if you locate the file RosenMaplePackages.mla on your computer and click Open, the correct directory will be added to **libname**. This command has also be set to be nonexecutable.

```
> libname := libname, FileTools[ParentDirectory](  
    Maplets[Utilities][GetFile](  
        'title' = "Locate RosenMaplePackages.mla",  
        'directory' = currentdir(homedir),  
        'filefilter' = "mla",  
        'filterdescription' = "Library Files"))
```

Once you have completed one of these methods, the following commands should execute without error.

```
> with(Chapter0):  
test()
```

Each chapter has an associated package, called “Chapter” followed by the number of the chapter. Each package contains the procedures defined in the respective chapter that may be of use to you as you explore discrete mathematics. All of these packages are defined in the RosenMaplePackages.mla repository.

Exercises

Exercise 1. Compute $17 \cdot (126^{34} - 93)$.

Exercise 2. Form the **list** (in the Maple sense) of the first 100 positive numbers that are 3 greater than a multiple of 7, using the **seq** command.

Exercise 3. Insert the number 300 between the 42nd and 43rd entries in the list you created in the previous exercise.

Exercise 4. Use a for loop to print the string “Hello World!” 10 times.

Exercise 5. Use a while loop to print the string “Hello World!” 10 times.

Exercise 6. **even** is a Maple type and thus can be used with the **type** command. Loop over the list you created in Exercise 2, and within the loop, use an if-else statement to print “even” or “odd” for each element of the list.

Exercise 7. Use **map** and a functional operator to apply the formula $x^2 + 3x - 2$ to the list $[-5, -4.5, -4, \dots, 4, 4.5, 5]$.

Exercise 8. Write a procedure (using **proc**) that has one parameter, a list, and outputs the list in reverse order. You should use only commands discussed in this Introduction.

1 The Foundations: Logic and Proofs

Introduction

This chapter describes how Maple can be used to further your understanding of logic and proofs. In particular, we describe how to construct truth tables, check the validity of logical arguments, and verify logical equivalence. In the final two sections, we provide examples of how Maple can be used as part of proofs, specifically to find counterexamples, carry out proofs by exhaustion, and to search for witnesses for existence proofs.

1.1 Propositional Logic

In this section, we will discuss how to use Maple to explore propositional logic. Specifically, we will see how to use logical connectives in Maple, describe the connection between logical implication and conditional statements in a program, show how Maple can be used to create truth tables for compound propositions, and demonstrate how Maple can be used to carry out bit operations.

In Maple, the truth values true and false are represented by the names **true** and **false**. These are constants in Maple, just like a numeric constant such as π . Propositions can be represented by names (variables) such as **p**, **q**, or **Prop1**. Note that if you have not yet made an assignment to a name, entering it will return the name.

```
> Prop1  
Prop1
```

(1.1)

Once you have assigned a value, Maple will evaluate the name to the assigned value whenever it appears.

```
> Prop1 := true  
Prop1 := true
```

(1.2)

```
> Prop1  
true
```

(1.3)

You can cause Maple to “forget” the assigned value with the following syntax:

```
> Prop1 := 'Prop1'  
Prop1 := Prop1
```

(1.4)

```
> Prop1  
Prop1
```

(1.5)

Right single quotes are used in Maple to delay evaluation of an expression. In the above, they are used to assign **Prop1** to the unevaluated **Prop1**; in other words, we set the value of the variable to the name of the variable which effectively unassigns the name. (The right single quote is on the same key as the quotation mark on a standard keyboard. In Maple, the right single quote is different from the left single quote, which is on the same key as the tilde, \sim .)

Logical Connectives

Maple supports the basic logical operators discussed in the textbook. We illustrate the logical operators of negation (**not**), conjunction (**and**), disjunction (**or**), exclusive or (**xor**), and implication (**implies**). Note that expressions formed using the logical connectives are called Boolean expressions, and variables that stand for true or false are called Boolean variables.

> **not** *true*
 false (1.6)

> *true and false*
 false (1.7)

> *true or false*
 true (1.8)

> *true xor false*
 true (1.9)

> *true implies false*
 false (1.10)

> *(true and false) implies true*
 true (1.11)

Note that the precedence of the Maple logical operators agrees with Table 8 of Section 1.1; so, the parentheses in the previous command is unnecessary, though they make for more readable propositions.

Maple supports other common logical operators indirectly. For example, the biconditional (**iff**) is translated into its definition in terms of implication and conjunction.

> *p iff q*
 $(p \Rightarrow q) \wedge (q \Rightarrow p)$ (1.12)

Conditional Statements

We saw above that Maple includes the operator **implies** for evaluating logical implication or conditional statements. In mathematical logic, “if p , then q ” has a very specific meaning, as described in detail in the text. In computer programming, and Maple in particular, conditional statements also appear very frequently, but have a slightly different meaning.

From the perspective of logic, a conditional statement is, like any other proposition, a sentence that is either true or false. In most computer programming languages, when we talk about a conditional statement, we are not referring to a kind of proposition. Rather, conditional statements are used to selectively execute portions of code. Consider the following example of a procedure, which adds 1 to the input value if the input is less than or equal to 5 and not otherwise.

```
1 | IfProc1 := proc (x)
2 |   local y;
3 |   y := x;
```

```

4   if y <= 5 then
5     y := y + 1;
6   else
7     y := y - 1;
8   end if ;
9   return y;
10 end proc;

```

*IfProc1 := proc(x) local y; y := x; if y <= 5 then y := y + 1
else y := y - 1 end if; return y end proc;*

To execute a procedure that we have written, give the name of the procedure followed by arguments, in parentheses. Observe that this procedure works as promised:

> *IfProc1(3)* (1.13)
4

> *IfProc1(7)* (1.14)
6

Because this is our first Maple procedure, let us spend a moment breaking it down. First, we have the name of the procedure, **IfProc1**, followed by the assignment operator **:=**. Then, we use the keyword **proc**, short for procedure. After the keyword **proc**, we list, in parentheses, names for the parameters, or inputs, to the procedure. In this example, there is only one parameter, which we name **x**.

After closing the parentheses to terminate the list of parameters, we use the keyword **local** followed by the name **y**. This tells Maple that **y** is a name that is used only in the procedure, so that if the name **y** is being used someplace else in the worksheet, they would not interfere with each other. In particular, if you are using **y** as a name for some value and then you run this procedure, having declared **y** as local to the procedure guarantees that the **y** you assigned outside of the procedure is not changed. You should always declare variables you use in a procedure as local and this declaration needs to be the first statement after the parameter list. If you have more than one variable to declare as local, you list them separated by commas and end the entire list with a semicolon. If you write a procedure and use variables that are not declared local, Maple will warn you about them. Also note that the names you use for parameters to the function are implicitly local to the procedure and should not be declared as local.

After declaring **y** to be local in our **IfProc1** procedure, we begin the body of the procedure, called the statement sequence. The statement sequence can be as long as is required.

For this procedure, the first statement in the statement sequence is to assign **y** to be equal to the value of **x**. It may seem strange to “copy” the value of **x** to **y** and then work with **y** rather than just using **x**. The reason is that Maple will not let you assign values to a parameter. If you try including **x := x + 1** in the procedure (go ahead and try), Maple will raise an error when you try to apply the procedure to an input value. There is a very good reason for this: it prevents you from accidentally modifying an input. For example, suppose you had a name **important** and stored some value in that variable, perhaps a value obtained after considerable work. If you then wrote a procedure that accidentally modifies its input parameter and you ran your procedure on **important**, then the value would be lost and you would have to redo all your work. Therefore, Maple prevents you from

assigning to parameters in the body of a procedure in order to help you avoid making that kind of mistake. (There is a way around this, but we will not discuss it now.)

Next are the five lines comprising the conditional statement: **if $y \leq 5$ then $y := y + 1$; else $y := y - 1$; end if;**. Again, this conditional statement is not a proposition; it is a command. When Maple comes to this part of the procedure, it sees the keyword **if** and knows that what follows the **if** up to the **then** keyword, is a conditional expression (which is a proposition). Maple checks to see if the conditional expression is true or not. If the conditional expression is true, then Maple knows to execute the statement(s) after the **then** keyword. If the conditional expression is false, then Maple executes the statement(s) after the **else** keyword.

Note also the way in which we change the value of y : **$y := y + 1$** ; This is another instance of Maple trying to make sure we “really mean it” when we want to change a value. The command **$y + 1$** ; would not change the value of y . To change the value of y we have to add 1 to y , and then assign that number to the name y .

Finally, the last line of the statement sequence is **return y** . By default, a Maple procedure’s output is the result of the last statement that is executed. Imagine the **IfProc1** procedure without the **return y** ; line. In that case, we would not be able to tell in advance what the last command to be executed is. For small values, the last command would be the assignment **$y := y + 1$** ; but for inputs greater than 5, **$y := x$** ; would be the last statement executed. It is better programming style to make it clear what the output of the procedure will be, since (especially in more complicated programs) it can get quite difficult to tell what the output will be. In this example, we made the output explicit with the **return y** ; command. Since this follows the end of the conditional statement, we know it will always be the last statement executed.

The **return** keyword is used in this case simply to be explicit that the output from the procedure is the value of y . The **return** command, as we will see in future procedures, is useful as it can be used for short-circuiting procedures. This means that execution of the procedure immediately stops and the value following the **return** keyword is output by the procedure.

Finally, we end the procedure with the statement **end proc;**. As is usual for assignment statements, Maple’s output is a repetition of the assignment. For this reason, we will generally use a colon rather than a semicolon to terminate the **end proc:** statement. This suppresses the output of the assignment command but does not alter the procedure.

Built-In Truth Table Command

In the textbook, you saw how to construct truth tables by hand. Here, we will see how to have Maple create them for us.

First, note that Maple includes a built-in command for producing truth tables. The **TruthTable** command is part of the **Logic** package. In order to use a command that is part of a package, as opposed to those that are top-level commands, there are two choices. You can either bind the package using **with** in order to make the function name available in its short form or you can use a long form of the function name.

If you are going to use a command from a package once or twice, it is easy enough to use the long form name. The general structure of the long form name of a command from a package is to give the name of the package followed by the function’s name in brackets. After the closing bracket

following the function name, arguments are passed to the function in parentheses. In the case of **TruthTable**, the only required argument is a Boolean expression. For example, to obtain the truth table for $\neg p \wedge \neg q$, we enter the following. Note that **p** and **q** are names that have not been assigned values.

```
> Logic[TruthTable] ((not p) and (not q))
```

	<i>p</i>	<i>q</i>	<i>value</i>
1	false	false	true
2	false	true	false
3	true	false	false
4	true	true	false

(1.15)

Since we will be using a few different commands from the **Logic** package, we will use **with** in order to be able to execute its commands without repeating the package name.

```
> with (Logic)
```

```
[&and, &iff, &implies, &nand, &nor, &not, &or, &xor, BooleanSimplify,
 Canonicalize, Complement, Contradiction, Convert, Dual,
 Environment, Equivalent, Export, Implies, Import, Normalize, Parity,
 Random, Satisfiable, Satisfy, Tautology, TruthTable, Tseitin]
```

(1.16)

```
> TruthTable ((not p) and (not q))
```

	<i>p</i>	<i>q</i>	<i>value</i>
1	false	false	true
2	false	true	false
3	true	false	false
4	true	true	false

(1.17)

When we execute the **with** command to make the **Logic** package commands readily available, it lists the names of the commands provided by the package. While somewhat technical, it is worth commenting on the fact that the **Logic** package includes versions of the logical operators, whose names are preceded by ampersands. The reason for this is that the main Boolean operators **and**, **or**, and so on are three-valued operators. In addition to their behavior with **true** and **false**, they also treat **FAIL** as a truth value. This is useful in a computer programming context, but not the expected situation for studying mathematical logic. So the operators **&and**, **&or**, and the others in the **Logic** package are the usual two-valued operators.

The **Logic** package commands automatically convert the standard operators (**and**, **or**, etc.) into the **Logic** package operators and work correctly without comment. The functions **Import** and **Export** in the **Logic** package can be used to explicitly convert between them. Both can optionally be given the second argument **form = boolean** to make explicit the form of the argument or result. Also note that, assuming the **Logic** package has been loaded using **with**, the package's operators appear as their logical symbols.

```
> Import(p and q)
```

```
(p & q)
```

(1.18)

```
> Export((&not p) &or q)
```

```
not p or q
```

(1.19)

It is important to be careful with the **Logic** package operators, as they do not obey the usual order of precedence. It is recommended to use them fully parenthesized. In this chapter, we will trust Maple to automatically convert from the usual operators even when using **Logic** package functions.

Truth Tables and Loops

While the **TruthTable** command provides an easy way to construct a truth table, it will be more instructive, both in terms of understanding the mathematics and in terms of understanding programming in Maple, to build truth tables with more fundamental commands.

We begin by considering the simplest case of a compound proposition: the negation of a single propositional variable.

```
> Prop2 := not p  
Prop2 := not p
```

(1.20)

Note that we defined the proposition **Prop2** as an expression in terms of the name **p**, which has not been assigned a variable. We can determine the truth value of **Prop2** in one of two ways. The obvious way is to assign a truth value to **p** and then ask Maple for the value of **Prop2** as follows.

```
> p := false  
p := false
```

(1.21)

```
> Prop2  
true
```

(1.22)

The drawback of this approach, however, is that our variable **p** is now identified with `false` and if we want to use it as a name again, we need to manually unassign it.

```
> p := 'p'  
p := p
```

(1.23)

The other approach is to use the **eval** command. Although **eval**, short for evaluate, has several different uses, the most common is for evaluating an expression (including a Boolean expression) at given values for the variables in the expression. For example,

```
> eval(Prop2, p = true)  
false
```

(1.24)

In this usage, the command takes two arguments. The first argument is the expression to be evaluated. The second argument specifies the values that are to be substituted for the variables in the expression. If there is only one variable to be replaced, as in the **Prop2** example, the second argument can be given as the single equation **variable=value**. If there are multiple variables, you must provide a list of such equations as the second argument, as in the next example.

```
> eval(p and not q, [p = true, q = false])  
true
```

(1.25)

To make a truth table for a proposition, we need to evaluate the proposition at all possible truth values of all of the different variables. To do this, we make use of loops (refer to the Introduction for a

general discussion of loops in Maple). Specifically, we want to loop over the two possible truth values, true and false, so we will construct a for loop over the list [true, false].

```
> for loopP in [true, false] do
    print(loopP, eval(Prop2, p = loopP));
end do;
    true, false
    false, true
(1.26)
```

The **for** keyword identifies this as a for loop. The name following the **for** keyword, **loopP**, is used as the index variable in the loop. The **in [true, false]** clause tells Maple that we want the index variable to be sequentially assigned to the elements in the list [true, false]. The **do** keyword indicates the beginning of the body of the loop.

The body of the loop consists of a single statement involving two commands. The **print** command can take any number of arguments and causes Maple to display the arguments to the command. The first argument is the index variable **loopP**, meaning that the first thing displayed in each iteration of the loop is the value of the Boolean variable.

The second argument to **print** is a call to the **eval** command: **eval(Prop2, p=loopP)**. In evaluating this command, Maple first evaluates the names **Prop2** and **loopP**, replacing them with their values. Thus, **Prop2** is replaced with **not p** and **loopP** is replaced with whichever truth value it is assigned to in the iteration of the loop. Then, the **eval** command causes Maple to replace the variable **p** in **Prop2** with the current value of **loopP**. Finally, Maple evaluates the Boolean expression so that the **print** command causes the truth value to be displayed.

When there are multiple propositional variables, we use multiple for loops. This is called “nesting” the loops. For example,

```
> Prop3 := (p and q) implies p
Prop3 := p  $\wedge$  q  $\Rightarrow$  p
(1.27)
```

```
> for loopQ in [true, false] do
    for loopP in [true, false] do
        print(loopP, loopQ, eval(Prop3, [p = loopP, q = loopQ]));
    end do;
end do;
    true, true, true
    false, true, true
    true, false, true
    false, false, true
(1.28)
```

Note that the output indicates that the proposition, $(p \wedge q) \rightarrow p$, is a tautology. In fact, this is a rule of inference called simplification, discussed in Section 1.6 of the textbook.

Logic and Bit Operations

We can also use Maple to explore the bit operations of OR, AND, and XOR. Recall that bit operations correspond to logical operators by equating 1 with true and 0 with false. Maple has a package,

called **Bits**, that provides a lot of support for working with bits and bit strings. For the purposes of this manual, however, we will create some of this functionality from scratch. This gives us the opportunity to introduce some more general Maple commands that will be useful in a variety of contexts.

First of all, we write the bitwise **AND** procedure. This procedure will take two arguments, which we call **a** and **b**, and should return 1 if both of the inputs are 1 and 0 if not. The body of the procedure will be an if statement that tests the proposition “ $a = 1$ and $b = 1$.” If this proposition is true, then the procedure will return the value 1. If not, in the else clause, it returns 0.

```
1 AND := proc (a, b)
2   if a=1 and b=1 then
3     return 1;
4   else
5     return 0;
6   end if ;
7 end proc;
```

> *AND*(1, 0)

0

(1.29)

> *AND*(1, 1)

1

(1.30)

The **zip** command

Now that we have a bitwise **AND** procedure, we can apply it to bit strings using Maple’s **zip** command. The **zip** command requires three arguments. The first argument must be a binary procedure or function, that is, a procedure that takes two arguments. The second and third arguments must be two lists (or a similar data structure that holds multiple values, e.g., a matrix). The result of the command is the list whose entries are the result of applying the procedure to the corresponding values of the list.

> *zip*(*AND*, [1, 1, 0, 0], [1, 0, 1, 0])

[1, 0, 0, 0]

(1.31)

In the example above, the first element in the result is obtained by applying **AND** to the pair of first elements, namely (1,1). The second result value is obtained from the pair of values consisting of the second elements in the given list, namely (1,0), the third result from (0,1), and the last result from (0,0).

If the lists that are given to **zip** are of different lengths, the command will stop at the end of the shorter list.

> *zip*(*AND*, [1, 1], [1, 0, 1, 0])

[1, 0]

(1.32)

If you want, however, you can provide an optional fourth argument to the command. This fourth argument will act as the default value for whichever list is shorter.

> *zip*(*AND*, [1, 1], [1, 0, 1, 0], 0)

[1, 0, 0, 0]

(1.33)

In this example, giving 0 as the optional argument means that when it gets to the third and fourth computations, it uses 0 as the value for the first (shorter) list. It is just as if we used [1,1,0,0] for the first list.

Although this last argument is helpful in many circumstances, for operations on bit strings, it is not entirely satisfactory. The problem is that it would be more natural, in the context of **AND** on bit strings, to increase the length of the list by adding 0s on the left, rather than the right. That is, the result of **AND** on [1,1] and [1,0,1,0] should be the same as [0,0,1,1] and [1,0,1,0].

To overcome this, we will rewrite the **AND** procedure. Our new and improved **AND** will still take two inputs, but the inputs will be allowed to be either bits or bit strings. If the inputs are both bits (0 or 1), then we check to see if they are both 1 and return 1 if they are or 0 if not. If the inputs are lists, then we make sure the lists are the same length (possibly by adding 0s on the front of the list), and then use **zip** to **AND** the individual bits. Finally, if the input values are not bits and not lists, then we cause an **error** to be generated.

Implementing this will require some more Maple commands. The first thing that needs to be done is to determine the type of input. This will be done with an **if-elif-else** structure. First, the **if** condition is tested. In this case, the **if** condition will be that the input values are 1s or 0s. If the **if** condition passes, then the commands following the **then** keyword are executed. Otherwise, we jump ahead to the **elif** keyword which, like **if**, is followed by a Boolean condition (that the input are lists) and the **then** keyword. If that condition is true, then the commands following the **elif**'s **then** are executed. Otherwise, we jump ahead to the **else** keyword and execute those commands.

We already know how to handle the first condition, that the inputs are 0s and 1s. This amounts to testing the proposition $(a = 0 \vee a = 1) \wedge (b = 0 \vee b = 1)$. If that is the case, then we can use the same code from the original **AND** procedure above.

Using type to check for lists

For the second, **elif**, condition, we need to determine if the inputs are lists. We do this with the **type** command. Every Maple object, from the number **5** to the procedure **AND** is of one or more types. In mathematics, we classify objects as integers, real numbers, matrices, or functions. Similarly, Maple classifies objects as **integer**, **realcons** (real constant), **Matrix**, or **procedure**. We test an object to see if it is of a particular type with the **type** command. For example, to see that 5 is an **integer** and a **realcons** but not a **Matrix**, but that **AND** is a **procedure** we issue the following commands.

```
> type(5, integer)
true
(1.34)
```

```
> type(5, realcons)
true
(1.35)
```

```
> type(5, Matrix)
false
(1.36)
```

```
> type(AND, procedure)
true
(1.37)
```

To see whether an object is a list, we can test to see if it is of type **list**.

```
> type([1, 2, 3], list)
true
```

(1.38)

```
> type(17, list)
false
```

(1.39)

So, our **elif** condition will use **type** to make sure both inputs are lists.

Padding lists with 0s

If the input are lists, we need to check their lengths and possibly add 0s to the front to make them the same length. We can get the length of a list using the **nops** command, short for number of operands. It allows only one argument, which can be any expression. Its most common use is for determining the number of elements of a list or set.

```
> nops([1, 0, 1, 1, 0, 1])
6
```

(1.40)

To compare the lengths of the two lists, we use the **max** command, which returns the largest of its arguments.

```
> max(3, 5)
5
```

(1.41)

Closely related to **nops** is the **op** command. This is another command with a variety of uses. In this case, we will use **op** to extract the elements of a list. To see why this is needed, consider the problem of adding a 0 to a list. The natural approach is to create a new list that has 0 as the first element, followed by the rest of the original list. You might try this:

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

(1.42)

```
> [0, exList]
[0, [1, 1, 1, 1]]
```

(1.43)

Instead of obtaining a list consisting of 0 followed by the **exList** elements, we get a list of two elements. The first element is 0 and the second element is the list **exList**. The elements of a list can be anything, including other lists. You can think of **exList** as a box in which we have stored five 1s. When we made the new list above, we made a new box and put a 0 and the **exList** in it producing a box within a box. In order to get the list we want, we need to extract the elements of the **exList**. We use **op** to take the data out of the **exList** box so that we can put them directly in this new box.

```
> [0, op(exList)]
[0, 1, 1, 1, 1]
```

(1.44)

In order to add more than one 0 at a time, we will use the **seq** command. This command creates a sequence of values. In its most basic form, **seq** requires two arguments. The first argument is an

expression that determines the values of the sequence. The second argument has the form **i=m..n**, where **i** is an index variable that, like in a for loop, is sequentially assigned the integers between **m** and **n**, inclusive. Typically, the first argument will depend in some way on the value of **i**. For example, to produce the first few perfect squares, we can do the following.

```
> seq(i^2,i=1..10);
1, 4, 9, 16, 25, 36, 49, 64, 81, 100
```

(1.45)

In our **AND** procedure, we will want all 0s, so the first argument will be 0 and the second argument will control how many are to be produced. For example, to make a sequence of three 0s, we can issue the following command.

```
> seq(0,i = 1 ..3)
0, 0, 0
```

(1.46)

Note that we can put this in place of the single 0 from above to extend the **exList**.

```
> [seq(0,i = 1 ..3), op(exList)]
[0, 0, 0, 1, 1, 1]
```

(1.47)

In addition, the bounds of the range can be expressions that depend on another variable. For example,

```
> upperBound := 10:
> seq(0,i = 1 ..upperBound - 2)
0, 0, 0, 0, 0, 0, 0, 0
```

(1.48)

For example, if we want to extend **exList** to a list of length 10, we would need to add

```
> upperBound - nops(exList)
6
```

(1.49)

0s to the front of **exList**. We can obtain these six 0s by

```
> seq(0,i = 1 ..upperBound - nops(exList))
0, 0, 0, 0, 0, 0
```

(1.50)

The new list can thus be obtained as follows:

```
> [seq(0,i = 1 ..upperBound - nops(exList)), op(exList)]
[0, 0, 0, 0, 0, 0, 1, 1, 1]
```

(1.51)

```
> nops((1.51))
10
```

(1.52)

Finally, if the range $i=m..n$ is empty, that is, n is less than m , then `seq` outputs nothing. This means that we do not need to test the length of `exList` against the desired length because `seq` will not add anything if it does not need to.

```
> smallBound := 4 :  
> [seq(0, i = 1 .. smallBound - nops(exList)), op(exList)]  
[1, 1, 1, 1] (1.53)
```

We have all the pieces in place to write the new and improved **AND** procedure.

```
1 AND := proc(A, B)  
2   local i, maxlen, newA, newB;  
3   if (A=0 or A=1) and (B=0 or B=1) then  
4     if A=1 and B=1 then  
5       return 1;  
6     else  
7       return 0;  
8     end if;  
9   elif type(A, list) and type(B, list) then  
10    maxlen := max(nops(A), nops(B));  
11    newA := [seq(0, i=1..maxlen-nops(A)), op(A)];  
12    newB := [seq(0, i=1..maxlen-nops(B)), op(B)];  
13    return zip(AND, newA, newB);  
14  else  
15    error "Only bits and bit strings are allowed.";  
16  end if;  
17 end proc;
```

```
> AND([1, 1, 1], [1, 0, 1, 1])  
[0, 0, 1, 1] (1.54)
```

We leave the procedures for the other operations, NOT, OR, and XOR, to the reader.

1.2 Applications of Propositional Logic

In this section, we will describe how Maple's computational abilities can be used to solve applied problems in propositional logic. In particular, we will consider consistency for system specifications and Smullyan logic puzzles.

System Specifications

The textbook describes how system specifications can be translated into propositional logic and how it is important that the specifications be consistent. As suggested by the textbook, one way to determine whether a set of specifications is consistent is with truth tables.

Recall that a collection of propositions is consistent when there is an assignment of truth values to the propositional variables which makes all of the propositions in the collection true simultaneously. For example, consider the following collection of compound propositions: $p \rightarrow q \wedge r$, $p \vee q$, and $p \vee \neg r$. We can see that these propositions are consistent because we can satisfy all three with the

assignment $p = \text{false}$, $q = \text{true}$, $r = \text{false}$. In Maple, we can confirm this by evaluating the list of propositions with that assignment of truth values.

```
> eval([p implies q and r, p or q, p or not r], [p = false, q = true, r = false])
[true, true, true] (1.55)
```

To determine if a collection of propositions is consistent, we can create a truth table.

Consider Example 4 from Section 1.2 of the text. We translate the three specifications as the following list of propositions.

```
> specEx4 := [p or q, not p, p implies q]
specEx4 := [p or q, not p, p => q] (1.56)
```

Then, we can construct the truth table exactly as we did in the previous section.

```
> for loopQ in [true, false] do
    for loopP in [true, false] do
        print(loopP, loopQ, eval(specEx4, [p = loopP, q = loopQ]));
    end do;
end do;
true, true, [true, false, true]
false, true, [true, true, true]
true, false, [true, false, false]
false, false, [false, true, true] (1.57)
```

We see that the only assignment of truth values that results in all three statements being satisfied is with $p = \text{false}$ and $q = \text{true}$.

We can make the output a bit easier to read if, instead of considering the truth table for the list of the propositions, we consider the proposition formed by the conjunction of the individual propositions: $(p \vee q) \wedge (\neg p) \wedge (p \Rightarrow q)$.

```
> specEx4b := (p or q) and (not p) and (p => q)
specEx4b := (p or q) and not p and (p => q) (1.58)
```

We can then build the truth table with loops.

```
> for loopQ in [true, false] do
    for loopP in [true, false] do
        print(loopP, loopQ, eval(specEx4b, [p = loopP, q = loopQ]));
    end do;
end do;
true, true, false
false, true, true
true, false, false
false, false, false (1.59)
```

Alternatively, we can use the **TruthTable** command from the **Logic** package. Note that this command could not be used when the system specifications were provided as a list, because the command requires its argument to be a single Boolean expression. Recall that we earlier used the **with** command to make the short form of the package command names available.

> <i>TruthTable</i> (specEx4b)		
<i>p</i>	<i>q</i>	<i>value</i>
1	false	false
2	false	true
3	true	false
4	true	false

(1.60)

Using either method, we can see that the final truth value in the second row is true, which means that the assignment of truth values satisfies all of the propositions in the system specification.

The **Logic** package also includes functions for checking for consistency that are more efficient than inspecting the truth table. The **Satisfiable** command accepts the same argument as **TruthTable**, namely a Boolean expression. Note that you may not give a list of expressions as the argument of **Satisfiable**.

> <i>Satisfiable</i> (specEx4b)	
	true

(1.61)

The **Satisfy** command will generate an assignment of truth values to the variables that do in fact satisfy the proposition, assuming it is satisfiable.

> <i>Satisfy</i> (specEx4b)	
	{ <i>p</i> = false, <i>q</i> = true}

(1.62)

If we add, as in Example 5, the proposition $\neg q$, we see that all of the assignments yield false for the conjunction of all four propositions.

> specEx5 := specEx4b and (not q)	
	specEx5 := (<i>p</i> or <i>q</i>) and not <i>p</i> and (<i>p</i> \Rightarrow <i>q</i>) and not <i>q</i>

(1.63)

> <i>TruthTable</i> (specEx5)		
<i>p</i>	<i>q</i>	<i>value</i>
1	false	false
2	false	false
3	true	false
4	true	false

(1.64)

In addition, note that **Satisfiable** returns **false** and **Satisfy** returns **NULL**, producing no output, to indicate that the proposition is not satisfiable.

```
> Satisfiable(specEx5)
false
> Satisfy(specEx5)
```

(1.65)

Logic Puzzles

Recall the knights and knaves puzzle presented in Example 7 of Section 1.2 of the text. In this puzzle, you are asked to imagine an island on which each inhabitant is either a knight and always tells the truth or is a knave and always lies. You meet two people named A and B. Person A says “B is a knight” and person B says “The two of us are opposite types.” The puzzle is to determine who kind of inhabitants A and B are.

We can solve this problem with Maple using truth tables. First, we must write A and B’s statements as propositions. Let a represent the statement that A is a knight and b the statement that B is a knight. Then, A’s statement is “ b ” and B’s statement is “ $a \wedge \neg b \vee \neg a \wedge b$,” as discussed in the text.

While these propositions precisely express the content of A and B’s assertions, it does not capture the additional information that A and B are making the statements. We know, for instance, that A either always tells the truth (knight) or always lies (knave). If A is a knight, then we know the statement “ b ” is true. If A is not a knight, then we know the statement is false. In other words, the truth value of the proposition a , that A is a knight, is the same as the truth value of A’s statement, and likewise for B. Therefore, we can capture the meaning of “A says proposition p ” by the proposition $a \leftrightarrow p$. We can express the two statements in the puzzle in Maple as follows.

```
> Ex7A := a iff b
Ex7A := (a ⇒ b) and (b ⇒ a)
> Ex7B := b iff ((a and not b) or (not a and b))
Ex7B := (b ⇒ a and not b or not a and b) and
(a and not b or not a and b ⇒ b)
```

(1.66)(1.67)

Like the system specifications above, a solution to this puzzle will consist of an assignment of truth values to the propositions a and b that make both people’s statements true.

```
> Satisfy(Ex7A and Ex7B)
{a = false, b = false}
```

(1.68)

We see that both statements are satisfied when both a and b are false. That is, when A and B are both knaves. Both individuals being knaves is, in fact, the only possibility consistent with their statements, as a truth table can confirm.

1.3 Propositional Equivalences

In this section, we consider logical equivalence of propositions. We will first look at the built-in functions for testing equivalence, and then we will create a function from scratch to accomplish the

same goal. Additionally, we will use Mathematica to solve the n -Queens problem as a satisfiability problem.

Built-In Functions

The **Logic** package includes the command **Equivalent** to determine whether logical propositions are equivalent. There are two required arguments: the two Boolean expressions.

As mentioned above, the **Logic** package operators, such as **&and**, are subtly different from the main operators, such as **and**. However, it is not usually required to use the **Logic** package operators, since Maple will convert them automatically. We can therefore use **Equivalent** to confirm one of De Morgan's laws as illustrated below.

```
> Equivalent(not(p and q), (not p) or (not q))  
true
```

(1.69)

The **Equivalent** command can also accept a name as an optional third argument. If the two propositions are not equivalent, then an assignment of truth values to variables that demonstrates their nonequivalence will be stored in the name.

```
> Equivalent(not(p and q), (not p) or (not q), valuation)  
false
```

(1.70)

```
> valuation  
{p = false, q = true}
```

(1.71)

This output indicates that when p is false and q true, the propositions $\neg(p \wedge q)$ and $\neg p \wedge \neg q$ evaluate to different truth values.

Built from Scratch Function

The **Logic** package provides built-in support for working with logical propositions and, in particular, checking propositional equivalence. Here, however, we are going to build a new function for checking whether or not two propositions are logically equivalent using a minimum of high-level functions. In fact, other than asking Maple to evaluate propositional expressions for particular truth values assigned to propositional variables, we will make use only of the Maple programming language's essential programming functionality.

There are two goals here. First, to illustrate more of Maple's programming abilities. Second, to reveal some of the more fundamental concepts and methods used in the Maple programming language.

Recall that propositions p and q are said to be logically equivalent if the biconditional $p \leftrightarrow q$ is a tautology. With Maple, we can test logical equivalence fairly easily by producing a truth table for the biconditional. For example, we can demonstrate De Morgan's laws as follows.

```
> demorgan1 := (not(p and q)) iff (not p or not q) colon  
> demorgan2 := (not(p or q)) iff (not p and not q) :
```

```

> for loopP in [true, false] do
    for loopQ in [true, false] do
        print(loopP, loopQ, eval(demorgan1, [p = loopP, q = loopQ]));
    end do;
end do;
true, true, true
true, false, true
false, true, true
false, false, true

```

(1.72)

```

> for loopP in [true, false] do
    for loopQ in [true, false] do
        print(loopP, loopQ, eval(demorgan2, [p = loopP, q = loopQ]));
    end do;
end do;
true, true, true
true, false, true
false, true, true
false, false, true

```

(1.73)

We would like to have a procedure, **AreEquivalent**, that would take two propositions and determine whether or not they are equivalent. Such a proposition will be our next goal, but it will require quite a bit of work. The main hurdles for such a procedure are: (1) having Maple determine what propositional variables are used in the given compound propositions, and (2) without *a priori* knowledge of the number of propositional variables, having Maple test every possible assignment of truth values. Note that we could avoid both of these hurdles by insisting that the propositional variables be limited to a certain small set of names, perhaps p , q , r , and s . Then, we could implement the procedure as four nested for loops. Often, not all four would be needed, which would add redundancy but would not impact functionality.

However, the two hurdles mentioned are not insurmountable, will provide a much more elegant and flexible procedure, and will also give us the opportunity to see examples of some important programming constructs.

Extracting Variables

The first hurdle is to get Maple to determine the variables used in a logical expression. Consider the following example.

```

> variableEx := ((p and q) or (p and not r)) and (s implies r)
variableEx := (p and q or p and not r) and (s ⇒ r)

```

(1.74)

Our goal is to write a procedure that will, given the above expression, tells us that the variables in use are p , q , r , and s .

The op Command and the Name Type

We have already discussed how the **op** command applied to a list returns the sequence underlying the list. However, **op** can be applied to any expression. If you apply **op** to an algebraic expression like **a*b**, it will return the sequence consisting of the operands a and b .

> $\text{op}(a \cdot b)$
 a, b (1.75)

Applied to a more complicated algebraic expression, say **a*b+c**, **op** effectively obeys order of operations. In this expression, the addition is the last operation to be performed. In other words, the expression **a*b+c** is the sum of **a*b** and **c**. Thus, **op** returns **a*b** and **c** as the operands to the addition.

> $\text{op}(a \cdot b + c)$
 $a \cdot b, c$ (1.76)

(Note the multiplication sign is not printed but is implied.)

If the expression involved multiple additions, Maple considers the expression to be an addition of three terms. Note that the fraction is obtained by typing **b/c**.

> $\text{op}\left(a + \frac{b}{c} + d\right)$
 $a, \frac{b}{c}, d$ (1.77)

Boolean expressions are essentially the same, except **not**, **and**, and **or** are the operators instead of the arithmetic operations.

> $\text{op}(\text{variableEx})$
 $p \text{ and } q \text{ or } p \text{ and not } r, s \Rightarrow r$ (1.78)

In this case, **op** indicates that the Boolean expression had two operands which were joined by the final **and** operator. Note that **op** has the effect of removing the operator.

If a call to a procedure is involved, the result is the sequence of arguments of the procedure. For instance,

> $\text{op}(\text{not}(p \text{ and } q) \text{ implies } (p \text{ or not } q))$
 $\text{not}(p \text{ and } q), (p \text{ or not } q)$ (1.79)

Since we are trying to obtain the variables in use, this is ideal. Our strategy is to keep applying the **op** command until we get down to variables. Doing this is a bit more complicated than saying it, of course.

How do we know when we have a variable as opposed to a compound proposition? In Maple, whenever you need to distinguish between different kinds of things, it makes sense to consider types. In fact, one of Maple's fundamental types is the **name** type. We use the **type** command as follows to see that $s \rightarrow r$ is not a name, but that s is.

> $\text{type}(s \Rightarrow r, \text{name})$
 false (1.80)

> $\text{type}(s, \text{name})$
 true (1.81)

Illustrating with an Example

We can now remove operators to obtain simpler expressions, and we have a way to test whether an expression is a variable or not. The general idea is that we keep applying the **op** command to the operands until we are down to nothing but names.

It is natural to create a list that will store the results of intermediate steps, so we start by initializing it to the list consisting of just one element, the expression we are analyzing.

```
> varExList := [variableEx]
varExList := [(p and q or p and not r) and (s => r)]
```

(1.82)

To apply the **op** command to our expression, which is now the first element of the list, we use the list selection operation.

```
> op(varExList[1])
p and q or p and not r, s => r
```

(1.83)

Next, we replace the original expression with the result. We can replace elements in a list with the **subsop** command. This command can be used in a variety of ways depending on the particular arguments given. The form we use here will include two arguments. The second argument will be the list **varExList**. The first argument will be an equation of the form **i=e**, where **i** is an integer that refers to an index in the list and **e** is the expression that will replace whatever is currently in that position of the list. For example, we can replace the 5 with a 6 in the list **[2,4,5,8,10]** with the following command.

```
> subsop(3 = 6, [2, 4, 5, 8, 10])
[2, 4, 6, 8, 10]
```

(1.84)

We emphasize that the left side of the **i=e** equation is an integer representing the index (i.e., the location) of the element of the list that we are replacing while the right side is the replacement. In the above, the element 5 is in position 3 and is being replaced by the element 6, hence the use of **3=6**. Also note that this command does not modify an existing list (it creates a new list object), so we will need to reassign the result to the name of our list.

Using **subsop** on **varExList**, we obtain the following.

```
> varExList := subsop(1 = op(varExList[1]), varExList)
varExList := [p and q or p and not r, s => r]
```

(1.85)

The above command is substituting, for the first element of **varExList**, the result of applying the **op** command to the first element of **varExList**. Now repeat.

```
> varExList := subsop(1 = op(varExList[1]), varExList)
varExList := [p and q, p and not r, s => r]
```

(1.86)

```
> varExList := subsop(1 = op(varExList[1]), varExList)
varExList := [p, q, p and not r, s => r]
```

(1.87)

Observe that it required three applications of **op** to the first element of the list. Continued use of **op** on **varExList[1]** will have no effect since the first element is a name. Likewise, the second element is a name, so we move to element 3.

```
> varExList := subsop (3 = op (varExList[3]), varExList)
varExList := [p, q, p, not r, s ⇒ r] (1.88)
```

The Procedure

The explicit example gives us the outline of our procedure:

1. Initialize a list, **L**, to the list consisting of the input expression as the sole element. Initialize an index variable, **i**, to one.
2. Test, using **type**, to see if the element of **L** referred to by the index **i** is a **name** or an expression. If it is a **name**, move on to the next element of the list by incrementing **i**.
3. If **L[i]** is not a **name**, the substitute that element of **L** with the result of applying the command **op** to it.
4. Repeat steps 2 and 3 until reaching the end of the list. This repetition is controlled by a **while** loop which continues as long as **i** does not exceed the length of the list.

Here is the implementation.

```

1 GetVars := proc (exp)
2   local L, i, j;
3   L := [exp];
4   i := 1;
5   while i <= nops (L) do
6     if type (L[i], name) then
7       i := i + 1;
8     else
9       L := subsop (i=op (L[i]), L);
10    end if;
11  end do;
12  return [op ({op (L)})];
13 end proc;
```

Note that the last line of this procedure makes use of Maple's set object to remove repetitions from the list of variables. We turn the list **L** into a set by applying **op** and enclosing the result in braces and then back into a list by applying **op** and enclosing it in brackets.

```
> GetVars (((p and q) or (p and not r)) and (s implies r))
[p, q, r, s] (1.89)
```

```
> GetVars (prop23 implies ((Q or q) iff (P and p)))
[P, Q, p, prop23, q] (1.90)
```

Truth Value Assignments

The second hurdle that we mentioned at the beginning of this section is that we do not know the number of propositional variables in advance. If we knew there would always be two variables, we would use two nested for loops. However, since we want our procedure to work with any number of variables, we need a different approach.

First, note that since our **GetVars** procedure produces a list of variables, it is natural to model an assignment of truth values to variables as a list of truth values. For example,

```
> variableExVars := GetVars(variableEx)
variableExVars := [p, q, r, s] (1.91)
```

```
> TAex := [true, true, false, true]
TAex := [true, true, false, true] (1.92)
```

We consider the **TAex** list (for Truth Assignment example) to indicate that we assign the first variable of **variableExVars** to true, the second variable to true, the third to false, and the fourth to true.

The eval Command

Recall the use of the **eval** command introduced above. In particular, by giving a list of equations of the form **var=val** as the second argument, we can evaluate the truth value of the proposition specified in the first argument with the assignments of the **vals** to the **vars**.

```
> eval(variableEx, [p = true, q = true, r = false, s = true])
false (1.93)
```

We can produce a list of assignments as follows. We saw how the **zip** command applies a binary function to the elements of two lists. We will use **zip** with the function that takes a variable and a truth value and creates the appropriate equation. A numeric example will be most illustrative, but the syntax is identical for logical expressions.

```
> zip((a, b) → a = b, [x, y, z], [5, 3, 9])
[x = 5, y = 3, z = 9] (1.94)
```

```
> eval(x + 2y - z, zip((a, b) → a = b, [x, y, z], [5, 3, 9]))
2 (1.95)
```

Indeed, $5 + 2 \cdot 3 - 9 = 2$. Note that the first argument to **zip**, **(a,b) -> a=b**, is a functional operator. A functional operator is a particular kind of procedure designed to mimic function notation. The arrow is formed by typing a hyphen followed by a greater-than symbol (**->**). On the left-hand side of the arrow are the input variables, and on the right-hand side is an expression in terms of the variables that yields the value of the function.

The same approach will work with the assignment of truth values to propositional variables.

```
> eval(variableEx, zip((a, b) → a = b, variableExVars, TAex))
false (1.96)
```

Finding All Possible Truth Assignments

Now that we know that we can effectively use lists of truth values to represent truth value assignments, we need a way to produce all such lists. We will use a strategy similar to binary counting. Start with the list of all falses. Get the next list by changing the first element to true. For the next assignment, change the first element back to false and the second element to true. Then change the first element to true. Then, change the first true to false, the second true to false, and the third element becomes true. Continue in this pattern: given a list of truth values,

obtain the next list by changing the left-most false to true and changing all trues up until that first false into false. (It is left to the reader to verify that this produces all possible truth value assignments.)

We implement this idea in the **NextTA** procedure (for Next Truth Assignment). The **NextTA** procedure will accept a list of truth values as input and return the “next” list. The main work of this procedure is done inside of a **for** loop. The for loop considers each position in the list of truth values in turn. If the value in the current position is true, then it is changed to false. On the other hand, if the value is false, then it is changed to true and the procedure is terminated by returning the list of truth values. If the for loop ends without having returned a new list, then the input to the procedure was all trues, which is the last possibility, and the procedure returns **NULL** to indicate that there is no next truth assignment.

```

1  NextTA := proc (A)
2    local i, new;
3    new := A;
4    for i from 1 to nops (A) do
5      if new[i] then
6        new[i] := false;
7      else
8        new[i] := true;
9        return new;
10     end if;
11   end do;
12   return NULL;
13 end proc:
```

Note that the **new** variable is necessary because procedure arguments cannot be assigned to. Also note that, in this procedure, we are able to modify list elements using the assignment operator with the **list[index] := value;** syntax. This was not possible in **GetVars**, because in that case we were potentially replacing a single element with a sequence of values.

Logical Equivalence Implementation

We now have the necessary pieces in place to write the promised **AreEquivalent** procedure. This procedure accepts two propositions as arguments and returns true if they are equivalent and false otherwise.

The procedure proceeds as follows:

1. First, we assign the function **(a,b) -> a=b** used in the **zip** command to the name **eqZip**. (This is a bit more efficient than including the definition of the function in the **zip** command.)
2. Then, we create the biconditional, which we name **Bicond**, that asserts the equivalence of the two propositions. Note that we form the biconditional using **implies** and **and** since **iff** is not a fundamental logical operator in Maple. We also use the **GetVars** procedure to determine the list of variables used in the propositions and initialize the truth assignment variable **TA** to the appropriately sized list of all false values.
3. Then, we begin a while loop. As long as **TA** is not **NULL**, we evaluate the biconditional **Bicond** on the truth assignment. If this truth value is false, we know that the biconditional is not a tautology and thus the propositions are not equivalent and we immediately return false. Otherwise, we use **NextTA** to update **TA** to the next truth assignment.

4. If the while loop terminates, it indicates that all possible truth assignments have been applied to the biconditional and that each one evaluated true; otherwise, the procedure would have returned false and terminated. Hence, the biconditional is a tautology and true is returned.

Here is the implementation.

```

1 AreEquivalent := proc (P, Q)
2   local eqZip, Bicond, Vars, numVars, i, TA, val;
3   eqZip := (a, b) -> a=b;
4   Bicond := (P implies Q) and (Q implies P);
5   Vars := GetVars(Bicond);
6   numVars := nops(Vars);
7   TA := [seq(false, i=1..numVars)];
8   while TA <> NULL do
9     val := eval(Bicond, zip(eqZip, Vars, TA));
10    if not val then
11      return false;
12    end if;
13    TA := NextTA(TA);
14  end do;
15  return true;
16 end proc;
```

We can use this to computationally verify that $\neg(p \vee (\neg p \wedge q)) \equiv \neg p \wedge \neg q$. This was shown in Example 7 of Section 1.3 of the text via equivalences.

> *AreEquivalent (not (p or (not p and q)), not p and not q)*
true

(1.97)

The *n*-Queens Problem

The textbook describes the *n*-Queens problem and shows how propositions can be formed to solve it via propositional satisfiability. With Maple's **Satisfy** command, the main challenge is building the propositions. This will give us an opportunity to see how we can write procedures that, instead of computing a value, actually create objects that can be used in other procedures. In particular, we will define **nQueens1** through **nQueens5** whose outputs are the propositions Q_1 through Q_5 , as defined in the main text.

First, we will need to form the basic propositions, those the main text refers to as $p(i, j)$. Maple will allow us to use indexed expressions such as $p_{2,3}$ as propositional variables. These are entered on the keyboard as **p[2,3]**.

> *p[2,3]*
p_{2,3}

(1.98)

We can use **seq** nested within itself to create the list of propositions.

> *seq (seq (p[i,j], i = 1 .. 4), j = 1 .. 4)*
P_{1,1}, P_{2,1}, P_{3,1}, P_{4,1}, P_{1,2}, P_{2,2}, P_{3,2}, P_{4,2}, P_{1,3}, P_{2,3}, P_{3,3}, P_{4,3}, P_{1,4}, P_{2,4}, P_{3,4}, P_{4,4}

(1.99)

Now we will form the proposition $Q_1 = \bigwedge_{i=1}^n \bigvee_{j=1}^n p_{i,j}$. Initially, consider just the inner disjunction, $\bigvee_{j=1}^n p_{i,j}$. We can produce this disjunction by using **seq** to create the sequence of propositions and then transforming the sequence into a disjunction by applying **or** to the sequence. Previously, we have used the logical operators only between the propositions they join. But the logical operators can also be applied at the head of a parenthesized sequence of parameters, like typical procedures. For example, $p \vee q \vee r$ can be formed as shown below.

```
> or(p, q, r)
      p or q or r (1.100)
```

For $n = 7$, we apply **seq** and conjunction with **or**.

```
> or(seq(p[i,j], j = 1 .. 7))
      p..1 or p..2 or p..3 or p..4 or p..5 or p..6 or p..7 (1.101)
```

To complete Q_1 , we simply make that the first argument of another use of **seq** and apply **and**. We add some line breaks to make the structure of the command clearer, though they are not necessary for Maple.

```
> and(seq(or(seq(p[i,j], j = 1 .. 7)),
      i = 1 .. 7))
      (p..1,1 or p..1,2 or p..1,3 or p..1,4 or p..1,5 or p..1,6 or p..1,7) and (p..2,1 or p..2,2 or
      p..2,3 or p..2,4 or p..2,5 or p..2,6 or p..2,7) and (p..3,1 or p..3,2 or p..3,3 or p..3,4 or
      p..3,5 or p..3,6 or p..3,7) and (p..4,1 or p..4,2 or p..4,3 or p..4,4 or p..4,5 or p..4,6 or
      p..4,7) and (p..5,1 or p..5,2 or p..5,3 or p..5,4 or p..5,5 or p..5,6 or p..5,7) and
      (p..6,1 or p..6,2 or p..6,3 or p..6,4 or p..6,5 or p..6,6 or p..6,7) and (p..7,1 or p..7,2 or
      p..7,3 or p..7,4 or p..7,5 or p..7,6 or p..7,7) (1.102)
```

We generalize this process into a functional operator by replacing the specific value 7 with a parameter.

```
> nQueens1 := n → and(seq(or(seq(p[i,j], i = 1 .. n)), j = 1 .. n)):
```

Other than needing to use the **min** function for Q_4 and Q_5 , the other propositions can be formed similarly. The definitions of the propositions are displayed below, followed by the definitions of the functions.

$$\begin{aligned} Q_2 &= \bigwedge_{i=1}^n \bigwedge_{j=1}^{n-1} \bigwedge_{k=j+1}^n (\neg p_{i,j} \vee \neg p_{i,k}) \\ Q_3 &= \bigwedge_{j=1}^n \bigwedge_{i=1}^{n-1} \bigwedge_{k=i+1}^n (\neg p_{i,j} \vee \neg p_{k,j}) \\ Q_4 &= \bigwedge_{i=2}^n \bigwedge_{j=1}^{n-1} \bigwedge_{k=1}^{\min(i-1, n-j)} (\neg p_{i,j} \vee \neg p_{i-k, k+j}) \\ Q_5 &= \bigwedge_{i=1}^{n-1} \bigwedge_{j=1}^{n-1} \bigwedge_{k=1}^{\min(n-i, n-j)} (\neg p_{i,j} \vee \neg p_{i+k, j+k}) \end{aligned}$$

```

> nQueens2 := n → and(seq(and(seq(and(seq((not p[i,j]) or (not p[i,k]), k = j + 1 ..n)), j = 1 ..n - 1)), i = 1 ..n)) :
> nQueens3 := n → and(seq(and(seq(and(seq((not p[i,j]) or (not p[k,j]), k = i + 1 ..n)), i = 1 ..n - 1)), j = 1 ..n)) :
> nQueens4 := n → and(seq(and(seq(and(seq((not p[i,j]) or (not p[i - k, k + j]), k = 1 ..min(i - 1, n - j))), j = 1 ..n - 1)), i = 2 ..n)) :
> nQueens5 := n → and(seq(and(seq(and(seq((not p[i,j]) or (not p[i + k, j + k]), k = i ..min(n - i, n - j))), j = 1 ..n - 1)), i = 1 ..n - 1)) :

```

With these functions in place to form the pieces of the full proposition describing the n -Queens problem, we create a procedure to solve the problem. The procedure will first form the proposition. We then check whether the proposition is satisfiable or not, using **Satisfiable** from the **Logic** package. If not, the procedure will return false. However, if there is a solution, then we use **Satisfy** to determine the solution.

```

1 nQueens := proc (n)
2   local prop;
3   prop := nQueens1 (n) and nQueens2 (n) and nQueens3 (n) and
4     nQueens4 (n) and nQueens5 (n);
5   if not Logic[Satisfiable] (prop) then
6     return false;
7   else
8     return Logic[Satisfy] (prop);
9   end if;
end proc;

```

Note that we use the long form version of calls to the **Logic** package functions. This is good programming practice as it ensures that the procedures you write do not depend on your having remembered to execute the **with** command.

Observe that the output of the procedure produces a solution of the 8-Queens problem different from the one shown in the main text.

```

> nQueens(8)
{p1,1 = false, p1,2 = false, p1,3 = false, p1,4 = false, p1,5 = true,
 p1,6 = false, p1,7 = false, p1,8 = false, p2,1 = false, p2,2 = false,
 p2,3 = false, p2,4 = false, p2,5 = false, p2,6 = false, p2,7 = true,
 p2,8 = false, p3,1 = false, p3,2 = true, p3,3 = false, p3,4 = false,
 p3,5 = false, p3,6 = false, p3,7 = false, p3,8 = false, p4,1 = false,
 p4,2 = false, p4,3 = false, p4,4 = false, p4,5 = false, p4,6 = true,
 p4,7 = false, p4,8 = false, p5,1 = false, p5,2 = false, p5,3 = true,
 p5,4 = false, p5,5 = false, p5,6 = false, p5,7 = false, p5,8 = false,
 p6,1 = true, p6,2 = false, p6,3 = false, p6,4 = false, p6,5 = false,
 p6,6 = false, p6,7 = false, p6,8 = false, p7,1 = false, p7,2 = false,
 p7,3 = false, p7,4 = true, p7,5 = false, p7,6 = false, p7,7 = false,
 p7,8 = false, p8,1 = false, p8,2 = false, p8,3 = false, p8,4 = false,
 p8,5 = false, p8,6 = false, p8,7 = false, p8,8 = true} (1.103)

```

1.4 Predicates and Quantifiers

In this section, we will see how Maple can be used to explore propositional functions and their quantification over a finite universe. We can think about a propositional function P as a function, or procedure, that takes as input a member of the domain and that outputs a truth value.

For example, let $P(x)$ denote the statement “ $x > 0$.” We will construct a procedure that takes x as input and returns true or false as appropriate. However, note that Maple does not automatically evaluate inequalities to their truth value.

```
> 3 > 0  
0 < 3
```

(1.104)

To have Maple evaluate such expressions to true or false, use the command **evalb**, for evaluate Boolean.

```
> evalb(3 > 0)  
true
```

(1.105)

We construct our procedure using the functional operator syntax that was discussed in the previous section. In this case, we assign the function to the name **GT0** (for greater than 0) and we include **evalb** in the definition of the propositional function.

```
> GT0 := x → evalb(x > 0)  
GT0 := x ↦ evalb(0 < x)
```

(1.106)

```
> GT0(3)  
true
```

(1.107)

```
> GT0(-2)  
false
```

(1.108)

Quantifiers

For finite domains, we can use Maple to determine the truth value of universally and existentially quantified statements. We will create two procedures, **Universal** and **Existential**. Both of these procedures will accept two arguments: a propositional function of one variable and a list or set representing the domain.

First, consider universal quantification. This procedure should return true only when all members of the domain satisfy the propositional function and false if even one element of the domain fails to satisfy the predicate. The procedure will loop through the elements of the domain. If the propositional function ever returns false, then the procedure will immediately return false. On the other hand, if we get through the loop without encountering false, then we can conclude that the universally quantified statement is true.

```
1 Universal := proc(P, D)  
2   local d;  
3   for d in D do  
4     if not P(d) then  
5       return false;
```

```

6      end if ;
7  end do ;
8  return true ;
9 end proc;

```

> *Universal*($GT0, [1, 2, 3, 4, 5]$)

true

(1.109)

> *Universal*($GT0, [1, 2, 3, 4, 5, -7]$)

false

(1.110)

For existential quantification, if we ever encounter a member of the domain for which the propositional function returns true, then the existential statement is true. If the for loop terminates without having found such a value, then we return false.

```

1 Existential := proc (P, D)
2   local d;
3   for d in D do
4     if P(d) then
5       return true;
6     end if;
7   end do;
8   return false;
9 end proc;

```

> *Existential*($GT0, [-3, -4, -5, -6, -7]$)

false

(1.111)

> *Existential*($GT0, [-3, -4, -5, 11, -7]$)

true

(1.112)

1.5 Nested Quantifiers

In this section, we consider propositions with nested quantifiers such as $\exists y \forall x (x + y = 0)$ and $\forall y \exists x (x + y = 0)$. These propositions are the subject of Example 4 of Section 1.5 in the textbook. Recall that the first statement is false while the second is true. Nested quantification can be difficult to understand, but writing procedures to test propositions such as those two examples can help make their interpretation clearer.

We will create a procedure, **ExistsForAll**, which will accept two arguments, a propositional function with two input values and a domain which will be considered common for both variables. Consider the example mentioned in the previous paragraph:

> *sumto0* := $(x, y) \rightarrow evalb(x + y = 0)$

sumto0 := $(x, y) \mapsto evalb(x + y = 0)$

(1.113)

> *sumto0Domain* := [*seq*($i, i = -10..10$)]

sumto0Domain := $[-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$

(1.114)

Now, we will see how to write **ExistsForAll**. Note that we will keep our variables consistent with those in the examples above. That is, the “outer,” existentially quantified, variable will be y and the “inner,” universally quantified, variable will be x . To see how to write this procedure, think about how we would go about testing $\exists y \forall x (x + y = 0)$. The claim is that there is some y so that all values of x satisfy the propositional function. Therefore, we will test each value of y in turn. Suppose we start with -10 . Then, we will check to see if $\forall x (x - 10 = 0)$. We consider, in sequence, each possible value for x . If any value of x fails to satisfy the proposition, then we know that the universal statement is false and we can stop checking values of x and move on to the next possible y . On the other hand, if all of the x values satisfy the proposition, then we have found a y that demonstrates that the proposition is true.

We use two for loops to check all possible values of y and x , respectively. Note that the inner, x loop, can be stopped as soon as we find a value that fails to satisfy the proposition. We can abort a loop with the **break** command, which causes the innermost loop to terminate. In this procedure, that means that we move on to the next possible y .

Also note that the outer loop can be stopped as soon as we find a y value that satisfies the predicate for all possible x . This is a bit trickier to program. What we will do is to assume that a y value “works” (*i.e.*, satisfies the proposition for all x) until we discover otherwise. We set a variable, **yworks**, equal to true as the first statement of the y loop. Immediately after each new y value is chosen, **yworks** is set to true indicating that we “believe” that the current y value satisfies the requirements. If an x is found so that the proposition fails, then before using **break** to end the x loop, we set **yworks** to false. Thus, at the conclusion of the inner x loop, **yworks** is true when the current value of y causes every possible value of x to satisfy the proposition. In that case, we can abort the procedure and return true. If the y loop terminates, then no such y was found and so the procedure returns false.

Here is the procedure.

```

1 ExistsForAll := proc (P, D)
2   local x, y, yworks;
3   for y in D do
4     yworks := true;
5     for x in D do
6       if not P(x, y) then
7         yworks := false;
8         break;
9       end if ;
10      end do;
11      if yworks then
12        return true;
13      end if ;
14    end do;
15    return false;
16  end proc;
```

As was mentioned, this should return false for **sumto0**.

> *ExistsForAll(sumto0, sumto0Domain)*
false

(1.115)

On the other hand, consider the predicate that asserts that the product is 0.

```
> multo0 := (x,y) → evalb(x · y = 0)
    multo0 := (x,y) ↪ evalb(yx = 0) (1.116)
```

```
> ExistsForAll(multo0, sumto0Domain)
    true (1.117)
```

As it can be a useful way to help better understand nested quantifiers, we will leave it to the reader to write the procedure for $\forall y \exists x P(x, y)$.

1.6 Rules of Inference

In this section, we will see how Maple can be used to verify the validity of arguments in propositional logic. In particular, we will write a procedure, that, given a list of premises and a possible conclusion, will determine whether or not the conclusion necessarily follows from the premises. Recall from Definition 1 in the text that an argument is defined to be a sequence of propositions, the last of which is called the conclusion and all others are premises. Also recall that an argument p_1, p_2, \dots, p_n, q is said to be valid when $(p_1 \wedge p_2 \wedge \dots \wedge p_n) \rightarrow q$ is a tautology.

Similar to the **Equivalent** command, Maple's **Logic** package includes an **Implies** command. Note that this is different from the logical operator **implies**. The **Implies** command requires two Boolean expressions as its arguments and returns true if the first implies the second; that is, it evaluates whether or not the implication is a tautology.

As a simple example, we see that from $\neg p$ and $p \vee q$, we can conclude q . That, of course, is the disjunctive syllogism.

```
> Implies((not p) and (p or q), q)
    true (1.118)
```

Keeping in mind that an argument is a sequence of statements, to determine if an argument is valid, we need to form the conjunction of the premises, and then test whether the combined premises entail the conclusion with **Implies**. The **IsValid** procedure below will accept as input an argument, that is, a list of propositions, and return true if the argument is valid.

```
1 IsValid := proc (L::list)
2   local premises, i;
3   premises := L[1];
4   for i from 2 to nops(L)-1 do
5     premises := premises and L[i];
6   end do;
7   Logic[Implies](premises, L[-1]);
8 end proc;
```

Make note of the parameter sequence in this procedure definition. The double colon syntax allows us to specify a required type for the parameter. In this procedure, we are insisting that the argument be a list.

The main work of the algorithm is to create the premises statement, which is initialized to the first statement in the list and then the others statements, except for the last, are conjoined one at a time. Note that we use **L[-1]** to obtain the last element of the list, which is the conclusion of the argument. It is a useful property of selection that negative integers count from the end of the list.

We can use this procedure to verify that the argument described in Exercise 12 of Section 1.6 of the text is in fact valid.

> *IsValid*([(**p and t**) implies (**r or s**), **q implies (u and t)**, **u implies p**, **not s**, **q implies r**])
true (1.119)

Finding Conclusions (optional)

In the remainder of this section, we will consider a slightly different question: given a list of premises, what conclusions can you draw using valid arguments? We will approach this problem in Maple in a straightforward (and naive) way: generate possible conclusions and use **IsValid** to determine which are valid conclusions.

Making Compound Propositions

To generate possible conclusions, we will use the following procedure, **AllCompound**. This procedure takes a list of propositions and produces all possible propositions formed from one logical connective (from **and**, **or**, and **implies**) and two of the given propositions, along with the negations of the propositions. To avoid including some trivialities, we will exclude those propositions that are tautologies or contradictions.

The procedure is provided below. We provide no additional explanation other than to mention the use, once again, of the **[op({op(L)})]** structure to remove duplicates. In addition, note that the original propositions are included in the result since these are equivalent to their conjunction with themselves which the procedure includes in the output.

```

1 AllCompound := proc (L)
2   local p, q, tempL, PropList;
3   PropList := [];
4   tempL := L;
5   for p in L do
6     tempL := [op(tempL), not p];
7   end do;
8   for p in tempL do
9     for q in tempL do
10       if not Logic[Equivalent](p and q, true) and not
11         Logic[Equivalent](p and q, false) then
12         PropList := [op(PropList), p and q];
13       end if;
14       if not Logic[Equivalent](p or q, true) and not
15         Logic[Equivalent](p or q, false) then
16         PropList := [op(PropList), p or q];
17       end if;
18       if not Logic[Equivalent](p implies q, true) and not
19         Logic[Equivalent](p implies q, false) then
20         PropList := [op(PropList), p implies q];
21       end if;
22     end for;
23   end for;
24   return PropList;
25 end proc;
```

```

18     end if ;
19 end do ;
20 end do ;
21 return [ op ( { op (PropList) } ) ] ;
22 end proc;

```

Finding Valid Conclusions

We now write a procedure to explore possible conclusions given a set of premises. This procedure will take two arguments. The first will be a list of premises. The second argument, a positive integer, indicates the number of times that **AllCompound** should, recursively, be used to generate possibilities. You will generally not want to use any number other than 1 for this second value as the time requirement for this procedure can be quite substantial.

The operation of this procedure is very straightforward. First, it determines the variables used in the premises. These become the basis for the potential conclusions. Second, the procedure applies the **AllCompound** procedure to generate the propositions formed using one logical connective and two of the variables and their negations. (Note that the resulting list will include the variables with no connective as well.) Third, the procedure joins the premises into one proposition by conjunction. Finally, each possible conclusion is evaluated with the **IsValid** procedure.

```

1 FindConsequences := proc (Premises, level)
2   local Vars, possibleC, C, c, P, i;
3   Vars := GetVars (Premises) ;
4   possibleC := Vars;
5   for i from 1 to level do
6     possibleC := AllCompound (possibleC);
7   end do;
8   C := [ ];
9   P := Premises [1];
10  for i from 2 to nops (Premises) do
11    P := P and Premises [i];
12  end do;
13  for c in possibleC do
14    if Logic [Implies] (P, c) then
15      C := [ op (C), c ];
16    end if;
17  end do;
18  return C;
19 end proc;

```

Here is the result of applying **FindConsequences** to the premises of Exercise 12 with only one iteration of **AllCompound**. (With two iterations of **AllCompound**, the procedure takes quite some time to complete and produces thousands of valid conclusions.)

- > *FindConsequences([(p and t) implies (r or s), q implies (u and t),
u implies p, not s], 1)*

[**not** *s*, **not** (*p* **and** *s*), **not** (*q* **and** *s*), **not** (*r* **and** *s*), **not** (*s* **and** *p*),
not (*s* **and** *q*), **not** (*s* **and** *r*), **not** (*s* **and** *t*), **not** (*s* **and** *u*),
not (*t* **and** *s*), **not** (*u* **and** *s*), *p* **or** **not** *q*, *p* **or** **not** *s*, *p* **or** **not** *u*,
q **or** **not** *s*, *r* **or** **not** *q*, *r* **or** **not** *s*, *t* **or** **not** *q*, *t* **or** **not** *s*, *u* **or** **not** *q*,
u **or** **not** *s*, **not** *q* **or** *p*, **not** *q* **or** *r*, **not** *q* **or** *t*, **not** *q* **or** *u*, **not** *s* **or** *p*,
not *s* **or** *q*, **not** *s* **or** *r*, **not** *s* **or** *t*, **not** *s* **or** *u*, **not** *u* **or** *p*, *p* \Rightarrow **not** *s*,
q \Rightarrow *p*, *q* \Rightarrow *r*, *q* \Rightarrow *u*, *q* \Rightarrow **not** *s*, *r* \Rightarrow **not** *s*, *s* \Rightarrow *p*, *s* \Rightarrow *q*, *s* \Rightarrow *r*,
s \Rightarrow *t*, *s* \Rightarrow *u*, *s* \Rightarrow **not** *p*, *s* \Rightarrow **not** *q*, *s* \Rightarrow **not** *r*, *s* \Rightarrow **not** *s*, *s* \Rightarrow **not** *t*,
s \Rightarrow **not** *u*, *t* \Rightarrow **not** *s*, *u* \Rightarrow *p*, *u* \Rightarrow **not** *s*, **not** *p* \Rightarrow **not** *q*,
not *p* \Rightarrow **not** *s*, **not** *p* \Rightarrow **not** *u*, **not** *q* \Rightarrow **not** *s*, **not** *r* \Rightarrow **not** *q*,
not *r* \Rightarrow **not** *s*, **not** *t* \Rightarrow **not** *q*, **not** *t* \Rightarrow **not** *s*, **not** *u* \Rightarrow **not** *q*,
not *u* \Rightarrow **not** *s*]

(1.120)

> *nops*((1.120))

62

(1.121)

Observe that some of the conclusions are just merely restating premises. However, even after eliminating those, there are still 60 valid conclusions involving at most two of the propositional variables. Most of those conclusions are going to be fairly uninteresting in any particular context. This illustrates a fundamental difficulty with computer assisted proof. Neither checking the validity of conclusions nor generating valid conclusions from a list of premises are particularly difficult. The difficulty is in creating heuristics and other mechanisms to help direct the computer to useful results.

1.7 Introduction to Proofs

In this section, we will see how Maple can be used to find counterexamples. This is, perhaps, the proof technique most suitable to Maple's computational abilities.

Example 14 of Section 1.7 of the textbook considers the statement "Every positive integer is the sum of the squares of two integers." This is demonstrated to be false with 3 as a counterexample. Here, we will consider the related statement that "every positive integer is the sum of the squares of three integers." This statement is also false.

Finding a Counterexample

To find a counterexample, we create a procedure that, given an integer, looks for three integers such that the sum of their squares are equal to the given integer. If the procedure finds three such integers, it will return a list containing them. On the other hand, if it cannot find three such integers, it will return false. Here is the procedure:

```

1 Find3Squares := proc (n)
2   local a, b, c, max;
3   max := floor(sqrt(n));
4   for a from 0 to max do
5     for b from 0 to max do
6       for c from 0 to max do
7         if n = a^2 + b^2 + c^2 then
8           return [a, b, c];
9         end if;
10      end do;

```

```

11   end do;
12 end do;
13 return false;
14 end proc:
```

The **Find3Squares** procedure is fairly straightforward. We use three **for** loops to check all possible values of a , b , and c . Each loop can range from 0 to the floor of \sqrt{n} (the floor of a number is the largest integer that is less than or equal to the number). Note that these bounds are sufficient to guarantee that if n can be written as the sum of the squares of three integers, then this procedure will find them. We observe that 3, the counterexample from Example 14, can be written as the sum of three squares.

> *Find3Squares*(3)

[1, 1, 1]

(1.122)

To find a counterexample, we write a procedure that, starting with 1, tests numbers with **Find3Squares**, until it finds a value that causes **Find3Squares** to return false.

```

1 Find3Counter := proc()
2   local n;
3   n := 1;
4   while Find3Squares(n) <> false do
5     n := n + 1;
6   end do;
7   return n;
8 end proc:
```

First, observe that this procedure does not take an argument; however, the parentheses are still required. Note that the **while** loop is controlled by the return value of the **Find3Squares** procedure. This is a fairly common approach when you are looking for an input value that will cause another procedure to return a desired result.

To find the counterexample, all we need to do is run the procedure.

> *Find3Counter*()

7

(1.123)

This indicates that 7 is an integer that is not the sum of the squares of three integers.

Let us take a step back and review what we did. Our goal was to disprove the statement $\forall n P(n)$ where $P(n)$ is the statement that “ n can be written as the sum of the squares of three integers.” We first wrote **Find3Squares**, which is a procedure whose goal is to find three integers whose squares add to n . Observe that if **Find3Squares** returns three values for a given n , then we know $P(n)$ is true for that n . Only after we wrote the **Find3Squares** procedure did we write **Find3Counter**, whose task was to find a counterexample to the universal statement. This is a common strategy when using a computer to find a counterexample—write a program that seeks to verify the $P(n)$ statement for n and then look to find a value of n that causes the program to fail.

Proof

We have not yet actually disproved the statement that “every positive integer is the sum of the squares of three integers.” The procedures we wrote found a candidate for a counterexample, but we do not yet know for sure that it is in fact a counterexample (after all, our program could be flawed). To prove the statement is false, we must prove that 7 is in fact a counterexample. We can approach this in one of two ways. The first approach is to follow the Solution to Example 17 in Section 1.8 of the text.

The alternative is to prove the correctness of our algorithm. Specifically, we need to prove the statement: “The positive integer n can be written as the sum of the squares of three integers if and only if **Find3Squares(n)** returns a list of three integers.” Let us prove this biconditional.

First, we prove that if the positive integer n can be written as the sum of the squares of three integers, then **Find3Squares(n)** returns a list of three integers. We use a direct proof. We assume that n can be written as the sum of three squares. Say $n = a^2 + b^2 + c^2$ for integers a, b, c . Note that we may take a, b , and c to be nonnegative integers, since an integer and its negative have the same square. In addition, $n = a^2 + b^2 + c^2 \geq a^2$. So $n \geq a^2$ and $a \geq 0$, which means that $a \leq \sqrt{n}$. Since a is an integer and is less than or equal to the square root of n , a must be less than or equal to the floor of \sqrt{n} since the floor of a real number is the greatest integer less than or equal to the real number. The same argument applies to b and c . We started with $n = a^2 + b^2 + c^2$ and have now shown that a, b , and c can be assumed to be nonnegative integers and must be less than or equal to the floor of the square root of n . The nested for loops in **Find3Squares** set **a**, **b**, and **c** equal to every possible combination of integers between 0 and **max**, which is the floor of the square root of **n**. Hence, **a**, **b**, and **c** must, at some point during the execution of **Find3Squares**, be set to a, b , and c , and thus the condition that **n=a^2+b^2+c^2** will be satisfied and **[a,b,c]** will be returned by the procedure. We have assumed that n can be written as the sum of three squares and concluded that **Find3Squares(n)** must return a list of the integers.

The converse is: if **Find3Squares** returns a list of three integers, then n can be written as the sum of the squares of three integers. This is nearly obvious, since if **Find3Squares(n)** returns **[a,b,c]**, it must have been because **n=a^2+b^2+c^2** was found to be true.

Therefore, the **Find3Squares** procedure is correct and since **Find3Squares(7)** returns false, we can conclude that 7 is, in fact, a counterexample to the assertion that every positive integer is the sum of the squares of three integers.

We will typically not be proving the correctness of procedures in this manual—that is a topic for another course. The above merely serves to illustrate how you can approach such a proof and to reinforce the principle that just because a program produces output does not guarantee that the program or the output is correct.

1.8 Proof Methods and Strategy

In this section, we will consider two additional proof techniques that Maple can assist with: exhaustive proofs and existence proofs.

Exhaustive Proof

In an exhaustive proof, we must check all possibilities. For an exhaustive proof to be feasible by hand, there must be a fairly small number of possibilities. With computer software such as Maple,

though, the number of possibilities can be greatly expanded. Consider Example 2 from Section 1.8 of the text. There it was determined by hand that the only consecutive positive integers not exceeding 100 that are perfect powers are 8 and 9.

We will consider a variation of this problem: prove that the only consecutive positive integers not exceeding 100 000 000 that are perfect powers are 8 and 9.

Our approach will be the same as was used in the text. We will generate all the perfect powers not exceeding the maximum value and then we will check to see which of the perfect powers occur as a consecutive pair. We will implement this strategy with two procedures. The first procedure, **FindPowers**, will accept as an argument the maximum value to consider (e.g., 100) and will return all of the perfect powers no greater than that maximum. The second procedure, **FindConsecutive-Powers**, will also accept the maximum value as its input. It will use **FindPowers** to generate the powers and then check them for consecutive pairs.

For the first procedure, **FindPowers**, we need to generate all perfect powers up to the given limit. To do this we use a nested pair of loops for the exponent (**p**) and the base (**b**). Both of the loops will be while loops controlled by a Boolean variable, **continuep** and **continueb**. In the inner loop, we check to see if b^p is greater than the limit. If it is, then we set **continueb** to false, which terminates the loop, and if not, we add b^p to the list of perfect powers and increment **b**. Once the **b** loop has terminated, we increment the power **p**. If 2^p exceeds the limit, then we know that no more exponents need to be checked and we terminate the outer loop by setting **continuep** to false.

```

1 FindPowers := proc (n : : posint)
2   local L, b, p, continuep, continueb;
3   L := [];
4   p := 2;
5   continuep := true;
6   while continuep do
7     b := 1;
8     continueb := true;
9     while continueb do
10       if b^p > n then
11         continueb := false;
12       else
13         L := [op(L), b^p];
14         b := b + 1;
15       end if;
16     end do;
17     p := p + 1;
18     if 2^p > n then
19       continuep := false;
20     end if;
21   end do;
22   return {op(L)};
23 end proc;
```

(Note that we return the set formed from the elements of the list of perfect powers so as to remove duplicates.) We can confirm that the list of powers produced by this algorithm is the same as the powers considered in Example 2 from the text.

> *FindPowers*(100)
{1, 4, 8, 9, 16, 25, 27, 32, 36, 49, 64, 81, 100} (1.124)

The second procedure, **FindConsecutivePowers**, begins by calling **FindPowers** and storing the set of perfect powers as **powers**. Then, we begin a for loop, using the **for x in S** structure with **x** a variable and **S** a set or list. This sets the variable **x** equal to each element of the set or list **S** in turn. In our procedure, we are considering each perfect power **x** in the set **powers**. In the body of the loop, we check to see if the next consecutive integer, **x+1**, is also a perfect power by using the **in** operator. The syntax **element in obj** tests to see if the **element** is a member of the set or list **obj**. When we find consecutive perfect powers, we **print** them. At the end of the procedure, the value **NULL** is returned, which means that the procedure will not display anything other than what was printed.

```

1 FindConsecutivePowers := proc (n)
2   local powers, x;
3   powers := FindPowers (n);
4   for x in powers do
5     if x + 1 in powers then
6       print(x, x+1);
7     end if;
8   end do;
9   return NULL;
10 end proc:
```

Subject to the correctness of our procedures, we can demonstrate that the only consecutive perfect powers less than 100 000 000 are 8 and 9 by running the procedure.

> *FindConsecutivePowers*(100 000 000)
8, 9 (1.125)

It is worth pointing out that in fact, 8 and 9 are the only consecutive perfect powers. That assertion was conjectured by Eugéne Charles Catalan in 1844 and was finally proven in 2002 by Preda Mihăilescu.

Existence Proofs

Proofs of existence can also benefit from Maple. Consider Example 10 in Section 1.8 of the text. This example asks, “Show that there is a positive integer that can be written as the sum of cubes of positive integers in two different ways.” The solution reports that 1729 is such an integer and indicates that a computer search was used to find that value. Let us see how this can be done.

The basic idea will be to generate numbers that can be written as the sum of cubes. If we generate a number twice, that will tell us that the number can be written as the sum of cubes in two different ways. We create a list **L** and, every time we generate a new sum of two cubes, we check to see if that

number is already in **L** using the **in** operator. If the new value is already in **L**, then that is the number we are looking for. Otherwise, we add the new number to **L** and generate a new sum of two cubes.

We will generate the sums of cubes with two nested loops that control integers **a** and **b**. The inner loop will be a for loop that causes **b** to range from 1 to the value of **a**. Using **a** as the maximum value means that **b** will always be less than or equal to **a** and so the procedure will not falsely report results coming from commutativity of addition (e.g., $9 = 2^3 + 1^3 = 1^3 + 2^3$). The outer loop will be a **while true do** loop. The value of **a** will be initialized to 1 and incremented by 1 after the inner **b** loop completes. The **while true do** loop is called an infinite loop because it will never stop on its own. When the procedure finds an integer which can be written as the sum of cubes in two different ways, the procedure will return that value which ends the procedure. The infinite loop means that the value of **a** will continue getting larger and larger with no upper bound. This is useful because we do not know how large the numbers will need to be in order to find the example. However, infinite loops should be used with caution, especially if you are not certain that the procedure will terminate in a reasonable amount of time.

Here is the procedure and its result.

```

1 TwoCubes := proc ( )
2   local L, a, b, n;
3   L := [];
4   a := 1;
5   while true do
6     for b from 1 to a do
7       n := a^3 + b^3;
8       if n in L then
9         return n;
10      else
11        L := [op(L), n];
12      end if;
13    end do;
14    a := a + 1;
15  end do;
16 end proc;
```

> *TwoCubes()*

1729

(1.126)

Solutions to Computer Projects and Computations and Explorations

Computer Projects 3

Given a compound proposition, determine whether it is satisfiable by checking its truth value for all positive assignments of truth values to its propositional variables.

Solution: While the **Logic** package command **Satisfiable** answers the underlying question, the question implies that we should write our own procedure.

Recall that a proposition is satisfiable if there is at least one assignment of truth values to variables that results in a true proposition. Our approach will be similar to the way we checked for logical equivalence in the **AreEquivalent** procedure in Section 1.3.

We create a procedure, **IsSatisfiable**, that checks all possible assignments of truth values to the propositional variables. The **IsSatisfiable** procedure accepts one argument, a logical expression. It will print out all, if any, truth value assignments that satisfy the proposition. We will initialize a **result** variable to false. When an assignment that satisfies the proposition is found, this variable is set to true and the assignment is printed. After all possible assignments are considered, the procedure returns the **result** variable.

Since this procedure is otherwise very similar to **AreEquivalent**, we offer no further explanation.

```

1  IsSatisfiable := proc (P)
2      local eqZip, Vars, numVars, i, TA, val, TAEqns, result;
3      result := false;
4      eqZip := (a, b) -> a=b;
5      Vars := GetVars (P);
6      numVars := nops (Vars);
7      TA := [seq (false, i=1..numVars)];
8      while TA <> NULL do
9          TAEqns := zip (eqZip, Vars, TA);
10         val := eval (P, TAEqns);
11         if val then
12             result := true;
13             print (TAEqns);
14         end if;
15         TA := NextTA (TA);
16     end do;
17     return result;
18 end proc;
```

We apply this procedure to the propositions in Example 9 of Section 1.3 of the text.

> *IsSatisfiable ((p or not q) and (q or not r) and (r or not p))*
 $[p = \text{false}, q = \text{false}, r = \text{false}]$
 $[p = \text{true}, q = \text{true}, r = \text{true}]$
 true (1.127)

> *IsSatisfiable ((p or q or r) and (not p or not q or not r))*
 $[p = \text{true}, q = \text{false}, r = \text{false}]$
 $[p = \text{false}, q = \text{true}, r = \text{false}]$
 $[p = \text{true}, q = \text{true}, r = \text{false}]$
 $[p = \text{false}, q = \text{false}, r = \text{true}]$
 $[p = \text{true}, q = \text{false}, r = \text{true}]$
 $[p = \text{false}, q = \text{true}, r = \text{true}]$
 true (1.128)

> *IsSatisfiable ((p or not q) and (q or not r) and (r or not p) and
(p or q or r) and (not p or not q or not r))*
 false (1.129)

Computations and Explorations 1

Look for positive integers that are not the sum of the cubes of eight positive integers.

Solution: We will find integers n such that $n \neq a_1^3 + a_2^3 + \cdots + a_8^3$ for any integers a_1, a_2, \dots, a_8 . We can restate the problem as finding a counterexample to the assertion that every integer can be written as the sum of eight cubes.

Our approach will be to generate all of the integers that are equal to the sum of eight cubes and then check to see what integers are missing. For this, we need to set a limit n , that is, the maximum integer that we are considering as a possible answer to the question. For instance, we might restrict our search to integers less than 100. Then, we know that each a_i is at most the cube root of this limit, since $a_i^3 \leq n$.

We also want to make our approach as efficient as possible in order to find as many such integers as we can. We make the following observations.

Every number that can be expressed as the sum of eight cubes can be expressed as the sum of two integers each of which is the sum of four cubes. Those, in turn, can be expressed as the sum of two integers which are the sum of two cubes each. That is,

$$n = [(a_1^3 + a_2^3) + (a_3^3 + a_4^3)] + [(a_5^3 + a_6^3) + (a_7^3 + a_8^3)]$$

This means that we do not need to write a procedure to find all possible sums of eight cubes. Instead, we will write a procedure that, given a list of numbers, will find all possible sums of two numbers that are both in that list. If we apply this procedure to the cubes of the numbers from 0 through $\sqrt[3]{n}$, that will produce all numbers that are the sums of two cubes. Applying the procedure again to that result will give all numbers that are the sum of four cubes. Applying it once again to that result will produce the numbers (up to n) that are the sum of eight cubes.

Additionally, when we find all the possible sums of two integers, we will exclude any sum that exceeds our maximum. Recall that we have determined that if an integer less than or equal to n can be written as the sum of cubes, then it can be written as the sum of cubes with each a_i between 0 and $\sqrt[3]{n}$. There will be numbers greater than n that are generated as the sum of cubes of integers less than $\sqrt[3]{n}$; however, these do not provide us with any information about numbers that cannot be generated as the sum of eight cubes. Excluding them at each step of the process decreases the number of sums that need to be computed.

Finally, we may assume that the second number is at least as large as the first. Since we add $2^3 + 5^3$ to our list of sums, there is no need to also include $5^3 + 2^3$.

Here is the procedure that finds all possible sums of pairs of integers from the given list **L** up to the specified maximum value **max**. Note that we again use the **[op({op(sums)})]** structure to turn the list into a set and then back into a list. This removes redundancies and also puts the list in increasing order.

```
1 AllPairSums := proc(L, max)
2     local a, b, s, sums, num;
3     sums := [];
```

```

4 num := nops (L) ;
5 a := 1 ;
6 while a <= num do
7     b := a ;
8     while b <= num do
9         s := L [a] + L [b] ;
10        if s <= max then
11            sums := [op (sums), s] ;
12        else
13            b := num ;
14        end if ;
15        b := b + 1 ;
16    end do ;
17    a := a + 1 ;
18 end do ;
19 return [op ({op (sums)})] ;
20 end proc;

```

With this procedure in place, we need to apply it (three times) to a list of cubes. We consider cubes up to 7^3 , and including 0. Note that i^3 is entered by typing **i³**.

```

> somecubes := [seq (i3, i = 0 .. 7)]
somecubes := [0, 1, 8, 27, 64, 125, 216, 343] (1.130)

```

Applying the **AllPairSums** procedure once gives us all sums of cubes.

```

> TwoCubes := AllPairSums (somecubes, 343)
TwoCubes := [0, 1, 2, 8, 9, 16, 27, 28, 35, 54, 64, 65, 72, 91, 125, 126, 128,
133, 152, 189, 216, 217, 224, 243, 250, 280, 341, 343] (1.131)

```

Applying it to that result gives all possible sums of four cubes (up to 343).

```

> FourCubes := AllPairSums (TwoCubes, 343)
FourCubes := [0, 1, 2, 3, 4, 8, 9, 10, 11, 16, 17, 18, 24, 25, 27, 28, 29,
30, 32, 35, 36, 37, 43, 44, 51, 54, 55, 56, 62, 63, 64, 65, 66, 67, 70,
72, 73, 74, 80, 81, 82, 88, 89, 91, 92, 93, 99, 100, 107, 108, 118,
119, 125, 126, 127, 128, 129, 130, 133, 134, 135, 136, 137, 141,
142, 144, 145, 149, 152, 153, 154, 155, 156, 160, 161, 163, 168,
179, 180, 182, 187, 189, 190, 191, 192, 193, 197, 198, 200, 205,
206, 216, 217, 218, 219, 224, 225, 226, 232, 233, 240, 243, 244,
245, 250, 251, 252, 253, 254, 256, 258, 259, 261, 266, 270, 271,
277, 278, 280, 281, 282, 285, 288, 289, 296, 297, 304, 307, 308,
314, 315, 317, 322, 334, 341, 342, 343] (1.132)

```

Once again, we obtain all integers up to 343 which can be obtained as the sum of eight cubes.

```

> EightCubes := AllPairSums (FourCubes, 343)

```

```

EightCubes := [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
 17, 18, 19, 20, 21, 22, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53,
 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71,
 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105,
 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119,
 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133,
 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147,
 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160,
 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173,
 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186,
 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199,
 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212,
 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225,
 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238,
 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252,
 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265,
 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278,
 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291,
 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304,
 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317,
 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330,
 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343]

```

(1.133)

Finally, we print out the integers that are missing from the list.

```

> for i to 343 do
  if not i in EightCubes then
    print(i);
  end if;
end do;
23
239

```

(1.134)

Exercises

Exercise 1. Write procedures **NOT**, **OR**, and **XOR** to implement those bit string operators.

Exercise 2. Use Maple to solve Exercises 23 through 27 in Section 1.2 using the knights and knaves puzzle that was solved earlier in this chapter as a guide.

Exercise 3. Implement the solution of Sudoku puzzles as a satisfiability problem described in the main text.

Exercise 4. Write a Maple procedure to find the dual of a proposition. Dual is defined in the Exercises of Section 1.3. (Hint: you may find it useful to know that ‘**and**’ and ‘**or**’ are considered types in Maple and, thus, can be used as the second argument to the **type** command.)

Exercise 5. Write a procedure **UniqueExists**, similar to the **Universal** and **Existential** procedures in Section 1.4 of this manual. This procedure should accept as its arguments a propositional function and a finite domain and return true if there is a unique element of the domain that satisfies the proposition and false otherwise.

Exercise 6. Write a procedure **ForAllExists** that is analogous to the **ExistsForAll** procedure given in Section 1.5 of this manual.

Exercise 7. Write a Maple procedure that plays the obligato game in the role of the student, as described in the Supplementary Exercises of Chapter 1. Specifically, the procedure should accept two arguments. The first argument is the new statement that you, as the teacher, provide. The second argument should be the list of Maple's responses to all the previous statements. For example, suppose the teacher's first statement is $p \rightarrow q \vee r$, the second statement is $\neg p \vee q$, and the third statement is r . If the procedure/student accepts the first statement and denies the second statement, then you would obtain the response to the third statement by executing

Obligato(r, [p implies (q or r), not(not p or q)]);

The procedure must accept the statement r and return the list with this response included:

[p implies (q or r), not(not p or q), r]

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

Introduction

Chapter 2 of the textbook covers mathematical objects that are essential 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 represent 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 essential in 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} (2.1)
```

```
> {"a","b","c"}
 {"a","b","c"} (2.2)
```

```
> {{1,2},{1,3},{2,3}}
 {{1,2},{1,3},{2,3}} (2.3)
```

```
> {}
∅ (2.4)
```

```
> {[1,2],[2,5],[3,11]}
 {[1,2],[2,5],[3,11]} (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, which we can interpret as a set containing three ordered pairs.

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}  
set1 := {1,2,3} (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} (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 (2.8)
```

Selection

A useful consequence of the fact that Maple stores sets in a particular 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 := {"a","b","c","d","e","f"} (2.9)
```

```
> set3[2]  
"b" (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]  
"e" (2.11)
```

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

```
> set3[3..5]  
{"c","d","e"} (2.12)
```

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

```
> set3[[1,3,5]]
{“a”, “c”, “e”} (2.13)
```

The seq Command

One of the most useful commands for constructing sets or lists is the **seq** (for sequence) command. For example, we can use **seq** to build the set consisting of the squares of some 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 (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 the sequence in braces; we also create the set of fourth powers of integers between -10 and 10.

```
> set4 := {seq1}
set4 := {0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100} (2.15)
```

```
> set5 := {seq (i^4, i = -10 .. 10)}
set5 := {0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10 000} (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.

```
> evalb(4 in set4)
true (2.17)
```

```
> evalb(5 in set4)
false (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 ∈ {0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100} (2.19)
```

As an alternative to the **in** operator, we can use the **member** command. This command does not require the application of **evalb** to return a Boolean result, but is called as a function applied to

arguments rather than the infix operator form. The first argument of **member** is the object and the second is a set or list or other larger structure.

```
> member(4, set4)
      true
```

(2.20)

One advantage of **member** is that it accepts an unevaluated name as an optional third argument. Upon execution, the location of the object, or first location if it appears more than once, will be stored in the name, provided it is present. If the object is not present, no assignment is made to the name.

```
> member(36, set4, 'locationof36')
      true
```

(2.21)

```
> locationof36
      7
```

(2.22)

Note the use of single quotes around **locationof36** in the call to **member**. This prevents the name being evaluated to any value that may have been previously assigned to it and ensures that the function will execute even if the name has previously been used. The single quotes are not required, and omitting them can be useful if you want to avoid overwriting data.

The inclusion in a parameter sequence of an optional unevaluated name is common in Maple functions, and it is worth briefly illustrating one reason it is a useful feature. Suppose we are writing a procedure that needs to check whether a set (or list or other appropriate data structure) contains an object and then take action on that member. For example, we may wish to replace 81 with 18 in **set4**. We can take advantage of the fact that **member** has the side-effect of recording the location to do this.

```
> if member(81, set4, 'locationof81') then
      set4 := subsop(locationof81 = 18, set4)
    else
      print("Not found")
    end if;
      set4 := {0, 1, 4, 9, 16, 18, 25, 36, 49, 64, 100}
```

(2.23)

Recall that the first argument of **subsop** is an equation with a position on the left-hand side and a new value on the right-hand side. Also recall that it does not modify the data structure given as the second argument, so we reassign the list in order to update it.

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
```

(2.24)

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

(2.25)

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

(2.26)

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
```

(2.27)

Another way to find the size of a finite set or the length of a list is the **numelems** command. For a list or a set, **numelems** and **nops** return the same result.

```
> numelems(set5)
11
```

(2.28)

Also note that both of them return the number of top-level elements. For example, on each list of lists below, they give the number of sublists, not the total number of objects. This is consistent with the mathematical definition of the size of a set.

```
> numelems([[1,2,3,4,5],[6,7,8]])
2
```

(2.29)

```
> nops([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
4
```

(2.30)

However, **numelems** is more flexible; for example, returning the number of characters in a string, which **nops** would treat as a single entity.

```
> numelems("helloworld")
11
```

(2.31)

```
> nops("helloworld")
1
```

(2.32)

The **numelems** command will also delve down into data structures such as matrices to report the number of entries, whereas **nops** would return the number of rows. In this way, **numelems** generally produces the more natural output for the context, while **nops** is focused on the formal representation of objects.

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.

We saw in the first chapter that 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.

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

(2.33)

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, firstcomb,
firstpart, firstperm, graycode, inttovector, lastcomb, lastpart, lastperm,
multinomial, nextcomb, nextpart, nextperm, numbcomb, numbcomp,
numbpart, numbperm, partition, permute, powerset, prevcomb,
prevpart, prevperm, randcomb, randpart, randperm, rankcomb,
rankperm, setpartition, stirling1, stirling2, subsets,
unrankcomb, unrankperm, vectoint] (2.34)
```

The result of this command is to list the commands that have been made available. Typically, you will end a **with** statement with a colon to suppress this output. Note that you can selectively load only those commands from a package that you actually need by following the name of the package with the names of the commands separated by commas.

```
> with(combinat, powerset, subsets, cartprod)
[powerset, subsets, cartprod] (2.35)
```

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"} } (2.36)
```

Keep in mind that it is often best to use the long form names in procedures you write, so that the procedure is not dependent on the package having been loaded.

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} (2.37)
```

Note the alternate form of the **seq** command, which is equivalent to **seq(i,i=1..10)**.

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 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]()
∅
```

(2.38)

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

```
> subsetTen[nextvalue]()
{1}
```

(2.39)

```
> subsetTen[nextvalue]()
{2}
```

(2.40)

```
> subsetTen[nextvalue]()
{3}
```

(2.41)

```
> subsetTen[nextvalue]()
{4}
```

(2.42)

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

```
> subsetTen[finished]
false
```

(2.43)

Once the **subsetTen[nextvalue]** procedure has returned the “last” subset, which will always be the original set itself, this Boolean is set to true. 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, we will search for the subsets of the first five positive integers which have their own cardinality as a member, that is, those sets S such that $|S| \in S$. We do this by considering each subset in turn and checking whether its size, obtained using the **nops** command, is **in** the set.

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

```
1 selfSize := proc()
2   local onetofive, pSet, S, n;
3   onetofive := {seq(1..5)};
4   pSet := combinat[subsets](onetofive);
5   while not pSet[finished] do
```

```

6      S := pSet[nextvalue]();
7      if nops(S) in S then
8          print(S);
9      end if;
10     end do;
11 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, 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}                                         (2.44)

```

It is 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 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 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]0**.

```
> while not Ex17[finished] do  
    print(Ex17[nextvalue]0)  
end do;  
[1, "a"]  
[1, "b"]  
[1, "c"]  
[2, "a"]  
[2, "b"]  
[2, "c"]
```

(2.45)

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. Note that this command accepts either sets or 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 to illustrate the command’s flexibility. (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,e]]):  
 > while not cartesian3[finished] do  
     print(cartesian3[nextvalue]0)  
 end do;  
[1, "a", π]  
[1, "a", e]  
[1, "b", π]  
[1, "b", e]  
[1, "c", π]  
[1, "c", e]  
[2, "a", π]  
[2, "a", e]  
[2, "b", π]
```

```
[2, "b", e]
[2, "c", π]
[2, "c", e] (2.46)
```

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} (2.47)
```

```
> odds := {1, 3, 5, 7, 9, 11, 13}
odds := {1, 3, 5, 7, 9, 11, 13} (2.48)
```

We compute their union and intersection as follows.

```
> primes union odds
{1, 2, 3, 5, 7, 9, 11, 13} (2.49)
```

```
> primes intersect odds
{3, 5, 7, 11, 13} (2.50)
```

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, that is, $A - B$ is generally not the same as $B - A$.

```
> primes minus odds
{2} (2.51)
```

```
> odds minus primes
{1, 9} (2.52)
```

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 from 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} (2.53)
```

> *universe minus primes*
 $\{1, 4, 6, 8, 9, 10, 12\}$ (2.54)

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.

```

1 tenSquares := proc ()
2   local S, i;
3   S := {};
4   for i from 1 to 10 do
5     S := S union {i^2};
6   end do;
7   return S;
8 end proc;
```

> *squares := tenSquares()*
 $\{1, 4, 9, 16, 25, 36, 49, 64, 81, 100\}$ (2.55)

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, and requires much more typing than the **seq** command would have, it illustrates a common technique.

All three operators can also be used as commands, and, in the cases of **union** and **intersect**, 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*
 $\{1, 2, 3, 4, 5, 7, 9, 11, 13, 16, 25, 36, 49, 64, 81, 100\}$ (2.56)

> '*union*'(*primes, odds, squares*);
 $\{1, 2, 3, 4, 5, 7, 9, 11, 13, 16, 25, 36, 49, 64, 81, 100\}$ (2.57)

Note that the single left quotes are required in the second option, since it is in 1-D notation. 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. In 2-D notation, they are not required.

Set Identities and Membership Tables

The textbook discusses how membership tables can be used to prove set identities. We will 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 us 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 $A \cup B = \overline{A} \cap \overline{B}$.

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		

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 value in the universe of row numbers is in A if there is a 1 in the column for A in that row. Thus $A = \{1, 2\}$ because rows 1 and 2 have 1s in the column for A . Likewise, B is defined to be $\{1, 3\}$.

```
> rows := {1,2,3,4}  
rows := {1,2,3,4} (2.58)
```

```
> setA := {1,2}  
setA := {1,2} (2.59)
```

```
> setB := {1,3}  
setB := {1,3} (2.60)
```

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} (2.61)
```

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} (2.62)
```

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, we create expressions representing the two sides of the identity from Example 14 of the text. We use the name **U** for the universe.

$$\begin{aligned} > Ex14L := U \text{ minus } (A \cup (B \cap C)) \\ Ex14L := U \setminus (A \cup B \cap C) \end{aligned} \quad (2.63)$$

$$\begin{aligned} > Ex14R := ((U \text{ minus } C) \cup (U \text{ minus } B)) \cap (U \text{ minus } A) \\ Ex14R := (U \setminus A) \cap (U \setminus B \cup (U \setminus C)) \end{aligned} \quad (2.64)$$

$$\begin{aligned} > Ex14 := Ex14L = Ex14R \\ Ex14 := U \setminus (A \cup B \cap C) = (U \setminus A) \cap (U \setminus B \cup (U \setminus C)) \end{aligned} \quad (2.65)$$

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

```

1  GetVars := proc(exp)
2    local L, i, j;
3    L := [exp];
4    i := 1;
5    while i <= nops(L) do
6      if type(L[i], name) then
7        i := i + 1;
8      else
9        L := subsop(i=op(L[i]), L);
10     end if;
11   end do;
12   L := {op(L)} minus {U};
13   return [op(L)];
14 end proc;
```

$$\begin{aligned} > Ex14Vars := GetVars(Ex14) \\ Ex14Vars := [A, B, C] \end{aligned} \quad (2.66)$$

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 is not 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

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

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

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)]) colon
```

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] (2.67)
```

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

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 will 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 := [\emptyset, \emptyset, \emptyset] (2.68)
```

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}
MTableSets2 := {5} (2.69)
```

```
> MTableSets
[[], {5}, []] (2.70)
```

We reinitialize this so we can use it below.

```
> MTableSets := [seq({}, i = 1 .. 3)]
MTableSets := [[], [], []] (2.71)
```

We reexecute 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 (2.72)
```

Now, we use the standard Cartesian product while loop, but instead of just calculating and displaying the 3-tuple, we 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 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}] (2.73)
```

We also need the universe represented.

```
> Ex14U := {seq(1 .. MTrownum)}
Ex14U := {1, 2, 3, 4, 5, 6, 7, 8} (2.74)
```

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}]
```

(2.75)

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}]
```

(2.76)

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

```
> eval(Ex14, Ex14eqns)
Ex14L = {1,2,3}
```

(2.77)

```
> evalb((2.77))
false
```

(2.78)

The Procedure

Finally, we combine it all into a single procedure.

```
1 MemberTable := proc(identity)
2   local vars, numvars, cartesian, setList, rowNum, curTuple, vI,
3     Universe, eqns, U;
4   vars := GetVars(identity);
5   numvars := nops(vars);
6   cartesian := combinat[cartprod]([seq({0,1}, i=1..numvars)]);
7   setList := [seq({}, i=1..numvars)];
8   rowNum := 0;
9   while not cartesian[finished] do
10    rowNum := rowNum + 1;
11    curTuple := cartesian[nextvalue]();
12    for vI from 1 to numvars do
13      if curTuple[vI] = 1 then
14        setList[vI] := setList[vI] union {rowNum};
15      end if;
16    end do;
17    Universe := {seq(i, i=1..rowNum)};
18    eqns := zip((a,b) → a=b, vars, setList);
19    eqns := [op(eqns), U=Universe];
20    return evalb(eval(identity, eqns));
21 end proc;
```

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

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

(2.79)

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
```

(2.80)

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 73 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 will represent fuzzy sets in three 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]}
```

(2.81)

We 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. Suppose 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 := [“a”, “b”, “c”, “d”, “e”, “f”, “g”]
```

(2.82)

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 nonmembership.

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

(2.83)

The third approach is to make use of Maple’s built-in support for multisets. Recall from the textbook that in a multiset, objects can appear more than once. In a Maple **MultiSet**, objects are allowed to have nonnegative integers as their multiplicities. To represent fuzzy sets, we need to use the generalized version of the **MultiSet**, which allows the multiplicities to be any real numbers. The reason for the distinction is that for nongeneralized multisets, it is possible to iterate over the members of the set, which is not possible when the degrees of membership are not nonnegative integers.

A generalized **MultiSet** is created by applying the command to a specification of the elements of the set. One of the ways to specify the membership is the same as the roster representation above.

```
> fuzzyM := MultiSet[generalized]([“a”, 0.3], [“b”, 0.7], [“e”, 0.1])
fuzzyM := {[“a”, 0.3], [“b”, 0.7], [“e”, 0.1]} (2.84)
```

If you wish, instead of lists containing the object and its degree of membership, you can specify the elements by equations, as illustrated below.

```
> MultiSet[generalized](“a” = 0.3, “b” = 0.7, “e” = 0.1)
{[“a”, 0.3], [“b”, 0.7], [“e”, 0.1]} (2.85)
```

Note that the display of a **MultiSet** is identical to the roster representation, but they are in fact different objects. Our roster representation is a set, while this representation is not.

```
> type(fuzzyM, set)
false (2.86)
```

The usual set operations, **union**, **intersect**, and **minus**, can be applied to **MultiSet** objects. When applied to representations of fuzzy sets, **union** and **intersect** produce the expected result. However, because a generalized **MultiSet** is allowed to have negative values for its degree of membership, is defined to simply subtract the respective degrees of membership and can result in sets that are not fuzzy sets.

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 nonzero, then we add to the roster the pair consisting of the element from the universe and the degree of membership.

```
1 BitToRoster := proc(bitstring, universe)
2   local S, i;
3   S := {};
4   for i from 1 to nops(universe) do
5     if bitstring[i] <> 0 then
6       S := S union {[universe[i], bitstring[i]]};
7     end if;
8   end do;
9   return S;
10 end proc;
```

```
> BitToRoster(fuzzyBitS,fuzzyU)
{[“a”, 0.3], [“b”, 0.7], [“e”, 0.1]} (2.87)
```

Converting from Roster to Bit String Representation

In the other direction, we 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 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
```

(2.88)

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

(2.89)

As we mentioned above, **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
```

(2.90)

```
> pos
5
```

(2.91)

Now, we can write the **RosterToBit** procedure.

```

1 RosterToBit := proc(roster,universe)
2   local B, i, e, pos;
3   B := [seq(0,i=1..nops(universe))];
4   for e in roster do
5     if member(e[1],universe,'pos') then
6       B[pos] := e[2];
7     else
8       error "Roster contained a member not in the universe.";
9     end if;
10    end do;
11    return B;
12  end proc;
```

```
> RosterToBit(fuzzyR,fuzzyU)
[0.3, 0.7, 0, 0, 0.1, 0, 0]
```

(2.92)

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.

Converting between MultiSet and Roster Representations

Recall that even though a generalized **MultiSet** is displayed as if it were the same as our roster representation, it is actually not a set. However, the display suggests, and it is true, that converting a **MultiSet** into a list of pairs of objects and their multiplicities is easy. Simply apply the function **Entries**.

> *Entries(fuzzyM)*
 $\{["a", 0.3], ["b", 0.7], ["e", 0.1]\}$ (2.93)

In the other direction, we simply apply **MultiSet** to the roster representation.

> *MultiSet[generalized](fuzzyR)*
 $\{["a", 0.3], ["b", 0.7], ["e", 0.1]\}$ (2.94)

While the usual syntax for the membership specification of a **MultiSet** is a sequence of such pairs, not a set of them, the command will automatically convert a set or list of object-degree pairs. Also note that **MultiSet** will convert a set of objects into a multiset in which each object has degree of membership equal to 1, and will convert a list into a multiset in which each object's degree of membership is its multiplicity in the list.

2.3 Functions

In this section, 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, **GetVars** 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.

```

1 Floor1 := proc (x : float) :: integer;
2     return floor(x);
3 end proc;
```

The body of our **Floor1** procedure is merely a call to Maple's internal **floor** command. The focus of the example is to illustrate 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, which prints and then decreases its argument by 1 until it reaches 0.

```
1 loopy := proc(n::posint)
2   local m;
3   m := n;
4   while m <> 0 do
5     print(m);
6     m := m - 1;
7   end do;
8 end proc;
```

Applying this procedure with a positive integer as its argument has the desired result.

```
> loopy(3)
3
2
1
0
```

(2.95)

If you call the function with a value that is not of type **posint**, it will raise an error.

```
> 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. In addition, Maple provides a method for easily creating new types via the structured type syntax. For example, our **Floor1** procedure has a slight problem, as the following illustrates.

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

5

(2.96)

> *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.

```
1 Floor2 := proc (x::{float,integer})  
2     return floor(x);  
3 end proc;
```

> *Floor2*(5.), *Floor2*(5)

5, 5 (2.97)

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. We 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.

```
1 Floor3 := proc (L::list)  
2     return map(floor,L);  
3 end proc;
```

> *Floor3*([12.5, 13.9, -2.5])

[12, 13, -3] (2.98)

Note that we have 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.

```
1 Floor4 := proc (L::list({float,integer}))  
2     return map(floor,L);  
3 end proc;
```

> *Floor4*([47.298, 3, 13.7])

[47, 3, 13] (2.99)

> *Floor4*([5.6, 3, $\frac{22}{7}$, 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 second and third applications of **Floor4** 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. We can make our procedure accept either single values or lists as follows.

```
1 Floor := proc(v::{numeric,list(numeric)}) 
2   if type(v, numeric) then
3     return floor(v);
4   else
5     return map(floor, v);
6   end if;
7 end proc;
```

> *Floor* $\left([5.6, 3, \frac{22}{7}, 6.8]\right)$
[5, 3, 3, 6] (2.100)

> *Floor*(3.2)
3 (2.101)

Functional Operators

Next, we will 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  $\mapsto x^2$  (2.102)
```

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. If you wish the function to accept 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. (Note that we presented this first example in 1-D input mode; when using 2-D mode, the hyphen–greater-than pair is converted into an arrow as you type.)

You apply the functional operator just as you would expect.

```
> f(3)
9 (2.103)
```

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.

$$\begin{aligned}> g := x \rightarrow x + 1 \\ g := x \mapsto x + 1\end{aligned}\tag{2.104}$$

$$\begin{aligned}> (f + g)(x) \\ x^2 + x + 1\end{aligned}\tag{2.105}$$

$$\begin{aligned}> \left(\frac{f}{g}\right)(t) \\ \frac{t^2}{t + 1}\end{aligned}\tag{2.106}$$

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**.

$$\begin{aligned}> f/g(t); \\ \frac{f}{t + 1}\end{aligned}\tag{2.107}$$

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

$$\begin{aligned}> (g @ f)(x) \\ x^2 + 1\end{aligned}\tag{2.108}$$

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.

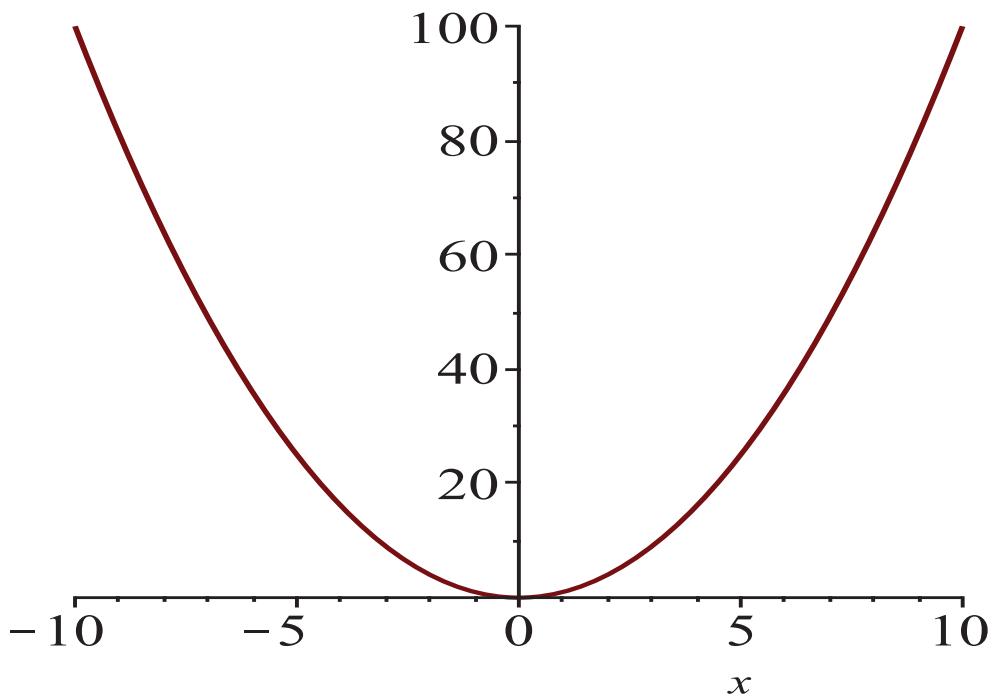
$$\begin{aligned}> squareFloor := f @ floor \\ squareFloor := f @ floor\end{aligned}\tag{2.109}$$

$$\begin{aligned}> squareFloor(3.2) \\ 9\end{aligned}\tag{2.110}$$

Plotting Graphs of Functions

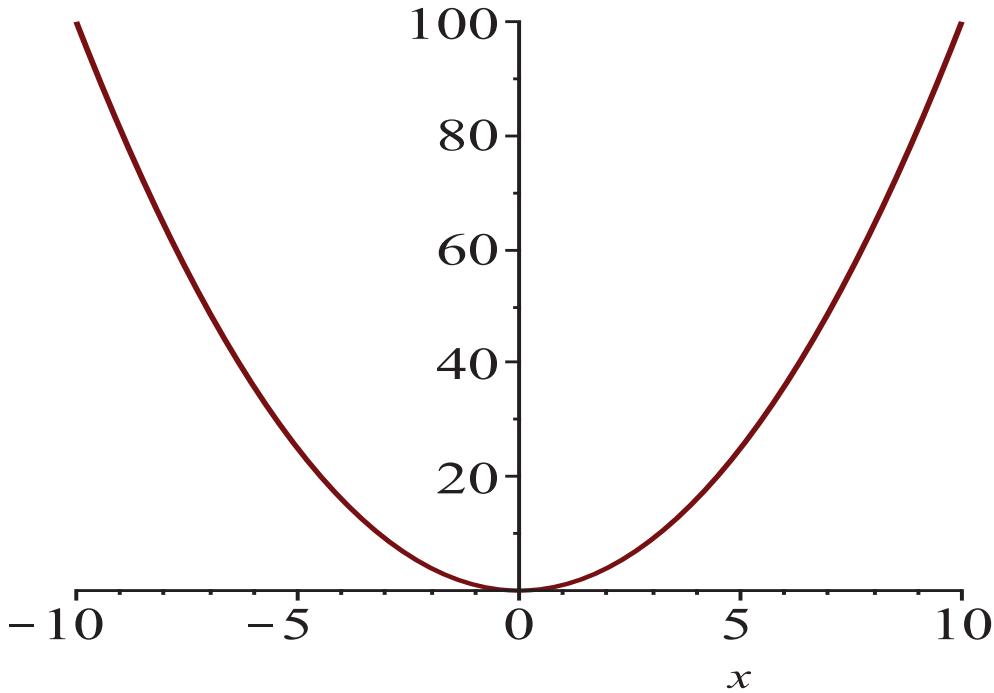
You can use Maple to draw the graph of a function by using the **plot** command. The most basic syntax requires only two arguments: the function to be graphed in terms of an independent variable and the variable. The first argument can be given as an expression as follows.

$$> plot(x^2, x)$$

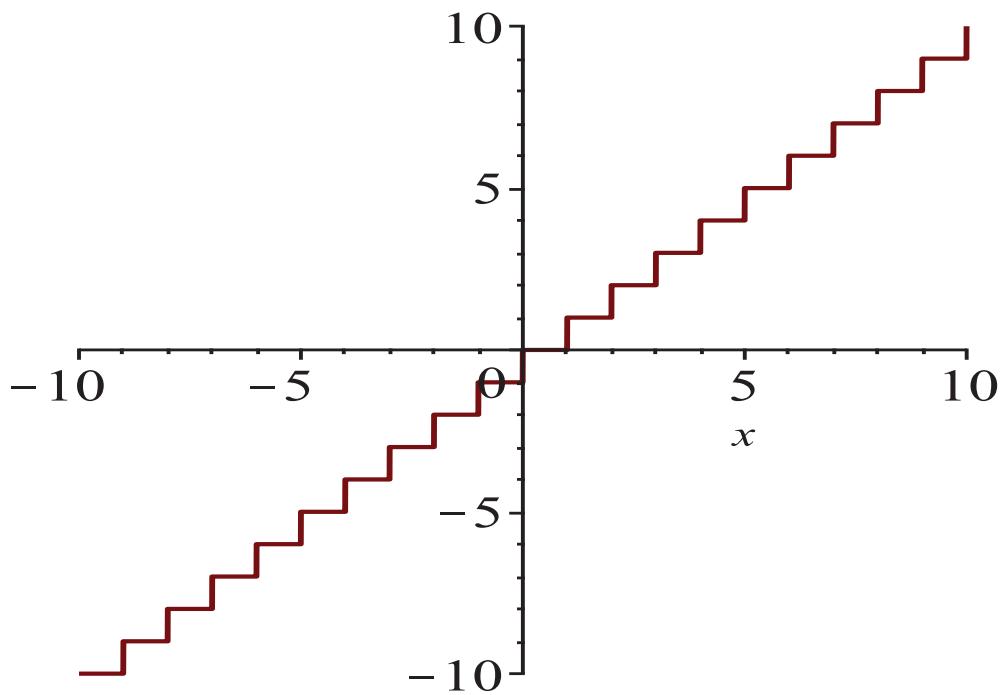


The first argument can also be given as a functional operator or even a procedure as the following two examples illustrate. The essential requirement is that the first argument must evaluate to a numeric value whenever the independent variable is assigned a value. That is to say, the function must have appropriate domain and range.

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

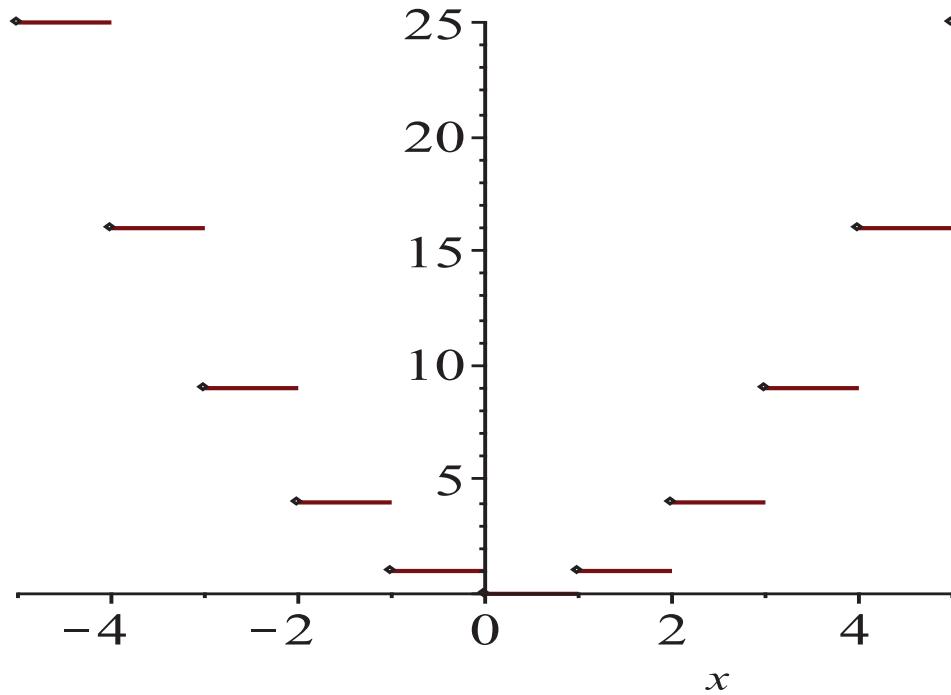


```
> plot(floor(x),x)
```



In the plot of **floor**, 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 indicate 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 $f(Ann) = 83$, $f(Bob) = 79$, $f(Carlos) = 91$, and $f(Delia) = 72$. 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, “Carlos” = 91, “Delia” = 72])  
exams := table([“Ann” = 83, “Bob” = 79, “Carlos” = 91, “Delia” = 72]) (2.111)
```

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

```
> exams[“Carlos”]  
91 (2.112)
```

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

```
> exams[“Ann”] := 84  
exams“Ann” := 84 (2.113)
```

```
> exams[“Ernie”] := 86  
exams“Ernie” := 86 (2.114)
```

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

```
> exams  
exams (2.115)
```

```
> op(exams)  
table([“Ann” = 84, “Ernie” = 86, “Bob” = 79, “Carlos” = 91, “Delia” = 72]) (2.116)
```

```
> op(op(exams))  
[“Ann” = 84, “Ernie” = 86, “Bob” = 79, “Carlos” = 91, “Delia” = 72] (2.117)
```

Domain and Range

Since tables are finite, we can write procedures to check various properties. First, we find the domain (that is, 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 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)  
[“Ann”, “Ernie”, “Bob”, “Carlos”, “Delia”] (2.118)
```

```
> entries(exams)  
[84], [86], [79], [91], [72] (2.119)
```

The **indices** command returns a sequence of lists where each list contains an index of the table. The reason for this is to allow 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([(1,1)=1,(1,2)=0,(2,1)=0,(2,2)=1])
```

(2.120)

```
> indices(M)
[1,1], [1,2], [2,1], [2,2]
```

(2.121)

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

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

(2.122)

(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. 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)
“Ann”, “Ernie”, “Bob”, “Carlos”, “Delia”
```

(2.123)

```
> entries(exams, nolist)
84, 86, 79, 91, 72
```

(2.124)

Note that while Maple chooses the order 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 listed first is the entry corresponding to the index 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.

1	FindDomain := proc (T : :table)
2	return {indices(T, 'nolist')};
3	end proc;
4	FindRange := proc (T : :table)

```

5   return {entries(T,'nolist')};
6 end proc;

```

> *FindDomain(exams)*
{“Ann”, “Bob”, “Carlos”, “Delia”, “Ernie”} (2.125)

> *FindRange(exams)*
{72, 79, 84, 86, 91} (2.126)

Injective and Surjective

We will create a few more examples and then 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 $D = \{-5, -4, 5, \dots\}$.

Thus far, we have defined tables by manually creating all of the index–entry pairs. Here, we will use **seq** to form the equations that define the table. We simply provide the equation as the first argument.

> *f1 := table([seq(x = x², x = -5 .. 5)])*
f1 := table([-1 = 1, 0 = 0, -2 = 4, 1 = 1, -3 = 9, 2 = 4,
-4 = 16, 3 = 9, 4 = 16, -5 = 25, 5 = 25]) (2.127)

> *f2 := table([seq(x = x³, x = -5 .. 5)])*
f2 := table([-1 = -1, 0 = 0, -2 = -8, 1 = 1, -3 = -27,
2 = 8, -4 = -64, 3 = 27, 4 = 64, -5 = -125, 5 = 125]) (2.128)

> *f3 := table([seq(x = |x|, x = -5 .. 5)])*
f3 := table([-1 = 1, 0 = 0, -2 = 2, 1 = 1, -3 = 3, 2 = 2,
-4 = 4, 3 = 3, 4 = 4, -5 = 5, 5 = 5]) (2.129)

The domain and range functions defined earlier produce the expected results.

> *FindDomain(f1)*
{-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5} (2.130)

> *FindRange(f1)*
{0, 1, 4, 9, 16, 25} (2.131)

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

```

1 IsOnto := proc(T::table, codomain::set)
2   return evalb(FindRange(T) = codomain);
3 end proc;

```

> *IsOnto(f1, {0, 1, 2, 3, 4, 5})*
false (2.132)

> *IsOnto(f3, {0, 1, 2, 3, 4, 5})*
true (2.133)

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**.

```

1 IsOneToOne := proc (T : :table)
2   return evalb(nops(FindDomain(T)) = nops(FindRange(T)));
3 end proc;
```

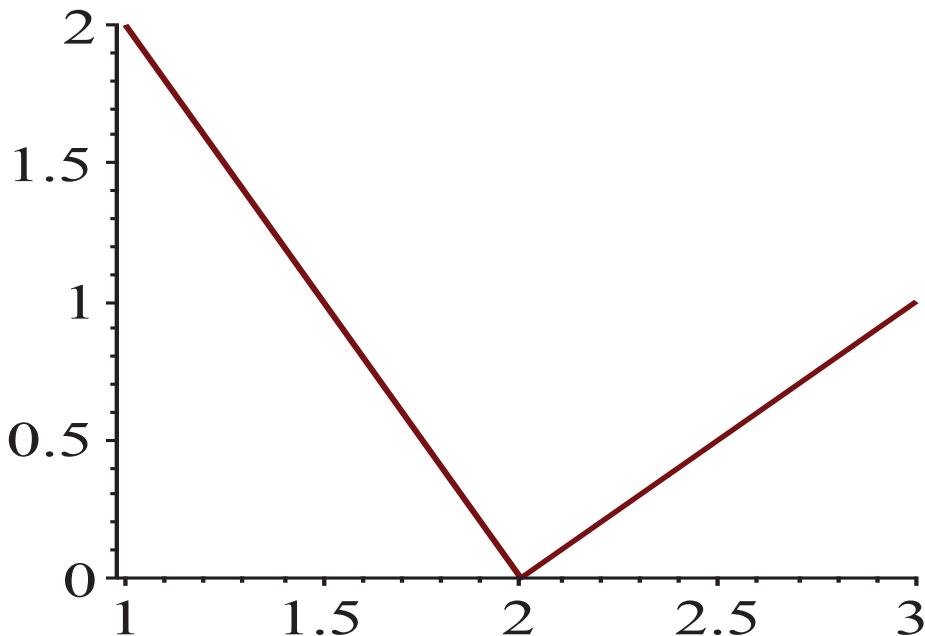
> *IsOneToOne(f1)*
false (2.134)

> *IsOneToOne(f2)*
true (2.135)

Graphing a Function from a Table

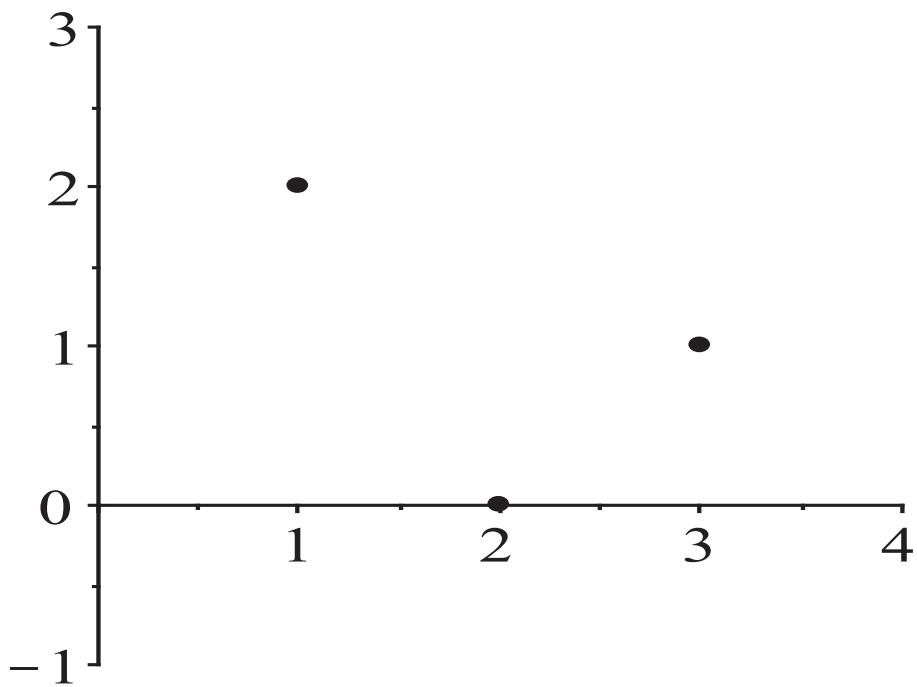
Finally, we see how to graph a function defined by a table. An alternative form of the **plot** command accepts two arguments: the first argument is the list of x -values and the second argument is the list of y -values. For example,

> *plot([1, 2, 3], [2, 0, 1])*



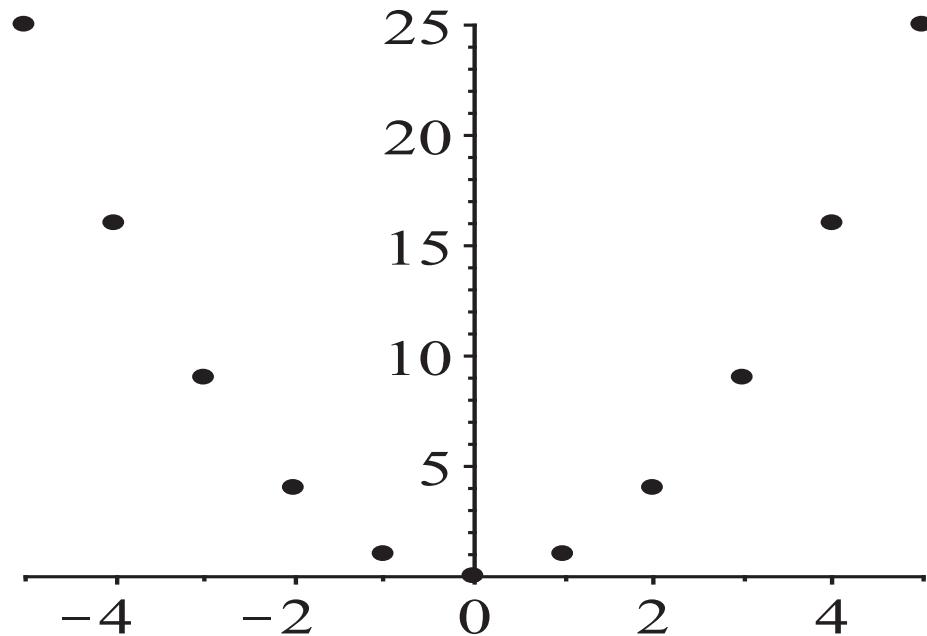
Note that Maple connected the points $(1, 2)$, $(2, 0)$, and $(3, 1)$ to draw the graph. We can use the **style=point** option to draw points instead of connecting the dots. In addition, the options **symbol=solidcircle** and **symbolsize=15** will cause the points to be drawn as solid circles 15 points 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])

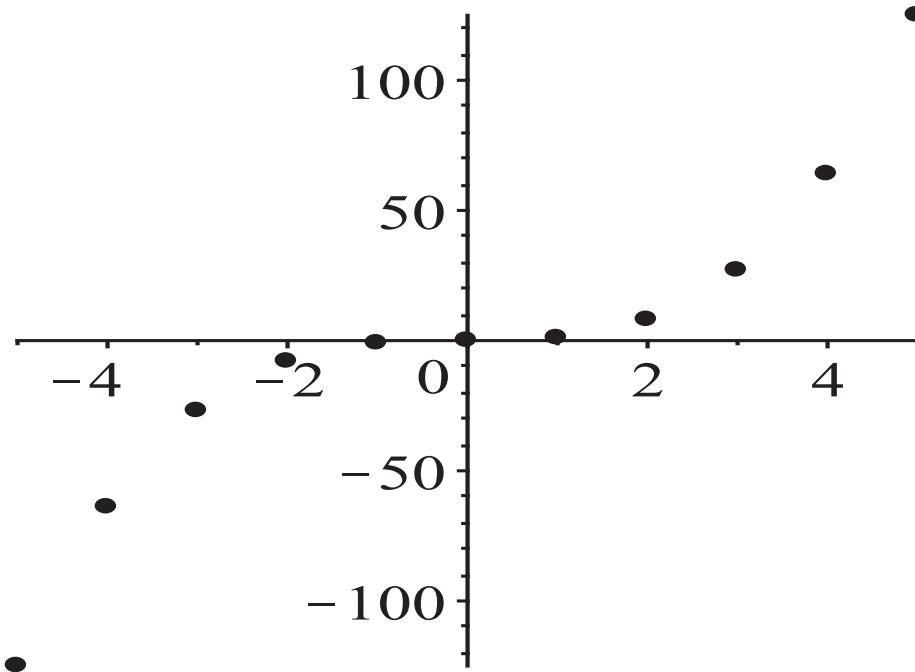


Graphing a function defined by a table can be done in the same way, using the **indices** and **entries** commands to create lists of the x and y coordinates of the desired points. Remember that Maple orders the indices and entries so that they are consistent, that is, the x and y values will correspond.

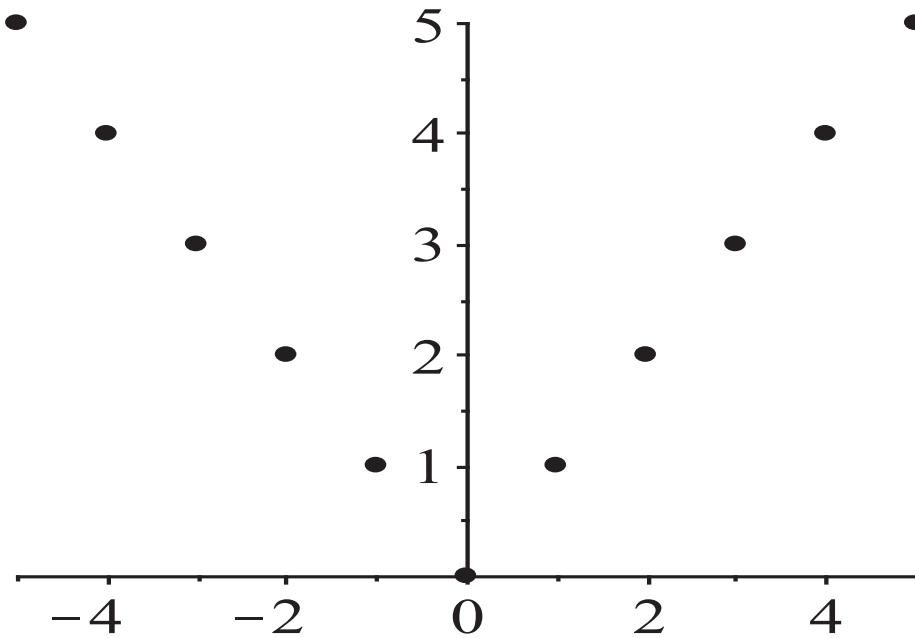
```
> plot([indices(f1, nolist)], [entries(f1, nolist)], style = point,
      symbol = solidcircle, symbolsize = 15, view = [-5 .. 5, 0 .. 25])
```



```
> 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 have 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  
(2.136)
```

```
> ceil(2.7)  
3  
(2.137)
```

There are some additional functions related to those two. 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 toward 0. And the **frac** command returns the fractional part of the number.

```
> round(2.7)  
3  
(2.138)
```

```
> trunc(2.7)  
2  
(2.139)
```

```
> trunc(-2.7)  
-2  
(2.140)
```

```
> frac(2.7)  
0.7  
(2.141)
```

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  
(2.142)
```

```
> factorial(6)  
720  
(2.143)
```

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, 3x2 + 5, {"a", "b", "c"}  
aSequence := 1, "a", x, π, 3x2 + 5, {"a", "b", "c"}  
(2.144)
```

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 way as lists and sets, with the selection operation, as follows.

```
> aSequence3
      x
```

(2.145)

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
```

(2.146)

```
> 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**. In procedures, we will often 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 displaying output.

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

Building Sequences: The 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
```

(2.147)

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, "a", x, π, 3x2 + 5, {"a", "b", "c"}, 1, 2, 3, 4
```

(2.148)

```
> sequence2 := sequence2, 15
sequence2 := 1, 2, 3, 4, 15
```

(2.149)

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 $a, ar, ar^2, ar^3, \dots, ar^n$.

```

1 GeometricSeq := proc (a, r, n)
2   local S, i;
3   S := NULL;
4   for i from 0 to n do
5     S := S, a*r^i;
6   end do;
7   return S;
8 end proc;
```

> *GeometricSeq*(3, 4, 10)
3, 12, 48, 192, 768, 3072, 12288, 49152, 196608,
786432, 3145728 (2.150)

In that procedure, the output sequence **S** 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: The seq Command

We have already seen several examples of the **seq** command. We 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*($3 \cdot 4^i, i = 0 \dots 10$)
3, 12, 48, 192, 768, 3072, 12288, 49152, 196608,
786432, 3145728 (2.151)

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, that is, 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*($3 \cdot 4^i, i = 0 \dots 10, 2$)
3, 48, 768, 12288, 196608, 3145728 (2.152)

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

> *seq*($i, i = 2.3 \dots 5.6, 0.25$)
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 (2.153)

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
```

(2.154)

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
```

(2.155)

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, 3, 5, 7, 11, 13})
4, 9, 25, 49, 121, 169
```

(2.156)

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
```

(2.157)

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
```

(2.158)

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 = 3x^5 + 2x^4 - x^3 + 7x^2 - 8x + 9)
9x^10, 4x^8, x^6, 49x^4, 64x^2, 81
```

(2.159)

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

Building Sequences: The \$ Operator

The **\$** operator is an alternative to the **seq** command, though it is somewhat 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^k $ k = 0 .. 10
3, 12, 48, 192, 768, 3072, 12288, 49152, 196608,
786432, 3145728
```

(2.160)

Note that this is the same as **seq(3*4^k,k=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
```

(2.161)

```
> 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
```

(2.162)

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

```
> seq(anIndex · 2 + 3, anIndex = 1 .. 5)  
5, 7, 9, 11, 13
```

(2.163)

The **\$** operator is particularly useful in the following two situations. First, 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
```

(2.164)

Second, 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"
```

(2.165)

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

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

(2.166)

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, and we will follow the Maple convention.) 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 ↪ Fib(n - 1) + Fib(n - 2) (2.167)
```

Next, we set the initial values as follows.

```
> Fib(1) := 0
Fib(1) := 0 (2.168)
```

```
> Fib(2) := 1
Fib(2) := 1 (2.169)
```

Now, Maple will compute values of the sequence.

```
> Fib(7)
8 (2.170)
```

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

```
> seq(fib(n), n = 1 .. 20)
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,
1597, 2584, 4181 (2.171)
```

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 causes Maple to “remember” any values for the procedure that it has already computed by storing them in a table.

```
1 Fib2 := proc(n :: posint)
2   option remember;
3   if n <= 2 then
4     return 1;
5   else
6     return Fib2(n-1) + Fib2(n-2);
7   end if;
8 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 us see how long it takes to compute the 10 000th Fibonacci number.

```
> st := time() : Fib2(10000) : time() - st
0.030
```

(2.172)

The **time** command returns the total CPU time used in the current Maple session. Thus, 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 10 000th Fibonacci number using **Fib2**. Note that if we repeat the computation,

```
> st := time() : Fib2(10000) : time() - st
0.007
```

(2.173)

the total time taken drops considerably. This is because Maple does not 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
0.516
```

(2.174)

Note that the purely recursive implementation **Fib** cannot be used to compute the 10 000th Fibonacci number. In fact, to compute the 10 000th Fibonacci number, **Fib** would need to be invoked approximately

```
> Fib2(9999)
2.08 × 102089
```

(2.175)

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

$\frac{(2.175)}{1000000000}$

$2.079360824 \times 10^{2080}$

(2.176)

seconds, or

$$> \frac{(2.176)}{60 \cdot 60 \cdot 24 \cdot 365} \\ 6.593609919 \times 10^{2072} \quad (2.177)$$

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.

$$> \text{add}(i^2, i = 1 .. 10) \\ 385 \quad (2.178)$$

To compute the sum of the members of a list, you can enter:

$$> \text{add}(i, i \text{ in } [1, 2, 4, 6, 9, 11, 14]) \\ 47 \quad (2.179)$$

This can be shortened by providing the list as the sole argument.

$$> \text{add}([1, 2, 4, 6, 9, 11, 14]) \\ 47 \quad (2.180)$$

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, $\sum_{k=1}^n k^2$.

$$> \text{sum}(k^2, k = 1 .. n) \\ \frac{(n+1)^3}{3} - \frac{(n+1)^2}{2} + \frac{n}{6} + \frac{1}{6} \quad (2.181)$$

We can also compute the sum of the terms with even index up to $2n$ in a geometric series, that is,

$$\sum_{k=0}^n ar^{2k}.$$

$$> \text{sum}(a \cdot r^{2k}, k = 0 .. n)$$

$$\frac{a(r^2)^{n+1}}{r^2 - 1} - \frac{a}{r^2 - 1} \quad (2.182)$$

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

$$> \text{sum}(k \cdot x^{k-1}, k = 1 .. \text{infinity}) \text{ assuming } |x| < 1$$

$$\frac{1}{(x - 1)^2} \quad (2.183)$$

Note the keyword **assuming** followed by a condition expressing the radius of convergence. 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 explore the cardinality 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?

Enumerating the Positive Rationals

We begin by reviewing the description in Example 4. The first element of the list is the rational number $\frac{1}{1}$. Second, we list the positive rationals $\frac{p}{q}$ such that $p + q = 3$. Next 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 refer to n as the stage, so that in stage 5, for example, we are 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 implement this as a for loop with index variable n .

Within each stage, that is, 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. Therefore, 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.

```
1 ListRationals := proc (max::posint)
2   local L, n, p, q;
3   L := [];
4   for n from 2 to max do
5     for q from 1 to n-1 do
6       p := n-q;
7       if not (p/q in L) then
8         L := [op(L), p/q];
9       end if;
10      end do;
11    end do;
12    return L;
13  end proc:
```

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

```
> ListRationals(6)
[1, 2, 1/2, 3, 1/3, 4, 3/2, 2/3, 1/4, 5, 1/5] (2.184)
```

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. Thus, we compute the list up to stage 64.

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

We suppress the output because this is a long list:

```
> nops(RatsTo64)
1259 (2.185)
```

Now, we work backwards from the end of the list until we find the desired fraction. A simple loop will help with this. Including a **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 for each iteration. Once we find the location of the desired fraction, we display the location and **break** the loop.

```
> for i from 1259 by -1 to 1 do
  if RatsTo64[i] = 29/35 then
    print(i);
    break;
  end if;
end do;
1245 (2.186)
```

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
(2.187)
```

```
> denom(29/35)
35
(2.188)
```

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

```
1 LocateRational := proc(r::rational)
2   local stage, L, i;
3   if r <= 0 then
4     error "Input value must be positive .";
5   end if;
6   stage := numer(r) + denom(r);
7   L := ListRationals(stage);
8   for i from nops(L) to 1 by -1 do
9     if L[i] = r then
10       return i;
11     end if;
12   end do;
13 end proc;
```

```
> LocateRational(75/197)
22566
(2.189)
```

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 100th 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. Therefore, 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.

$$> \text{sum}(k - 1, k = 2 .. n)$$

$$\frac{(n + 1)^2}{2} - \frac{3n}{2} - \frac{1}{2} \quad (2.190)$$

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

$$> \text{factor} \quad (2.190)$$

$$\frac{n(n - 1)}{2} \quad (2.191)$$

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 . Alternatively, $F(n)$ is the number of distinct positive rational numbers $\frac{p}{q}$ such that $p + q \leq n$. Thus, we have determined that

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

We now return to the question of how many stages we need to compute in order to find the 100th rational number. We can restate this as follows: find n such that $F(n) \geq 100$. Combining our inequalities, we have that $\frac{n(n - 1)}{2} \geq 100$. 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.)

$$> \text{solve} \left(\frac{n(n - 1)}{2} = 100. \right)$$

$$14.65097170, -13.65097170 \quad (2.192)$$

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

$$> \text{RatsTo15} := \text{ListRationals}(15);$$

$$> \text{nops}(\text{RatsTo15})$$

$$71 \quad (2.193)$$

$$> \text{RatsTo17} := \text{ListRationals}(17);$$

$$> \text{nops}(\text{RatsTo17})$$

$$95 \quad (2.194)$$

$$> \text{RatsTo18} := \text{ListRationals}(18);$$

$$> \text{nops}(\text{RatsTo18})$$

$$101 \quad (2.195)$$

```
> RatsTo18[100]
```

$$\frac{5}{13}$$

(2.196)

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 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. We 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 create the list by using the **seq** command.

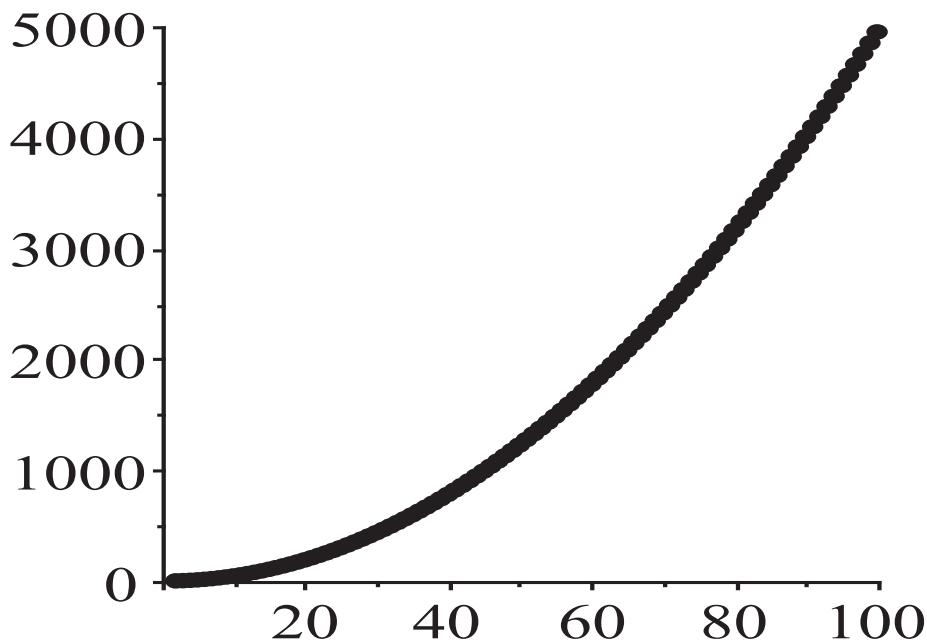
```
> xValues := [seq(n, n = 2 .. 100)]:
```

For the y -values, we use the **seq** command with the formula for the upper bound: $\frac{n(n - 1)}{2}$.

```
> boundValues := [seq(\frac{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])
```

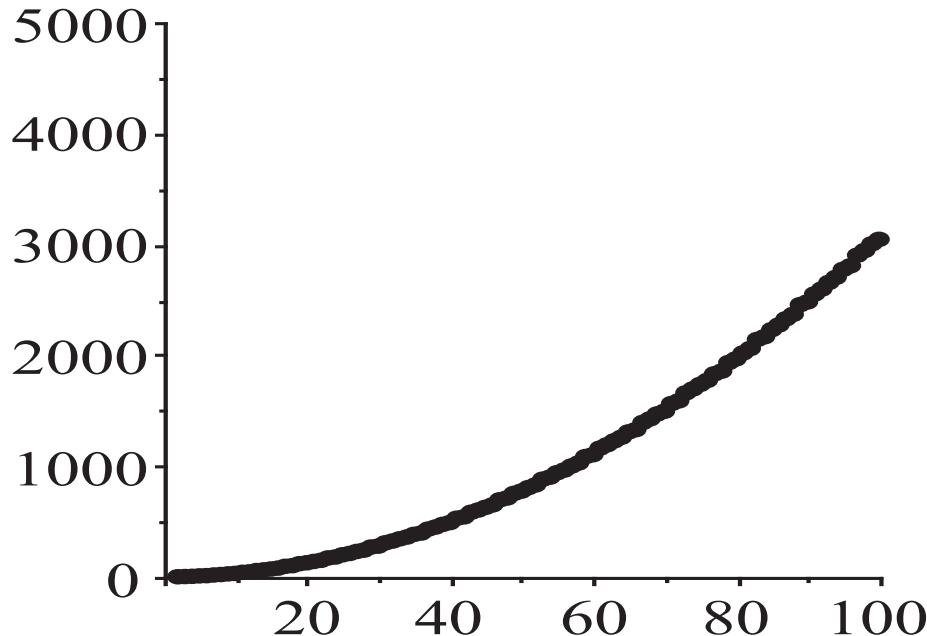


To find the actual values of $F(n)$, we need the size of the list returned by **ListRationals** applied to n . In other words, we apply **nops** to the result of **ListRationals(n)**. Again, we use **seq** to form the list of these counts.

```
> 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])
```

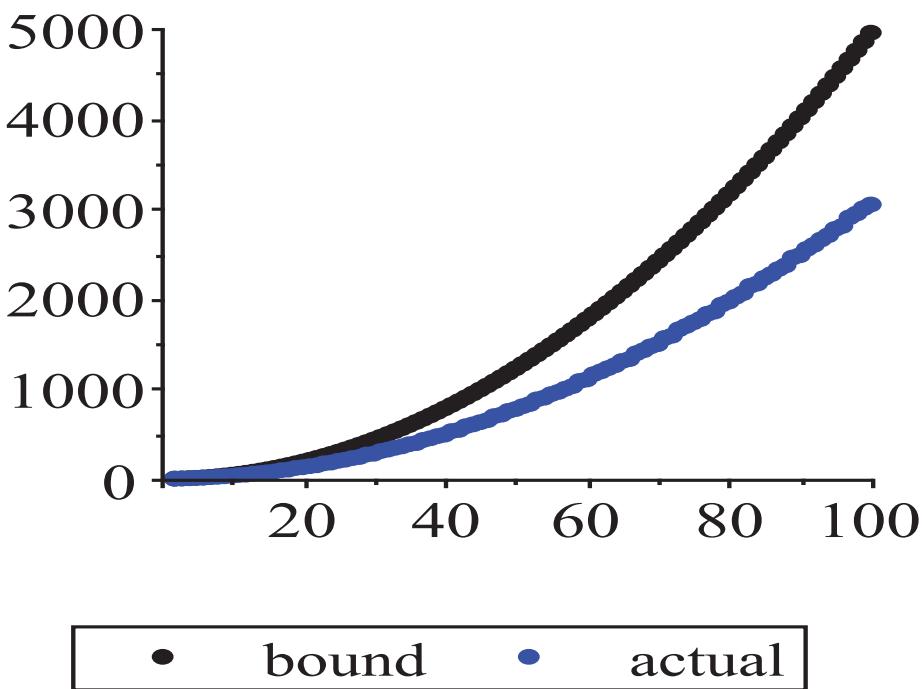


Since we used the same **view** for both graphs, you can see that the value of $F(n)$ is much smaller than the upper bound. We can make the comparison easier by overlaying the graphs. We do this as follows. First, we assign the graphs to names. (We also change the color of the graph of the $F(n)$ data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

Finally, 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 begin this section by describing a variety of ways to construct matrices in Maple. Then, we 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 := 
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$
 (2.197)
```

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 the first two arguments to the **Matrix** command.

```
> m2 := Matrix(2,3,[[1,2,3],[4,5,6]])
m2 := 
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$
 (2.198)
```

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)

However, 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 := 
$$\begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & 5 & 6 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$
 (2.199)
```

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 := 
$$\begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & 5 & 6 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$
 (2.200)
```

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 := 
$$\begin{bmatrix} 5 & 5 & 5 \\ 5 & 5 & 5 \\ 5 & 5 & 5 \end{bmatrix}$$
 (2.201)
```

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 := 
$$\begin{bmatrix} 5 & 5 & 5 & 2 \\ 5 & 5 & 5 & 2 \\ 5 & 5 & 5 & 2 \end{bmatrix}$$
 (2.202)
```

For very small matrices, you can use a shortcut notation, as illustrated below. The entries of the matrix are listed between a pair of angle brackets (\langle and \rangle), with the entries in each row listed in order separated by commas and the rows separated by semicolons.

$$\begin{aligned} > m8 := \langle 1, 2, 3, 4; 5, 6, 7, 8; 9, 10, 11, 12 \rangle \\ m8 := \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \end{aligned} \quad (2.203)$$

If you prefer to enter the values as columns, you use vertical bars (|) in place of the semicolons.

$$\begin{aligned} > m9 := \langle 1, 2, 3, 4 | 5, 6, 7, 8 | 9, 10, 11, 12 \rangle \\ m9 := \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix} \end{aligned} \quad (2.204)$$

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.

$$\begin{aligned} > m6[3, 1] := 6 \\ m6_{3,1} := 6 \end{aligned} \quad (2.205)$$

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

$$\begin{aligned} > m6 \\ m6 := \begin{bmatrix} 5 & 5 & 5 \\ 5 & 5 & 5 \\ 6 & 5 & 5 \end{bmatrix} \end{aligned} \quad (2.206)$$

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.

$$\begin{aligned} > m7copy := m7 \\ m7copy := \begin{bmatrix} 5 & 5 & 5 & 2 \\ 5 & 5 & 5 & 2 \\ 5 & 5 & 5 & 2 \end{bmatrix} \end{aligned} \quad (2.207)$$

```
> m7[1,2] := 11
m71,2 := 11
```

(2.208)

```
> m7

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

(2.209)

```

```
> m7copy

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

(2.210)

```

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 :=  $\begin{bmatrix} 5 & 11 & 5 & 2 \\ 5 & 5 & 5 & 2 \\ 5 & 5 & 5 & 2 \end{bmatrix}$ 
(2.211)

```

```
> m7[2,4] := 23
m72,4 := 23
```

(2.212)

```
> m7, m7copy, m7realcopy

$$\begin{bmatrix} 5 & 11 & 5 & 2 \\ 5 & 5 & 5 & 23 \\ 5 & 5 & 5 & 2 \end{bmatrix}, \begin{bmatrix} 5 & 11 & 5 & 2 \\ 5 & 5 & 5 & 23 \\ 5 & 5 & 5 & 2 \end{bmatrix}, \begin{bmatrix} 5 & 11 & 5 & 2 \\ 5 & 5 & 5 & 2 \\ 5 & 5 & 5 & 2 \end{bmatrix}$$

(2.213)

```

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.

$$> m10 := \text{Matrix}(2, 3, [1, 2, 3, 4, 5, 6])$$

$$m10 := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad (2.214)$$

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

$$> m11table := \text{table}([(1, 2) = 5, (1, 3) = 6, (2, 1) = -2])$$

$$m11table := \text{table}([(1, 2) = 5, (1, 3) = 6, (2, 1) = -2]) \quad (2.215)$$

$$> m11 := \text{Matrix}(2, 3, m11table)$$

$$m11 := \begin{bmatrix} 0 & 5 & 6 \\ -2 & 0 & 0 \end{bmatrix} \quad (2.216)$$

As a shortcut, the data defining the table can be passed as a set to form the matrix.

$$> \text{Matrix}(2, 3, \{(1, 2) = 5, (1, 3) = 6, (2, 1) = -2\})$$

$$\begin{bmatrix} 0 & 5 & 6 \\ -2 & 0 & 0 \end{bmatrix} \quad (2.217)$$

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.

$$> m12F := (i, j) \rightarrow i + j$$

$$m12F := (i, j) \mapsto i + j \quad (2.218)$$

$$> m12 := \text{Matrix}(4, m12F)$$

$$m12 := \begin{bmatrix} 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \\ 5 & 6 & 7 & 8 \end{bmatrix} \quad (2.219)$$

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.

$$> m13 := \text{Matrix}([[1, 2, 3], [4, 5, 6]])$$

$$m13 := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad (2.220)$$

$$> m14 := \text{Matrix}([[-2, 3, -1], [1, 5, 2]])$$

$$m14 := \begin{bmatrix} -2 & 3 & -1 \\ 1 & 5 & 2 \end{bmatrix} \quad (2.221)$$

$$> m13 + m14$$

$$\begin{bmatrix} -1 & 5 & 2 \\ 5 & 10 & 8 \end{bmatrix} \quad (2.222)$$

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

$$> 3m14$$

$$\begin{bmatrix} -6 & 9 & -3 \\ 3 & 15 & 6 \end{bmatrix} \quad (2.223)$$

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

Matrix multiplication is computed with the **.** (dot, typed as a period) operator instead of an asterisk. This is to emphasize that matrix multiplication is not commutative.

$$> m15 := \text{Matrix}([[3, 6, 11, 1], [-2, 5, 2, 0], [4, 8, 9, -3]])$$

$$m15 := \begin{bmatrix} 3 & 6 & 11 & 1 \\ -2 & 5 & 2 & 0 \\ 4 & 8 & 9 & -3 \end{bmatrix} \quad (2.224)$$

$$> m16 := \text{Matrix}([[2, 5], [1, -2], [3, 7], [-1, 0]])$$

$$m16 := \begin{bmatrix} 2 & 5 \\ 1 & -2 \\ 3 & 7 \\ -1 & 0 \end{bmatrix} \quad (2.225)$$

$$> m15 \bullet m16$$

$$\begin{bmatrix} 44 & 80 \\ 7 & -6 \\ 46 & 67 \end{bmatrix} \quad (2.226)$$

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.

```
> m17 := Matrix([[1,2,3],[4,5,6],[7,8,9]])
```

$$m17 := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad (2.227)$$

```
> m175
```

$$\begin{bmatrix} 121824 & 149688 & 177552 \\ 275886 & 338985 & 402084 \\ 429948 & 528282 & 626616 \end{bmatrix} \quad (2.228)$$

The transpose of a matrix is most easily computed with the $\wedge+$ operator, as follows. (We show the 1-D input and the appearance of the input in 2-D mode.)

```
> m16+;
```

$$\begin{bmatrix} 2 & 1 & 3 & -1 \\ 5 & -2 & 7 & 0 \end{bmatrix} \quad (2.229)$$

```
> m16+
```

$$\begin{bmatrix} 2 & 1 & 3 & -1 \\ 5 & -2 & 7 & 0 \end{bmatrix} \quad (2.230)$$

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

```
> LinearAlgebra[Transpose](m16)
```

$$\begin{bmatrix} 2 & 1 & 3 & -1 \\ 5 & -2 & 7 & 0 \end{bmatrix} \quad (2.231)$$

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.

```
> m18 := Matrix([[1,2],[3,4]])
```

$$m18 := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad (2.232)$$

```
> m19 := Matrix([[1,2],[3,4]])
```

$$m19 := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad (2.233)$$

```
> evalb(m18 = m19)
```

$$\text{false} \quad (2.234)$$

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](m18, m19)
true
```

(2.235)

Zero–One Matrices

With Maple, we can create and manipulate zero–one matrices as well. In particular, we will 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 will 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 instead make use of Maple’s **Bits** package. This package provides several commands related to performing operations on bits and bit strings. We will only make use of two commands: **And** and **Or**. First we load the package.

```
> with(Bits, And, Or) :
```

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

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

(2.236)

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

(2.237)

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

(2.238)

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

(2.239)

(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 := 
$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

```

(2.240)

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 will 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 want to be more specific. We will 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 use **{0,1}** as the type of entry allowed in the matrix. Remember that braces in a structured type indicate that any of the options inside the braces are acceptable. (Note: the single right quotes are required and delay evaluation so that Maple does not 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
```

(2.241)

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

(2.242)

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.

```

1  BoolJoin := proc(A::'Matrix'({0,1}),B::'Matrix'({0,1}))
2    local numrows, numcols, R, r, c;
3    uses LinearAlgebra, Bits;
4    numrows := RowDimension(A);
5    numcols := ColumnDimension(A);
6    if numrows <> RowDimension(B) or numcols <> ColumnDimension(B)
7      then
        error "Input matrices must be of the same size.";
```

```

8   end if ;
9   R := Matrix (numrows , numcols ) ;
10  for r from 1 to numrows do
11    for c from 1 to numcols do
12      R [r , c ] := Or (A [r , c ] , B [r , c ]) ;
13    end do ;
14  end do ;
15  return R ;
16 end proc :

```

Below we apply this procedure to two example matrices.

$$> zo1 \\ \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad (2.243)$$

$$> zo2 := Matrix ([[1, 0, 0], [1, 1, 1], [0, 0, 0]]) \\ zo2 := \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad (2.244)$$

$$> BoolJoin (zo1, zo2) \\ \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad (2.245)$$

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 **Ands**, as you might hope. Instead, we 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
```

(2.246)

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
```

(2.247)

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

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

(2.248)

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

(2.249)

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

(2.250)

In terms of the generic formula, we initialize $c = (a_{i1} \wedge b_j)$. 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**.

```

1  BoolProduct := proc(A::'Matrix'({0,1}),B::'Matrix'({0,1}))
2    local m, k, n, C, i, j, c, p;
3    uses LinearAlgebra, Bits;
4    m := RowDimension(A);
5    k := ColumnDimension(A);
6    if k <> RowDimension(B) then
7      error "Dimension mismatch.";
8    end if;
9    n := ColumnDimension(B);
10   C := Matrix(m,n);
11   for i from 1 to m do
12     for j from 1 to n do
13       c := And(A[i,1],B[1,j]);
```

```

14   for p from 2 to k do
15     c := Or(c, And(A[i, p], B[p, j]));
16   end do;
17   C[i, j] := c;
18 end do;
19 end do;
20 return C;
21 end proc;

```

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

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

$$\text{Ex8A} := \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \quad (2.251)$$

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

$$\text{Ex8B} := \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \quad (2.252)$$

> *BoolProduct*(*Ex8A*, *Ex8B*)

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} \quad (2.253)$$

Solutions to Computer Projects and Computations and Explorations

Computer Projects 3

Given fuzzy sets A and B , find \bar{A} , $A \cup B$, and $A \cap B$ (see preamble to Exercise 73 of Section 2.2).

Solution: In Section 2.2 of this manual, we described Maple's built-in **MultiSet** object, which provides one solution for this problem. That approach would yield little understanding of either fuzzy sets or Maple, however. Instead, we will create a procedure from scratch. We will compute the union and leave complement and intersection to the reader. Recall, from Exercise 74, 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 procedures to convert between them. We 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. However, 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.

```

1 FuzzyUnion := proc (A, B)
2   local U, e, Abits, Bbits, Cbits, C;
3   U := { };
4   for e in A do
5     U := U union {e[1]};
6   end do;
7   for e in B do
8     U := U union {e[1]};
9   end do;
10  Abits := RosterToBit (A, U);
11  Bbits := RosterToBit (B, U);
12  Cbits := zip (max, Abits, Bbits);
13  C := BitToRoster (Cbits, U);
14  return C;
15 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]} (2.254)

> *fuzzyB* := {[“a”, 0.5], [“b”, 0.1], [“d”, 0.2]}
fuzzyB := {[“a”, 0.5], [“b”, 0.1], [“d”, 0.2]} (2.255)

> *FuzzyUnion* (*fuzzyA,fuzzyB*)
{[“a”, 0.5], [“b”, 0.3], [“c”, 0.7], [“d”, 0.2]} (2.256)

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. Thus, 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.

```

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

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

$$\text{symmetricMatrix} := \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix} \quad (2.257)$$

> *IsSymmetric* (*symmetricMatrix*)

true (2.258)

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

$$\text{notSymmetricMatrix} := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad (2.259)$$

> *IsSymmetric* (*notSymmetricMatrix*)

false (2.260)

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 will 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 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.

```

1 NextBitS := proc (Bits :: list ({0, 1})) 
2   local newBits, i;
3   newBits := Bits;
4   for i from 1 to nops (newBits) do
5     if newBits[i] = 1 then
6       newBits[i] := 0;
7     else
8       newBits[i] := 1;
9     return newBits;
10    end if;
11  end do;
12  return NULL;
13 end proc;
```

> *NextBitS*([1, 1, 1, 0, 0, 1, 0])
[0, 0, 0, 1, 0, 1, 0] (2.261)

Next, we will 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.

```

1 BitToSubset := proc (Bits :: list ({0, 1}), S :: set)
2   local subS, i;
3   subS := {};
4   for i from 1 to nops (Bits) do
5     if Bits[i] = 1 then
6       subS := subS union {S[i]};
7     end if;
8   end do;
9   return subS;
10 end proc;
```

> *BitToSubset*([0, 1, 1, 0, 0], {"a", "b", "c", "d", "e"})
{"b", "c"} (2.262)

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

```

1 Subsets := proc (S :: set)
2   local Bits;
```

```

3 Bits := [0 $ nops(S)];
4 while Bits <> NULL do
5     print(BitToSubset(Bits, S));
6     Bits := NextBits(Bits);
7 end do;
8 return NULL;
9 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”} (2.263)

```

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, without using the built-in **MultiSet** object type. Instead, represent a multiset as a set of pairs **[a,m]** where **m** is the multiplicity of the element **a**. (Refer to Section 2.2 for information about multisets.)

Exercise 5. Write procedures to compute the Jaccard similarity and Jaccard distance between sets. (Refer to the preamble of Exercise 71 in Section 2.2 for the definitions of Jaccard similarity and distance.)

Exercise 6. 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 7. Write a procedure to find the inverse of a function defined by a table.

Exercise 8. Write a procedure to find the composition of functions defined by tables.

Exercise 9. 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 10. Write a procedure **ArithmeticSeq**, similar to **GeometricSeq** from above, that produces an arithmetic sequence.

Exercise 11. Find the first 20 terms of the sequences defined by the recurrence relations below.

- a) $a_n = 2a_{n-1} + 3a_{n-2}$, with $a_1 = 1$ and $a_2 = 0$.
- b) $a_n = a_{n-1} + na_{n-2} + n^2a_{n-3}$, with $a_1 = 1$, $a_2 = 1$, and $a_3 = 3$.
- c) $a_n = a_{n-1}a_{n-2} + 1$, with $a_1 = a_2 = 1$.

Exercise 12. 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 13. 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 14. Use Maple to find formulae for the sum of the n th powers of the first k positive integers for n up to 10.

Exercise 15. 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**, that is, the value $F(n)$, is recorded in a list.

Exercise 16. Find a value R so that the graph of $R \cdot \frac{n(n-1)}{2}$ is just above the graph of $F(n)$. Use your **listActuals** procedure to expand the data and refine the value of R .

Exercise 17. Use Maple to find the 100th positive rational number in the list generated by **ListRationals**. What about the 1000th? 10 000th? (If you completed it, the result of the previous exercise can be helpful.)

Exercise 18. Write a procedure **BoolMeet** implementing the Boolean meet operation on zero–one matrices.

3 Algorithms

Introduction

In this chapter, we supplement the discussion of algorithms presented in the text with their implementation in Maple. In Section 3.1, we discuss the process of turning step-by-step instructions describing a procedure and pseudocode for a procedure into Maple code. In the second section, we make use of Maple's graphing capabilities to visualize functions related by the big-O notation. In Section 3.3, we explore the average-case complexity of algorithms by considering the performance of a procedure on input.

In this chapter, please keep in mind the difference between an algorithm and its implementation (referred to as a procedure in Maple parlance). An algorithm refers to an approach to solving a particular problem, while a procedure is the implementation of the algorithm within Maple. In this manual, we will also distinguish between complexity and performance. Complexity is a measure of an algorithm and is generally measured by counting basic operations such as comparisons, while performance takes into account additional factors related to the specifics of an implementation and can be measured by recording the time it takes for the procedure to complete. Some of the factors affecting performance may include: choice of data structures and how the system implements those structures, the kinds of loops employed and how those are implemented by the computer language, and various improvements to efficiency handled by the system (for example, many computer languages, when evaluating a Boolean expression such as $p \wedge q$, will not bother checking q if p is found to be false thereby decreasing the number of operations that need to be performed).

3.1 Algorithms

It is impossible to overemphasize the importance and utility of writing either pseudocode or step-by-step instructions for an algorithm before you write the actual code for the procedure. Doing so helps you organize your ideas about how to solve the problem without the rigid constraints of the particular programming language. The textbook serves as an excellent model for you as you learn how to turn mathematical concepts and solutions to problems into algorithms. Those algorithms can then be turned into procedures in any programming language you choose. This manual will help you turn your pseudocode or step-by-step instructions for algorithms into procedures written in Maple.

Section 3.1 of the textbook describes several algorithms, with an emphasis on how you can describe these algorithms using both English descriptions and pseudocode. Here, we will see how to implement several of these algorithms in Maple. In this chapter, it will be especially important for you to have the text alongside you, as we will not reproduce the descriptions of the algorithms from the text.

Finding the Maximum

The first algorithm we will implement is the algorithm for finding the maximum in an unsorted sequence. This is described in Section 3.1 in the solution to Example 1 as step-by-step instructions and as pseudocode in Algorithm 1. We will build the procedure according to the step-by-step instructions. In order to make the connection between the instructions and the code as clear as

possible, we begin with a procedure with no statements and successively revise it to show the addition of each step. Be warned that the incomplete versions of the procedure will produce errors if you attempt to execute them. In addition, provided that you have not turned off this feature in the system settings, Maple will highlight syntax errors in the code edit regions.

We begin with the basic elements of the procedure definition without code. Specifically, we include the assignment to the name of the procedure, the input with its type, and, since we will have local variables, the local keyword.

```
1 FindMax := proc (L :: list (integer) )  
2   local ;  
3  
4 end proc:
```

Error, ‘;’ unexpected

Note that the parameter declaration indicates that **L** is the name we will use to refer to the parameter, which must be a list and that all of the elements of the list must be integers.

Step 1 in the step-by-step instructions is to “set the temporary maximum equal to the first integer” in the list. We declare a local variable to store the temporary maximum and add a statement in the body of the procedure to assign the first integer in the list to this value.

```
1 FindMax := proc (L :: list (integer) )  
2   local tempMax;  
3   tempMax := L [1] ;  
4  
5 end proc:
```

Step 2, according to Example 1 of the main text, is to compare the next integer to the temporary maximum and update the temporary maximum if necessary. This requires an **if** statement to make the comparison.

```
1 FindMax := proc (L :: list (integer) )  
2   local tempMax;  
3   tempMax := L [1] ;  
4   if tempMax < L [2] then  
5     tempMax := L [2] ;  
6   end if ;  
7  
8 end proc:
```

(We are intentionally following the step-by-step instructions to the letter, so in step 2, “the next integer” is **L[2]**.)

Step 3 tells us to repeat step 2 for all of the integers in the list. We need to revise our code as follows. The fact that we are supposed to repeat step 2 means that we need a loop. Since this loop is

supposed to consider all of the elements of the list beyond the first means that we use a **for** loop over the indices of the list starting at 2. We put the code we wrote for step 2 inside this loop, since that is the instruction being repeated, and replace the specific index “2” with the loop variable. Finally, the loop variable needs to be added to the **local** list.

```
1 FindMax := proc (L :: list (integer) )
2   local tempMax, i;
3   tempMax := L [1];
4   for i from 2 to nops (L) do
5     if tempMax < L [i] then
6       tempMax := L [i];
7     end if ;
8   end do;
9 end proc;
```

Finally, step 4 tells us that once the loop is completed, the value of the temporary maximum is the maximum, so we return that value.

```
1 FindMax := proc (L :: list (integer) )
2   local tempMax, i;
3   tempMax := L [1];
4   for i from 2 to nops (L) do
5     if tempMax < L [i] then
6       tempMax := L [i];
7     end if ;
8   end do;
9   return tempMax;
10 end proc;
```

> *FindMax* ([3, 18, -5, 72, 6, 0])

72

(3.1)

Admittedly, this example may be a bit too simple to warrant such an elaborate process. However, it illustrates an essential point, namely, that a well-written set of step-by-step instructions describing a procedure can be easily turned into working and correct code. Moreover, for nontrivial algorithms, the two-step process of writing instructions for the procedure and then implementing the procedure based on those instructions typically results in the production of a correct implementation more quickly than attempting the implementation without writing instructions.

Take a moment to compare the procedure above with the pseudocode given in Algorithm 1. You will notice that, in this example, they are extremely similar. This is one of the benefits of pseudocode in comparison to step-by-step instructions. However, step-by-step instructions are often easier to write as a first step toward a working procedure. Step-by-step instructions are also typically easier to understand, especially for novice programmers, as they can more easily accommodate explanation and other information that is out of place in pseudocode.

Binary Search

The second example is the binary search algorithm, presented as Algorithm 3 in Section 3.1. The previous example showed how you can use step-by-step instructions to build up the procedure. Starting from pseudocode, writing the final implementation involves translating statements in the pseudocode into legal statements in the programming language and filling in the missing details.

As before, we will go through several iterations as we translate the pseudocode in the text to actual Maple code. Initially, we need to make sure that the input is specified in a way appropriate for Maple; declare the local variables that are indicated in the pseudocode; and make basic syntax adjustments, specifically the **while** loops, **if** statements, and mathematical expressions such as $\lfloor i/2 + j/2 \rfloor$ must be translated into their Maple counterparts. In this case, we turn the sequence a_1, a_2, \dots, a_n of integers specified as input in the text into a list **A**.

```
1 binarysearch := proc (x::integer, A::list(integer) )
2   local i, j, m, location;
3   i := 1;
4   j := n;
5   while i < j do
6     m := floor((i+j)/2);
7     if x > A[m] then
8       i := m + 1;
9     else
10      j := m;
11    end if;
12  end do;
13  if x = A[i] then
14    location := i;
15  else
16    location := 0;
17  end if;
18  return location;
19 end proc:
```

It is too much to expect this first version to properly execute.

```
> binarysearch(19,[1,2,3,5,6,7,8,10,12,13,15,16,18,19,20,22])
```

Error, (in binarysearch) cannot determine if this expression is true or false: $1 < n$

The problem is a simple one to correct. The pseudocode used n as the last index of the sequence of integers, so we need to make that assignment in our code.

```
1 binarysearch := proc (x::integer, A::list(integer) )
2   local n, i, j, m, location;
3   n := nops(A);
4   i := 1;
5   j := n;
6   while i < j do
```

```

7      m := floor( (i+j)/2 );
8      if x > A[m] then
9          i := m + 1;
10     else
11         j := m;
12     end if;
13 end do;
14 if x = A[i] then
15     location := i;
16 else
17     location := 0;
18 end if;
19 return location;
20 end proc:
```

Now we try running it again:

```
> binarysearch(19,[1,2,3,5,6,7,8,10,12,13,15,16,18,19,20,22])
14
```

(3.2)

Observe that the only commands that we added were the declaration of local variables, the assignment of n , and ending the loop and if statements. We also added additional line breaks to be consistent with the style of code used in this manual. Otherwise, the Maple code and pseudocode match very closely. It is, of course, not always quite this straightforward, but well-written pseudocode should contain all the essential elements. Like with proofs, as you become more familiar with pseudocode, you will find yourself more comfortable with leaving some details out.

Bubble Sort

We will next implement the bubble sort, presented in the text as Algorithm 4 of Section 3.1.

For our first attempt at implementing Algorithm 4, we need to specify the input, declare the local variables (which can, in part, be gleaned from the pseudocode), and correct the syntax for the **for** loops and **if** statements.

```

1 bubblesort := proc (A::list(realcons))
2   local i, j;
3   for i from 1 to n - 1 do
4     for j from 1 to n - i do
5       if A[j] > A[j+1] then
6         # interchange A[j] and A[j+1]
7         end if;
8       end do;
9     end do;
10    end proc:
```

(The **realcons** type is Maple's type for real numbers.)

The implementation above gets us close to a correct implementation of the bubble sort algorithm, but there are some problems. The simplest to fix is that **n** is not defined. You can correct this two ways. Either declare **n** as a local variable and set it equal to the number of elements of **A** or replace the two occurrences of **n** with **nops(A)**. There is little difference between these solutions.

```

1 bubblesort := proc (A::list(realcons))
2   local i, j, n;
3   n := nops(A);
4   for i from 1 to n - 1 do
5     for j from 1 to n - i do
6       if A[j] > A[j+1] then
7         # interchange A[j] and A[j+1]
8         end if;
9       end do;
10      end do;
11    end proc;
```

The next problem is the instruction in the pseudocode to interchange a_j with a_{j+1} . Maple contains no such command. Therefore, we will either have to make an **interchange** procedure that can be used within the **bubblesort** procedure or flesh out the code to interchange the two list elements within **bubblesort** itself. We take the first approach here so as to preserve as close a connection as possible between our implementation and the pseudocode.

Before creating an **interchange** procedure, however, there is another issue to take into consideration. Namely, in Maple, within a procedure, you cannot make assignments to an argument. Instead, we need to copy the parameter to a local variable, both in **bubblesort** and in the **interchange** procedure we are about to write.

Our **interchange** procedure will take two arguments: the list of numbers and the index of the smaller of the two positions to be swapped. It will proceed as follows:

1. Set a temporary variable equal to the first value to be swapped.
2. Set the value of the first position equal to the second value.
3. Set the value of the second position equal to the value stored in the temporary variable.

```

1 interchange := proc (L::list, i::posint)
2   local M, temp;
3   M := L;
4   temp := M[i];
5   M[i] := M[i+1];
6   M[i+1] := temp;
7   return M;
8 end proc;
```

Now, we finish our implementation of bubble sort by copying the list to a local name and replacing all the occurrences of the parameter with the local variable; applying the **interchange** procedure; and ending the procedure by explicitly returning the local copy of the list.

```

1 bubblesort := proc (A::list(realcons) )
2   local B, i, j, n;
3   B := A;
4   n := nops (B) ;
5   for i from 1 to n - 1 do
6     for j from 1 to n - i do
7       if B [j] > B [j+1] then
8         B := interchange (B, j);
9       end if;
10      end do;
11    end do;
12    return B;
13  end proc;

```

> *bubblesort* ([3, 18, -5, 72, 6, 0])
[−5, 0, 3, 6, 18, 72] (3.3)

3.2 The Growth of Functions

In this section, we will use Maple to computationally explore the growth of functions. In particular, we will graph functions in order to visually convince ourselves that the big-O relationship is satisfied. We will also see how to use graphs to determine possible witnesses for the constants C and k in the definition of big-O notation. Since, as the textbook mentions, $f(x)$ is $O(g(x))$ if and only if $g(x)$ is $\Omega(f(x))$, the techniques we explore in this section apply also to big-Omega and big-Theta notation.

We begin by considering the function $f(x) = 5x^3 + 4x^2 + 3x + 9$. Theorem 1 from Section 3.2 tells us that this is $O(x^3)$, but we will use this function as an example of using Maple to find values for C and k such that $|f(x)| \leq C|g(x)|$ for all $k \leq x$.

The plot Command

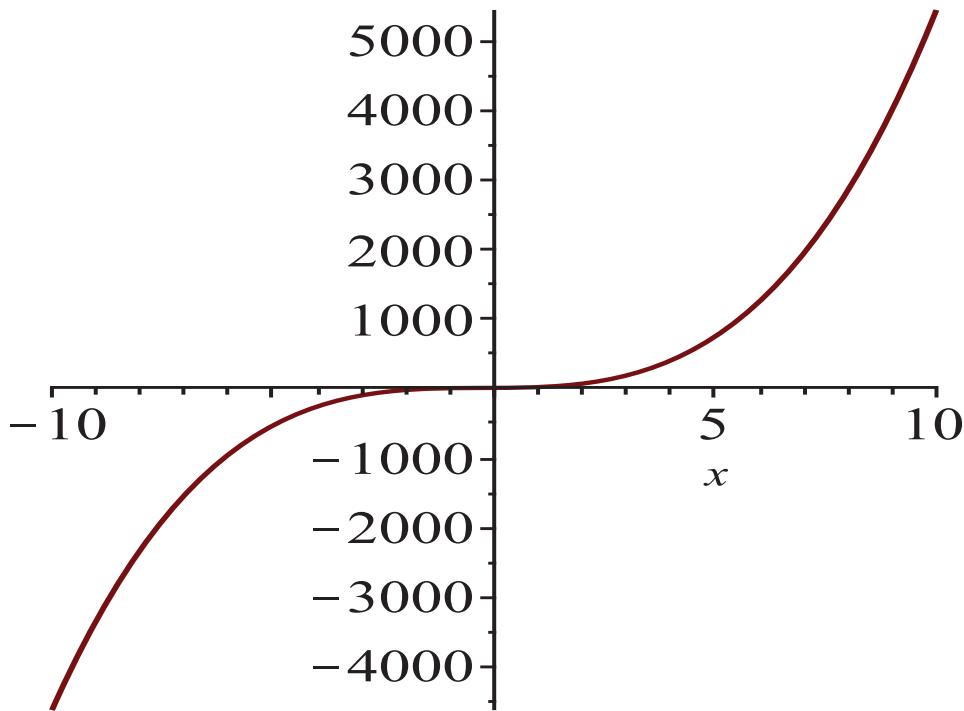
We first look at options to the **plot** command that will be useful in this context. Start by giving names to the formulas.

> *f1* := $5x^3 + 4x^2 + 3x + 9$
f1 := $5x^3 + 4x^2 + 3x + 9$ (3.4)

> *g1* := x^3
g1 := x^3 (3.5)

Graphing $f(x)$ can be as simple as the following.

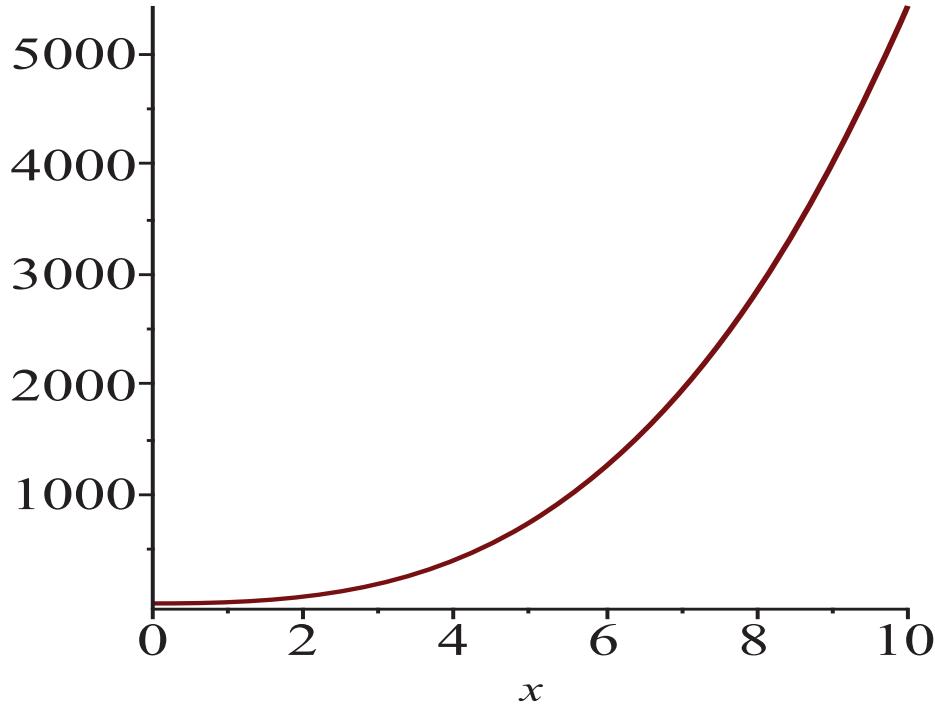
> *plot(f1,x)*



The first argument is the function in terms of an independent variable and the second argument is the name of the variable.

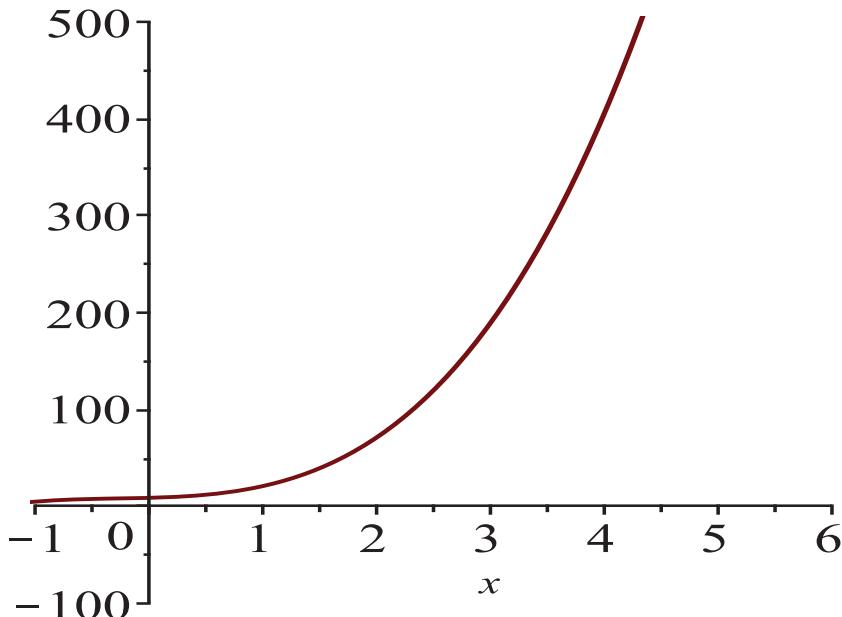
By default, Maple displays the graph with the horizontal axis ranging from -10 to 10 . You can specify a different range of x values as follows:

```
> plot(f1, x = 0 .. 10)
```



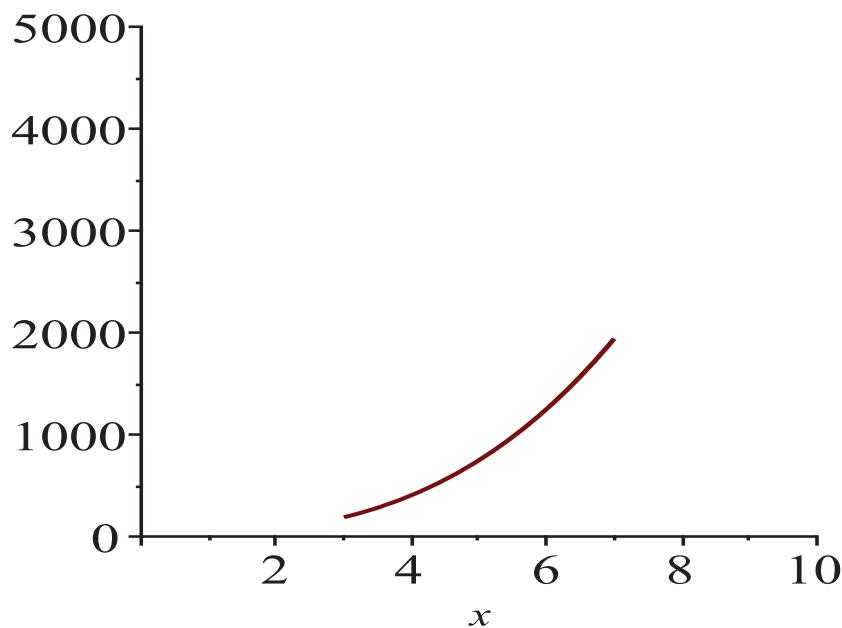
Note that Maple automatically selects the vertical range of the graph. You can control this with the **view** option. You use the view option by setting the keyword **view** equal to a two-element list. The first element of the list consists of the horizontal range to be displayed and the second member of the list is the vertical range to be displayed. For example, to display our graph with x ranging from -1 to 6 and y restricted between -100 and 500 , enter the following command.

```
> plot(f1,x,view = [-1..6,-100..500])
```



It should be observed that, despite a somewhat similar effect, the **view** option is quite different from setting the range of the independent variable. We illustrate the difference with the following example.

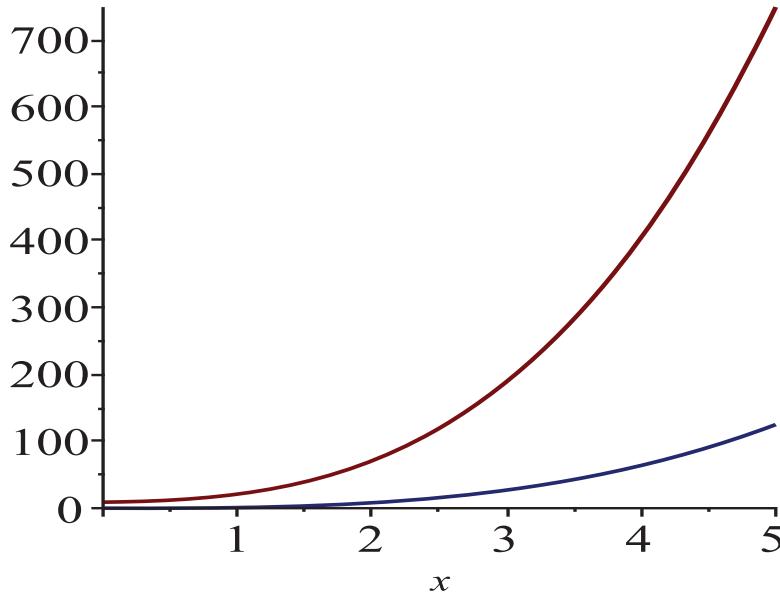
```
> plot(f1,x = 3..7,view = [0..10,0..5000])
```



You see above that the **view** option is specifying the extent of the graph. On the other hand, setting the range **x=3..7** is actually a restriction of the domain of the function.

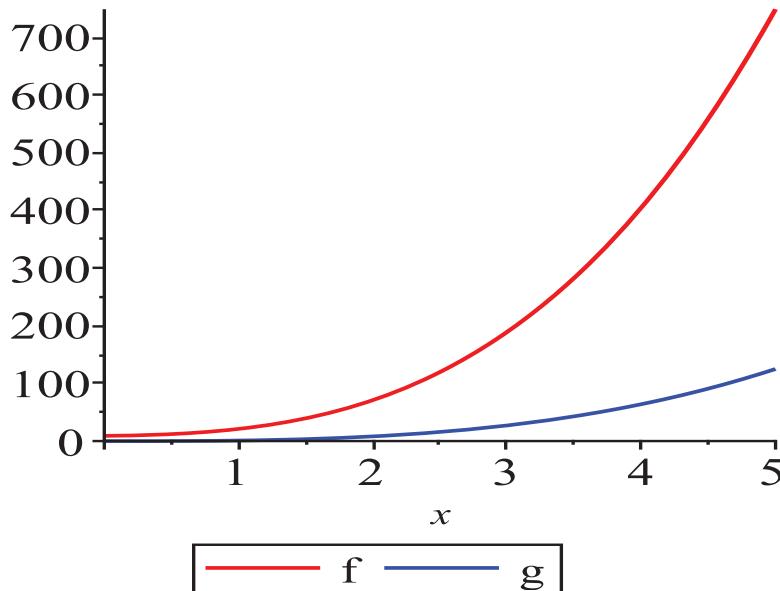
To plot multiple functions in the same graph, we merely issue the **plot** command with a list of the functions as the first argument.

```
> plot([f1, g1], x = 0 .. 5)
```



Note that Maple automatically selects colors for the two functions. You can manually select the colors you want with the **color** option. If you set the **color** keyword equal to a list of color names, the first color is assigned to the first function, the second color to the second function, and so on. You can also create a legend by setting the **legend** keyword equal to a list of strings identifying the functions.

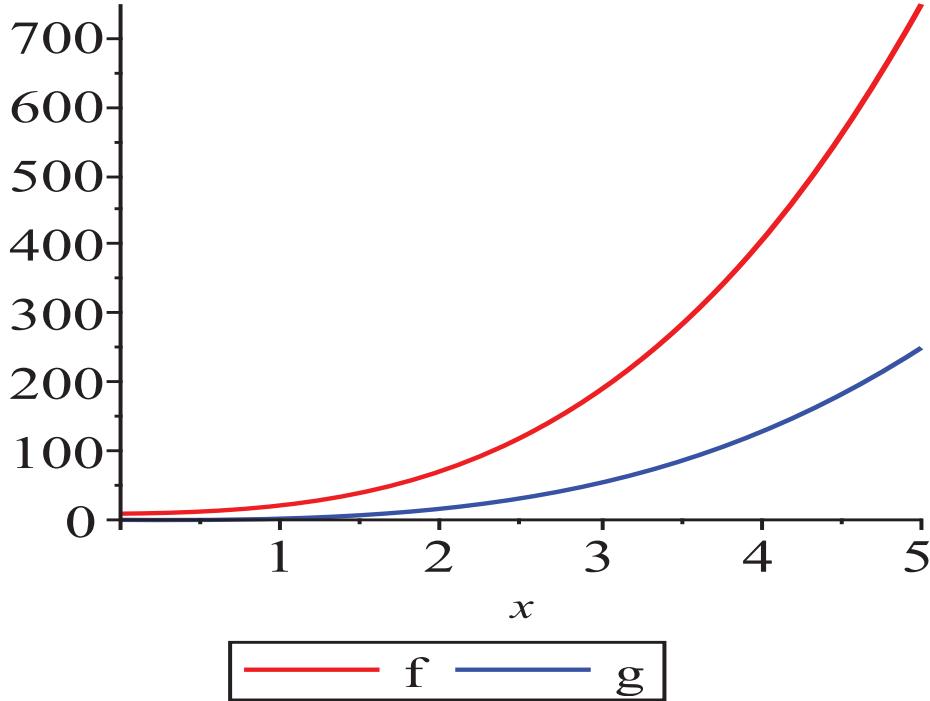
```
> plot([f1, g1], x = 0 .. 5, color = [red, blue], legend = ["f", "g"])
```



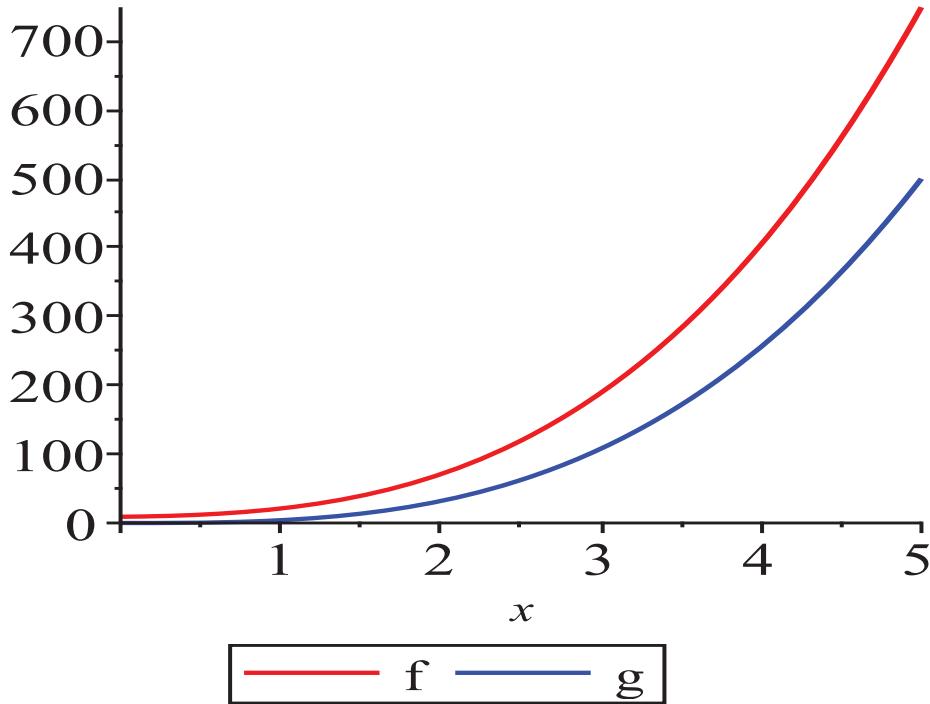
Finding Values for C and k

Now, we can start exploring different values of C for which the equation $f(x) \leq Cg(x)$ is satisfied. To do this, we just have to multiply $\mathbf{g1(x)}$ by different values within the list of functions. We will choose several values until we see a clear crossing.

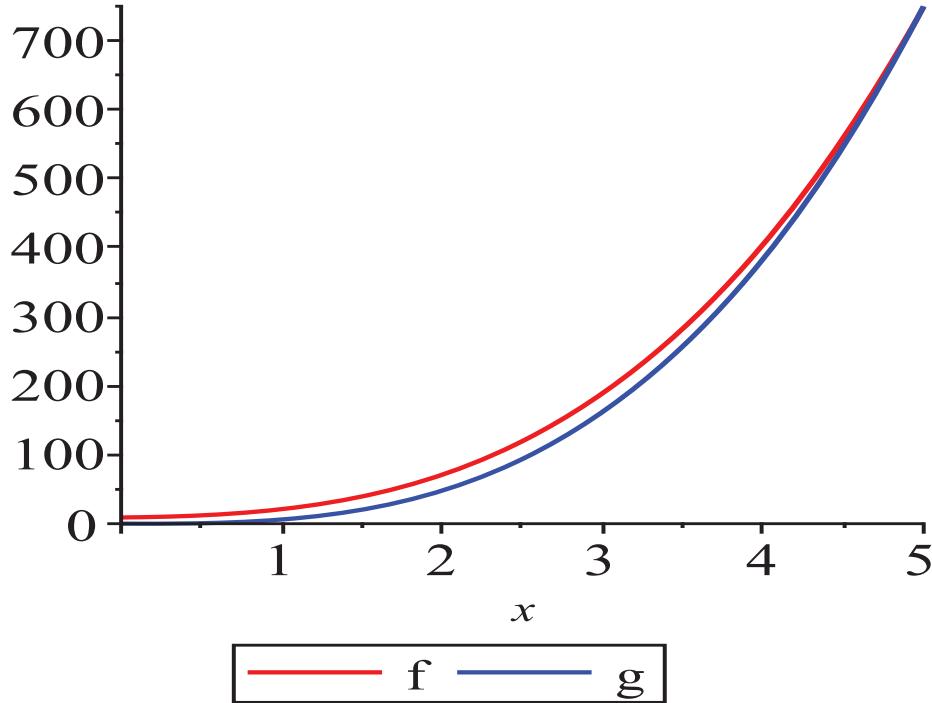
```
> plot([f1, 2 · g1], x = 0 .. 5, color = [red, blue], legend = ["f", "g"])
```



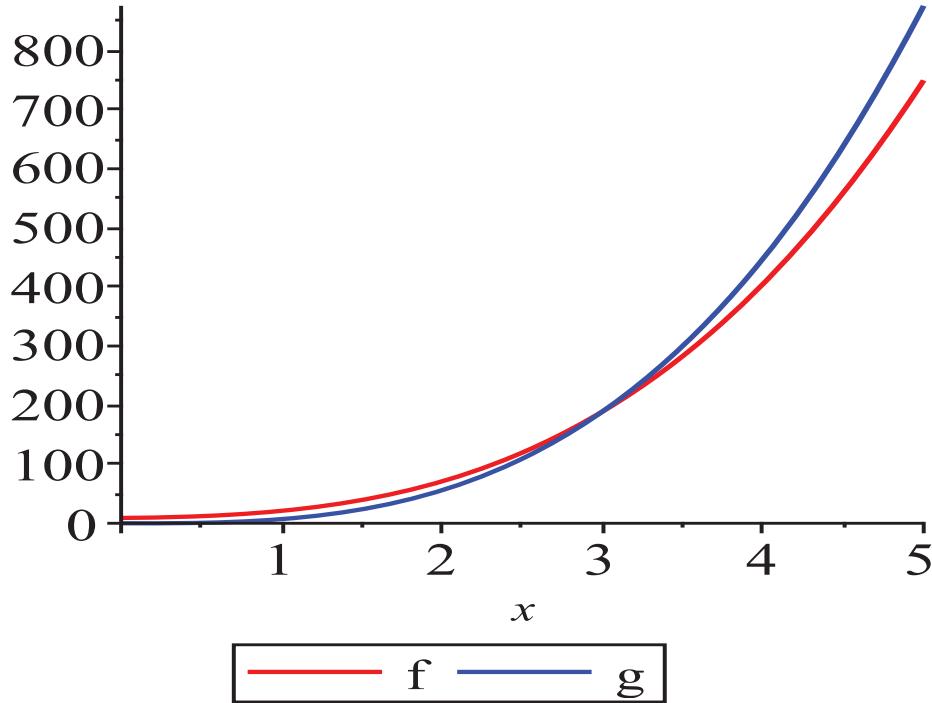
```
> plot([f1, 4 · g1], x = 0 .. 5, color = [red, blue], legend = ["f", "g"])
```



```
> plot([f1, 6 · g1], x = 0 .. 5, color = [red, blue], legend = ["f", "g"])
```

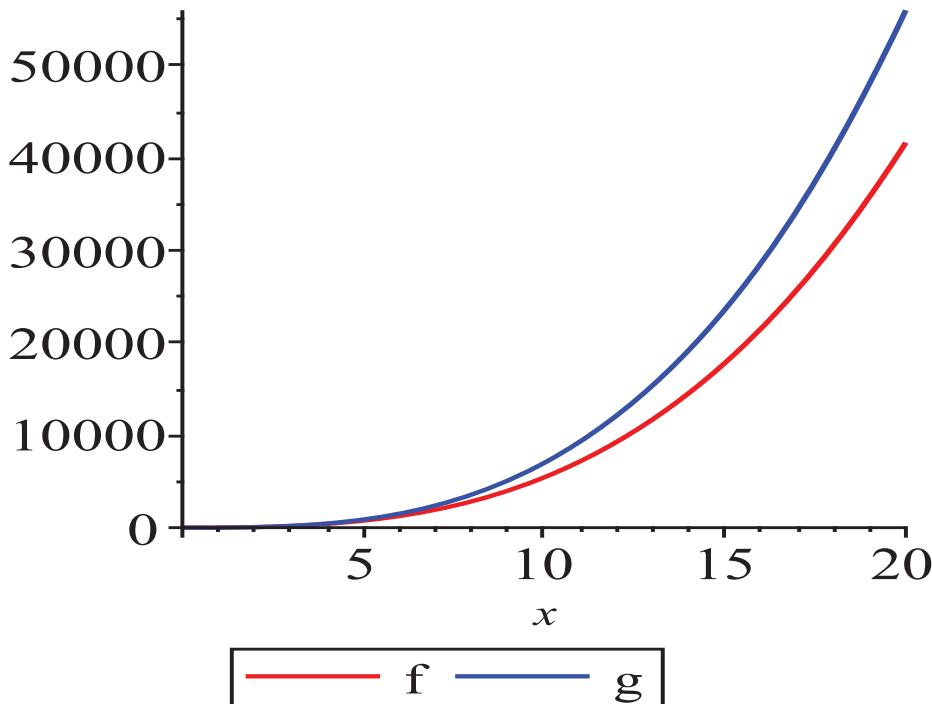


```
> plot([f1, 7 · g1], x = 0 .. 5, color = [red, blue], legend = ["f", "g"])
```



By expanding the range of x -values, we can produce a graph that provides fairly convincing evidence that $C = 7$ and $k = 3$ witness for the assertion that $f(x)$ is $O(x^3)$.

```
> plot([f1, 7 · g1], x = 0 .. 20, color = [red, blue], legend = ["f", "g"])
```



It is important to note that the graph above is not *proof* that $5x^3 + 4x^2 + 3x + 9$ is $O(x^3)$. A formal proof must follow the model provided by the examples in the text.

A Second Example

As a second example, consider $f(x) = 3x^5 + x^3 \ln(x^2 + 2x + 1)$. We claim that this is $O(x^n)$ for some value of n . We need to first determine the smallest value of n and then find witnesses for C and k . We assign a name for the formula for $f(x)$.

$$\begin{aligned} > f2 := 3x^5 + x^3 \log(x^2 + 2x + 1) \\ & f2 := 3x^5 + x^3 \ln(x^2 + 2x + 1) \end{aligned} \tag{3.6}$$

We could proceed as in the previous example and display a selection of graphs comparing $f2$ and x^n for different exponents in order to find a likely choice of n and then explore the coefficients. However, Maple's Exploration Assistant allows for a more interactive approach. There are two ways to create interactive elements in Maple. One is to enter a command that produces the output you wish to explore, but in terms of variables that have not been specified. For example:

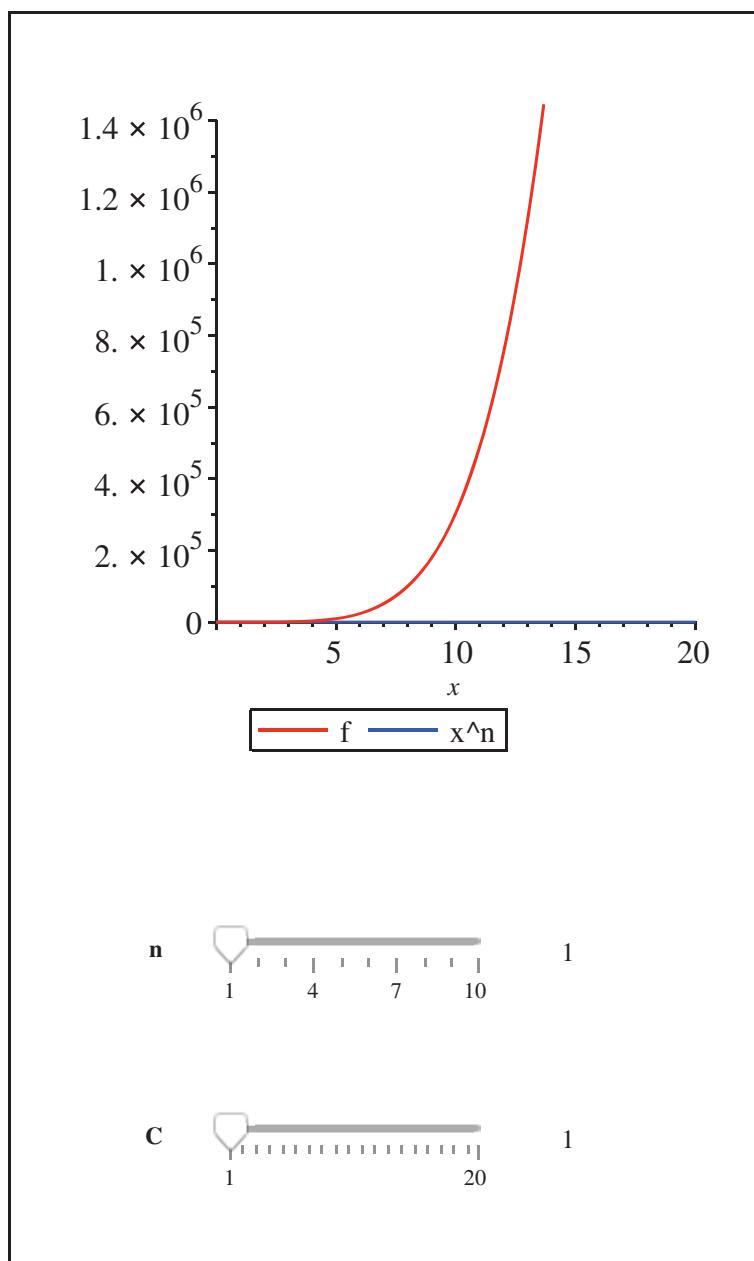
$$> plot([f2, Cx^n], x = 0 .. 20, color = [red, blue], legend = ["f", "g"])$$

Executing this command would produce an error (in the Maple version of this manual, it has been set to be not executable). However, if you right-click anywhere on the line above, one of the options in the menu that pops up should be “Explore.” Likewise, if you have the Context Panel open and the cursor is anywhere in the code for the plot, one of the options will be “Explore.” Clicking on “Explore” in either menu will cause a dialog window to open allowing you to specify certain options. Clicking on the “Explore” button at the bottom of that window will result in an interactive application with which you can explore the effects of the parameters.

The other approach is to type the commands to create the interactive exploration. This is the approach we will take in this manual, since it is more explicit and is typically easier to replicate. The name of the command used to create interactive explorations is **Explore**. The first argument is always a function call in terms of one or more names that will be altered by sliders or other controls, such as the call to **plot** above. If that is the only argument, then the result of executing it is to open the dialog window described in the previous paragraph.

If you don't want to use the dialog window, the simplest way to use **Explore** is to follow the command you wish to explore with equations setting the names of the parameters to ranges specifying possible values. For example, to allow the exponent to be integers from 1 to 10 and the constant between 1 and 20, we enter the following.

```
> Explore(plot([f2,Cxn],x=0..20,color=[red,blue],
legend=[“f”, “xn”]),n=1..10,C=1..20)
```

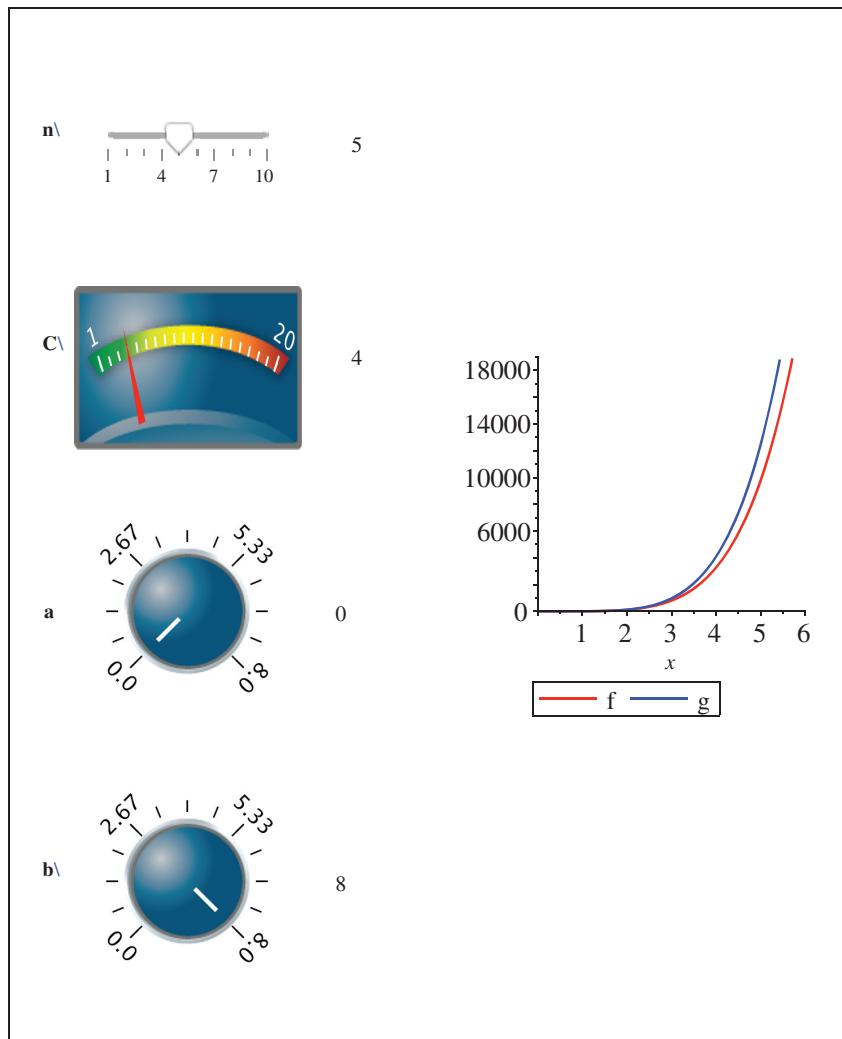


This is sufficient to find n for which f_2 is $O(x^n)$. Simply start dragging the slider for n . For $n \leq 3$, x^n grows so slowly relative to $f(x)$ that it is virtually invisible on the graph. For $n = 4$, the graph of x^4 is visible, but clearly growing at a much slower rate than $f(x)$. For $n = 5$, however, there is some hope that multiplying by a constant may enable x^5 to catch up to $f(x)$. If you then start changing the value of C , you see that $C = 4$ appears sufficient for $4x^5$ to dominate $f(x)$.

To find a value for k , we will add parameters, a and b , for the endpoints of the range of x -values being plotted, which will effectively allow us to dynamically zoom in. Note that all that is required for a parameter to be allowed to take on noninteger values within its range is to give at least one of the endpoints as a noninteger. Keep in mind that for Maple, the number **8.** (with a decimal point) is a noninteger.

We will take this opportunity to also explore some of the options associated to an **Explore**. We first provide the command and then explain the options.

```
> Explore(plot([f2, Cxn], x = a .. b, color = [red, blue], legend = ["f", "g"]),
parameters = [n = 1 .. 10, [C = 1 .. 20, controller = meter],
[a = 0 .. 8.0, controller = dial], [b = 0 .. 8.0, controller = dial]],
initialvalues = [n = 5, C = 4, b = 8], placement = left,
widthmode = pixels, width = 500)
```



First, rather than just providing the parameters and ranges, we use the **parameters** keyword in order to make some choices about how we interact with the parameters. The **parameters** keyword is set to a list. Each element in the list is the specification for a parameter. That specification can be a simple “parameter equals range,” or, in place of the range, a list. In the above, that is what was done for the parameter n , resulting in a slider. The other parameters are each specified by a sublist within the main **parameters** list. The first entry in each sublist is an equation identifying a parameter with a range or list of values. In addition, we have specified the **controller** for the parameter. The possible controllers are **slider**, **volumegaage**, **dial**, **meter**, **rotarygauge**, **checkbox**, **combobox**, **listbox**, and **textarea**. Note that some of these require the parameter be specified as having a list of values, rather than a range. For example, if you wish to have the exponent displayed as a combobox, also known as a drop-down menu, it would need to be specified as

```
[n=[\$1..10], controller=combobox]
```

The $\$$ creates the sequence of integers and the brackets form the list of those values.

Second, after the list of lists specifying the parameters, we use the **initialvalues** option to specify values to which the parameters should be initially set. This option is set to a list of equations specifying the initial values, with any omitted parameters defaulting to the lowest value in their range or the first value in their list. This was an important option to use in this example; without it, both a and b would have started at 0, making the plot empty.

Third, we used the **placement** option to place the controllers to the left of the plot. Possible values are **left**, **right**, **bottom**, and **top**. Note that **placement** can also be used within the parameter specifications to position the controllers for different parameters separately.

Finally, the **width** option is used to specify the width of the exploration. The value is always a non-negative real number, but the interpretation of the value depends on the **widthmode** option, which can be set to either **percentage** to specify the width as a percentage of the worksheet’s width or **pixels** to specify an absolute width.

Using the interactive application and by tightening the range of x values, you can see that $k = 1.5$ appears to be sufficient.

3.3 Complexity of Algorithms

Section 3.3 of the textbook emphasizes worst-case complexity and shows you how it can be deductively determined. The textbook also mentions average-case complexity and shows how to compute the average-case complexity of the linear search algorithm (Example 4).

Average-case complexity is typically more difficult to analyze deductively, but is still very important. From a practical standpoint, average-case complexity can help differentiate algorithms whose worst-case complexities are of the same order. Moreover, algorithms that have very poor worst-case complexity may have reasonable average-case complexity, provided that the “bad” inputs that produce the worst case are rare.

While average-case complexity is difficult to analyze, average-case performance can be computed fairly directly. Recall from the introduction to this chapter that we distinguish complexity

of an algorithm from performance of a procedure. In this section, we will see how Maple can be used to analyze the average-case performance of procedures experimentally. We will use the **bubblesort** procedure developed in Section 3.1 of this manual as an example. Our goal will be to produce a graph displaying the empirically determined average-case time performance of the procedure.

It is essential to note that performance depends on many factors besides the algorithm, including the programming language used. In particular, different languages are designed with different purposes, leading to different efficiencies. This means, for example, that in comparing two algorithms, it is possible that one will outperform the other when implemented in one language and the reverse could be true if they are implemented in a different language.

We begin by explaining the standard approach to timing procedures in Maple. The **time** command returns the total CPU time that has been used by Maple since the start of the Maple session. By computing this value before running a procedure and again after the procedure has been completed, the difference in the values will be a very good estimate of the time taken to run the procedure.

Here is an example of the use of the **time** command.

```
> st := time() :  
FindMax([\$1..100 000]) :  
time () - st  
0.075
```

(3.7)

The name **st** stands for “start time.” Recall that **[\$1..100000]** produces the list consisting of the integers 1 through 100 000. Note that colons are used to suppress all the output except the final command that reports the elapsed time. This is important because if the procedure produces output, the display of the output takes time. We do not want the time it takes for Maple to display the results included as part of the time performance of the procedure.

Average Input

By average-case performance, we mean the average performance of a procedure on a random input selected from all possible inputs of the given size. The particulars of how the random input is selected is a necessary component in the analysis. It is natural to assume that each possible input will appear with the same likelihood as every other, but it is important to recognize that this may not always be the case. It may be that, in the circumstances under which the algorithm is intended to be used, some inputs may appear with relatively higher or lower frequency.

In our test of **bubblesort**, we have no particular application in mind and so will assume that all inputs are equally likely. In order to generate a random input, we will use the **randperm** command from the **combinat** package. Applied to a positive integer n , **randperm** will produce a list of the integers from 1 to n in random order. (We describe the commands of the **combinat** package in more detail in Chapter 6.)

```
> combinat[randperm] (20)  
[13, 4, 5, 20, 17, 7, 16, 10, 15, 14, 6, 3, 12, 8, 2, 11, 9, 19, 1, 18]
```

(3.8)

We can apply the **bubblesort** algorithm directly to the result and time how long it takes to execute.

```
> st := time() :  
    bubblesort(combinat[randperm](100)) :  
    time() - st  
    0.022
```

(3.9)

Since we are after average-case performance, we will need to execute **bubblesort** on some number, say 100, of different random inputs and average the time taken by each execution. To collect the 100 times, we can use a for loop to build a list in which the times are stored.

```
> timings := [] :  
for i to 100 do  
    st := time();  
    bubblesort(combinat[randperm](100));  
    et := time() - st;  
    timings := [op(timings), et]  
end do :  
  
> timings  
[0.078, 0.013, 0.014, 0.015, 0.072, 0.016, 0.016, 0.014, 0.070, 0.014,  
 0.014, 0.015, 0.070, 0.015, 0.015, 0.014, 0.070, 0.014, 0.012,  
 0.015, 0.070, 0.016, 0.012, 0.014, 0.070, 0.016, 0.014, 0.013,  
 0.066, 0.015, 0.014, 0.014, 0.072, 0.016, 0.015, 0.013, 0.069,  
 0.016, 0.015, 0.019, 0.076, 0.016, 0.014, 0.014, 0.015, 0.070,  
 0.014, 0.014, 0.014, 0.070, 0.014, 0.014, 0.014, 0.069, 0.013,  
 0.013, 0.014, 0.070, 0.014, 0.013, 0.014, 0.066, 0.012, 0.014,  
 0.013, 0.068, 0.016, 0.015, 0.014, 0.071, 0.013, 0.014, 0.014,  
 0.069, 0.014, 0.013, 0.015, 0.067, 0.014, 0.013, 0.013, 0.014,  
 0.066, 0.015, 0.012, 0.015, 0.069, 0.012, 0.013, 0.015, 0.072,  
 0.013, 0.013, 0.015, 0.069, 0.015, 0.013, 0.016, 0.067, 0.014]
```

(3.10)

(Depending on the speed of your computer, you may need to increase or decrease the size of the input list.)

To average the times, we apply the **Mean** command from the **Statistics** package to the list of values.

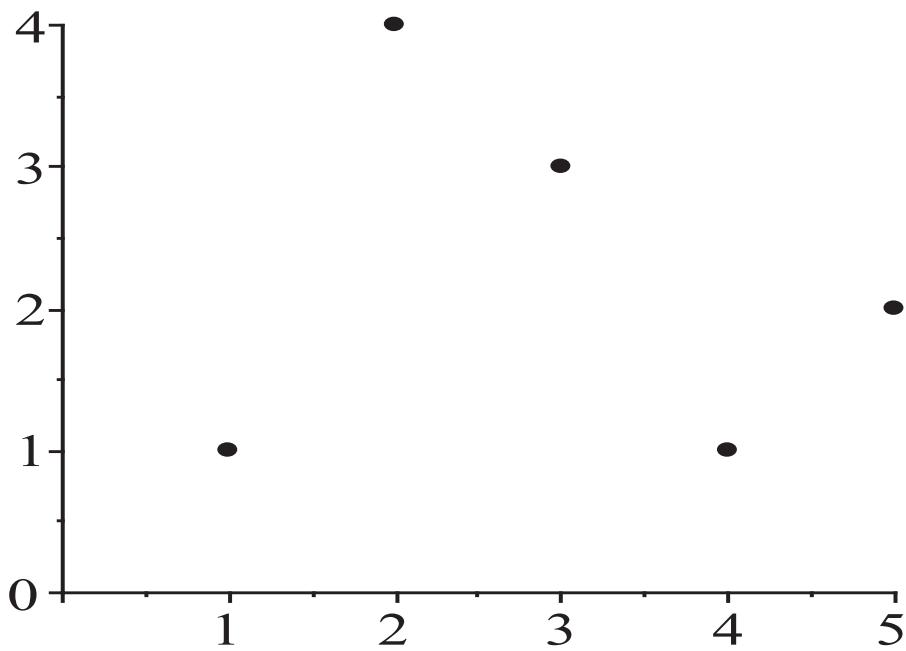
```
> Statistics[Mean](timings)  
0.0281000000000000
```

(3.11)

Graphing the Empirically Calculated Average-Case Complexity

To graph the average time data, we use the **plot** command. We saw above how to use **plot** to graph functions defined by formulas. We can also use **plot** to draw graphs by giving it specific values for the x and y coordinates. The coordinates must be given as two lists—the first list consisting of the x values and the second list the corresponding y values.

```
> plot([1, 2, 3, 4, 5], [1, 4, 3, 1, 2], view = [0 .. 5, 0 .. 4], style = point,  
      symbol = solidcircle, symbolsize = 15)
```



We have already seen the **view** option. The option **style=point** causes Maple to display only the points specified in the lists. Omitting **style=point** results in a graph in which the data are connected by straight line segments. The **symbol=solidcircle** determines the symbol used to plot the points. Other options include **asterisk**, **box**, **solidbox**, and **circle**. Finally, **symbolsize=15** increases the size of the dots to 15 points.

We will now write a procedure that produces the two lists required by **plot**. This procedure will accept no arguments, but will compute the average, over 100 trials, of the time taken to execute **bubblesort** on randomly generated lists of size 10, 20, 30, 40, and 50.

```

1  getTimes := proc()
2    local sizes, s, avgTimes, times, trials, data, st, t, i;
3    sizes := [seq(10*i, i=1..5)];
4    avgTimes := [];
5    for s in sizes do
6      times := [];
7      for trials from 1 to 100 do
8        data := combinat[randperm](s);
9        st := time();
10       bubblesort(data);
11       t := time() - st;
12       times := [op(times), t];
13    end do;
14    avgTimes := [op(avgTimes), Statistics[Mean](times)];
15  end do;
16  return [sizes, avgTimes];
17 end proc;
```

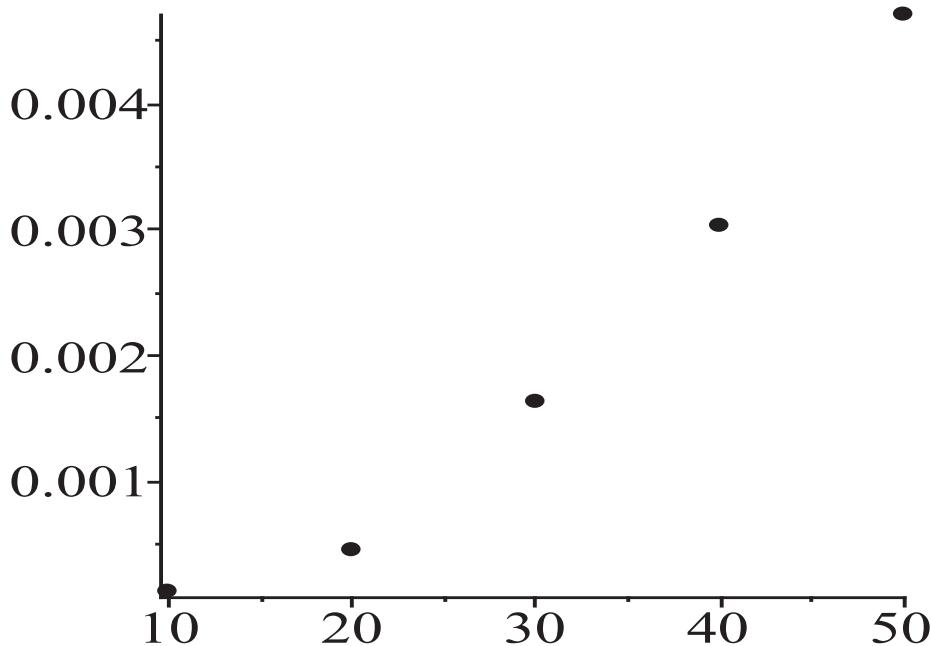
The procedure begins by generating the list **sizes**. These values indicate the sizes of the lists to which **bubblesort** will be applied. The **avgTimes** list will hold the averages of the times for the

corresponding input size. For each possible size, the procedure empties the **times** list and then runs 100 trials in which a randomly ordered list of the appropriate size is generated and the time it takes for **bubblesort** to sort the list is recorded and added to the **times** list. After the 100 trials are complete, the average of the times is computed and added to the **avgTimes** list. The procedure returns a two-element list consisting of the **sizes** list and the **avgTimes** list. Note that we moved the call to **randperm** to before the assignment of **st**, rather than calling **randperm** within the argument to **bubblesort**. This is so that the time it takes Maple to generate the random permutation is not counted as part of the time it takes **bubblesort** to execute.

We now apply the **getTimes** procedure and use its output to create a graph.

```
> getTimesOut := getTimes()
getTimesOut := [[10, 20, 30, 40, 50], [0.0001200000000000000,
0.000450000000000000, 0.00163000000000000,
0.00303000000000000, 0.00471000000000000]]
(3.12)

> plot(getTimesOut[1], getTimesOut[2], style = point,
symbol = solidcircle, symbolsize = 15)
```



From the shape of the graph, it appears that the average-case performance of **bubblesort** is polynomial. This suggests that the complexity of the algorithm is also polynomial. Of course, a proof of that fact would require an analysis of the kind given in Example 4 of Section 3.3.

The reader can modify the definition of the **sizes** list in order to produce finer detail (by decreasing the step between the input sizes) and to obtain data for larger input lists (by increasing the maximum value of **i**). However, note that for a list of length greater than 100, attempting to modify an element will produce an error. This is because lists in Maple are immutable meaning that changing a single element actually creates a new modified copy of the list, which can quickly become memory intensive. Therefore, the maximum possible list size that **bubblesort**, as currently written, can

handle is 100. For larger sized inputs, you would need to rewrite **bubblesort** and **interchange** to uses **Arrays** instead of lists.

We conclude with a caveat. The empirical testing we have done in this section is an example of a way to get an idea of the average-case performance of a procedure. It can be used to compare two or more algorithms with each other and can indicate major differences in worst-case and average-case complexity (for instance, in an algorithm with exponential worst-case complexity and polynomial average-case complexity). However, beyond generalities, the implementation of the algorithm, the computer running it, the computer language it is written in, and a host of other factors can play a sufficiently significant role that this approach is generally not helpful for making finer distinctions (e.g., between quadratic and cubic complexity).

The reader should refer to the solution of Computer Project 9 for a method of analyzing average case complexity that modifies the procedure in order to count the number of operations used with the input values.

Solutions to Computer Projects and Computations and Explorations

Computer Project 7

Given two strings of characters use the naive string matching algorithm to determine whether the shorter string occurs in the longer string.

Solution: We begin, as usual, by following the pseudocode presented in the main text as Algorithm 6. As before, we will need to adjust basic syntax for the loops and other elements. However, the pseudocode indicates that the input should include two sequences of characters. That makes for a clunky user experience. We would much rather be able to enter

stringMatch[7, 3, “eceyeye”, “eye”]

rather than

stringMatch[7, 3, “e”, “c”, “e”, “y”, “e”, “y”, “e”, “e”, “y”, “e”]

We will implement the more user-friendly version by applying the **seq** command to transform a string into the sequence of its characters. We then wrap the sequence in brackets to produce the list of characters.

> [seq(“eceyeye”)]
[“e”, “c”, “e”, “y”, “e”, “y”, “e”] (3.13)

Here is our first attempt.

```
1  stringMatch1 := proc (n : :integer, m : :integer, t : :string, p : :string)
2      local T, P, s, j;
3      T := [seq(t)];
4      P := [seq(p)];
5      for s from 0 to n-m do
6          j := 1;
7          while j <= m and T[s+j] = P[j] do
8              j := j + 1;
```

```

 9   end do;
10  if j > m then
11    print(cat(s, " is a valid shift "));
12  end if;
13 end do;
14 end proc:
```

Note that when **print** is applied to multiple arguments, it will print the comma-separated sequence. We apply **cat**, for concatenate, to merge the sequence into a single string.

Having written the function, we test it.

```
> stringMatch1(7,3,"eceeeye","eye")
2 is a valid shift
4 is a valid shift
```

(3.14)

It is often a good idea, having written a working procedure, to reflect on it and think about whether it could be made simpler. In this case, you might start wondering why the lengths of the text and pattern are included as arguments to the function. Remember that the syntax for executing this function suggested by the pseudocode is

stringMatch[7,3,"e","c","e","y","e","y","e","e","y","e"]

With the text and pattern given as individual characters, the lengths are necessary in order to determine where the text stops and the pattern begins. In our implementation, the text and pattern are separate arguments, so the lengths can be computed by the function rather than entered by the user. Our final version of the naive string matcher is below.

```

1 stringMatch := proc(t :: string, p :: string)
2   local T, n, P, m, s, j;
3   T := [seq(t)]; n := numelems(t);
4   P := [seq(p)]; m := numelems(p);
5   for s from 0 to n-m do
6     j := 1;
7     while j <= m and T[s+j] = P[j] do
8       j := j + 1;
9     end do;
10    if j > m then
11      print(cat(s, " is a valid shift "));
12    end if;
13  end do;
14 end proc:
```

```
> stringMatch("eceeeye","eye")
2 is a valid shift
4 is a valid shift
```

(3.15)

Maple, of course, includes extensive support for string manipulation, including a command that performs the same function as our naive string matcher called **SearchAll**, which is part of the

StringTools package. The arguments for this function are in the reverse order as ours, with the pattern first. The result is a sequence of the locations of the first characters of the matches or, if there is no match, the command returns 0.

> *StringTools[SearchAll]("eye", "eceyeye")*

3, 5

(3.16)

The difference in output is that the built-in function, rather than indicating valid shifts, returns the starting positions of where the pattern occurs in the text.

Computer Project 10

Given an ordered list of n integers and an integer x in the list, find the number of comparisons used to determine the position of x in the list using a linear search and using a binary search.

Solution: There is no loss of generality to assume that the list of n integers is the list of integers from 1 to n .

For the linear search algorithm provided as Algorithm 2 in Section 3.1 of the text, each step in the search requires 2 comparisons, one tests whether the end of the list has been reached and one tests whether the current element is the element being searched for. These are both contained in the Boolean expression that controls the while loop. A final comparison is used after the while loop is completed to determine whether the element was found or not. In the list of n integers 1 through n , the integer x is therefore found after $2x + 1$ comparisons.

To determine the number of comparisons needed to find x via the binary search algorithm, we modify the procedure we wrote in Section 3.1 of this manual to count comparisons. For reference, here is the original **binarysearch** procedure.

```
1  binarysearch := proc (x::integer, A::list(integer))
2      local n, i, j, m, location;
3      n := nops(A);
4      i := 1;
5      j := n;
6      while i < j do
7          m := floor((i+j)/2);
8          if x > A[m] then
9              i := m + 1;
10         else
11             j := m;
12         end if;
13     end do;
14     if x = A[i] then
15         location := i;
16     else
17         location := 0;
18     end if;
19     return location;
20 end proc;
```

We will modify this procedure to count comparisons. Each time through the while loop accounts for two comparisons, the $i < j$ comparison that controls the loop and the $a_m < x$ comparison in the if statement. Thus, we add a line of code to increment the comparison count by two at the start of the while loop. In addition, we need to add one to the comparison count after the end of the loop to account for the comparison that terminates the loop. One final comparison is done to determine if the search has succeeded or not.

The modified procedure returns the count of comparisons instead of the position of the element.

```

1  binarysearchC := proc (x::integer, A::list(integer) )
2      local n, i, j, m, location, count;
3      count := 0;
4      n := nops (A) ;
5      i := 1;
6      j := n;
7      while i < j do
8          count := count + 2;
9          m := floor ((i+j)/2);
10         if x > A[m] then
11             i := m + 1;
12         else
13             j := m;
14         end if;
15     end do;
16     count := count + 1;
17     if x = A[i] then
18         location := i;
19     else
20         location := 0;
21     end if;
22     count := count + 1;
23     return count;
24 end proc:
```

For example, to find 15 in the list from 1 to 20, it takes

> *binarysearchC*(15, [\$1 .. 20])
10 (3.17)

comparisons.

We can use the information above to compare the average number of comparisons required in a list of n elements. We need to determine the number of comparisons needed to find each value from 1 to n in the list from 1 to n and average these numbers of comparisons. For the linear search, we know that it takes $2x + 1$ comparisons, so the average can be found from

$$\frac{\sum_{x=1}^n 2x + 1}{n} .$$

We can use Maple's symbolic summation capabilities (discussed in Section 2.4 of this manual).

$$\begin{aligned} > \frac{\sum(2 \cdot x + 1, x = 1 .. n)}{n} \\ & \frac{(n + 1)^2 - 1}{n} \end{aligned} \tag{3.18}$$

$$\begin{aligned} > \text{simplify}(\%) \\ & n + 2 \end{aligned} \tag{3.19}$$

(The **simplify** command forces Maple to simplify expressions.)

For the binary search procedure, we can find the average by applying our procedure above to each integer in turn and taking the average. The following procedure will produce the average number of comparisons required for a given value of n .

```

1 binaryAvg := proc(n::posint)
2   local comps, L, x;
3   comps := [];
4   L := [$1..n];
5   for x from 1 to n do
6     comps := [op(comps), binarysearchC(x, L)];
7   end do;
8   return Statistics[Mean](comps);
9 end proc;
```

For example, in the list from 1 to 20, it requires an average of $20 + 2 = 22$ comparisons using the linear search, and an average of 10.8 comparisons using the binary search.

$$\begin{aligned} > \text{binaryAvg}(20) \\ & 10.80000000000000 \end{aligned} \tag{3.20}$$

Next, we graph the average number of comparisons as n ranges from 1 to 100. To do this, we first create the necessary inputs to the **plot** command. Recall from Section 3.2 of this manual that plot requires a list of the x values and a list of the y values. The x values will be the values of n .

> *nList* := [\$1..100] :

For the linear search algorithm, the y values are $n + 2$.

> *linearAverages* := [seq($n + 2$, $n = 1 .. 100$)]

For the binary search algorithm, the y values are obtained from the procedure **binaryAvg**.

> *binaryAverages* := [seq(*binaryAvg*(n), $n = 1 .. 100$)] :

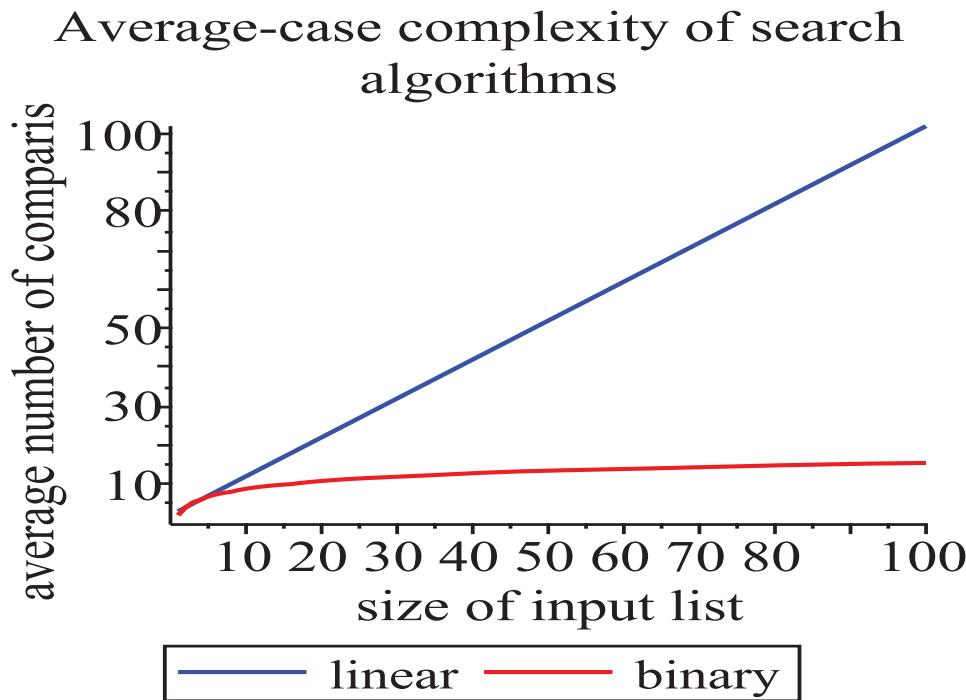
We want to overlay the graphs for the two algorithms. To do this, we assign the results of the **plot** command to a name instead of displaying them. We use different colors for the plots and ensure that the views are the same.

```
> linearPlot := itplot(nList, linearAverages, color = blue,
view = [1..100, 1..102], legend = "linear"):

> binaryPlot := itplot(nList, binaryAverages, color = red,
view = [1..100, 1..102], legend = "binary"):
```

To overlay the two plots, we use the **display** command, which is part of the **plots** package. In its simplest form, this command accepts a list of plot structures (such as what you obtain by applying the **plot** command) and overlays the plots. It can also accept most of the options that are available for **plot**. Below, we demonstrate the **display** command along with the use of several options to provide an informative graph.

```
> plots[display] ([linearPlot, binaryPlot,
title = "Average-case complexity of search algorithms",
labels = ["size of input list", "average number of comparisons"],
labeldirections = [horizontal, vertical])
```



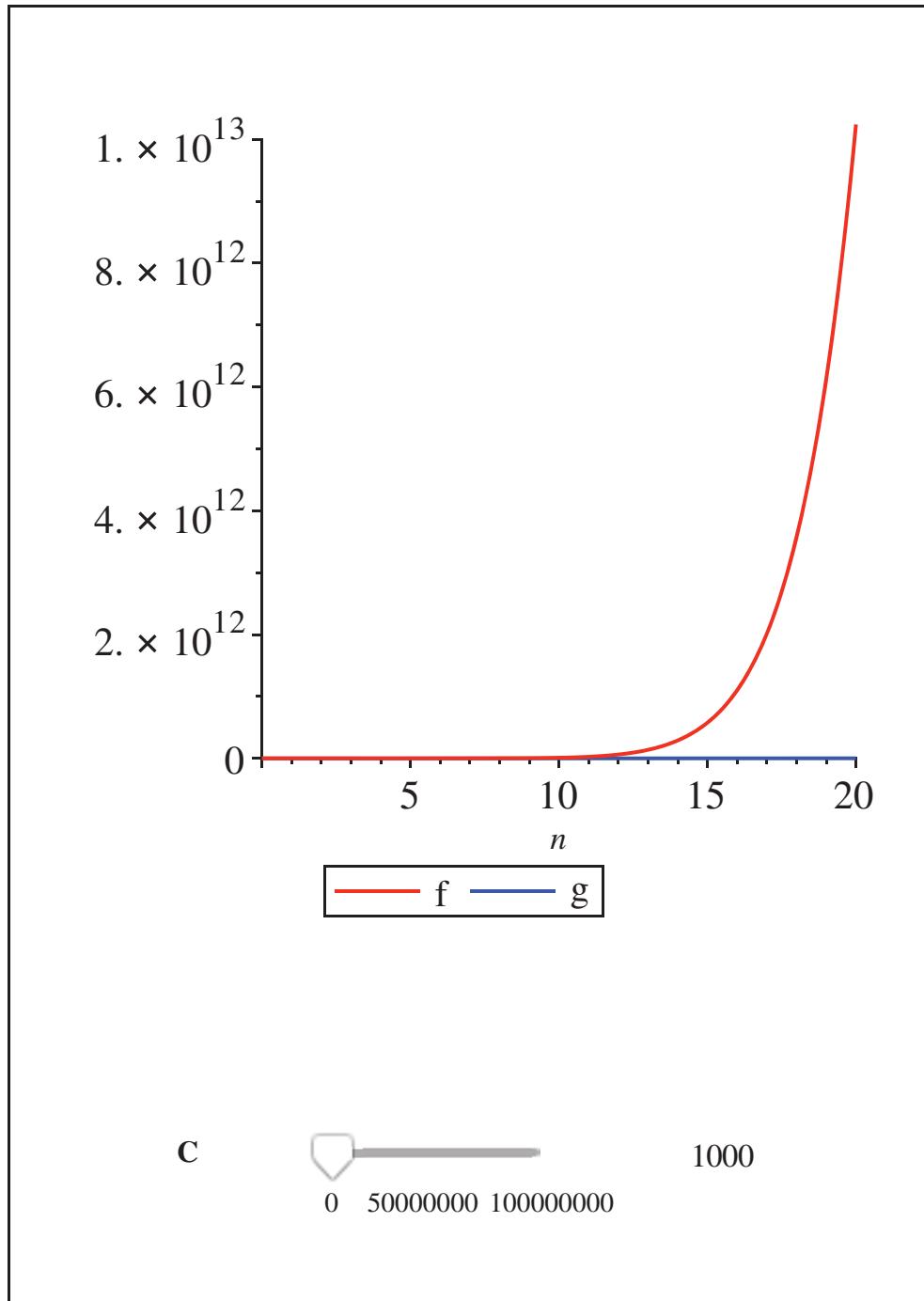
Computations and Explorations 1

We know that n^b is $O(d^n)$ when b and d are positive numbers with $2 \leq d$. Give values of the constants C and k such that $n^b \leq Cd^n$ whenever $k < n$ for each of these sets of values: .

Solution: For $b = 10$ and $d = 2$, we need to compare the functions $f(n) = n^{10}$ to $g(n) = 2^n$. Following the approach we took in Section 3.2, we will use Explore to graph the functions while dynamically changing the value of C . Note that the values of C that will suffice with $k < 20$ are

extremely large. When exploring other values of b and d , you may have to modify the maximum values for C . It is also possible to find smaller values of C by increasing the horizontal extent of the graph.

```
> Explore(plot([n^10, C · 2^n], n = 0 .. 20, view = [0 .. 20, 0 .. 10^13],
color = [red, blue], legend = ["f", "g"]), C = 0 .. 10^8,
initialvalues = [C = 1000])
```



Once you find a potential value for C , you can confirm that it is correct and also find an exact value for k . For example, it appears that with $g(n) = C2^n$ dominates $f(n) = n^{10}$ for $n > 20$. We can

confirm this by having Maple solve the inequality $n^{10} \leq 10^7 \cdot 2^n$ with the **solve** command. The **solve** command applied to an equation or inequality in one variable will find the solution to the equation or inequality. We apply **evalf** to force a floating-point result.

```
> evalf(solve(n^10 ≤ 10^7 · 2^n))
[-3.840511598, 10.07828296], [19.87621431, ∞) (3.21)
```

This indicates that $C = 10^7$ and $k = 19.9$ witness for n^{10} being $O(2^n)$. Note that smaller values of C will work provided the value of k is made sufficiently large.

```
> evalf(solve(n^10 ≤ 10^5 · 2^n))
[-2.634476634, 4.243747408], [34.45805187, ∞) (3.22)
```

```
> evalf(solve(n^10 ≤ 10^3 · 2^n))
[-1.765447237, 2.347895073], [44.93042315, ∞) (3.23)
```

```
> evalf(solve(n^10 ≤ 1 · 2^n))
[-0.9371092012, 1.077550150], [58.77010593, ∞) (3.24)
```

Exercises

Exercise 1. Write step-by-step instructions, then pseudocode, and then implement in Maple an algorithm to determine the k largest integers in a list of integers.

Exercise 2. Implement the linear search presented as Algorithm 2 in Section 3.1 of the text.

Exercise 3. Implement the insertion sort presented as Algorithm 5 in Section 3.1 of the text.

Exercise 4. Implement the cashier's algorithm presented as Algorithm 7 in Section 3.1 of the text.

Exercise 5. Implement the algorithm for scheduling talks presented as Algorithm 8 in Section 3.1 of the text.

Exercise 6. Implement the brute-force algorithm for finding the closest pair of points as presented in Algorithm 3 in Section 3.3 of the text.

Exercise 7. Modify the **bubblesort** procedure so that it terminates when no more interchanges are necessary. (See Exercise 39 from Section 3.1.)

Exercise 8. Implement the selection sort algorithm in Maple. (Refer to the preamble to Exercise 43 in Section 3.1 for information on selection sort.)

Exercise 9. Implement the binary insertion sort in Maple. (Refer to the preamble to Exercise 49 in Section 3.1 for information on the binary insertion sort.)

Exercise 10. Implement the deferred acceptance algorithm in Maple. (Refer to the preamble to Exercise 65 in Section 3.1 for information on the deferred acceptance algorithm.)

Exercise 11. Implement the Boyer–Moore majority vote algorithm in Maple. (Refer to the preamble to Exercise 68 in Section 3.1 for information on the majority vote algorithm.)

Exercise 12. Following the solution to Computations and Explorations 1, use Maple to determine values for C and k that witness for the fact that $f(x)$ is $O(g(x))$ for each of the pairs of functions given below.

- a) $f(x) = 7 \ln(3x^2 - 2x + 5); g(x) = x.$
- b) $f(x) = \frac{x^4}{x^2 - 4x - 4}; g(x) = x^2.$
- c) $f(x) = [x] \lceil x \rceil; g(x) = x^2.$
- d) $f(x) = n \ln(n); g(x) = \ln(n!).$

Exercise 13. Using the approach described in Section 3.3 of this manual, compare the average-case performance of the **bubblesort** procedure presented in Section 3.1 to Maple's **sort** command.

Exercise 14. Using the solution to Computer Project 10 as a model, compare the average-case complexity (as measured by the number of comparisons) of the **bubblesort** procedure with the modified procedure that you implemented as Exercise 7.

Exercise 15. Using the solution to Computer Project 10 as a model, compare the average-case complexity (as measured by the number of comparisons) of the **bubblesort** procedure with the other sort procedures you wrote (e.g., insertion sort, selection sort, or binary selection sort).

Exercise 16. Implement the two algorithms suggested by Exercise 31 of Section 3.1 for determining whether two strings are anagrams. Then, using the approach described in Section 3.3 of this manual, compare the average-case performance of the two algorithms. Compare the results of your performance analysis with the big-O estimates you found in Exercise 38 of Section 3.3.

Exercise 17. Implement the two algorithms suggested by Exercise 32 of Section 3.1 for finding the closest of n real numbers. Then, using the approach described in Section 3.3 of this manual, compare the average-case performance of the two algorithms. Compare the results of your performance analysis with the big-O estimates you found in Exercise 39 of Section 3.3.

4 Number Theory and Cryptography

Introduction

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

4.1 Divisibility and Modular Arithmetic

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

Quotient, Remainder, and Divisibility

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

```
> iquo(99, 13)  
7  
(4.1)
```

```
> irem(99, 13)  
8  
(4.2)
```

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

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

```
> iquo(99, 13, 'remainderName')  
7  
(4.3)
```

```
> remainderName  
8  
(4.4)
```

Checking Divisibility

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

```
> irem(132, 3)  
0
```

(4.5)

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

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

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

(4.6)

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

(4.7)

The mod Operator

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

```
> 99 mod 13  
8
```

(4.8)

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

```
> -27 mod 5  
3
```

(4.9)

However, there may be times when it is more useful to allow negative values. For example, consider the following question: “It is now 11:00 AM. What time will it be 142 hours from now?” If we compute $124 \text{ mod } 24$,

```
> 142 mod 24  
22
```

(4.10)

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

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

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

(4.11)

The **modp** command returns the “positive” representation.

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

(4.12)

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

Congruences

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

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

(4.13)

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

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

(4.14)

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

```
> evalb(modp(428 = 530, 17))
true
```

(4.15)

```
> evalb(modp(289 = 311, 17))
false
```

(4.16)

Solving Congruences

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

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

(4.17)

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

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

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

```
> msolve (3^i = 4, 7)
```

```
{i = 4 + 6_Z1}
```

(4.18)

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

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

```
true
```

(4.19)

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

```
> msolve (3^i = 4, C, 7)
```

```
{i = 4 + 6 C}
```

(4.20)

Arithmetic Modulo m

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

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

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

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

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

```
&+ := (a, b) ↪ modp(a + b, 11)
```

(4.21)

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

```
&* := (a, b) ↪ modp(b a, 11)
```

(4.22)

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

```
> 7 &+ (9 &* 2)
3
```

(4.23)

Addition and Multiplication Tables

We conclude Section 4.1 by producing addition and multiplication tables.

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

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

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

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

```

(4.24)

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

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

```

(4.25)

4.2 Integer Representations and Algorithms

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

Base Conversion

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

```
> convert([1, 2, 3], set)  
{1, 2, 3} (4.26)
```

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

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

```
> convert(12345, octal)  
30071 (4.27)
```

```
> convert(177130, hex)  
'2B3EA' (4.28)
```

```
> convert(241, binary)  
1111 0001 (4.29)
```

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

```
> convert(30071, decimal, octal)  
12345 (4.30)
```

```
> convert(1111 0001, decimal, binary)  
241 (4.31)
```

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

```
> convert("2B3EA", decimal, hex)  
177130 (4.32)
```

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

```
> convert(1111 0001, decimal, 2)  
241 (4.33)
```

```
> convert(12, decimal, 3)  
5 (4.34)
```

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

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

```
> convert(194, base, 5)
[4, 3, 2, 1] (4.35)
```

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

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

```
> convert(12345, base, 8)
[1, 7, 0, 0, 3] (4.36)
```

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

```
> convert(12345, base, 10)
[5, 4, 3, 2, 1] (4.37)
```

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

```
> convert([3, 2, 1], base, 5, 3)
[2, 0, 1, 1] (4.38)
```

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

Binary and the Bits Package

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

Split and Join

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

```
> with(Bits):
> Split(241)
[1, 0, 0, 0, 1, 1, 1, 1] (4.39)
```

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

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

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

(4.40)

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

```
> Join([1, 0, 0, 0, 1, 1, 1])
241
```

(4.41)

The GetBits Command

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

```
> GetBits(241, 0 .. -1)
[1, 0, 0, 0, 1, 1, 1]
```

(4.42)

```
> GetBits(241, 0 .. 3)
[1, 0, 0, 0]
```

(4.43)

```
> GetBits(241, 0, 4 .. -1)
[1, 1, 1, 1, 1]
```

(4.44)

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

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

```
> GetBits(241, -1 .. 0)
[1, 1, 1, 1, 0, 0, 0, 1]
```

(4.45)

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

```
> GetBits(241, 1 .. 4)
[0, 0, 0, 1]
```

(4.46)

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

(4.47)

Bitwise Operations

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

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

```
> Split(43)
[1, 1, 0, 1, 0, 1] (4.48)
```

```
> Split(44)
[0, 0, 1, 1, 0, 1] (4.49)
```

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

```
> zip(And, Split(43), Split(44))
[0, 0, 0, 1, 0, 1] (4.50)
```

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

```
> Join([0, 0, 0, 1, 0, 1])
40 (4.51)
```

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

```
> And(43, 44)
40 (4.52)
```

Binary Addition

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

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

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

```

10      S [ j ] := A [ j ] + B [ j ] + c - 2*d;
11      C := d;
12  end do;
13  S [ n+1 ] := c;
14  return S;
15 end proc;

```

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

> *Addition* ([0, 1, 0, 1], [0, 1, 0, 1, 1, 1])
[0, 0, 1, 0, 0, 0, 1] (4.53)

> *Join* (%)
68 (4.54)

Binary Multiplication

Finally, we implement a multiplication algorithm, presented as Algorithm 3 in Section 4.2. Once again, our procedure will accept the binary representations of positive integers as the inputs. This time, however, it is not necessary for them to have the same length.

The shift that occurs when $b_j = 1$ will be accomplished as follows. To shift the list **[1,1,1,1]** by 5 places, we must prepend 5 zeros on front of the list. (Remember, our binary representations have least significant digit first, which is why 0s are added to the front of the list instead of the back.) We do this by creating a new list that begins with 5 zeros and is followed by the elements of the original list.

> *shiftExample* := [1, 1, 1, 1]
shiftExample := [1, 1, 1, 1] (4.55)

> [0 \$ 5, *op*(*shiftExample*)]
[0, 0, 0, 0, 0, 1, 1, 1, 1] (4.56)

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

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

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

Here is our implementation of Algorithm 3.

```

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

```

```

7   else
8     C[j] := [0];
9   end if;
10  end do;
11  p := [0];
12  for j from 1 to nops(b) do
13    p := Addition(p, C[j]);
14  end do;
15  return p;
16 end proc:

```

We test our procedure using Example 10 from Section 4.2.

> *Multiplication* ([0, 1, 1], [1, 0, 1])
[0, 1, 1, 1, 1, 0] (4.57)

4.3 Primes and Greatest Common Divisors

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

Primes

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

Testing for Primality

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

> *isprime* (5)
true (4.58)

> *isprime* (10)
false (4.59)

> *isprime* ($2^{13} - 1$)
true (4.60)

Unlike the trial division algorithm discussed in the book, which checks all possible divisors to see if a number is prime or composite, **isprime** uses a probabilistic primality test. This probabilistic test gains much faster performance at the cost of a small possibility that the command will return an incorrect result. As the help page asserts, there is no known example of an integer for which **isprime** is incorrect and any such example must be exceptionally large. Therefore, **isprime** is reliable.

Listing Primes

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

> *ithprime* (1)
2 (4.61)

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

(4.62)

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

```

```
> ithprime(100 000)
1 299 709
```

(4.64)

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

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

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

(4.65)

Similarly, the prime number before that one is

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

(4.66)

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

Inspecting Procedure Definitions

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

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

(4.67)

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

Prime Factorization

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

```
> ifactor(100)
(2)2(5)2 (4.68)
```

```
> ifactor(123 456 789)
(3)2(3803)(3607) (4.69)
```

```
> ifactor(-987 654 321)
-(3)2(17)2(379 721) (4.70)
```

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

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

```
> expand((4.70))
-987 654 321 (4.71)
```

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

```
> ifactor(236 914 830 635 411 777 378 758 175 934 586 404 476 822)
(2)(197)(509)(32 129 861)2(10 459 723)3 (4.72)
```

```
> ifactor(236 914 830 635 411 777 378 758 175 934 586 404 476 822, easy)
(2)(197)(509)_c37_1 (4.73)
```

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

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

```
> ifactors(100)
[1, [[2, 2], [5, 2]]] (4.74)
```

```
> ifactors(123 456 789)
[1, [[3, 2], [3607, 1], [3803, 1]]] (4.75)
```

```
> ifactors(-987 654 321)
[-1, [[3, 2], [17, 2], [379 721, 1]]] (4.76)
```

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

The Distribution of Primes

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

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

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

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

```
> NumberTheory[pi](1000)
168 (4.77)
```

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

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

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

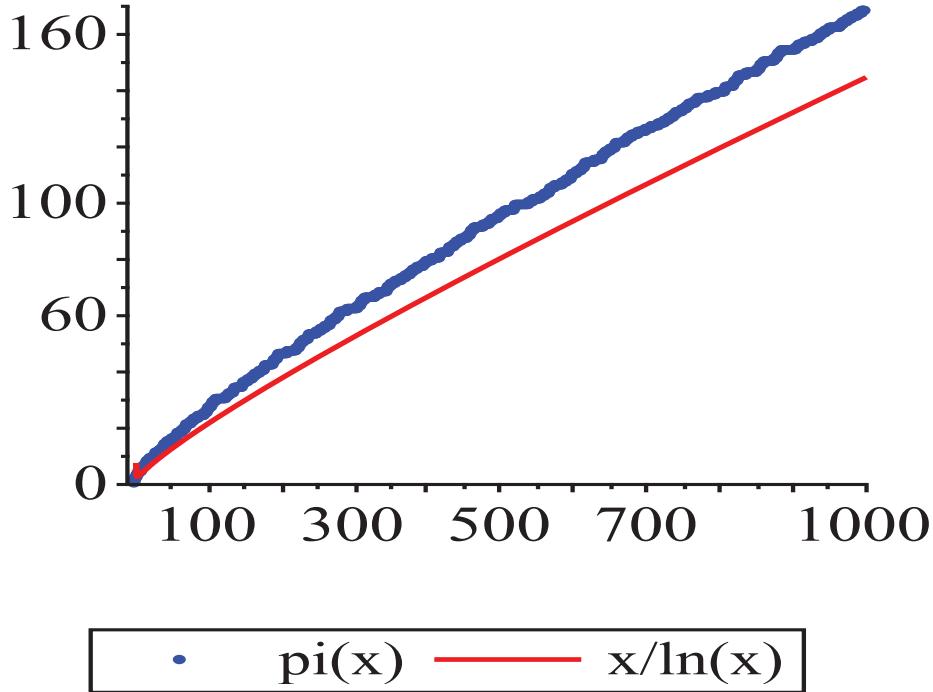
We are going to graphically compare the values of $\pi(x)$ to the function $\frac{x}{\ln(x)}$. We will define two plot objects and then combine them with the **display** command. Refer to Chapter 3 of this manual, particularly Section 3.3 and the solution to Computer Project 10, for detailed information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

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

```

> xlnxPlot := plot \left( \frac{x}{\ln(x)}, x = 1 .. 1000, color = red, view = [1 .. 1000, 0 .. 25],
  legend = "x/\ln(x)" \right) :
> plots[display](piPlot, xlnxPlot)

```



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

Greatest Common Divisors and Least Common Multiples

Maple provides commands **igcd** and **ilcm** for computing the greatest common divisor and the least common multiple of integers. To compute the greatest common divisor of two integers, you apply the **igcd** command to them.

```

> igcd(6, 9)
3

```

(4.78)

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

```

> igcd(6, 9, 12)
3

```

(4.79)

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

```
> ilcm(6, 9)
18
```

(4.80)

```
> ilcm(12, 18, 33)
396
```

(4.81)

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

Relatively Prime

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

```
> igcd(10, 21)
1
```

(4.82)

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

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

```
1 AreRelPrime := proc(a::integer, b::integer)
2   if igcd(a, b) = 1 then
3     return true;
4   else
5     return false;
6   end if;
7 end proc:
```

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

(4.83)

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

(4.84)

Pairwise Relatively Prime

Recall that a list of integers a_1, a_2, \dots, a_n is said to be pairwise relatively prime if $\gcd(a_i, a_j) = 1$ whenever $1 \leq i < j \leq n$. That is, when every pair is relatively prime.

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

```
> igcd(4, 6, 9)
1
```

(4.85)

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

```

1 ArePairwisePrime := proc (A : : list (integer) )
2   local n, i, j;
3   n := nops (A) ;
4   for i from 1 to n-1 do
5     for j from i+1 to n do
6       if gcd (A[i], A[j]) <> 1 then
7         return false;
8       end if ;
9     end do;
10    end do;
11    return true;
12  end proc:
```

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

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

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

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

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

The Extended Euclidean Algorithm

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

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

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

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

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

4.4 Solving Congruences

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

Modular Inverses

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

Finding Inverses with **igcdex**

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

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

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

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

> *s*
 374
(4.92)

> *t*
 -31
(4.93)

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

> $374 \cdot 264 \bmod 3185$
 1
(4.94)

Finding Inverses with $\wedge(-1)$

Maple actually provides a simpler way to compute the modular inverse. The textbook uses the notation \bar{a} to indicate the modular inverse of an integer. An alternate notation is a^{-1} , which calls to mind the notation used in algebra for reciprocals, as in $3^{-1} = 1/3$. Maple interprets an exponent of -1 , within the context of modular arithmetic, as the modular inverse. For example, we can obtain the inverse of 264 modulo 3185 as follows.

```
> 264-1 mod 3185  
374
```

(4.95)

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

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

Error, the modular inverse does not exist

Solving Congruences

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

```
> msolve(4x = 3, 11)  
{x = 9}
```

(4.96)

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

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

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

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

```
> msolve(3x = 9, 12)  
{x = 3}, {x = 7}, {x = 11}
```

(4.97)

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

The Chinese Remainder Theorem

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

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

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

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

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

> *chrem* ([2, 4, 6, 10], [3, 5, 7, 11])

1154

(4.98)

Creating Our Own Procedure

In the remainder of this section we will provide an implementation of the method for solving systems of congruences. This implementation will be based on the construction given in the proof of the Chinese remainder theorem. While this will be less efficient than Maple's **chrem** command, implementing the algorithm can help you to better understand the proof of the theorem.

Our procedure, which we call **CRTheorem**, will accept as input two lists, **a** and **m**, representing the values and the moduli of the congruences. It will begin with two tests to check that the lists are the same length and that the moduli are in fact pairwise relatively prime, as is required by the assumptions of the theorem. We use **ArePairwisePrime** from Section 4.3 of this manual to check that the moduli are pairwise relatively prime.

Once the preliminary tests are complete, the procedure sets **P** equal to the product of the moduli. (Note that **P** corresponds to m in the statement of the theorem in the text. This is the only notational difference between our procedure and the text.) Recall that the Maple command **mul** computes the product of the values obtained by evaluating the first argument at each element in the range given in the second argument.

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

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

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

```

1 CRTheorem := proc (a : : list (integer) , m : : list (posint) )
2   local P, M, y, i, x;
3   if not nops (a) = nops (m) then
4     error "Lists must be the same length .";
5   end if ;
6   if not ArePairwisePrime (m) then
7     error "Moduli must be pairwise relatively prime .";
8   end if ;
9   P := mul (m [i] , i=1..nops (m) );
10  M := table () ;
11  y := table () ;
12  for i from 1 to nops (m) do
13    M [i] := P/m [i];
14    y [i] := M [i] ^ (-1) mod m [i];
15  end do ;
16  x := add (a [i]*M [i]*y [i] , i=1..nops (m) );
17  return x mod P;
18 end proc:
```

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

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

Pseudoprimes

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

The algorithm is fairly straightforward. We will use a for loop beginning at 3, ending with the specified maximum and increasing by 2 each time (so as to skip even integers). Within the loop, we first test the congruence. If the congruence holds, then we use **isprime** to check whether the number is prime or composite. If it is composite, then it is added to the list of pseudoprimes.

```

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

```

8      end if ;
9      end if ;
10     end do ;
11     return PList ;
12 end proc;

```

Note that we used $\&^$ in the calculation of the congruence instead of $^$. The syntax $\&^$ is the “inert” version of the exponent operator. When we use $^$, the integer exponentiation is performed first, which can result in an extremely large number. With the inert $\&^$ operator, Maple performs the exponentiation intelligently, using techniques such as those discussed in Section 4.2 of the text for performing efficient modular exponentiation.

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

```

> FindPseudoprimes(2, 100000)
[341, 561, 645, 1105, 1387, 1729, 1905, 2047, 2465, 2701, 2821
3277, 4033, 4369, 4371, 4681, 5461, 6601, 7957, 8321, 8481,
8911, 10261, 10585, 11305, 12801, 13741, 13747, 13981, 14491,
15709, 15841, 16705, 18705, 18721, 19951, 23001, 23377, 25761,
29341, 30121, 30889, 31417, 31609, 31621, 33153, 34945, 35333,
39865, 41041, 41665, 42799, 46657, 49141, 49981, 52633, 55245,
57421, 60701, 60787, 62745, 63973, 65077, 65281, 68101, 72885,
74665, 75361, 80581, 83333, 83665, 85489, 87249, 88357, 88561,
90751, 91001, 93961]                                         (4.100)

```

Primitive Roots and Discrete Logarithms

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

```
> with(NumberTheory) :
```

Primitive Roots

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

```

> PrimitiveRoot(11)
2                                         (4.101)

```

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

```

> PrimitiveRoot(11, greaterthan = 2)
6                                         (4.102)

```

```

> PrimitiveRoot(11, greaterthan = 6)
7                                         (4.103)

```

> *PrimitiveRoot(11, greaterthan = 7)*
8

(4.104)

> *PrimitiveRoot(11, greaterthan = 8)*

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

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

```
1 AllPrimRoots := proc (n::posint)
2   local L, x;
3   uses NumberTheory;
4   L := [];
5   x := 0;
6   try
7     do
8       x := PrimitiveRoot (n, greaterthan=x);
9       L := [op(L), x];
10    end do;
11  catch:
12    return L;
13  end try;
14 end proc;
```

> *AllPrimRoots(11)*
[2, 6, 7, 8]

(4.105)

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

Note that Maple's **PrimitiveRoot** command applies to nonprime moduli as well. Maple is using a definition of primitive root that is more general than the definition in the text. Specifically, an integer r is a primitive root modulo an integer n if every positive integer that is both less than n and relatively prime to n can be obtained as a power of r .

Discrete Logarithms

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

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

(4.106)

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

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

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

(4.107)

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

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

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

(4.108)

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

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

(4.109)

Exploring the Structure of Primitive Roots

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

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

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

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

```

1 DisplayPowers := proc (p :: prime)
2   local r, x, L;
3   for r from 1 to p-1 do
4     L := [r];
5     x := r;
6     while x <> 1 do
7       x := x * r mod p;
8       L := [op(L), x];
9     end do;
10    print(r, L);
11  end do;
12 end proc;
```

> *DisplayPowers(11)*

```

1, [1]
2, [2, 4, 8, 5, 10, 9, 7, 3, 6, 1]
3, [3, 9, 5, 4, 1]
4, [4, 5, 9, 3, 1]
5, [5, 3, 4, 9, 1]
6, [6, 3, 7, 9, 10, 5, 8, 4, 2, 1]
7, [7, 5, 2, 3, 10, 4, 6, 9, 8, 1]
8, [8, 9, 6, 4, 10, 3, 2, 5, 7, 1]
9, [9, 4, 3, 5, 1]
10, [10, 1]
```

(4.110)

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

4.5 Applications of Congruences

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

Hashing Functions

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

Initial Examples

Each student record will be stored as a table with indices “ID” and “GPA”. Here are three example students.

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

(4.111)

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

(4.112)

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

(4.113)

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

```
> student1[“GPA”]  
3.21
```

(4.114)

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

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

```
> studentRecords := Array(1 ..57)  
studentRecords := 
$$\begin{bmatrix} 1 ..57 \text{ Array} \\ \text{Data Type: anything} \\ \text{Storage: rectangular} \\ \text{Order: Fortran_order} \end{bmatrix}$$

```

(4.115)

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

In order to store a student record in the array (which represents the school’s database), we need to apply a hashing function to the unique student ID. The hashing function we will use is $h(k) = k \bmod 57 + 1$. Note that the addition of 1 is to occur after the computation of $k \bmod 57$. It is included in

our function because the indices in our **studentRecords** array run from 1 to 57 while the values of $k \bmod 57$ range from 0 to 56.

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

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

For example,

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

(4.116)

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

```
> studentRecords[24] := student1
studentRecords24 := student1
```

(4.117)

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

```
> studentRecords[24]
student1
```

(4.118)

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

```
> studentRecords[24]["ID"]
7319
```

(4.119)

```
> studentRecords[24]["GPA"]
3.21
```

(4.120)

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

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

(4.121)

```
> studentRecords[2] := student2
studentRecords2 := student2
```

(4.122)

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

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

(4.123)

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

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

(4.124)

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

```
> studentRecords[25] := student3  
studentRecords25 := student3
```

(4.125)

Printing Records

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

```
> studentRecords  
[ 1 ..57 Array  
  Data Type: anything  
  Storage: rectangular  
  Order: Fortran_order ]
```

(4.126)

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

```
> op(studentRecords)  
1 ..57, {2 = student2, 24 = student1, 25 = student3}, datatype = anything,  
storage = rectangular, order = Fortran_order
```

(4.127)

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

```
1 PrintRecords := proc ()  
2   local i;  
3   global studentRecords;  
4   for i from 1 to 57 do  
5     if studentRecords[i] <> 0 then  
6       print(i, entries(studentRecords[i]));  
7     end if;  
8   end do;  
9 end proc;
```

```
> PrintRecords()  
2, [2.89], [2908]  
24, [3.21], [7319]  
25, [3.42], [6578]
```

(4.128)

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

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

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

Storing New Records

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

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

Here is the completed **Store** procedure.

```
1 Store := proc(id :: integer, gpa :: float)
2   local hash, i, newrecord;
3   global studentRecords;
4   newrecord := table( ["ID"=id, "GPA"=gpa] );
5   for i from 0 to 56 do
6     hash := calculateHash(id + i);
7     if studentRecords [hash] = 0 then
8       break;
9     end if;
10    end do;
11    studentRecords [hash] := newrecord;
12    return NULL;
13  end proc;
```

We now add a few records.

- > *Store*(2216, 1.98)
- > *Store*(1325, 3.14)
- > *Store*(7061, 3.51)

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

```
> op(studentRecords)
1 ..57, {2 = student2, 15 = newrecord, 24 = student1, 25 = student3,
      51 = newrecord, 52 = newrecord} datatype = anything,
      storage = rectangular, order = Fortran_order
```

(4.129)

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

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

(4.130)

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

Retrieving Records

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

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

Assuming the location is not 0, we check to see if the ID of the record in that position is the ID we are looking for. If so, we return the table by applying **eval** to the entry in **studentRecords**. (Without the **eval**, the procedure would return the name of the table rather than the table itself.) If the ID is not the one we are searching for, it must have been the case that our record was pushed down the line because of a collision and we continue the loop.

```
1 Retrieve := proc(id: :integer)
2   local hash, i;
3   global studentRecords;
4   for i from 0 to 56 do
5     hash := calculateHash(id + i);
6     if studentRecords[hash] = 0 then
7       return FAIL;
8     end if;
9     if studentRecords[hash] ["ID"] = id then
```

```

10      return eval(studentRecords[hash]);
11  end if;
12 end do;
13 return FAIL;
14 end proc:
```

> *Retrieve*(1325)
 $\text{table}([\text{"GPA"} = 3.14, \text{"ID"} = 1325])$ (4.131)

> *Retrieve*(7061)
 $\text{table}([\text{"GPA"} = 3.51, \text{"ID"} = 7061])$ (4.132)

Pseudorandom Numbers

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

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

> $\text{floor}(\text{time}[\text{real}]() \cdot 1000)$
96865119 (4.133)

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

We will write two procedures that generate random student IDs and GPAs that we can use to add some random records to our **studentRecords** from above. We first write the procedure **randomIDs**, which will accept a positive integer as input to control the number of IDs to generate. It will return a sequence of that number of random student IDs.

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

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

```

1 randomIDs := proc (n : : posint)
2   local S, m, a, c, x;
3   S := NULL;
4   m := 8999;
5   a := 57;
6   c := 328;
7   x := modp(floor(time[real] () * 1000), m);
8   from 1 to n do
9     x := modp(a*x+c, m);
10    S := S, x+1000;
11  end do;
12  return S;
13 end proc:
```

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

```

> someIDs := [randomIDs(10)]
someIDs := [3874, 3164, 7689, 4643, 2002, 4448, 8885,
9822, 9237, 2889] (4.134)
```

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

```

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

```

> someGPAs := [randomGPAs(10)]
someGPAs := [1.09, 0.538, 2.39, 0.933, 3.86, 1.97, 1.24,
1.23, 1.11, 1.20] (4.135)
```

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

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

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

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

Check Digits

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

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

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

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

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

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

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

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

```
> parse(isbnExample[3])  
7  
(4.139)
```

Our procedure will compute the sum indicated by the formula above using the **add** command. Recall that the first argument to **add** is an expression in terms of an index variable and the second argument is the range for the variable. Once the value of x_{10} is determined, we compare it to the check digit. This is only slightly complicated by the fact that a check digit of 10 corresponds to the symbol X.

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

```
> checkISBN(isbnExample)  
true  
(4.140)
```

```
> checkISBN("084 930 149X")  
false  
(4.141)
```

```
> checkISBN("232 150 031X")  
true  
(4.142)
```

4.6 Cryptography

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

Encoding Strings

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

Some Commands for Working with Strings

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

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

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

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

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

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

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

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

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

Mapping Characters to Integers

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

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

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

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

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

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

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

Converting Between a String and a Numeric Representation

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

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

```
1 StringToNums := proc(s :: string)
2   local S;
3   uses StringTools;
4   global CharToNum;
5   S := Explode(UpperCase(s));
6   S := map(c -> CharToNum[c], S);
7   return S;
8 end proc;
```

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

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

```
1 NumsToString := proc(S :: list)
2   local s;
3   global NumToChar;
4   s := map(c -> NumToChar[c], S);
5   s := Implode(s);
6   return s;
7 end proc;
```

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

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

Classical Cryptography

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

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

where p is an integer corresponding to a character that is to be encrypted. We will refer to the pair a, b as the key to the cipher. For decryption to be feasible, the key must be chosen so that f is a

bijection. This amounts to choosing a relatively prime to 27. (Note that the text uses a modulus of 26 where we use 27 because we are considering space to be an encodable character.)

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

```

1  AffineCipher := proc (s : :string, a : :integer, b : :integer)
2    local S, T;
3    S := StringToNums (s);
4    T := map (p -> modp (a*p+b, 27), S);
5    return NumToString (T);
6  end proc:
```

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

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

> *AffineCipher*(“The quick brown fox”, 5, 3)
“VPACG URDCMLXJSCFXO” (4.150)

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

$$p \equiv a^{-1}(c - b) \pmod{27} \equiv a^{-1}c - a^{-1}b \pmod{27}.$$

In other words, to decrypt a message encrypted using the key (a, b) , we use the same procedure but with key $(a^{-1}, -a^{-1}b)$.

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

> $5^{-1} \bmod 27$
11 (4.151)

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

> $-5^{-1} \cdot 3 \bmod 27$
21 (4.152)

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

> *AffineCipher*(4.150), 11, 21)
“THE QUICK BROWN FOX” (4.153)

RSA Encryption

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

To construct keys in the RSA system, we need to find pairs of large primes, say with 300 digits each. Since messages can be decrypted by anyone who can factor the product of these primes, the two primes must be large enough so that their product is extremely difficult to factor.

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

Key Generation

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

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

```

1 GenerateKeys := proc (p::prime, q::prime)
2   local n, phin, e, d;
3   e := 13;
4   n := p * q;
5   phin := (p - 1) * (q - 1);
6   d := modp(e^(-1), phin);
7   return [[n, e], [n, d]];
8 end proc:
```

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

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

```
> keys := GenerateKeys(59, 71)
keys := [[4189, 13], [4189, 937]] (4.154)
```

The public and private keys are

```
> publickey := keys[1]
publickey := [4189, 13] (4.155)
```

```
> privatekey := keys[2]
privatekey := [4189, 937] (4.156)
```

Encoding

Now that we have the keys, we turn to encoding the message. As described in the text, we encode the message in much the same way as for affine ciphers, except that we block groups of characters into single integers. The block length must be chosen so that, after conversion, the largest integer produced is less than the modulus n . Here, we have $n = 4189$ and the largest block that can be produced is 2626 for “ZZ”.

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

For a specific example, consider the message “SECRET MESSAGE”. We can use our **StringToNums** procedure from above to get the numeric representation of each character.

```
> messageString := StringToNums("SECRET MESSAGE")
messageString := [19, 5, 3, 18, 5, 20, 0, 13, 5, 19, 19, 1, 7, 5] (4.157)
```

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

```
> messageCode := []:
> for i from 2 to nops(messageString) by 2 do
    messageCode := [op(messageCode),
      100 · messageString[i - 1] + messageString[i]] :
end do :
> messageCode
[1905, 318, 520, 13, 519, 1901, 705] (4.158)
```

Encryption

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

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

Our “SECRET MESSAGE” is encrypted as

```
> cipherText := RSA(publickey, messageCode)
cipherText := [723, 3360, 2306, 1979, 2695, 917, 1863] (4.159)
```

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

```
> RSA(privatekey, cipherText)
[1905, 318, 520, 13, 519, 1901, 705] (4.160)
```

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

Generating Large Primes

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

Using Maple’s computational abilities, we can generate fairly large prime numbers for use in an RSA key. Remember that what is needed is a pair of prime numbers, each of about 300 digits. Moreover, they should be selected in an unpredictable fashion. To do this in Maple, we can use the **rand** command to produce a random 300 digit number. Then, we use the **nextprime** command to find the smallest prime number that exceeds our random number. This will guarantee that the prime number has at least 300 digits.

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

```
> bigInt := rand(10299 .. 10300)
> a := bigInt()
a :=
235 726 808 767 240 131 892 387 679 230 487 763 079 953 883 726
554 914 270 335 331 988 936 067 036 004 248 743 352 686 385 930
952 790 393 226 156 791 082 950 595 300 471 690 316 314 646 561
668 134 009 554 428 603 253 983 753 534 033 365 546 308 155 005
729 686 838 034 395 625 006 588 935 807 374 308 092 774 154 589
243 063 887 435 863 555 163 865 559 297 655 953 965 900 318 854
667 516 172 768 (4.161)
```

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

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

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

(4.163)

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

(4.164)

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

Homomorphic Encryption

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

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

```
> cloud := RSA(publickey, [23])[1]
cloud := 3655
```

(4.165)

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

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

```
> cloud := cloud · RSA(publickey, [45])[1]
cloud := 7894 800
```

(4.166)

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

> RSA(privatekey, [cloud])
[1035] (4.167)

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

> 23 · 45
1035 (4.168)

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

Solutions to Computer Projects and Computations and Explorations

Computer Projects 3

Given a positive integer, find the Cantor expansion of this integer (see the preamble to Exercise 54 of Section 4.2).

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

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

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

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

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

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

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

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

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

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

```

3   A := [ ];
4   n := 1;
5   Y := a;
6   while Y <> 0 do
7       A := [ op(A), modp(Y, n+1) ];
8       Y := (Y-A[n]) / (n+1);
9       n := n+1;
10      end do;
11      return A;
12  end proc;

```

> *CantorExpression*(471)

[1, 1, 2, 4, 3]

(4.169)

Computer Projects 21

Generate a shared key using the Diffie–Hellman key exchange protocol.

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

(1) Alice and Bob agree on a prime number p and a primitive root a of p . For this example, we use a relatively small prime.

> *DHprime* := *nextprime*(*rand*())
DHprime := 395 718 860 549

(4.170)

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

> *DHroot* := *primroot*(*DHprime*)
DHroot := 3

(4.171)

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

> *AliceSends* := *DHroot* &[^] 421 **mod** *DHprime*
AliceSends := 287 654 735 840

(4.172)

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

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

> *BobSends* := *DHroot* &[^] *rand*() **mod** *DHprime*
BobSends := 346 411 045 536

(4.173)

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

```
> sharedKey := BobSends &^ 421 mod DHprime
      sharedKey := 318 885 707 608
```

(4.174)

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

Computations and Explorations I

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

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

```
1 CheckMersenne := proc (max::posint)
2   local p, L;
3   p := 2;
4   L := [];
5   while p <= max do
6     if isprime(2^p-1) then
7       L := [op(L), p];
8     end if;
9     p := nextprime(p);
10  end do;
11  return L;
12 end proc;
```

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

```
> CheckMersenne(100)
[2, 3, 5, 7, 13, 17, 19, 31, 61, 89]
```

(4.175)

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

```
> IsMersenne(19)
true
```

(4.176)

```
> IsMersenne(20)
false
```

(4.177)

```
> IsMersenne(457 237 649 731)
FAIL
```

(4.178)

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

```
> IthMersenne(5)
13
```

(4.179)

> *IthMersenne*(100)

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

It is of note that there is a better test for checking whether a Mersenne number is prime, called the Lucas–Lehmer test, that is more efficient and can be implemented in Maple. For a complete description of that algorithm, consult Rosen’s text on Number Theory.

Computations and Explorations 5

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

Solution: We write a Maple procedure that, given a maximum n , tests the integers of the given form.

```
1 CE5 := proc(max::posint)
2   local n, L;
3   L := [];
4   for n from 1 to max do
5     if isprime(n^2+1) then
6       L := [op(L), n^2+1];
7     end if;
8   end do;
9   return L;
10 end proc;
```

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

> *CE5*(100)

[2, 5, 17, 37, 101, 197, 257, 401, 577, 677, 1297, 1601, 2917,
3137, 4357, 5477, 7057, 8101, 8837]

(4.180)

Exercises

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

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

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

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

Exercise 5. Use Maple to determine which numbers are perfect squares modulo n for various values of the modulus n . For each perfect square s , determine how many square roots s has. That is, for how many values of x is $x^2 \equiv s \pmod{n}$. What conjectures can you make about the number of different square roots an integer has modulo n ? (The Maple functions **ModularSquareRoot** and **msolve** may be of use.)

Exercise 6. Use Maple to find the base 2 expansion of the 4th Fermat number $F_4 = 65\,537$. Do the following for several large integers n . Compute the time required to calculate the remainder modulo n of various bases b raised to the power F_4 (i.e., the time to calculate $b^{F_4} \pmod{n}$) using two different methods. First, do the calculation by a straightforward exponentiation. Second, do it using the binary expansion of F_4 with repeated squarings and multiplications. Why do you think F_4 is a good choice for the public exponent in the RSA encryption scheme?

Exercise 7. Modify the procedure **GenerateKeys** that we developed to produce the keys for the RSA system to incorporate the techniques for generating random large primes. Make your procedure take as an argument a “security” parameter which measures the number of digits in the primes.

Exercise 8. Write Maple routines to encode and decode English sentences into lists of integers appropriate for encryption with **RSA**. You may ignore punctuation and insist that all letters are uppercase. Your procedures should accept as input the block size.

Exercise 9. There are infinitely many primes of the form $4n + 1$ and infinitely many of the form $4n + 3$. Use Maple to determine for various values of x whether there are more primes of the form $4n + 1$ less than x than there are of the form $4n + 3$. What conjectures can you make from this evidence?

Exercise 10. Develop a procedure for determining whether Mersenne numbers are prime using the Lucas–Lehmer test as described in number theory books, such as *Elementary Number Theory and its Applications* by K. Rosen. How many Mersenne numbers can you test for primality using Maple?

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

Exercise 12. Compute the sequence of pseudorandom numbers generated by the linear congruential generator $x_{n+1} = (ax_n + c) \pmod{m}$ for various values of the multiplier a , the increment c , and the modulus m . For which values do you get a period of length m (period is defined in Exercise 14) for the sequence that you generate? Formulate a conjecture.

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

Exercise 14. A sequence a_1, a_2, a_3, \dots is called *periodic* if there are positive integers N and p for which $a_n = a_{n+p}$ for all $N \leq n$. The least integer p for which this is true is called the *period* of the sequence. The sequence is said to be *periodic modulo m*, for a positive integer m , if the sequence $a_1 \pmod{m}, a_2 \pmod{m}, a_3 \pmod{m}, \dots$ is periodic. Use Maple to determine whether the Fibonacci sequence is periodic modulo m for various integers m and, if so, find the period. Can you, by examining enough different values of m , make any conjectures concerning the relationship between m and the period? Do the same thing for other sequences that you find interesting.

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

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

5 Induction and Recursion

Introduction

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

5.1 Mathematical Induction

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

Summation Example 1

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

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

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

Listing and Graphing to Find the Formula

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

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

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

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

In our situation, we want to add the first several positive integers. For example, the sum of the first 10 positive integers is

> **add(i, i = 1 .. 10)**

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

```
> seq(add(i,i = 1 ..n), n = 1 ..50)
1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136, 153,
171, 190, 210, 231, 253, 276, 300, 325, 351, 378, 406, 435, 465, 496,
528, 561, 595, 630, 666, 703, 741, 780, 820, 861, 903, 946, 990, 1035,
1081, 1128, 1176, 1225, 1275
(5.2)
```

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

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

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

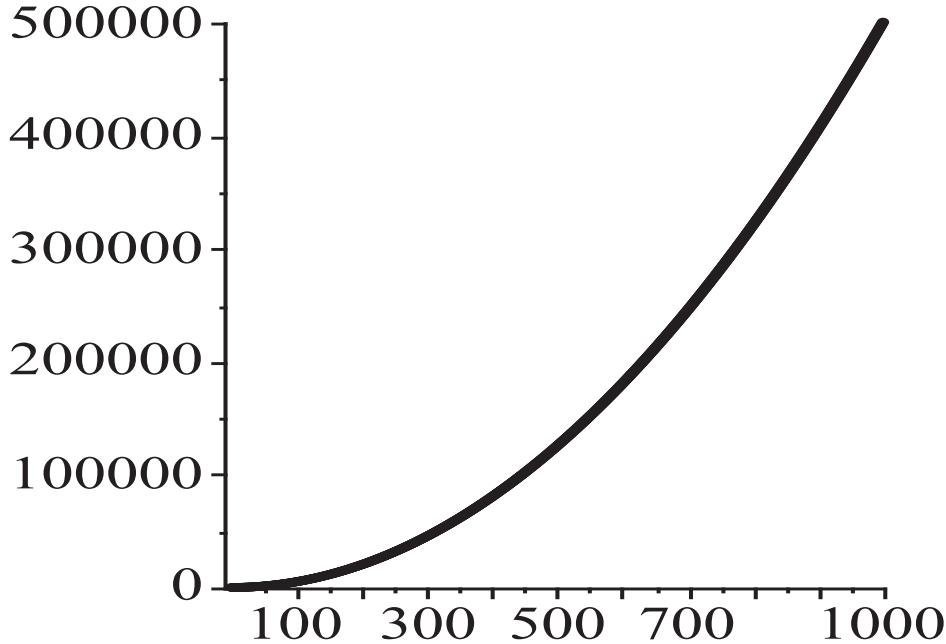
This still may not be enough to get an idea of what the formula could be, in which case a graph of the data might be of use. The **plot** command is used to graph functions and data, and was first discussed in Section 2.3 of this manual. In this situation, we want to plot the points that the previous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

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

These lists will be the first two arguments to **plot**, with the x -values as the first argument and the y -values the second. These are the only required arguments when graphing data, but we will use several options. As we have seen before, the **style=point** and **symbol=solidcircle** options will cause Maple to draw the graph as a series of dots. (The default is to “connect the dots” with straight line segments.) We will also increase the size of the dots slightly with the **symbolsize=10** option.

We actually have enough data in this example that even with these options, the graph will look like a smooth line, but they are useful options to be aware of.

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



Although not needed in this particular example, it is also worth remembering the option **view=[xmin..xmax,ymin..ymax]** which specifies the region that is displayed in the graph. Without it, Maple chooses a view window.

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

Finding the Coefficients

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

We already know numerous values for this function. Here are the first few again.

```
> seq([n, add(i, i = 1..n)], n = 1 .. 10)
[1, 1], [2, 3], [3, 6], [4, 10], [5, 15], [6, 21], [7, 28], [8, 36], [9, 45], [10, 55]      (5.4)
```

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

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

For $n = 1$, we have

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

For $n = 3$:

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

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

```
> eval(2x + 5, x = 3)  
11
```

(5.5)

evaluates the expression $2x + 5$ after substituting 3 for x . In our case, the first argument will be the equation $f(n) = an^2 + bn + c$. Instead of $f(n)$, we need to use our data **sumList**. So our first argument will be **sumList[n] = a*n^2+b*n+c**. With the second argument **n=2**, say, Maple will replace each **n** with 2. It will then look up the second entry in **sumList** and simplify the right side of the equation for us.

```
> eval(sumList[n] = an^2 + bn + c, n = 2)  
3 = 4a + 2b + c
```

(5.6)

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

```
> [seq(eval(sumList[n] = an^2 + bn + c, n = N), N = 1..10)]  
[1 = a + b + c, 3 = 4a + 2b + c, 6 = 9a + 3b + c, 10 = 16a + 4b + c,  
15 = 25a + 5b + c, 21 = 36a + 6b + c, 28 = 49a + 7b + c,  
36 = 64a + 8b + c, 45 = 81a + 9b + c, 55 = 100a + 10b + c]
```

(5.7)

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

```
> solve((5.7))  
{a = 1/2, b = 1/2, c = 0}
```

(5.8)

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

You may recall that to solve a system of equations in three unknowns, only three equations are required. In this situation, having more equations is useful. If we were wrong about the formula being quadratic but attempted to find coefficients with only three equations, Maple may still have

found values for a , b , and c that satisfied the three equations we chose. With ten equations, if the actual formula were not quadratic, there is a greater chance that no values of a , b , and c would satisfy all ten equations. In that case, Maple would have returned **NULL** and displayed no output, which would indicate the absence of a solution and that our guess about the kind of formula was incorrect.

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

The Induction Proof

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

$$\begin{aligned} > \text{sumF} := n \rightarrow \frac{n^2}{2} + \frac{n}{2} \\ &\text{sumF} := n \mapsto \frac{1}{2}n^2 + \frac{1}{2}n \end{aligned} \tag{5.9}$$

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

$$\begin{aligned} > \text{add}(i, i = 1 .. 1) \\ &1 \end{aligned} \tag{5.10}$$

$$\begin{aligned} > \text{sumF}(1) \\ &1 \end{aligned} \tag{5.11}$$

They are equal and the basis step is verified.

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

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

$$\begin{aligned} > \text{sumF}(k) + (k + 1) \\ &\frac{1}{2}k^2 + \frac{3}{2}k + 1 \end{aligned} \tag{5.12}$$

Note that the inductive hypothesis was used to replace the sum up to k with $f(k)$. We need to see if this is the same as $f(k+1)$.

$$\begin{aligned} > \text{sumF}(k+1) \\ & \frac{(k+1)^2}{2} + \frac{k}{2} + \frac{1}{2} \end{aligned} \tag{5.13}$$

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

$$\begin{aligned} > \text{evalb}(\text{simplify}(\text{sumF}(k) + (k+1) = \text{sumF}(k+1))) \\ & \text{true} \end{aligned} \tag{5.14}$$

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

Summation Example 2

As a second example of using Maple to find and prove summation formulae, consider the sum $\sum_{i=1}^n i^2(i+1)!$. For this example, we will not go through the process of computing values and graphing as we did above. That is a very valuable process, and we strongly recommend that you go through it yourself with a few examples. However, Maple includes a command that will give us the result.

Using sum to Determine the Formula

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

$$\begin{aligned} > \text{add}(i^2 \cdot (i+1)!, i = 1 .. 10) \\ & 4311014402 \end{aligned} \tag{5.15}$$

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

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

$$\begin{aligned} > \text{sum}(i^2 \cdot (i+1)!, i = 1 .. n) \\ & (n-1)(n+2)! + 2 \end{aligned} \tag{5.16}$$

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

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

The Induction Proof

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

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

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

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

$$\begin{aligned} > \text{sumF2} &:= n \rightarrow (n-1)(n+2)! + 2 \\ &\quad \text{sumF2} := n \mapsto (n-1)(n+2)! + 2 \end{aligned} \tag{5.17}$$

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

$$\begin{aligned} > 1^2(1+1)! \\ &\quad 2 \end{aligned} \tag{5.18}$$

And the formula applied to $n = 1$ is

$$\begin{aligned} > \text{sumF2}(1) \\ &\quad 2 \end{aligned} \tag{5.19}$$

The basis step is verified.

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

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

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

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

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

$$\begin{aligned} > \text{sumF2}(k) + (k+1)^2((k+1)+1)! \\ & (k-1)(k+2)! + 2 + (k+1)^2(k+2)! \end{aligned} \tag{5.20}$$

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

$$\begin{aligned} > \text{sumF2}(k+1) \\ & k(k+3)! + 2 \end{aligned} \tag{5.21}$$

$$\begin{aligned} > \text{evalb}(\text{simplify}(\text{sumF2}(k) + (k+1)^2((k+1)+1)! = \text{sumF2}(k+1))) \\ & \text{true} \end{aligned} \tag{5.22}$$

This completes the induction and proves (subject to Maple's computations being correct) the correctness of the formula.

Divisibility

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

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

$$\begin{aligned} > \text{divExpr} := n \rightarrow n^7 - n \\ & \text{divExpr} := n \mapsto n^7 - n \end{aligned} \tag{5.23}$$

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

$$\begin{aligned} > \text{divExpr}(0) \\ & 0 \end{aligned} \tag{5.24}$$

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

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

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

$$\begin{aligned} > \text{divExpr}(k+1) - \text{divExpr}(k) \\ & (k+1)^7 - 1 - k^7 \end{aligned} \tag{5.25}$$

$$\begin{aligned} > \text{simplify}(\%) \\ & 7k(k+1)(k^2+k+1)^2 \end{aligned} \tag{5.26}$$

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

> $\% \bmod 7$
0 (5.27)

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

5.2 Strong Induction and Well-Ordering

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

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

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

Making Postage

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

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

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

```
> for a from 0 to floor( $\frac{12}{4}$ ) do
  for b from 0 to floor( $\frac{12}{5}$ ) do
    if  $4 \cdot a + 5 \cdot b = 12$  then
      print([a, b]);
    end if;
  end do;
end do;
[3, 0] (5.28)
```

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

```

1 MakePostage := proc (A : : posint, B : : posint, postage : : posint)
2   local a, b;
3   for a from 0 to floor (postage/A) do
4     for b from 0 to floor (postage/B) do
5       if A*a + B*b = postage then
6         return [a, b];
7       end if;
8     end do;
9   end do;
10  return FAIL;
11 end proc;
```

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

> *MakePostage* (4, 5, 13)
[2, 1] (5.29)

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

> *MakePostage* (4, 5, 11)
FAIL (5.30)

Automating the Basis Step

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

> **for** p **from** 12 **to** 15 **do**
MakePostage(4, 5, p)
end do
[3, 0]
[2, 1]
[1, 2]
[0, 3] (5.31)

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

that you can make postage of $k + 1 - 4 = k - 3$ cents. This requires that $k - 3 \geq 12$, that is, $k \geq 15$.

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

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

```

1 PostageBasis := proc (A::posint, B::posint, minpost::posint)
2   local small, postList, post, R;
3   small := min(A, B);
4   postList := [];
5   for post from minpost to (minpost + small - 1) do
6     R := MakePostage(A, B, post);
7     if R <> FAIL then
8       postList := [op(postList), post=R];
9     else
10      return false;
11    end if;
12  end do;
13  return postList;
14 end proc:
```

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

> *PostageBasis*(4, 5, 12)
[12 = [3, 0], 13 = [2, 1], 14 = [1, 2], 15 = [0, 3]] (5.32)

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

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

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

> *PostageBasis*(4, 6, 12)
false (5.33)

5.3 Recursive Definitions and Structural Induction

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

A Simple Recursive Function

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

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

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

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

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

```
> functionEx := x → 3x2 - 2x + 4  
functionEx := x ↪ 3x2 - 2x + 4
```

(5.34)

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

```
> f := n → 2f(n - 1) + 3  
f := n ↪ 2f(n - 1) + 3
```

(5.35)

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

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

(5.36)

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

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

```
> f(10)
6141
```

(5.37)

You can compute the values of f from 1 to 9 using **seq**.

```
> seq(f(n), n = 1 .. 9)
9, 21, 45, 93, 189, 381, 765, 1533, 3069
```

(5.38)

Remember Tables

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

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

```
> op(eval(f))
n, operator, arrow, table([0 = 3])
```

(5.39)

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

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

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

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

For a functional operator, we have seen that you explicitly store a value in the remember table with the syntax **f(x) := v**. In order to have a functional operator automatically store values that it computes, you define the functional operator as follows.

```
> f2 := n → (f2(n) := 2 · f2(n - 1) + 3)
f2 := n ↪ (f2(n) := 2f2(n - 1) + 3)
```

(5.40)

```
> f2(0) := 3
f2(0) := 3
```

(5.41)

The above defines **f2** as a functional operator whose input is **n** and whose result is the assignment to **f2(n)** the result of the recursive formula. This means that when we execute **f2(10)**, we are telling Maple to make the assignment **f2(10) := ...**, which has the effect of storing a value in the remember table for **f2**.

```
> f2(10)
6141
```

(5.42)

Since the computation of $f2(10)$ required the recursive calculations $f2(9), f2(8), f2(7), \dots, f2(1)$, and each of those function calls result in an assignment statement, Maple has now stored all of those values in the remember table.

```
> op(eval(f2))
n, operator, arrow, table([0 = 3, 1 = 9, 2 = 21, 3 = 45, 4 = 93,
5 = 189, 6 = 381, 7 = 765, 9 = 3069, 8 = 1533, 10 = 6141])
```

(5.43)

You may occasionally want to clear a procedure or function's remember table. To do this, you apply the **forget** command to the name of the function or procedure, for example, **forget(f2)**.

A Second Recursive Function

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

We could define this function as a functional operator as we did in the previous subsection. However, modeling the function as a procedure will illustrate how to have Maple create a remember table for procedures.

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

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

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

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

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

```
> seq(F(n), n = 0 .. 10)
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89
```

(5.44)

Also note that **F** has added the results of its calculations to its remember table.

```
> op(eval(F))
n ::  $\mathbb{Z}^{(0,+)}$ , remember, table([0 = 1, 1 = 1, 2 = 2, 3 = 3, 4 = 5, 5 = 8,
6 = 13, 7 = 21, 9 = 55, 8 = 34, 10 = 89])
```

(5.45)

Comparing Complexity

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

Here are the two procedures.

```
1 gR := proc(n :: nonnegint)
2   option remember;
3   2*gR(n-1) + 3*gR(n-2);
4 end proc;
5 gR(0) := 5:
6 gR(1) := 2:
7
8 gF := proc(n :: nonnegint)
9   2*gF(n-1) + 3*gF(n-2);
10 end proc;
11 gF(0) := 5:
12 gF(1) := 2:
```

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

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

(5.46)

Tracing the Procedure with a Remember Table

Now, execute **gR** on 5.

```
> gR(5)
```

```
{--> enter gR, args = 5
{--> enter gR, args = 4
```

```

{--> enter gR, args = 3
{--> enter gR, args = 2
value remembered (in gR) : gR(1) ->2
value remembered (in gR) : gR(0) ->5
    19

<-- exit gR (now in gR) = 19}
value remembered (in gR) : gR(1) ->2
    44

<-- exit gR (now in gR) = 44}
value remembered (in gR) : gR(2) ->19
    145

<-- exit gR (now in gR) = 145}
value remembered (in gR) : gR(3) ->44
    422

<-- exit gR (now at top level) = 422}
    422

```

(5.47)

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

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

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

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

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

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

Once **gR(3)** returns 44, Maple continues “backing out of the recursion.” **gR(3)** was called in the computation of **gR(4)**, specifically the formula $2*gR(3)+3*gR(2)$. Since **gR(3)** just returned 44, Maple can turn this expression into $2*44+3*gR(2)$. Now it needs to determine the value of **gR(2)**. Thanks to the remember option, **gR(2)** was added to the remember table. The trace tells us this two lines below the blue 44. Maple replaces **gR(2)** with the remembered value of 19 and computes that **gR(4)** is 145. It returns this value and exits the **gR(4)** computation.

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

Tracing the Forgetful Procedure

Contrast the above to what happens when we execute the forgetful **gF** on 5.

```
> gF(5)

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

(5.48)
```

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

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

Maple then tracks up back to the execution of $\mathbf{gF(3)}$ and the expression $2*\mathbf{gF(2)}+3*\mathbf{gF(1)}$. It just returned the value of $\mathbf{gF(2)}$ and it looks up $\mathbf{gF(1)}$. So it returns 44 and exits $\mathbf{gF(3)}$.

The first 11 lines of the output from the trace are the same as for \mathbf{gR} . However, something different happens now. Once Maple exits $\mathbf{gF(3)}$, it is back to the execution of $\mathbf{gF(4)}$ and the formula $2*\mathbf{gF(3)}+3*\mathbf{gF(2)}$. It has just returned 44 for $\mathbf{gF(3)}$, so the expression has been partially resolved and stands as $2*44+3*\mathbf{gF(2)}$. When \mathbf{gR} was at this stage, it was able to look up the value of $\mathbf{gF(2)}$ because it had automatically stored that in the remember table. However, \mathbf{gF} is not recording output in its remember table. So to complete the computation of $\mathbf{gF(4)}$, it has to once again call \mathbf{gF} with argument 2.

After executing $\mathbf{gF(2)}$ and recomputing 19, $\mathbf{gF(4)}$ is able to complete and it returns 145.

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

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

A Recursive Function with Two Parameters

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

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

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

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

```

1 A := proc (m::nonnegint, n::nonnegint)
2   option remember;
3   if n=0 and m>0 then
4     return A(m-1, n)+1;

```

```

5   else
6       return A(m, n-1) + n;
7   end if ;
8 end proc;
9 A(0, 0) := 0;

```

We can now compute some values of A .

> $A(3, 2)$

6

(5.49)

> $A(5, 3)$

11

(5.50)

Displaying Values of A

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

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

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

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

> $Matrix(10, (i, j) \rightarrow A(i - 1, j - 1))$

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

(5.51)

A Recursively Defined Set

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

Let S be the subset of the integers defined by

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

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

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

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

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

```
> S := {4,7}
S := {4,7} (5.52)
```

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

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

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

```
1 recurseS := proc (S :: set)
2   local x, y, T;
3   T := S;
4   for x in S do
5     for y in S do
6       T := T union {x + y};
7     end do;
8   end do;
9   return T;
10 end proc:
```

We apply this procedure to \mathbf{S} .

```
> S := recurseS(S)
S := {4,7,8,11,14} (5.53)
```

After a second iteration:

```
> S := recurseS(S)
S := {4, 7, 8, 11, 12, 14, 15, 16, 18, 19, 21, 22, 25, 28} (5.54)
```

A third:

```
> S := recurseS(S)
S := {4, 7, 8, 11, 12, 14, 15, 16, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28,
      29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 46, 47, 49,
      50, 53, 56} (5.55)
```

A Set of Strings

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

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

```
> Alphabet := {"a", "b", "c", "d"}
Alphabet := {"a", "b", "c", "d"} (5.56)
```

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

```
> S2 := {""}
S2 := {""} (5.57)
```

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

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

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

```
1 buildStrings := proc (S::set, A::set)
2   local T, w, x;
3   T := S;
4   for w in T do
5     for x in A do
6       T := T union {cat(w, x)};
7     end do;
8   end do;
9   return T;
10 end proc;
```

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

```
> S2 := buildStrings(S2, Alphabet)
S2 := {"", "a", "b", "c", "d"}(5.58)
```

The second application adds all the two-character strings.

```
> S2 := buildStrings(S2, Alphabet)
S2 := {"", "a", "aa", "ab", "ac", "ad", "b", "ba", "bb", "bc", "bd",
       "c", "ca", "cb", "cc", "cd", "d", "da", "db", "dc", "dd"}(5.59)
```

The third application includes the three-character strings.

```
> S2 := buildStrings(S2, Alphabet)
S2 := {"", "a", "aa", "aaa", "aab", "aac", "aad", "ab", "aba", "abb",
       "abc", "abd", "ac", "aca", "acb", "acc", "acd", "ad", "ada", "adb",
       "adc", "add", "b", "ba", "baa", "bab", "bac", "bad", "bb", "bba",
       "bbb", "bbc", "bbd", "bc", "bca", "bcb", "bcc", "bcd", "bd", "bda",
       "bdb", "bdc", "bdd", "c", "ca", "caa", "cab", "cac", "cad", "cb",
       "cba", "cbb", "cbc", "cbd", "cc", "cca", "ccb", "ccc", "ccd", "cd",
       "cda", "cdb", "cdc", "cdd", "d", "da", "daa", "dab", "dac", "dad",
       "db", "dba", "dbb", "dbc", "dbd", "dc", "dca", "dcb", "dcc", "dcd",
       "dd", "dda", "ddb", "ddc", "ddd"}(5.60)
```

A General Procedure

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

```
1 AllStrings := proc (A::set, n::posint)
2   local S;
3   S := {""};
4   from 1 to n do
5     S := buildStrings (S, A);
6   end do;
7   return S;
8 end proc;
```

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

```
> AllStrings ({"ab", "ba"}, 4)
{ "", "ab", "abab", "ababab", "ababba", "abba", "abbaab", "abbaba",
  "ba", "baab", "baabab", "baabba", "baba", "babaab", "bababa",
  "abababab", "abababba", "ababbaab", "ababbaba", "abbaabab",
  "abbaabba", "abbabaab", "abbababa", "baababab", "baababba",
  "baabbaab", "baabbaba", "babaabab", "babaabba", "bababaab",
  "babababa"}(5.61)
```

5.4 Recursive Algorithms

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

Modular Exponentiation

Example 4 of Section 5.4 of the text describes two recursive approaches to computing $b^n \bmod m$. Both of these use the initial condition $b^0 \bmod m = 1$. The first approach is based on the fact that

$$b^n \bmod m = (b \cdot (b^{n-1} \bmod m) \bmod m).$$

The second approach is based on the observation for even exponents that we can compute via the formula

$$b^n \bmod m = (b^{n/2} \bmod m)^2 \bmod m.$$

On the other hand, if the exponent is odd, we can use the identity

$$b^n \bmod m = \left((b^{\lfloor n/2 \rfloor} \bmod m)^2 \bmod m \right) \cdot (b \bmod m) \bmod m.$$

Approach 1

First, we will implement exponentiation based on the initial condition $b^0 \bmod m = 1$ and the formula

$$b^n \bmod m = (b \cdot (b^{n-1} \bmod m) \bmod m).$$

The procedure **power1** will accept three arguments: the base b , the exponent n , and the modulus m .

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

```
1 power1 := proc(b::integer, n::nonnegint, m::posint)
2   if n=0 then
3     return 1;
4   else
5     return modp(b * power1(b, n-1, m), m);
6   end if;
7 end proc;
```

Note that we did not use the **remember** option in this procedure. The reason for this is that each iteration of the procedure only depends on one other call to the procedure. This is in contrast to the

gR and **gF** procedures from the previous section. Those procedures called themselves twice in each iteration. As a result, those procedures made use of the same value multiple times. Our **power1** procedure will not, unless you execute the procedure on the same input values. When deciding whether or not to use the **remember** option, you must weigh its potential benefit for not repeating computation with the cost of storage requirements.

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

```
> power1(3, 6, 7)
1
```

(5.62)

```
> 3^6 mod 7
1
```

(5.63)

Approach 2

The second approach computes the power based on Algorithm 4 from Section 5.4. For exponent 0, it returns 1, just as before. If the exponent is even, then the algorithm uses the formula

$$b^n \bmod m = (b^{n/2} \bmod m)^2 \bmod m,$$

and for odd powers, it computes the power using the identity

$$b^n \bmod m = ((b^{\lfloor n/2 \rfloor} \bmod m)^2 \bmod m) \cdot (b \bmod m) \bmod m.$$

Since there are three possibilities, 0, even, or odd, we use the **elif** clause as part of an **if** statement. Short for “else if,” an **elif** clause contains a condition and a statement sequence. The **elif** condition is only checked when the **if** condition and any other previous **elif** conditions failed. Note that **even** is a Maple type, so we can use the **type** command to form the condition.

Here is the implementation of Algorithm 4.

```

1 power2 := proc(b::integer, n::nonnegint, m::posint)
2   if n=0 then
3     return 1;
4   elif type(n, even) then
5     return modp(power2(b, n/2, m)^2, m);
6   else
7     return modp(modp(power2(b, floor(n/2), m)^2, m)*b, m);
8   end if;
9 end proc;
```

We apply **power2** to $3^6 \bmod 7$ as well.

```
> power2(3, 6, 7)
1
```

(5.64)

Comparing Performance of the Algorithms

Now that we have implemented these two algorithms, let us compare their performance on a variety of input values.

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

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

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

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

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

```
> max(times1)
0.003
(5.65)
```

```
> max(times2)
0.001
(5.66)
```

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

Now, we graph the time results. We define **P1** and **P2** to be the plots for **times1** and **times2**.

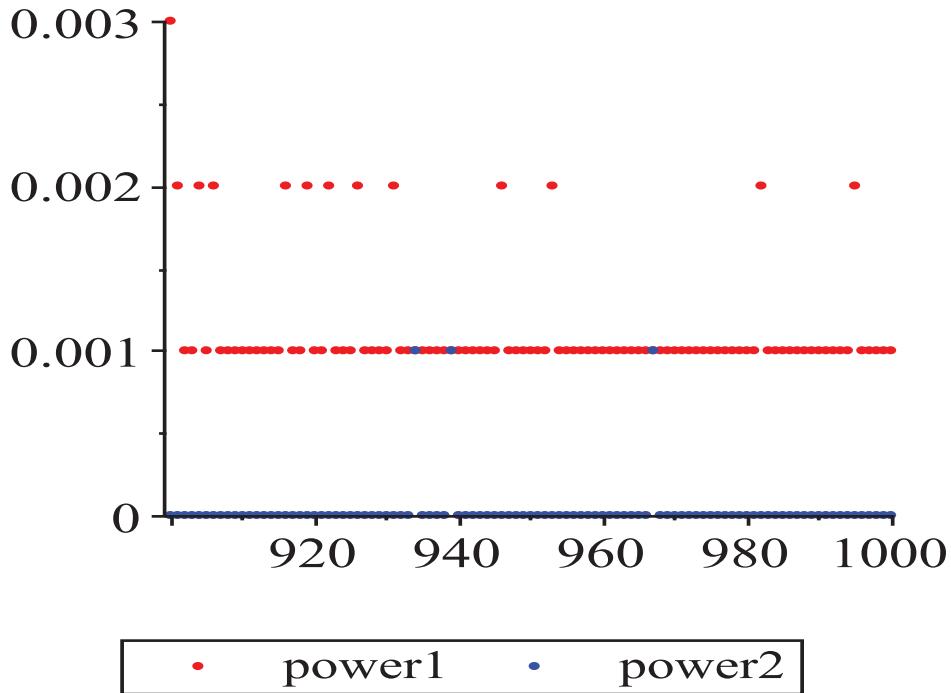
```
> P1 := plot(N, times1, style = point, symbol = solidcircle,
    view = [900..1000, 0..0.003], color = red, symbolsize = 8,
    legend = "power1"):

> P2 := plot(N, times2, style = point, symbol = solidcircle,
    view = [900..1000, 0..0.003], color = blue, symbolsize = 7,
    legend = "power2"):
```

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

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

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



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

Recursion and Iteration

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

Recursive Factorial

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

```

1 factorialR := proc (n : : nonnegint)
2   if n=0 then
3     return 1;
4   else
5     return n * factorialR(n-1);
6   end if;
7 end proc;
```

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

```
> factorialR(10)
3 628 800
```

(5.67)

```
> 10!
3 628 800
```

(5.68)

Iterative Factorial

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

```
1 factorialI := proc (n :: nonnegint)
2   local f, i;
3   f := 1;
4   for i from 1 to n do
5     f := f * i;
6   end do;
7   return f;
8 end proc;
```

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

```
> factorialI(10)
3 628 800
```

(5.69)

Comparing Recursion and Iteration

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

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

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

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

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

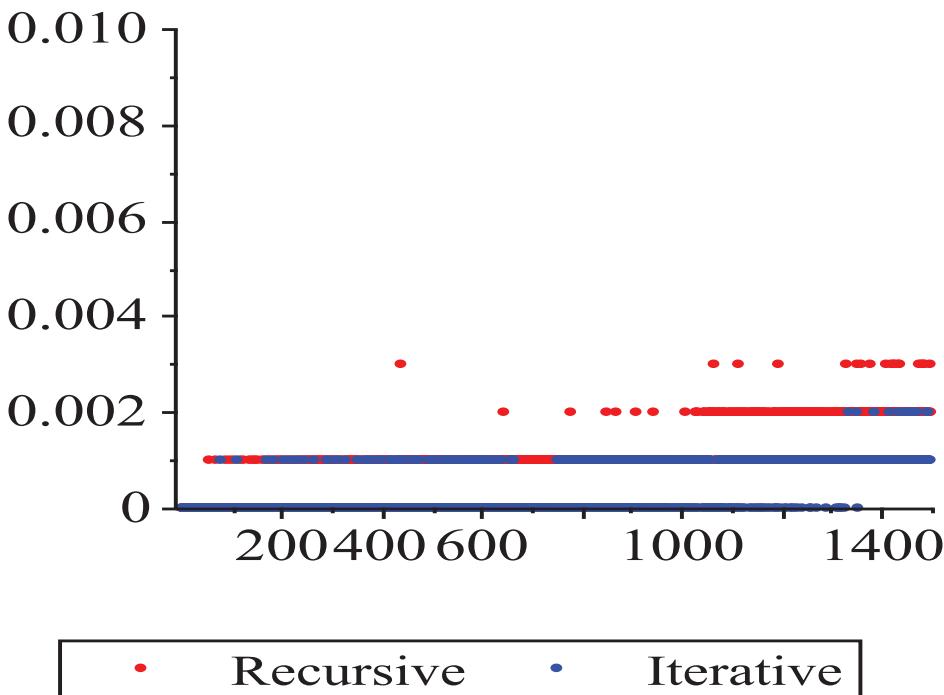
```

> PlotR := plot(M,timesR,style = point,symbol = solidcircle,
  view = [1 ..1500,0 ..0.01],color = red,symbolsize = 8,
  legend = "Recursive"):

> PlotI := plot(M,timesI,style = point,symbol = solidcircle,
  view = [1 ..1500,0 ..0.01],color = blue,symbolsize = 7,
  legend = "Iterative"):

> plots[display](PlotR,PlotI)

```



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

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

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

The iterative procedure involves two assignments (the explicit assignment of **f** and the implicit assignment of **i**) absent from the recursive procedure. However, two assignments are *much* less

costly than a procedure call. Recursive procedure calls require Maple to perform several operations in the background, both in order to execute the recursive call and to keep track of where Maple is in the chain of recursive calls. All of which takes time and memory that the iterative approach does not require.

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

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

Merge Sort

We conclude this section by implementing merge sort. Note that Maple’s **sort** command for sorting lists is an implementation of merge sort.

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

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

```
1 mergesort := proc (L::list(integer))
2   local m, L1, L2;
3   if nops(L) > 1 then
4     m := floor(nops(L)/2);
5     L1 := L[1..m];
6     L2 := L[(m+1)..-1];
7     return merge(mergesort(L1), mergesort(L2));
8   else
9     return L;
10  end if;
11 end proc;
```

Implementing Merge

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

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

The main work of **merge** is in a while loop conditioned on both lists being nonempty. Within this while loop there are two if statements. The first if statement will implement the instruction

to “remove smaller of first elements of L_1 and L_2 from its list; put it at the right end of L ” from Algorithm 10. This if statement will test whether the first element of L_1 is smaller than the first element of L_2 . If so, then the first element is removed from L_1 and added to L . If not, in the else clause, the first element of L_2 is moved to L .

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

Here is the implementation of **merge**.

```

1 merge := proc(l1 : :list(integer), l2 : :list(integer))
2   local L, L1, L2;
3   L1 := l1;
4   L2 := l2;
5   L := [];
6   while L1 <> [] and L2 <> [] do
7     if L1[1] < L2[1] then
8       L := [op(L), L1[1]];
9       L1 := L1[2..-1];
10    else
11      L := [op(L), L2[1]];
12      L2 := L2[2..-1];
13    end if;
14    if L1 = [] then
15      L := [op(L), op(L2)];
16      L2 := [];
17    elif L2 = [] then
18      L := [op(L), op(L1)];
19      L1 := [];
20    end if;
21  end do;
22  return L;
23 end proc;
```

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

```
> mergesort([7,4,1,5,2,3,6])
[1,2,3,4,5,6,7] (5.70)
```

Tracing Merge Sort

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

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

```
> mergesort([3,1,2])
```

```

{--> enter mergesort, args = [3, 1, 2]
  m := 1
  L1 := [3]
  L2 := [1,2]

{--> enter mergesort, args = [3]
<-- exit mergesort (now in mergesort) = [3] }
{--> enter mergesort, args = [1, 2]
  m := 1
  L1 := [1]
  L2 := [2]

{--> enter mergesort, args = [1]
<-- exit mergesort (now in mergesort) = [1] }
{--> enter mergesort, args = [2]
<-- exit mergesort (now in mergesort) = [2] }
{--> enter merge, args = [1], [2]
  L1 := [1]
  L2 := [2]
  L := []
  L := [1]
  L1 := []
  L := [1,2]
  L2 := []

<-- exit merge (now in mergesort) = [1, 2] }
<-- exit mergesort (now in mergesort) = [1, 2] }
{--> enter merge, args = [3], [1, 2]
  L1 := [3]
  L2 := [1,2]
  L := []
  L := [1]
  L2 := [2]
  L := [1,2]
  L2 := []
  L := [1,2,3]
  L1 := []

<-- exit merge (now in mergesort) = [1, 2, 3] }
<-- exit mergesort (now at top level) = [1, 2, 3] }
[1,2,3]                                         (5.71)

```

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

> *untrace(mergesort, merge)*:

5.5 Program Correctness

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

merge

For convenience, we repeat the definition of **merge**. In addition, we have added comments to indicate that we have broken the procedure into three segments: S_1 , S_2 , and S_3 .

```
1 merge := proc(l1 : :list(integer), l2 : :list(integer))
2   local L, L1, L2;
3   # begin S1
4   L1 := l1;
5   L2 := l2;
6   L := [];
7   # end S1
8   while L1 <> [] and L2 <> [] do
9     # begin S2
10    if L1[1] < L2[1] then
11      L := [op(L), L1[1]];
12      L1 := L1[2..-1];
13    else
14      L := [op(L), L2[1]];
15      L2 := L2[2..-1];
16    end if;
17    # end S2
18    # begin S3
19    if L1 = [] then
20      L := [op(L), op(L2)];
21      L2 := [];
22    elif L2 = [] then
23      L := [op(L), op(L1)];
24      L1 := [];
25    end if;
26    # end S3
27  end do;
28  return L;
29 end proc:
```

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

Let q_1 be the proposition that $L_1 = l_1$ and $L_2 = l_2$ and L is the empty list. It is clear that $p \{ S_1 \} p \wedge q_1$.

Define the following propositional variables:

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

We claim that $p \wedge q_1 \rightarrow r$. Assume $p \wedge q_1$. That is, l_1 and l_2 are ordered, nonempty, and disjoint lists of integers. In addition, $L_1 = l_1$, $L_2 = l_2$, and L is empty. Then r_1 and r_2 hold since a list is a sublist of itself. Proposition r_3 holds since L is empty and thus is ordered vacuously. That r_4 is true follows from substituting l_1 and l_2 and $[]$ for L_1 , L_2 , and L . Moreover, r_5 is vacuous since L is empty. From $p \{S_1\} p \wedge q_1$ and $(p \wedge q_1) \rightarrow r$, we have $p \{S_1\} r$.

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

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

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

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

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

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

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

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

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

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

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

mergesort

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

```

1 mergesort := proc (L::list(integer))
2   local m, L1, L2;
3   if nops(L) > 1 then
4     m := floor(nops(L)/2);
5     L1 := L[1..m];
6     L2 := L[(m+1)..-1];
7     return merge(mergesort(L1), mergesort(L2));
8   else
9     return L;
10  end if;
11 end proc;
```

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

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

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

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

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

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

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

Solutions to Computer Projects and Computations and Explorations

Computer Projects 2

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

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

To begin, we present a recursive definition of the set of well-formed formulae on the symbols.

Basis step: x, y , and z are well-formed formulae.

Recursive step: If F and G are well-formed formulae, then so are: $-F$, $F + G$, $F - G$, $F \cdot G$, and F/G .

Note that we will fully parenthesize the well-formed formulae so as to avoid ambiguity, but parentheses will not be considered symbols.

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

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

Generating Formulae

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

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

```

1 buildWFFs := proc (S : :set)
2   local T, f, g, o;
3   T := S;
4   for f in S do
5     T := T union {cat ("-", f, "")} ;
6   end do;
7   for o in [ "+", "*", "/", "-" ] do
8     for f in S do
9       for g in S do
10         T := T union {cat ("(", f, o, g, ")")};
11       end do;
12     end do;
13   end do;
14   return T;
15 end proc:
```

Let us confirm that this works as expected by applying it to the basis set $\{\text{"x"}, \text{"y"}, \text{"z"}\}$.

```
> buildWFFs ({“x”, “y”, “z”})
{“(−x)”, “(−y)”, “(−z)”, “(x * x)”, “(x * y)”, “(x * z)”, “(x + x)”,  

 “(x + y)”, “(x + z)”, “(x − x)”, “(x − y)”, “(x − z)”, “(x/x)”, “(x/y)”,  

 “(x/z)”, “(y * x)”, “(y * y)”, “(y * z)”, “(y + x)”, “(y + y)”, “(y + z)”,  

 “(y − x)”, “(y − y)”, “(y − z)”, “(y/x)”, “(y/y)”, “(y/z)”, “(z * x)”,  

 “(z * y)”, “(z * z)”, “(z + x)”, “(z + y)”, “(z + z)”, “(z − x)”, “(z − y)”,  

 “(z − z)”, “(z/x)”, “(z/y)”, “(z/z)”, “x”, “y”, “z”} (5.72)
```

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

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

```

1 AllWFFs := proc (m::posint)
2   local S;
3   S := { "x", "y", "z" } ;
4   from 1 to m do
5     S := buildWFFs (S) ;
6   end do;
7   return S;
8 end proc;

```

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

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

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

We suppressed the output since the output would be lengthy.

> *nops* (*AllWFF3*)

7101

(5.73)

Here is the set of every 300th formula.

> *AllWFF3*[[*seq* (*i*, *i* = 1 .. 7101, 300)]]

```

{ “(−(−x))”, “((−y) − (x * x))”, “((x * x) + (x − x))”,
  “((x * z) * (y * x))”, “((x + x)/(y − x))”, “((x + z) − (z * x))”,
  “((x − y) + (z − x))”, “((x/x) * x)”, “((x/z) * (x * x))”,
  “((y * x)/(x − x))”, “((y * z) − (y * x))”, “((y + y) + (y − x))”,
  “((y − x) * (z * x))”, “((y − y)/(z − x))”, “((y/x) − x)”,
  “((y/z) − (x * x))”, “((z * y) + (x − x))”, “((z + x) * (y * x))”,
  “((z + y)/(y − x))”, “((z − x) − (z * x))”, “((z − z) + (z − x))”,
  “((z/y) * x)”, “(−(x + x))”, “(y − (z − x))”}

```

(5.74)

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

Pruning the Set

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

> *length* (“abcde”)

5

(5.75)

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

```

1  countSymbols := proc (WFF :: string)
2      local count, i;
3      count := length (WFF);
4      for i in WFF do
5          if i = "(" or i = ")" then
6              count := count - 1;
7          end if;
8      end do;
9      return count;
10 end proc;
```

For example, the number of symbols in “ $((z - x) - (z \cdot x))$ ” is:

> *countSymbols* (“ $((z - x) - (z \cdot x))$ ”) 7 (5.76)

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

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

> *AllWFF3pruned* := *select*(*f* → *evalb*(*countSymbols*(*f*) ≤ 3), **AllWFF3**):

This results in a much more manageable number of results.

> *AllWFF3pruned*
{“ $(-(-x))$ ”, “ $(-(-y))$ ”, “ $(-(-z))$ ”, “ $(-x)$ ”, “ $(-y)$ ”, “ $(-z)$ ”, “ $(x * x)$ ”,
“ $(x * y)$ ”, “ $(x * z)$ ”, “ $(x + x)$ ”, “ $(x + y)$ ”, “ $(x + z)$ ”, “ $(x - x)$ ”, “ $(x - y)$ ”,
“ $(x - z)$ ”, “ (x/x) ”, “ (x/y) ”, “ (x/z) ”, “ $(y * x)$ ”, “ $(y * y)$ ”, “ $(y * z)$ ”,
“ $(y + x)$ ”, “ $(y + y)$ ”, “ $(y + z)$ ”, “ $(y - x)$ ”, “ $(y - y)$ ”, “ $(y - z)$ ”, “ (y/x) ”,
“ (y/y) ”, “ (y/z) ”, “ $(z * x)$ ”, “ $(z * y)$ ”, “ $(z * z)$ ”, “ $(z + x)$ ”, “ $(z + y)$ ”,
“ $(z + z)$ ”, “ $(z - x)$ ”, “ $(z - y)$ ”, “ $(z - z)$ ”, “ (z/x) ”, “ (z/y) ”, “ (z/z) ”,
“ x ”, “ y ”, “ z ”} (5.77)

Computations and Explorations 2

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

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

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

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

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

(5.78)

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

> fib5
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

(5.79)

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

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

1	testFib5 := proc (n :: posint)
2	local F;
3	uses combinat;
4	F := fibonacci(n);
5	if (n mod 5 = 0) and (F mod 5 <> 0) then
6	print(cat("n = ", n, " is divisible by 5, but Fn = ", F, " is not."));
7	elif (n mod 5 <> 0) and (F mod 5 = 0) then
8	print(cat("Fn = ", F, " is divisible by 5, but n = ", n, " is not."));
9	end if;
10	end proc;

In order to check as many Fibonacci numbers as possible, while not spending too much time on the process, we will apply the **timelimit** function. The first argument is the time, in seconds, that the expression should be allowed to run. The second argument is an expression to be evaluated. In this case, the expression will be an infinite loop that applies **testFib5**. We make the

loop infinite by declaring it to be a for loop while omitting the **to** keyword. Maple will allow the loop to run until the time is expired, at which point an error will be raised. Specifically, a “time expired” error. By catching the error in a try...catch block, we can have Maple report the largest value it was able to test in the given time. Note that between the **catch** keyword and the required colon, we enter the string “time expired” so that only that particular kind of error is caught.

The loop needs to be enclosed in a procedure in order to avoid the syntax errors that would arise trying to make a loop definition an argument to the **timelimit** function. The value being tested is declared global in the procedure so as to allow it to be referenced in the **catch**. Note that we subtract one from the value of the loop variable since we cannot be certain whether the current value was in fact checked or if it was aborted.

```

1 testFibInfinite := proc()
2   global fibvalue;
3   for fibvalue from 1 do
4     testFib5(fibvalue):
5   end do:
6 end proc:
```

Below, we allow the test to run for 3 seconds.

```

> try
  timelimit(3, testFibInfinite())
  catch "time expired" :
    print(cat("Checked through ", fibvalue - 1))
  end try;
"Checked through 35 439"                                (5.80)
```

How many Fibonacci numbers can be checked will depend on your computer. Proving the conjecture, as well as forming and proving conjectures for 7 and 11, is left to the reader.

Exercises

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

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

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

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

Exercise 5. The results of Supplementary Exercises 4 and 5 of the text suggest a more general conjecture. Use Maple’s **sum** command to produce evidence for this conjecture.

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

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

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

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

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

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

Exercise 12. Write a procedure to compute the number of partitions of a positive integer (see Exercise 49 in Section 5.3 of the text).

Exercise 13. Write a procedure to compute Ackerman's function (see the preamble to Exercise 450 in Section 5.3).

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

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

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

Exercise 17. Compare the performance of your implementations of Algorithm 5 and Algorithm 6 as follows: for a variety of values of n , let L be the list of integers from 1 to n . Randomly choose 100 integers between 1 and n and measure the average of the times taken for each algorithm to find the randomly chosen integers. Graph n versus the average times. (See Section 4.6 of this manual for a description of the **rand** command for generating random integers.)

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

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

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

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

Exercise 22. Write a procedure that models Knuth's up-arrow notation (see Writing Project 6). Be careful when evaluating this function, as the value grows very quickly.

6 Counting

Introduction

This chapter presents a variety of techniques that are available in Maple for counting a diverse collection of discrete objects, including combinations and permutations of finite sets. Objects can be counted using formulae or by using algorithms to list the objects and then directly counting the size of the list.

Most of the Maple commands relevant to this chapter dwell in the **combinat** package. Since this package will be used extensively in this chapter, we load it now.

```
> with(combinat):
```

Advanced readers may wish to explore the **combstruct** package on their own. This package will not be discussed in this manual as the **combinat** package is sufficient for the material covered here. The **combstruct** package contains commands related to combinatorial structures, and can be thought of as generalizing some of the commands provided by **combinat**.

6.1 The Basics of Counting

In this section, we will see how Maple can be used to perform the computations needed to solve basic counting problems. We will begin by looking at some examples. We will discuss computations involving large integers. Then, we will see how the principles of counting can be used to count the number of operations used by a Maple procedure. This section concludes by using Maple procedures to solve counting problems by enumerating all the possibilities.

Basic Examples

We begin with two basic examples to demonstrate the use of some useful Maple commands.

Counting One-to-One Functions

Recall Example 7 from Section 6.1 of the text. This example calculated that the number of one-to-one functions from a set with m elements to a set with n elements is

$$n \cdot (n - 1) \cdot (n - 2) \cdots (n - m + 1).$$

Note that we can rewrite this using product notation as

$$\prod_{i=0}^{m-1} n - i.$$

For small values of m , it is easy to enter this product in Maple. For instance, the statement below computes the number of one-to-one functions from a set of four elements to a set of 20 elements.

```
> 20 · 19 · 18 · 17
```

116 280

(6.1)

For larger values of m , it is more convenient to use the **mul** command. The **mul** command is used to multiply a sequence of values. Its syntax is similar to the syntax for **seq**. The first argument is an expression in terms of a variable (e.g., **i**) that evaluates to the values that are to be multiplied together. The second argument, in the most common usage, has the form **i=a..b** with the range **a..b** representing the values the variable is to take.

For example, we can recompute the number of one-to-one functions from a set of 20 elements to a set of four elements as follows.

```
> mul(20 - i, i = 0 .. 3)
116 280
```

(6.2)

The second argument indicates that the index variable **i** will be assigned the integers 0, 1, 2, and 3. These are then substituted into the first argument, **20-i**, producing the values 20, 19, 18, and 17, which are multiplied together.

We can easily compute the number of one-to-one functions from a set of 12 elements to a set of 300 elements.

```
> mul(300 - i, i = 0 .. 11)
425 270 752 192 695 317 567 218 560 000
```

(6.3)

Computer Passwords

Example 16 from Section 6.1 describes a computer system in which each user has a password that must be between six and eight characters long consisting of uppercase letters or digits. In addition, each password must contain at least one digit.

The solution to the example describes how to calculate this. For each password length, 6, 7, or 8, the number of passwords are

```
> P[6] := 366 - 266
P[6] := 1 867 866 560
```

(6.4)

```
> P[7] := 367 - 267
P[7] := 70 332 353 920
```

(6.5)

```
> P[8] := 368 - 268
P[8] := 2 612 282 842 880
```

(6.6)

Thus, the total number of possible passwords is

```
> P := P[6] + P[7] + P[8]
P := 2 684 483 063 360
```

(6.7)

We can use the **add** command to make this calculation a bit easier. The **add** command has the same syntax as **mul**, but is used to add its arguments rather than multiply them.

```
> add(36i - 26i, i = 6 .. 8)
2 684 483 063 360
```

(6.8)

This makes it easier to compute the number of valid passwords for larger ranges. For instance, it is common to require passwords to be between 8 and 64 characters. If we retain the rules that the password be uppercase letters or numbers and include at least one number, then the total number of possible passwords is calculated with the statement below.

```
> add (36i - 26i, i = 8 .. 64)  
4 126 620 251 202 066 828 551 300 229 999 712 527 129 717 696 057 592 889 450 099 703 \  
219 511 292 080 116 014 779 039 630 823 409 920  
(6.9)
```

Working with Large Integers

Maple's computational engine is able to work with arbitrarily large integers, subject only to the limitations imposed by the computer's memory and speed.

DNA

In Example 11, the text provides a brief description of DNA and concludes that there are at least 4^{10^8} different possible sequences of bases in the DNA for complex organisms.

To have Maple compute this value, we just enter the statement. (Note that since exponentiation is not associative, parentheses are required in this calculation.)

```
> DNAsequences := 4108  
[Length of output exceeds limit of 1 000 000] (6.10)
```

Maple reports that the length of the output exceeds a limit. This does not imply that Maple has not computed it, it only means that displaying the integer would require excessive space.

Maple has computed and stored the exact value and we can use this value in further computations. For example, we can find the last three digits of the number by computing the result modulo 1000.

```
> DNAsequences mod 1000  
376 (6.11)
```

We can calculate the number of digits in the result by applying the **ilog10** command. This command computes an integer approximation of the base 10 logarithm of its argument. Specifically, as long as the argument x is a real number, the result of **ilog10(x)** is an integer r such that $10^r < |x| < 10^{r+1}$. In particular, the result is one less than the number of digits of x .

Apply **ilog10** to the result for the number of possible sequences of bases in DNA.

```
> ilog10(DNAsequences)  
60 205 999 (6.12)
```

Remember that the approximation 4^{10^8} was a lower bound. In other words, the minimum number of possible sequences of bases in the DNA of a complex organism has over 60 million digits.

Suppose you wanted to print this number. Using a typical fixed-width 12-point font and 1-inch margins, you can fit about 64 digits in each line and 45 lines on a page. With these parameters, it would require

```

> evalf( (6.12) )
64 · 45
20 904.86076

```

(6.13)

pages to print the entire number.

Variable Names in Maple

Example 15 in Section 6.1 of the text calculated that in one version of the programming language BASIC, there were 957 different names for variables. We will calculate the number of possible names in Maple.

In Maple, the basic form of a name is a letter possibly followed by additional letters, digits, or underscores. Maple considers uppercase distinct from lowercase. There are 52 uppercase or lowercase letters that can be used as the first character. Including the underscore and the 10 digits, there are 63 possibilities for each character following the first.

The maximum length of a name depends on the computer. On a 32-bit system, the maximum length is 268 435 439 characters. On a 64-bit machine, the maximum is over 34 billion characters long. It is unlikely that you would ever use such a name, but you could if you wanted to.

How many possible names are there? The total number of possible names on a 32-bit system is

$$\sum_{i=0}^{268\,435\,438} 52 \cdot 63^i.$$

Attempting to calculate this would be rather time consuming.

We will ask a more reasonable question. How many names have at most 15 characters? To answer this question, we use the **add** command, as we did above. The formula is $52 \cdot 63^i$ where the index variable i represents the number of characters in the name beyond the first and ranges from 0 to 14.

```

> add( 52 · 63i, i = 0 .. 14)
819 822 618 169 347 817 599 853 876

```

(6.14)

We see that even limiting ourselves to a maximum of 15 characters, there are over 800 septillion distinct Maple names. (The number above is slightly inaccurate because it does not exclude Maple keywords and other protected names that you are not allowed to assign values to. Of course, those are relatively insignificant.)

Counting Operations in a Procedure

Next, we consider an example of counting the number of operations performed by a procedure. In Example 9 in Section 6.1 of the textbook, it is shown that the number of times that the innermost statement in a nested for loop is executed is the product of the number of iterations of each loop.

As an example of this, consider the **MakePostage** procedure from Section 5.1 of this manual. Recall that the purpose of this procedure was to determine the numbers of stamps of two given denominations that are required to make a given amount of postage. Here is the procedure definition again.

1	MakePostage := proc (A : : posint, B : : posint, postage : : posint)
2	local a, b;
3	for a from 0 to floor(postage/A) do

```

4   for b from 0 to floor(postage/B) do
5     if A*a + B*b = postage then
6       return [a, b];
7     end if;
8   end do;
9
10  end do;
11  return FAIL;
end proc:

```

We will count the number of multiplications and additions that this procedure requires in the worst case. The **return** command means that once the procedure has found a way to make the desired amount of postage, execution is immediately terminated. Therefore, knowing the number of iterations used for a particular input value is equivalent to knowing the output of the procedure. If there was a formula for that, we would not need the procedure. By considering the worst-case scenario, we can get an idea of the complexity of the algorithm without having to execute the procedure.

The worst-case scenario, that is, the situation that requires the most number of iterations of the loop, occurs when the desired postage cannot be made. In this case, the outer loop variable will range from 0 to $\left\lfloor \frac{\text{postage}}{A} \right\rfloor$ and the inner loop will range from 0 to $\left\lfloor \frac{\text{postage}}{B} \right\rfloor$. Thus, the number of times the if statement is executed is $\left(\left\lfloor \frac{\text{postage}}{A} \right\rfloor + 1 \right) \left(\left\lfloor \frac{\text{postage}}{B} \right\rfloor + 1 \right)$. Therefore, in the worst case, the **MakePostage** procedure requires that number of additions and twice as many multiplications.

Counting by Listing All Possibilities

At the end of Section 6.1, the text discusses using tree diagrams to solve counting problems. Tree diagrams provide a visual way to organize information so you can be sure that you arrive at all possible results. We will not, in this section, implement trees, as that is the focus of Chapter 11. The goal of a tree diagram is to list all of the possibilities. In this subsection, we will consider two problems that can be solved by using Maple procedures to list all possibilities.

Subsets

For the first example, we consider the following question: how many subsets of the set of the integers 1 through 10 have sums less than 15? (This is similar to Exercise 69 in Section 6.1.)

To solve this problem, we will consider all of the possible subsets and count those that satisfy the condition.

In order to generate all of the possible subsets of $\{1, 2, \dots, 10\}$, we use the **subsets** command, first introduced in Section 2.1 of this manual. The **subsets** command is part of the **combinat** package.

The **subsets** command accepts one argument: the set (or list) whose subsets are to be generated. It returns a table with two entries. The **nextvalue** entry is a procedure that takes no arguments and that, when executed, returns a subset. The **finished** entry is a Boolean value that is initially false but is set to true once the **nextvalue** procedure has returned the final subset.

Here is an example of using the **subsets** command to print all of the subsets of the set $\{1, 2\}$.

```
> subs12 := subsets({1, 2}):  
  
> while not subs12[finished] do  
    print(subs12[nextvalue]())  
end do  
  
∅  
{1}  
{2}  
{1, 2} (6.15)
```

The assignment to the result of **subsets** establishes **subs12** as the name storing the table. Then, **subs12[finished]** accesses the Boolean value of the **finished** entry from the table. As this is false until all subsets have been produced, its logical negation is a useful control for a while loop. To produce the subsets, **subs12[nextvalue]()** calls the **nextvalue** procedure, which returns the “next” subset.

We will use **subsets** to solve the problem of counting the number of subsets of $\{1, 2, 3, \dots, 10\}$ whose sum is less than 15. Instead of printing all subsets as the simple loop above does, we will test the subset produced by **nextvalue** to see if its sum is less than 15.

To calculate the sum of the elements of a set, we use the **add** command. To add the elements of a list or set, the **add** command will accept the list or set as the sole argument. For example, to add the elements of the set $\{1, 3, 7, 8\}$, we can execute the following expression.

```
> add({1, 3, 7, 8})  
19 (6.16)
```

Since the problem, as stated, was to count the number of subsets with sum less than 15, we use a variable that is initialized to 0 and incremented each time a subset has the desired property. We generalize the problem a bit and write a procedure that accepts a set of positive integers and a target value and counts all the subsets of the given set whose elements sum to a value less than the target.

```
1 SubsetSumCount := proc (S :: set (posint), target :: posint)  
2   uses combinat;  
3   local count, P, s, x;  
4   count := 0;  
5   P := subsets (S);  
6   while not P [finished] do  
7     s := P [nextvalue] ();  
8     x := add (s);  
9     if x < target then  
10       count := count + 1;  
11     end if;  
12   end do;  
13   return count;  
14 end proc;
```

Applying the procedure to $\{1, 2, 3, \dots, 10\}$ and 15 will answer the original question.

> *SubsetSumCount*($\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, 15)

99

(6.17)

Bit Strings

For the second example, we will consider a problem similar to Example 21. How many bit strings of length 10 do not have three consecutive 1s?

We could use an approach similar to the previous example and produce all bit strings of length 10 and then count the number that do not contain three consecutive 1s. However, the solution to Example 22 of Section 6.1, and especially Figure 4, suggests a more efficient solution. Instead of creating all the possible bit strings, we can build them in such a way as to only create those that satisfy the limitation on the number of consecutive 1s.

To solve this problem, we will use a recursive algorithm. The basis step will be the set consisting of all bit strings of length 2 (these cannot have three consecutive 1s). In the recursive step, the new version of the set will be constructed as follows. For each bit string in the previous set, we will append a 0. In addition, we will append a 1 to all of the bit strings whose last two bits are not both 1.

For this problem, we will model bit strings as lists of 0s and 1s. The algorithm described above will be implemented as two procedures. The first procedure will be responsible for extending a particular bit string. That is, given a bit string (a list of 0s and 1s), it will return either the two bit strings obtained by appending a 0 and by appending a 1, or it will return the single bit string formed by appending a 0 if appending a 1 would result in three consecutive 1s. The second procedure will apply the first to an entire set of bit strings.

First, we implement the procedure that extends a single bit string. The parameter to this procedure will be a single bit string. The procedure will first create a new bit string by appending a zero to the input.

Then, the procedure will test the last two elements of the original bit string to determine if they are both ones. We will accomplish this test by extracting the sublist consisting of the last two entries with the selection operator: $L[-2..-1]$. We can then compare the result against the list $[1, 1]$. If those lists are equal, that is, the last two bits are both 1, then the procedure only returns the list obtained by adding 0. Otherwise, it will create the list with 1 added and return both extended bit strings as a sequence.

Here is the implementation.

```
1 AddBit := proc (L : :list)
2   local L0, L1;
3   L0 := [op(L), 0];
4   if L[-2..-1] = [1, 1] then
5     return L0;
6   else
7     L1 := [op(L), 1];
8     return L0, L1;
9   end if;
10 end proc;
```

We test this procedure on two examples: $[1, 0, 1, 1]$ should produce only $[1, 0, 1, 1, 0]$, while applying the procedure to that result should produce $[1, 0, 1, 1, 0, 0]$ and $[1, 0, 1, 1, 0, 1]$.

> *AddBit* ([1, 0, 1, 1])
[1, 0, 1, 1, 0] (6.18)

> *AddBit* (%)
[1, 0, 1, 1, 0, 0], [1, 0, 1, 1, 0, 1] (6.19)

Now, we write the main procedure. It will accept as input a positive integer n representing the length of the bit strings to be output. In case this value is 2, it will return the four bit strings of length 2. For values of n larger than 2, it will recursively call itself on $n - 1$ and store the result of the recursion as S . It then initializes a new list T to the empty set. Finally, it loops through the set S , applying **AddBit** to each member and adding the result to T .

Here is the implementation.

```

1 FindBitStrings := proc(n::nonnegint)
2   local S, s, T;
3   if n = 2 then
4     return {[0,0],[0,1],[1,0],[1,1]};
5   else
6     S := FindBitStrings(n-1);
7     T := {};
8     for s in S do
9       T := T union {AddBit(s)};
10    end do;
11    return T;
12  end if;
13 end proc;
```

Applying the procedure to 10 and using **nops** will give us the number of bit strings of length 10 that do not include three successive 1s.

> *nops* (*FindBitStrings* (10))
504 (6.20)

6.2 The Pigeonhole Principle

In this section, we will see how Maple can be used to help explore two problems related to the pigeonhole principle: finding consecutive entries in a sequence with a given sum and finding increasing and decreasing subsequences.

Before considering those two problems, however, recall the **ceil** command. This command calculates the ceiling of an expression. For example, the solution to Example 8 in the text indicates that the minimum number of area codes needed to assign different phone numbers to 25 million phones is $\left\lceil \frac{25\,000\,000}{8\,000\,000} \right\rceil$. In Maple, this can be computed by the following statement.

$$> \text{ceil}\left(\frac{25000000}{8000000}\right)$$

4

(6.21)

Consecutive Entries with a Given Sum

Example 10 in Section 6.2 describes the solution to the following problem. In a month with 30 days, a baseball team plays at least one game per day but at most 45 games during the month. Then, there must be a period of consecutive days during which the team plays exactly 14 games.

The problem can be generalized. Given a sequence of d positive integers whose sum is at most S , there must be a consecutive subsequence with sum T for any $T < S - d$. We leave it to the reader to prove this assertion.

We will write a procedure that, given a sequence and target sum T , will find the consecutive terms whose sum is T . Our solution will be based on the approach described in the solution of Example 10.

First, we will calculate the numbers a_1, a_2, \dots, a_d with each a_j equal to the sum of the first j terms in the sequence. These values will be stored as a list, **A**. We will calculate these sums using the observation that each one is equal to the previous sum plus the next entry in the sequence.

Then, we will calculate $a_i + T$ for each i and use the **member** command to check to see if this value is in **A**. The **member** command requires two arguments: the first is the element being sought and the second is the list (or set) to be searched. The command returns true if the element is found and false otherwise. The **member** command also accepts an unevaluated (i.e., enclosed in right single quotes) name as a third, optional argument. If the element is found in the list, then this name is assigned to the position of the first occurrence. For example,

> *member*("d", ["a", "b", "c", "d", "e", "f", "g"], 'p')

true

(6.22)

> *p*

4

(6.23)

Finally, if $a_i + T$ is found in the list a_1, a_2, \dots, a_d , say at position j , then we know that $i + 1$ through j are the positions of the consecutive subsequence with the desired sum. The procedure returns the starting and ending positions as well as the subsequence.

Here is the procedure.

```

1 FindSubSum := proc (L :: list (posint) , T :: posint)
2   local A, i, j;
3   A := [L[1]];
4   for i from 2 to nops(L) do
5     A := [op(A), A[i-1]+L[i]];
6   end do;
7   for i from 1 to nops(L) do
8     if member (A[i]+T, A, 'j') then
9       return i+1, j, L[i+1..j];

```

```

10      end if ;
11      end do ;
12      return FAIL ;
13  end proc;

```

We can apply our procedure to the following sequence, representing the number of games a baseball team played on each day of a 30-day month:

2, 1, 3, 1, 1, 3, 1, 1, 1, 3, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1, 3, 1, 1, 1.

As in Example 10, we find the consecutive days during which the team played 14 games.

$$\begin{aligned}
 > & \text{FindSubSum}([2, 1, 3, 1, 1, 3, 1, 1, 1, 1, 3, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], \\
 & [3, 1, 3, 1, 1, 1, 3, 1, 3] \\
 & 6, 13, [3, 1, 1, 1, 1, 3, 1, 3]
 \end{aligned} \tag{6.24}$$

Strictly Increasing Subsequences

Theorem 3 of Section 6.2 asserts that every sequence of $n^2 + 1$ distinct real numbers contains a subsequence of length $n + 1$ that is either strictly increasing or strictly decreasing. We will develop a procedure that will find a longest strictly increasing subsequence.

The Patience Algorithm

To find the longest increasing subsequence, we will use a greedy strategy based on “Patience sorting” (the name refers to the solitaire card game also called Klondike). The idea is as follows. Imagine that the numbers in the sequence are written on cards. The cards are placed in a “deck” in the order they appear in the sequence and with the first element of the sequence on top. Now, play a “game” using the deck of cards based on the following rules.

The cards are “dealt” one at a time onto a series of piles on the table. Initially, there are no piles. The top card (the first element in the sequence) is the first card dealt and forms the first pile. To play the next card, check to see if it is less than or greater than the first card. If the second card (the second element of the sequence) is less than the first, then it is placed on the first pile, on top of the first card. If the second card is greater than the first, then it starts a new pile to the right of the first.

The “game” continues in this way. At each step, the table has on it a series of piles. To play the next card, you compare the value on the card to the card on top of the first pile. If the card to be played has a value smaller than the number showing on the first pile, then the new card is placed on top of the first pile. Otherwise, you look at the second pile. If the card being played is smaller than the value on the second pile, it is placed on top of the second pile. Continue in this fashion until either the card has been played or, if it is larger than the top card on every existing pile, then it begins a new pile to the right of all the others.

An illustration is in order. Consider the sequence 12, 18, 7, 11, 16, 3, 20, 17.

Step 1: Play the first entry, 12, as the first pile	12
Step 2: Play the second entry, 18. Since $18 > 12$, 18 starts a new pile.	12 18
Step 3: Play 7. Checking the first pile, note that $7 < 12$, so 7 is played on the first pile.	7 12 18

Step 4: Play 11. Checking the first pile, $11 > 7$, so do not play 11 on first pile. Checking the second pile, $11 < 18$, so play 11 on the second pile.	7 11 12 18
Step 5: Play 16. Checking the first pile, $16 > 7$ so do not play 16 on the first pile. Checking the second pile, $16 > 11$, so 16 begins a third pile.	7 11 12 18 16
Step 6: Play 3. Checking the first pile, $3 < 7$, so 3 is played on the first pile.	3 7 11 12 18 16
Step 7: Play 20. Checking the first pile, $20 > 3$. Checking the second pile, $20 > 11$. Checking the third pile, $20 > 16$, so 20 starts a new pile.	3 7 11 12 18 16 20
Step 8: Play 17. Checking the first pile, $17 > 3$. Checking the second, $17 > 11$. Checking the third, $17 > 16$, but $17 < 20$, so 17 is played on the fourth pile.	3 7 11 17 12 18 16 20

Once the “game” is complete, the length of the longest strictly increasing subsequence is equal to the number of piles.

Now that all of the cards are played, we obtain a strictly increasing subsequence by backtracking. The top card on the final pile is 17, so 17 will be the last entry in the subsequence. When 17 was placed on the pile, the top card on the pile before it was 16 (step 8); so, 16 precedes 17 in the subsequence. When 16 was placed on the third pile, the top card on the second pile was 11 (step 5); so, 11 precedes 16. And when 11 was placed on the second pile, the top card on the first pile was 7; so, 7 is first in the subsequence. Thus, 7, 11, 16, 17 is a strictly increasing subsequence of maximal length.

You are encouraged to “play” through this approach a few times with your own sequences to ensure that you understand the process before continuing on to the implementation of the procedure below. You can apply the **FindIncreasing** procedure defined below to make sure that you are arriving at the same result. Be sure to keep track of what was on top of the previous pile when each number is played, as you need that information in the backtracking stage.

Implementing the Algorithm

We will now implement the Patience algorithm. The given sequence will be input to the procedure as a list. Within the procedure, we need to track three kinds of information.

First, we need to know which “step” we are in, that is, which card is being played. This will be represented by the variable `a` for loop ranging from 1 to the size of the list.

Second, we need to know what cards are on the piles. Specifically, we need to know the top card of each pile. This will be represented as a list, **piles**. When a card is placed on top of an existing pile, we can replace the current value in that position with the selection-assignment syntax **piles[i] := x**; When a new pile is added to the list, we extend the list via the usual syntax **[op(piles),x]**.

Third, in order to backtrack and recover the longest increasing sequence, we need to store, for each member of the sequence, the value that was on the top of the pile to the left of the entry’s pile. For this, we will use a table, **pointers**, whose indices will be the members of the sequence and whose entries will be set to the previous pile’s top card. (We use the name **pointers** for this table because of the similarity to the “linked list” structure used in some programming languages.) For those numbers played in the first pile, the value in the table will be set to **NULL**.

The algorithm consists of two stages. The first stage will be the game stage. After initializing **piles** to the empty list and **pointers** to the empty table, we begin a for loop with loop variable **step** running from 1 to the length of the input sequence **S**. Within this main loop, two tasks are performed.

The first thing that happens within the **step** loop is determining on which pile to play the current card. Note that the value of the current card is accessed by **S[step]**. To determine the proper location for the current card, we do the following.

1. Initialize a variable **whichpile** to 0.
2. Use a for loop from 1 to the current number of piles. Within the for loop, compare the current card to the top of each pile. If the current card is smaller than the value in **piles**, set **whichpile** equal to that pile index.
3. Once the loop exits, check the value of **whichpile**. If it is 0 following termination of the loop, that means a pile was not found for the current card and, thus, the **piles** list must be extended to create a new pile for this card. Otherwise, the value of **whichpile** is not 0 and we update the corresponding entry in **piles** to indicate that the latest card is placed on top in that position.

The second task within the **step** loop is to update the **pointers** table. This also depends on the value of **whichpile**.

- If **whichpile** is 1 or if **piles** only contains 1 entry, then the latest card was played on the first pile and the associated value should be **NULL**.
- If **whichpile** is 0, then the value associated with the latest card is **piles[-2]**, the top card on what had been the last pile but is now the next to last pile. (Note that the previous condition, that **whichpile** is 1 or **piles** has only 1 entry will ensure that this second condition is only tested if **piles** has at least 2 entries and thus **piles[-2]** is a valid selection.)
- Otherwise, the value is **piles[whichpile-1]**.

That concludes the game stage. The second stage is the backtracking stage, which is much simpler. First, access the top card of the last pile with **piles[-1]**, and initialize the maximal increasing list, **iList**, to the list consisting of this value.

Then, we extend **iList** on the left with the entry in the **pointers** table associated to that value. Since we are building the list from right to left, **iList[1]** always contains the most recently added number. So **pointers[iList[1]]** is the new value, which is added via the expression **[pointers[iList[1]], op(iList)]**. Since the cards played in the first pile were associated with **NULL**, we can use a while loop with condition **pointers[iList[1]] <> NULL** to fill the **iList**. At the conclusion of the loop, the procedure returns **iList**.

Here, finally, is the procedure.

```

1 FindIncreasing := proc (S :: list)
2   local piles, pointers, step, whichpile, p, iList;
3   piles := [];
4   pointers := table ();
5   # game stage
6   for step from 1 to nops (S) do
7     # playing the card
8     whichpile := 0;
9     for p from 1 to nops (piles) do
10       if S[step] < piles[p] then
11         whichpile := p;

```

```

12         break;
13     end if ;
14 end do;
15 if whichpile = 0 then
16     piles := [op(piles), S[step]];
17 else
18     piles[whichpile] := S[step];
19 end if ;
20 # update pointers
21 if whichpile = 1 or nops(piles) = 1 then
22     pointers[S[step]] := NULL;
23 elif whichpile = 0 then
24     pointers[S[step]] := piles[-2];
25 else
26     pointers[S[step]] := piles[whichpile-1];
27 end if ;
28 end do;
29 # backtracking stage
30 iList := [piles[-1]];
31 while pointers[iList[1]] <> NULL do
32     iList := [pointers[iList[1]], op(iList)];
33 end do;
34 return iList;
35 end proc;

```

Example 12 from the text involved the sequence 8, 11, 9, 1, 4, 6, 12, 10, 5, 7.

> *FindIncreasing* ([8, 11, 9, 1, 4, 6, 12, 10, 5, 7])
[1, 4, 5, 7]

(6.25)

This is one of the four sequences given in the text.

Connection to the Pigeonhole Principle

Recall that Theorem 3 asserted that every sequence of $n^2 + 1$ distinct real numbers contains a subsequence of length $n + 1$ that is either strictly increasing or strictly decreasing. It may appear that the Patience algorithm has no connection to this theorem or to the pigeonhole principle.

However, the Patience algorithm does in fact suggest a proof of Theorem 3 via the pigeonhole principle. When the Patience algorithm is executed on a list of $n^2 + 1$ distinct real numbers, either there are at least $n + 1$ stacks or there are at most n stacks. If there are $n + 1$ stacks, then there is a strictly increasing subsequence of length $n + 1$.

On the other hand, assume that there are at most n stacks. Take the $n^2 + 1$ values to be the pigeons and the stacks the pigeonholes. When $n^2 + 1$ objects are placed in n boxes, by the generalized pigeonhole principle, there is a box containing at least $\left\lceil \frac{n^2 + 1}{n} \right\rceil = \left\lceil \frac{n^2}{n} + \frac{1}{n} \right\rceil = n + \left\lceil \frac{1}{n} \right\rceil = n + 1$ objects. Hence, some stack has $n + 1$ values. However, the rules of the game ensure that each stack

is a strictly decreasing subsequence, since one value is placed on top of another in a stack only when the second value is lesser than, and appears in the sequence later than, the lower value.

In short, either there are $n + 1$ stacks and hence an increasing subsequence of length $n + 1$ or there is a stack of size $n + 1$ and hence a decreasing subsequence of length $n + 1$.

6.3 Permutations and Combinations

The **combinat** package contains many functions pertaining to counting and generating combinatorial structures. We will be using **combinat** commands extensively in this section.

Permutations

We begin by looking at commands related to permutations of objects.

We have seen in previous chapters the use of the exclamation mark for factorial.

```
> 6!  
720
```

(6.26)

The **factorial** command can be used instead of the exclamation mark if you prefer. Otherwise they are equivalent. (Factorial does not depend on the **combinat** package.)

Counting Permutations

To compute the number of permutations, Maple provides the **numbperm** command. This command can be used in a few different ways.

You can give only one argument, an integer.

```
> numbperm(5)  
120
```

(6.27)

In this case, Maple computes the number of permutations, that is, the number of 5-permutations of a set with five elements. This, of course, is the same as computing $5!$.

Instead of an integer as the only argument, you can instead provide a set or list of objects.

```
> numbperm({“a”, “b”, “c”, “d”, “e”})  
120
```

(6.28)

With this argument, **numbperm** computes the number of permutations of the given set (or list). Since the set had five elements, this result is the same as the previous value.

To compute the number of r -permutations of a set with n objects, use **numbperm** with a second argument indicating the value of r . The number of 4-permutations of a set with seven distinct objects, that is, $P(7, 4)$, is computed by the following command.

```
> numbperm(7, 4)  
840
```

(6.29)

Once again, the first argument could be given as a set or a list of objects instead of the number of objects.

```
> numbperm(["a", "b", "c", "d", "e", "f", "g"], 4)  
840
```

(6.30)

You may wonder what the purpose is of allowing the first argument to **numbperm** to be a set or list rather than requiring it to always be a positive integer. We will explore this more in Section 6.5, but briefly, the reason is that the objects in a list do not have to be different. By giving the first element as a list of not-necessarily distinct objects, **numbperm** will compute the number of arrangements of those objects, taking into account any duplication of elements.

Listing Permutations

To obtain a list of all permutations, Maple provides the **permute** command. The syntax is the same as **numbperm**, but instead of reporting the number of permutations, a list of the permutations is provided.

Once again, the first argument can be an integer or a set or a list. If the first argument is given as an integer n , Maple will display the permutations of the set $\{1, 2, \dots, n\}$. For example, the following command lists the permutations of the first four integers.

```
> permute(4)  
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 4, 2], [1, 4, 2, 3], [1, 4, 3, 2], [2, 1, 3, 4],  
[2, 1, 4, 3], [2, 3, 1, 4], [2, 3, 4, 1], [2, 4, 1, 3], [2, 4, 3, 1], [3, 1, 2, 4],  
[3, 1, 4, 2], [3, 2, 1, 4], [3, 2, 4, 1], [3, 4, 1, 2], [3, 4, 2, 1], [4, 1, 2, 3],  
[4, 1, 3, 2], [4, 2, 1, 3], [4, 2, 3, 1], [4, 3, 1, 2], [4, 3, 2, 1]]
```

(6.31)

Providing a specific list or set as the first argument will produce the permutations of the objects in the given set or list. For example, the permutations of the letters a, b, and c are shown below.

```
> permute({"a", "b", "c"})  
[[{"a", "b", "c"}, {"a", "c", "b"}, {"b", "a", "c"}, {"b", "c", "a"},  
 {"c", "a", "b"}, {"c", "b", "a"}]
```

(6.32)

To obtain the r -permutations instead, you must provide r as the second argument. The following lists the 2-permutations of a set with four distinct objects.

```
> permute(4, 2)  
[[1, 2], [1, 3], [1, 4], [2, 1], [2, 3], [2, 4], [3, 1], [3, 2], [3, 4],  
 [4, 1], [4, 2], [4, 3]]
```

(6.33)

Or you can provide your own objects to be r -permuted.

```
> permute({"a", "b", "c", "d", "e"}, 2)  
[[{"a", "b"}, {"a", "c"}, {"a", "d"}, {"a", "e"}, {"b", "a"}, {"b", "c"},  
 {"b", "d"}, {"b", "e"}, {"c", "a"}, {"c", "b"}, {"c", "d"}, {"c", "e"}, {"d", "a"}, {"d", "b"}, {"d", "c"}, {"d", "e"}, {"e", "a"}, {"e", "b"}, {"e", "c"}, {"e", "d"}]
```

(6.34)

Random Permutations

Maple also provides a command, **randperm**, that will produce a randomly chosen permutation.

Once again, the argument to **randperm** can be an integer or a list or set of objects. If an integer n is given, then **randperm** returns a randomly selected permutation of the set $\{1, 2, \dots, n\}$.

```
> randperm(10)  
[3, 8, 9, 5, 1, 7, 2, 10, 4, 6] (6.35)
```

If a list or set is provided as the first argument, then the random permutation is produced using the elements of the given set or list.

```
> randperm(["a", "b", "c", "d"]  
["d", "a", "c", "b"]) (6.36)
```

Note that the permutation is selected so that each permutation has the same probability of being chosen.

Unlike the **numbperm** and **permute** commands, **randperm** does not accept a second argument. (If one is given, it is ignored.) In order to produce a random r -permutation, you must use one of the following approaches.

Suppose you need a random 4-permutation of the set $\{"a", "b", "c", "d", "e", "f"\}$. Your first choice is to apply the **randperm** command to the set, and then use the selection operator to select the first four elements.

```
> randperm({“a”, “b”, “c”, “d”, “e”, “f”})[1..4]  
[“f”, “b”, “a”, “e”] (6.37)
```

The second option is to first use **randcomb** (described below) to obtain a random subset of size 4 and then apply **randperm** to the randomly chosen subset. This approach will be demonstrated below, after describing the commands relevant to combinations. Both approaches will produce a random r -permutation of the given objects. The second approach is faster to execute, however.

Combinations

The commands related to combinations are very similar to those for permutations.

Counting Combinations

The number of combinations is obtained with the **numbcomb** command. Like **numbperm**, this command accepts one or two arguments and the first argument can be a number or a set or a list.

If you provide only one argument to **numbcomb**, it returns the total number of combinations. For instance,

```
> numbcomb(5)  
32 (6.38)
```

indicates that there are 32 ways to choose some (including all or none) of five objects. In other words, there are 32 subsets of a set of 5 elements.

The first argument can also be a set or a list.

```
> numbcomb({"a","b","c","d","e"})  
32
```

(6.39)

To compute the number of r -combinations, you provide r as the second argument. Once again, the first argument can be a set or list or a positive integer indicating the size of the set the objects are to be drawn from. The following compute $C(52, 5)$ and the number of ways to choose 3 from the set of vowels.

```
> numbcomb(52, 5)  
2 598 960
```

(6.40)

```
> numbcomb({"a","e","i","o","u"}, 3)  
10
```

(6.41)

The text mentions that $C(n, r)$ is also referred to as a binomial coefficient. In Maple, the command **binomial** can also be used to compute $C(n, r)$. For example, $C(52, 5)$ can be computed by:

```
> binomial(52, 5)  
2 598 960
```

(6.42)

Note that **binomial** and **numbcomb** agree when given two nonnegative integers as arguments. However, they are not identical commands. For one, **binomial** requires two arguments which must evaluate to algebraic expressions and cannot accept a list or set as the first argument. On the other hand, **binomial** uses a more general formula that can compute with rational and floating-point arguments. In addition, **binomial** is not part of the **combinat** package and is thus available without needing to load the package.

Listing Combinations

The **choose** command is the combination analog of **permute**.

Given only one argument, **choose** produces all possible combinations of every size. If the argument is a set or list, it will list the subsets or sublists of the argument.

```
> choose({"a","b","c","d"})  
{[], {"a"}, {"b"}, {"c"}, {"d"}, {"a", "b"}, {"a", "c"}, {"a", "d"}, {"b", "c"}, {"b", "d"}, {"c", "d"}, {"a", "b", "c"}, {"a", "b", "d"}, {"a", "c", "d"}, {"b", "c", "d"}, {"a", "b", "c", "d"}}
```

(6.43)

```
> choose(["a","b","c","d"])  
[[], ["a"], ["b"], ["c"], ["d"], ["a", "b"], ["a", "c"], ["a", "d"], ["b", "c"],  
["b", "d"], ["c", "d"], ["a", "b", "c"], ["a", "b", "d"], ["a", "c", "d"],  
["b", "c", "d"], ["a", "b", "c", "d"]]
```

(6.44)

Note that if the argument is a set, the result is a set of sets. On the other hand, if the argument is a list, the result is a list of lists. Many Maple commands accept either sets or lists as arguments, and they typically return the same kind of object they were given. This was not the case for **permute**, because in that case order is relevant, so it must return lists.

If the argument is a nonnegative integer n , it will use $[1, 2, \dots, n]$ by default.

```
> choose(3)
[[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]
```

(6.45)

Note that with only one argument, **choose** behaves very similarly to **powerset**, which was discussed in Section 2.1 of this manual. The **powerset** command accepts only one argument, which can be a set, list, or nonnegative integer. It returns the set of all subsets (or the list of all sublists) just as **choose** does.

```
> powerset({"a", "b", "c", "d"})
{[], {"a"}, {"b"}, {"c"}, {"d"}, {"a", "b"}, {"a", "c"}, {"a", "d"}, {"b", "c"}, {"b", "d"}, {"c", "d"}, {"a", "b", "c"}, {"a", "b", "d"}, {"a", "c", "d"}, {"b", "c", "d"}, {"a", "b", "c", "d"}}
```

(6.46)

With a second argument given to **choose**, you can specify the number of objects to be selected.

```
> choose({{"a", "b", "c", "d"}}, 2)
{{{"a", "b"}, {"a", "c"}, {"a", "d"}, {"b", "c"}, {"b", "d"}, {"c", "d"}}
```

(6.47)

```
> choose(5, 2)
[[1, 2], [1, 3], [1, 4], [1, 5], [2, 3], [2, 4], [2, 5], [3, 4], [3, 5], [4, 5]]
```

(6.48)

The **powerset** command does not allow a second argument.

Random Combinations

The **randcomb** command is used to produce a random combination.

As opposed to **randperm**, **randcomb** requires two arguments. The first can be a set or list of objects or a positive integer, and the second must be a nonnegative integer.

```
> randcomb({{"a", "b", "c", "d", "e"}}, 3)
{{"a", "c", "e"}}
```

(6.49)

```
> randcomb(5, 3)
{2, 4, 5}
```

(6.50)

As mentioned earlier, **randcomb** can be combined with **randperm** to produce a random permutation of a specified size. To obtain a random 4-permutation of $\{a, b, c, d, e, f\}$, for example, first use **randcomb** to pick a random 4-combination. Then, apply **randperm** to the selected combination.

```
> randcomb({{"a", "b", "c", "d", "e", "f"}}, 4)
{{"a", "c", "d", "f"}}
```

(6.51)

```
> randperm(%)
["f", "c", "a", "d"]
```

(6.52)

You can combine them into one statement, as follows.

```
> randperm(randcomb({"a","b","c","d","e","f"},4))  
["d","e","a","f"]
```

(6.53)

Circular Permutations

The preamble to Exercise 42 in Section 6.3 describes circular permutations. A circular r -permutation of n people is a seating of r of those n people at a circular table. Moreover, two seatings are considered the same if one can be obtained from the other by rotation.

The exercises ask you to compute the number of circular 3-permutations of 5 people and to arrive at a formula for that number. In this subsection, we will write a procedure to list all of the circular r -permutations of n people. Having such a procedure can help you more easily explore the concept and test your formula.

Rotating a Permutation

The key to listing all circular permutations is to devise a way to test whether two circular permutations are equal. According to the definition, two circular permutations are considered to be equal if one can be obtained from the other by a rotation. If we use a list to represent a permutation, a rotation will consist of moving the first element to the end of the list (or the last to the front).

Here is a procedure that, given a list representing a circular permutation, will rotate the permutation by one position.

```
1 | RotatePerm := proc (P : : list)  
2 |     return [op(2..-1, P), P[1]] ;  
3 | end proc;
```

For example, the seating Abe, Carol, Barbara, is the same as Carol, Barbara, Abe.

```
> RotatePerm(["Abe", "Carol", "Barbara"])  
["Carol", "Barbara", "Abe"]
```

(6.54)

Note that rotations start to repeat.

```
> RotatePerm(%)  
["Barbara", "Abe", "Carol"]
```

(6.55)

```
> RotatePerm(%)  
["Abe", "Carol", "Barbara"]
```

(6.56)

In fact, for an r -permutation, after r rotations, the list will return to its original state.

Equality of Circular Permutations

This observation indicates that, given two r -permutations, we can test to see if they are the same by rotating one of them $r - 1$ times. The procedure below returns true if the two input lists represent the same circular permutation and false otherwise. It first checks equality without performing rotation. Then, using a for loop, it rotates the second list, checking for equality after each rotation.

```

1 CPEquals := proc(L1::list, L2::list)
2   local i, Lr;
3   if L1 = L2 then
4     return true;
5   end if;
6   Lr := L2;
7   for i from 1 to nops(L2)-1 do
8     Lr := RotatePerm(Lr);
9     if L1 = Lr then
10       return true;
11     end if;
12   end do;
13   return false;
14 end proc;

```

We can use it to confirm equality of circular permutations. For example,

> *CPEquals*(["Charles", "Helen", "Dean"], ["Helen", "Dean", "Charles"])
true (6.57)

Listing all Circular Permutations

Now we are prepared to write a procedure that lists all circular permutations. First, we will use the **permute** command to generate all r -permutations of n people. We will initialize the set of all distinct circular permutations to the first element of the list of all permutations. That first permutation is then removed from the list of all permutations.

Within a while loop, consider the first element in the list of all permutations. Use **CPEquals** and a loop to see if the first element is identical to any of the members of the set of circular permutations. If not, add it to the set of circular permutations. In either case, it is deleted from the list of all permutations. This continues until the list of all permutations has been emptied.

Here is the implementation. Our procedure will accept a set as the first argument and r , the number that can be seated at the table, as the second argument.

```

1 AllCP := proc(S::set, r::posint)
2   local allP, allCP, isnew, p;
3   allP := combinat[permute](S, r);
4   allCP := {allP[1]};
5   allP := allP[2..-1];
6   while allP <> [] do
7     isnew := true;
8     for p in allCP do
9       if CPEquals(allP[1], p) then
10         isnew := false;
11         break;
12       end if;
13     end do;
14     if isnew then

```

```

15      allCP := allCP union {allP[1]};
16      end if;
17      allP := allP[2..-1];
18  end do;
19  return allCP;
20 end proc;

```

Note that the **isnew** Boolean is used to track whether or not the current first member of **allP** is new or not.

The following computes the possible circular 3-permutations of the set {"Abe", "Barbara", "Carol", "Dean", "Eve"}.

```

> AllCP({"Abe", "Barbara", "Carol", "Dean", "Eve"}, 3)
{["Abe", "Barbara", "Carol"], ["Abe", "Barbara", "Dean"],
 ["Abe", "Barbara", "Eve"], ["Abe", "Carol", "Barbara"],
 ["Abe", "Carol", "Dean"], ["Abe", "Carol", "Eve"],
 ["Abe", "Dean", "Barbara"], ["Abe", "Dean", "Carol"],
 ["Abe", "Dean", "Eve"], ["Abe", "Eve", "Barbara"],
 ["Abe", "Eve", "Carol"], ["Abe", "Eve", "Dean"],
 ["Barbara", "Carol", "Dean"], ["Barbara", "Carol", "Eve"],
 ["Barbara", "Dean", "Carol"], ["Barbara", "Dean", "Eve"],
 ["Barbara", "Eve", "Carol"], ["Barbara", "Eve", "Dean"],
 ["Carol", "Dean", "Eve"], ["Carol", "Eve", "Dean"]}          (6.58)

```

```

> nops(%)
20

```

(6.59)

It is left to the reader to experiment with other starting sets and values of r to determine a formula. Note that the procedures in this subsection were written using a very naive approach. There are simpler and more efficient approaches, but those would give away the key idea used to create the formula.

6.4 Binomial Coefficients and Identities

In this section, we will use Maple to compute binomial coefficients, to generate Pascal's triangle, and to verify identities.

The Binomial Theorem

Recall from the previous section that the Maple command **binomial** can be used to compute $\binom{n}{r}$, which is another notation for $C(n, r)$. With n and r positive integers, this command produces the same result as **numbcomb**. The **binomial** command is designed to be more general, in that it will compute coefficients that appear in Newton's generalized binomial theorem. The generalization is beyond the scope of this manual.

Here, we will consider questions such as Examples 2 through 4 from Section 6.4 of the text.

First, consider the problem of expanding $(x + y)^5$. In Maple, this can be done easily with the **expand** command. The **expand** command requires one argument, an algebraic expression. It returns the result of “expanding” the expression, that is, of distributing products over sums.

$$\begin{aligned} > \text{expand}((x+y)^5) \\ x^5 + 5x^4y + 10x^3y^2 + 10x^2y^3 + 5xy^4 + y^5 \end{aligned} \tag{6.60}$$

Now, consider the question of finding the coefficient of $x^{18}y^{12}$ in the expansion of $(x + y)^{30}$. The binomial theorem tells us that this coefficient is $\binom{30}{12}$. The **binomial** command with first argument 30 and second 12 will produce this value.

$$\begin{aligned} > \text{binomial}(30, 12) \\ 86\,493\,225 \end{aligned} \tag{6.61}$$

Thus, the expansion of $(x + y)^{30}$ contains the term $86\,493\,225 x^{18}y^{12}$.

Finding the coefficient of $x^{12}y^{13}$ in the expansion of $(2x - 3y)^{25}$ requires that we include the coefficients of x and y in the computation. As explained in the solution to Example 4 of the text, the expansion is

$$(2x + (-3y))^{25} = \sum_{j=0}^{25} \binom{25}{j} (2x)^{25-j} (-3y)^j.$$

The coefficient of $x^{12}y^{13}$ is found by taking $j = 13$:

$$\binom{25}{13} 2^{12}(-3)^{13}.$$

This is

$$\begin{aligned} > \text{binomial}(25, 13) \cdot 2^{12} \cdot (-3)^{13} \\ -33\,959\,763\,545\,702\,400 \end{aligned} \tag{6.62}$$

Pascal’s Triangle

As we have seen, it is very easy to compute binomial coefficients with Maple. To compute row n of Pascal’s triangle, we apply the **binomial** command with the second argument ranging from 0 to n . This can be done with the **seq** command. For example, the 25th row of Pascal’s triangle is shown below.

$$\begin{aligned} > \text{seq}(\text{binomial}(25, k), k = 0 .. 25) \\ 1, 25, 300, 2300, 12\,650, 53\,130, 177\,100, 480\,700, 1\,081\,575, 2\,042\,975, \\ 3\,268\,760, 4\,457\,400, 5\,200\,300, 5\,200\,300, 4\,457\,400, 3\,268\,760, \\ 2\,042\,975, 1\,081\,575, 480\,700, 177\,100, 53\,130, 12\,650, 2300, \\ 300, 25, 1 \end{aligned} \tag{6.63}$$

When calculating a single binomial coefficient or an isolated row, applying the formula may be the most efficient approach. However, if you wish to build a sizable portion of Pascal's triangle, making use of Pascal's identity (Theorem 2 of Section 6.4) and the symmetry property (Corollary 2 of Section 6.3) can be more effective.

In this subsection, we will write two procedures for computing binomial coefficients. The first will simply apply the formula $\frac{n!}{r!(n-r)!}$. The second will be a recursive procedure making use of Pascal's identity and symmetry. Then, we will compare the performance of the two procedures in building Pascal's triangle.

The first procedure will be a straightforward application of the formula. We name it **BinomialF** (for formula).

```

1 BinomialF := proc (n : : nonnegint, k : : nonnegint)
2     return n! / (k!* (n-k)!);
3 end proc;
```

A Recursive Procedure

The second procedure will be called **BinomialR** (for recursive). Recall Pascal's identity:

$$\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}.$$

Rewriting this in terms of n and $n - 1$, we have

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Also recall that the binomial coefficients are symmetric, that is,

$$\binom{n}{k} = \binom{n}{n-k}.$$

With these facts in mind, our recursive procedure will work as follows. We will declare the **remember** option so that Maple will automatically create a remember table. The body of the procedure will consist of an if-elif-else statement. In case the second argument, is 0, the procedure will return 1. That forms the basis case of the recursion. The elif clause will test whether $2k > n$. In this case, we make use of symmetry and call **BinomialR** on n and $n - k$. Finally, in the else clause, we apply Pascal's identity, making recursive calls to **BinomialR**. Here is the implementation.

```

1 BinomialR := proc (n : : nonnegint, k : : nonnegint)
2     option remember;
3     if k=0 then
4         return 1;
5     elif 2*k > n then
6         return BinomialR(n, n-k);
7     else
```

```

8      return BinomialR(n-1, k-1) + BinomialR(n-1, k);
9      end if;
10     end proc;

```

We use our two procedures to build Pascal's triangle using a loop to control the value of **n** and the **seq** command.

```

> for n from 0 to 5 do
    seq(BinomialF(n, k), k = 0 ..n)
end do
1
1, 1
1, 2, 1
1, 3, 3, 1
1, 4, 6, 4, 1
1, 5, 10, 10, 5, 1

```

(6.64)

```

> for n from 0 to 5 do
    seq(BinomialR(n, k), k = 0 ..n)
end do
1
1, 1
1, 2, 1
1, 3, 3, 1
1, 4, 6, 4, 1
1, 5, 10, 10, 5, 1

```

(6.65)

Comparing Performance

Now, we will compare the performance of the two procedures.

First, we use each of them to compute the first 1000 rows of Pascal's triangle and compare the time it takes. We must use **forget** to clear the remember table for **BinomialR**.

```

> forget(BinomialR)

> st := time():
for n from 0 to 1000 do
    seq(BinomialF(n, k), k = 0 ..n)
end do:
time() - st
9.524

```

(6.66)

```

> st := time():
for n from 0 to 1000 do
    seq(BinomialR(n, k), k = 0 ..n)
end do:
time() - st
1.900

```

(6.67)

You see that the difference between the two performance of the two procedures is substantial.

However, if we compute some isolated values, the situation is reversed. We will generate some random values of n between 100 and 1000 and random values of k between 0 and 100.

Recall that the **rand** command applied to a range of integers returns a procedure that generates random numbers in that range. For instance, the statement below assigns to the name **randN** a procedure. Calling **randN()** produces a random number.

```
> randN := rand(100..1000):  
> randN()  
286
```

(6.68)

Likewise for **randK**. Note that calling **rand** with a single integer as the argument is the same as the range from 0 to that value.

```
> randK := rand(100):  
> randK()  
15
```

(6.69)

Next, we generate a list of 100 randomly generated (n, k) pairs. We suppress the output since the list will be long, but note that the elements of this list are 2-element lists.

```
> NKlist := [seq([randN(), randK()], i = 1..100)]:
```

Now, we apply the two procedures to the elements of the randomly generated list. Note that we pass the arguments to both procedures with the syntax **BinomialF(op(NKpair))**. The **op** command applied to **NKpair**, an element of the **NKlist**, takes the 2-element list **NKpair** and returns the underlying expression sequence. The procedures were written to accept two integer arguments, so passing them the 2-element list (which is only one argument) would raise an error.

```
> st := time():  
for NKpair in NKlist do  
  BinomialF(op(NKpair))  
end do:  
time() - st  
0.019
```

(6.70)

```
> forget(BinomialR)  
> st := time():  
for NKpair in NKlist do  
  BinomialR(op(NKpair))  
end do:  
time() - st  
0.210
```

(6.71)

The reason for the difference in relative performance is that when generating Pascal's triangle, all of the values beginning with $n = 0, k = 0$ needed to be calculated. On the other hand, when calculating

certain isolated values, the recursive procedure still had to calculate all of the results for lower values of n and k , which the nonrecursive procedure did not need to compute.

Verifying Identities

Maple can help verify identities regarding the binomial coefficients. If you enter an expression using the binomial command that includes symbolic arguments, Maple will echo the statement. First, we ensure that **n** and **k** are unassigned.

```
> n := 'n';
k := 'k'
n := n
k := k
```

(6.72)

```
> binomial(n, k)

$$\binom{n}{k}$$

```

(6.73)

To have Maple produce an expression using factorials, use the **convert** command with the binomial expression as the first argument and the keyword **factorial** as the second argument.

```
> convert(binomial(n, k), factorial)

$$\frac{n!}{k! (n - k)!}$$

```

(6.74)

Symmetry

As a first example, we will verify the identity $C(n, k) = C(n, n - k)$, the symmetry identity.

Assign names to the left- and right-hand sides of the identity.

```
> left := binomial(n, k):
> right := binomial(n, n - k):
```

Remember that we need to use the **evalb** command to evaluate Boolean expressions, such as the equality **left=right**.

```
> evalb(left = right)
false
```

(6.75)

The negative result illustrates that a great deal of care is needed when using Maple to verify identities. Maple does not recognize the above as a true statement and therefore reports false. That does not mean that the identity is not in fact true. It only indicates that Maple was not able to confirm it.

In order to verify the symmetry identity, we need to apply two other commands. First, we use the **convert** command to replace the definitions of **left** and **right**.

```
> left := convert(left, factorial)
left :=  $\frac{n!}{k! (n - k)!}$ 
```

(6.76)

$$> \text{right} := \text{convert}(\text{right}, \text{factorial})$$

$$\text{right} := \frac{n!}{k! (n-k)!} \quad (6.77)$$

Second, we use **simplify** to help ensure that Maple will perform necessary algebraic manipulations in order for it to recognize the equality. It is often simplest to embed the call to **simplify** within the **evalb** statement. (In this example, **simplify** is not necessary, but it is typically needed.)

$$> \text{evalb}(\text{simplify}(\text{left} = \text{right}))$$

$$\text{true} \quad (6.78)$$

The Hockeystick Identity

Exercise 31 in Section 6.4 asks you to prove the hockeystick identity:

$$\sum_{k=0}^r \binom{n+k}{k} = \binom{n+r+1}{r}.$$

The exercise asks you to prove it using a combinatorial argument and using Pascal's identity. Maple will verify this identity for us using algebra based on the formula for the binomial coefficient.

First, we will give the right-hand side of the identity a name.

$$> \text{rightHockey} := \text{binomial}(n+r+1, r) :$$

The left-hand side is a summation. Since it is a symbolic sum, we must use **sum**, rather than **add** to represent it.

$$> \text{leftHockey} := \text{sum}(\text{binomial}(n+k, k), k = 0 .. r) :$$

To verify the identity, we need to go a step further and **convert** the expressions to factorials.

$$> \text{leftHockey} := \text{convert}(\text{leftHockey}, \text{factorial})$$

$$\text{leftHockey} := \frac{(n+r+1)!}{r! (n+1)!} \quad (6.79)$$

$$> \text{rightHockey} := \text{convert}(\text{rightHockey}, \text{factorial})$$

$$\text{rightHockey} := \frac{(n+r+1)!}{r! (n+1)!} \quad (6.80)$$

Now, **evalb** combined with **simplify** will confirm the identity.

$$> \text{evalb}(\text{simplify}(\text{leftHockey} = \text{rightHockey}))$$

$$\text{true} \quad (6.81)$$

Keep in mind when using Maple to check identities that it will only report true when the two sides of the expression are identical. If it does report true, you can be fairly confident that the identity does hold, though a truly convincing proof requires that you explicitly show the algebraic manipulations.

If Maple reports false, however, even after using **convert** and **simplify**, you cannot be certain whether the identity is false or if it is true but more manipulation is needed to get Maple to recognize it. To use Maple to demonstrate that a purported identity is false, you would find a counterexample by computing the values of both expressions and finding inputs that result in different values. (Refer to Section 1.7 for examples of finding counterexamples.)

6.5 Generalized Permutations and Combinations

In this section, we will introduce a variety of Maple commands related to permutations and combinations with repetition allowed and related to distributing objects in boxes where the objects and the boxes may or may not be distinguishable.

Permutations with Repetition

Recall from Theorem 1 that the number of r -permutations of n objects is n^r if repetition is allowed.

For example, the number of strings of length 5 that can be formed from the 26 uppercase letters of the English alphabet is

$$> 26^5 \\ 11\,881\,376 \quad (6.82)$$

As a second example, we compute the number of ways that four elements can be selected in order from a set with three elements when repetition is allowed.

$$> 3^4 \\ 81 \quad (6.83)$$

Recall from the previous section that the **numbperm** and **permute** commands can accept either a number or a list or a set as the first argument. In case you provide a list as the first argument, that list may have repeated elements. In case the list does contain repeated elements, those elements are treated as identical, but are allowed to repeat in the permutations.

For example, consider the statement below.

$$> \text{permute}([1, 1, 2]) \\ [[1, 1, 2], [1, 2, 1], [2, 1, 1]] \quad (6.84)$$

Given a list of n not necessarily distinct objects and no second argument, the **permute** command produces all of the n -permutations of the n objects. Note that 1 appeared twice in the input and thus appears twice in the results, while 2 appeared once in the input and so appears once in the output.

With a second argument, you can specify the length of the permutations.

$$> \text{permute}([1, 1, 2], 2) \\ [[1, 1], [1, 2], [2, 1]] \quad (6.85)$$

Since 1 appeared twice in the input list, it was allowed to appear twice in the results, but 2 appeared only one time in the input and thus was not allowed to repeat. This means that if you want to list the ways that four elements can be selected in order from a set with three elements when repetition

is allowed, you can use the **permute** command, so long as you repeat the elements in the input. In order to generate all r -permutations of n objects with repetition allowed, you must use as input the list consisting of the n objects each repeated r times. If an object is repeated fewer than r times in the input list, then that will limit the number of times it is allowed to repeat in the results.

To form 4-permutations with repetition allowed of $\{“a”, “b”, “c”\}$, we apply the **permute** command as shown below. Recall the use of the dollar sign operator with syntax **$\mathbf{o} \$ \mathbf{n}$** to produce a sequence of the object **o** repeated **n** times.

```
> abcRepeated := permute(["a" $ 4, "b" $ 4, "c" $ 4], 4)
abcRepeated := [[["a", "a", "a", "a"], ["a", "a", "a", "b"],
["a", "a", "a", "c"], ["a", "a", "b", "a"], ["a", "a", "b", "b"],
["a", "a", "b", "c"], ["a", "a", "c", "a"], ["a", "a", "c", "b"],
["a", "a", "c", "c"], ["a", "b", "a", "a"], ["a", "b", "a", "b"],
["a", "b", "a", "c"], ["a", "b", "b", "a"], ["a", "b", "b", "b"],
["a", "b", "b", "c"], ["a", "b", "c", "a"], ["a", "b", "c", "b"],
["a", "b", "c", "c"], ["a", "c", "a", "a"], ["a", "c", "a", "b"],
["a", "c", "a", "c"], ["a", "c", "b", "a"], ["a", "c", "b", "b"],
["a", "c", "b", "c"], ["a", "c", "c", "a"], ["a", "c", "c", "b"],
["a", "c", "c", "c"], ["a", "b", "a", "a"], ["a", "b", "a", "b"],
["a", "b", "a", "c"], ["a", "b", "b", "a"], ["a", "b", "b", "b"],
["a", "b", "b", "c"], ["a", "b", "c", "a"], ["a", "b", "c", "b"],
["a", "b", "c", "c"], ["a", "b", "a", "a"], ["a", "b", "a", "b"],
["a", "b", "a", "c"], ["a", "b", "b", "a"], ["a", "b", "b", "b"],
["a", "b", "b", "c"], ["a", "b", "c", "a"], ["a", "b", "c", "b"],
["a", "b", "c", "c"], ["a", "c", "a", "a"], ["a", "c", "a", "b"],
["a", "c", "a", "c"], ["a", "c", "b", "a"], ["a", "c", "b", "b"],
["a", "c", "b", "c"], ["a", "c", "c", "a"], ["a", "c", "c", "b"],
["a", "c", "c", "c"], ["a", "c", "a", "a"], ["a", "c", "a", "b"],
["a", "c", "a", "c"], ["a", "c", "b", "a"], ["a", "c", "b", "b"],
["a", "c", "b", "c"], ["a", "c", "c", "a"], ["a", "c", "c", "b"],
["a", "c", "c", "c"]]] (6.86)
```

```
> nops(abcRepeated)
81 (6.87)
```

Note that the size of the list produced by **permute** agrees with the answer given by the formula n^r .

The **numbperm** command has the same syntax as the **permute** command and will return the number of permutations without listing them all.

```
> numbperm(["a" $ 4, "b" $ 4, "c" $ 4], 4)
81 (6.88)
```

Combinations with Repetition

Combinations with repetition can be handled in Maple in much the same way as permutations with repetition are.

Theorem 2 of Section 6.5 asserts that the number of r -combinations of a set of n objects when repetition of elements is allowed is $C(n + r - 1, r)$. This suggests the following useful functional operator.

```
> numbcombrep := (n, r) → numbcomb(n + r - 1, r) :
```

Thus, we can compute, for example, the number of ways to select five bills from a cash box with seven types of bills (Example 3) as follows.

```
> numbcombrep(7, 5)
```

462

(6.89)

We can also make use of the **numbcomb** and **choose** commands in the same way as **numbperm** and **permute** were used above.

Consider Example 2 from Section 6.5. In this example, we are given a bowl of apples, oranges, and pears and are to select four pieces of fruit from the bowl provided that it contains at least four pieces of each kind of fruit. We have three ways to solve this problem with Maple.

First, we can use the **numbcombrep** functional operator that we created.

```
> numbcombrep(3, 4)
```

15

(6.90)

The second approach is to use **numbcomb**. The first argument is the list of “apple”, “orange”, and “pear” each repeated four times, and the second argument is 4 indicating that four objects are to be selected.

```
> numbcomb([“apple” $ 4, “orange” $ 4, “pear” $ 4], 4)
```

15

(6.91)

The third option is to use **choose** to list all the options. The arguments are the same as they were for **numbcomb**.

```
> choose([“apple” $ 4, “orange” $ 4, “pear” $ 4], 4)
```

```
[[“apple”, “apple”, “apple”, “apple”], [“apple”, “apple”, “apple”, “orange”],
 [“apple”, “apple”, “apple”, “pear”], [“apple”, “apple”, “orange”, “orange”],
 [“apple”, “apple”, “orange”, “pear”], [“apple”, “apple”, “pear”, “pear”],
 [“apple”, “orange”, “orange”, “orange”], [“apple”, “orange”, “orange”, “pear”],
 [“apple”, “orange”, “pear”, “pear”], [“apple”, “pear”, “pear”, “pear”],
 [“orange”, “orange”, “orange”, “orange”], [“orange”, “orange”, “orange”, “pear”],
 [“orange”, “orange”, “pear”, “pear”], [“orange”, “pear”, “pear”, “pear”],
 [“pear”, “pear”, “pear”, “pear”]]
```

(6.92)

```
> nops(%)
```

15

(6.93)

Note that when repetition is allowed, all of the commands **numbperm**, **permute**, **numbcomb**, and **choose** must be given a list as the first argument. If you were to use a set instead of a list, the repeated elements in the set would be automatically removed by Maple.

Permutations with Indistinguishable Objects

Maple handles permutations with indistinguishable objects in the same way as when repetition is allowed. The **numbperm** and **permute** commands both accept lists of objects as their first argument. If objects in the list are repeated, Maple considers them as indistinguishable and counts the permutations appropriately.

For example, to solve Example 7, finding the number of different strings that can be made from the letters of the word *SUCCESS*, we use the list $[S, U, C, C, E, S, S]$ as the argument to **numbperm**.

```
> numbperm(["S", "U", "C", "C", "E", "S", "S"])
420
```

(6.94)

Observe that this gives the same result as the formula given in Theorem 3.

Maple makes it easy to go a bit further than Theorem 3, which is restricted to the situation when you are permuting all n objects. To find the number of r -permutations of n objects where some objects are indistinguishable, you give r as the second argument.

For example, the number of strings of length 3 that can be made from the letters of the word *SUCCESS* can be calculated as follows.

```
> numbperm(["S", "U", "C", "C", "E", "S", "S"], 3)
43
```

(6.95)

Listing these words can be accomplished by use of the **permute** command with the same arguments.

```
> permute(["S", "U", "C", "C", "E", "S", "S"], 3)
[[["S", "U", "C"], ["S", "U", "E"], ["S", "U", "S"], ["S", "C", "U"],
  ["S", "C", "C"], ["S", "C", "E"], ["S", "C", "S"], ["S", "E", "U"],
  ["S", "E", "C"], ["S", "E", "S"], ["S", "S", "U"], ["S", "S", "C"],
  ["S", "S", "E"], ["S", "S", "S"], ["U", "S", "C"], ["U", "S", "E"],
  ["U", "S", "S"], ["U", "C", "S"], ["U", "C", "C"], ["U", "C", "E"],
  ["U", "E", "S"], ["U", "E", "C"], ["C", "S", "U"], ["C", "S", "C"],
  ["C", "S", "E"], ["C", "S", "S"], ["C", "U", "S"], ["C", "U", "C"],
  ["C", "U", "E"], ["C", "C", "S"], ["C", "C", "U"], ["C", "C", "E"],
  ["C", "E", "S"], ["C", "E", "U"], ["C", "E", "C"], ["E", "S", "U"],
  ["E", "S", "C"], ["E", "S", "S"], ["E", "U", "S"], ["E", "U", "C"],
  ["E", "C", "S"], ["E", "C", "U"], ["E", "C", "C"]]
```

(6.96)

A related question is the number of strings with three or more letters that can be made from the letters of the word *SUCCESS*. To do this, we compute the number of r -permutations with r equal to 3, 4, 5, 6, and 7 and sum these values. Using the **add** command, this can be done in one statement.

Remember that the **add** command accepts a first argument in terms of a variable such as **i**. With the second argument specifying the range of values that **i** can take, the command returns the sum of the numbers obtained by evaluating the first argument at each value for **i**.

```
> add(numbperm (["S","U","C","C","E","S","S"], i), i = 3 .. 7)
1247
```

(6.97)

Distinguishable Objects and Distinguishable Boxes

Example 8 asks how many ways there are to distribute hands of five cards to each of four players from a deck of 52 cards. There are several ways to compute this value in Maple.

First, we can use the expression in terms of combinations, $C(52, 5) C(47, 5) C(42, 5) C(37, 5)$ by using **numbcomb**.

```
> numbcomb(52, 5) · numbcomb(47, 5) · numbcomb(42, 5) · numbcomb(37, 5)
1 478 262 843 475 644 020 034 240
```

(6.98)

Second, we can use the formula from Theorem 4: $\frac{52!}{5! 5! 5! 32!}$.

```
> 52!
5! 5! 5! 32!
1 478 262 843 475 644 020 034 240
```

(6.99)

Finally, this same value can be computed using the **multinomial** command. This command requires at least two positive integers as arguments, but can accept any number of positive integers. Regardless of the number of arguments, the first must be equal to the sum of the rest. If n is the first argument and k_1, k_2, \dots, k_m are the rest of the arguments, then assuming $k_1 + k_2 + \dots + k_m = n$, the command returns

$$\frac{n!}{k_1! k_2! \cdots k_m!}.$$

That is, **multinomial** implements the formula of Theorem 4. (Exercise 65 of Section 6.5 explains the reason for the term multinomial.)

Computing the number of ways to deal cards, as described above, can be computed as follows.

```
> multinomial(52, 5, 5, 5, 5, 32)
1 478 262 843 475 644 020 034 240
```

(6.100)

Revising the multinomial Command

It is common in questions about distributing distinguishable objects into distinguishable boxes that you want to distribute only some of the objects. In Example 8, for instance, not all of the cards are distributed. The **multinomial** command requires that you include the remainder of the cards as an argument. Conceptually, you can think of making one more box to hold the objects that are not placed in any of the other boxes.

This is such a common occurrence, however, that it seems more natural to forget about this “discard box.” We will write a Maple command that will use the formula from Theorem 4 but will not require that we provide the size of the discard box.

In order to make our procedure work as much like **multinomial** as possible, we need it to be able to accept any number of arguments.

To do this, we apply the **seq** modifier to the parameter. As a simple example of how this works, consider the example below.

```

1 printInts := proc(x::seq(integer))
2   print(x);
3 end proc;
```

The parameter declaration **x::seq(integer)** indicates that the parameter is to accept a sequence of integers. When the procedure is called with a sequence of integers as arguments, the name **x** is assigned to the sequence.

```
> printInts(5, 2, 9, 11)
5, 2, 9, 11
(6.101)
```

If nonintegers are included in the call to the procedure, the parameter will only be assigned to the sequence of integers up until the first noninteger.

```
> printInts(-5, 8, 2, "hello", 7, 11)
-5, 8, 2
(6.102)
```

The type in parentheses after the **seq** keyword can be any Maple type. Note that such a parameter can match the empty sequence in which case it is assigned **NULL**.

Our procedure will have two parameters. The first will be a positive integer **n**. The second will be the sequence of positive integers **k**. First, we ensure that **k** is not empty. (If **k = NULL**, the procedure returns 1, as there is 1 way to discard all the objects.) Next, we calculate the difference of the parameter **n** and the sum of the rest of the arguments. Note that **k** must be turned into a list to be used with the **add** (and most other) commands. The difference is the number of objects that are discarded. If this is negative, the procedure will raise an error. Otherwise, it becomes the final k_m in the Theorem 4 formula.

Here is the implementation. Note that we use the **map** command to compute the factorials of each of the integers in the denominator of the formula.

```

1 distinguishable := proc(n::posint, k::seq(posint))
2   local discard, denomList, i;
3   if k = NULL then
4     return 1;
5   end if;
6   discard := n - add(i, i=[k]);
7   if discard < 0 then
8     error "first argument must be at least the sum of the others."
9   end if;
10  denomList := [k, discard];
```

```

11   denomList := map (factorial,denomList);
12   return n!/mul (i,i=denomList);
13 end proc;

```

With this procedure, we can compute the number of ways to deal five cards to each of four players from a deck of 52 cards as follows.

> *distinguishable*(52, 5, 5, 5)
1 478 262 843 475 644 020 034 240 (6.103)

Indistinguishable Objects and Distinguishable Boxes

The text describes the correspondence between questions about placing indistinguishable objects into distinguishable boxes and about combination with repetition questions.

Example 9 asks how many ways 10 indistinguishable balls can be placed in 8 bins. We can use the **numbcombrep** function written earlier.

> *numbcombrep*(8, 10)
19 448 (6.104)

It may seem that the arguments were reversed. Keep in mind that the connection to combinations with repetition is that you are selecting 10 bins from the 8 available bins with repetition allowed.

Compositions and Weak Compositions

We can also use the **composition** and **numbcomp** commands to answer questions of this kind. A k -composition of a positive integer n is a way of writing n as the sum of k positive integers where the order of the summands matters. For example, 4 has three distinct 2-compositions: $3 + 1$, $2 + 2$, and $1 + 3$.

A weak composition is similar, but the terms in the sum are allowed to be 0. Thus, 4 has five distinct weak 2-compositions: $4 + 0$ and $0 + 4$ in addition to the three listed before.

Note that the weak r -compositions of n correspond to the r -compositions of $n + r$. For suppose that $x_1 + x_2 + \cdots + x_r = n$ is a weak r -composition. Then, each x_i is nonnegative. Therefore,

$$(x_1 + 1) + (x_2 + 1) + \cdots + (x_r + 1) = n + r,$$

and each $x_i + 1$ is positive, and hence this is a composition of $n + r$. Likewise, any r -composition of $n + r$ can be transformed into a weak r -composition of n by subtracting 1 from each term.

Also note that weak r -compositions of n correspond to placing n indistinguishable balls into r distinguishable bins. Suppose $x_1 + x_2 + \cdots + x_r = n$ is a weak r -composition of n . This can be identified with placing x_1 of the objects into the first bin, x_2 objects in the second bin, etc.

Counting Weak Compositions

We now return to **numbcomp** and **composition**. The **numbcomp** command accepts two arguments, n and r . It returns the number of r -compositions of n . For example, as we saw, there are three 2-compositions of 4.

```
> numbcomp(4, 2)
3
```

(6.105)

We can create a functional operator to count the number of weak r -compositions of n using the fact that this is the same as the number of r -compositions of $n + r$.

```
> numbweakcomp := (n, r) → combinat[numbcomp](n + r, r) :
```

We saw there were five weak 2-compositions of 4.

```
> numbweakcomp(4, 2)
5
```

(6.106)

The number of ways that ten indistinguishable balls can be placed in eight bins is:

```
> numbweakcomp(10, 8)
19 448
```

(6.107)

Note that weak r -compositions were the subject of Example 5, which asked how many solutions there are to $x_1 + x_2 + x_3 = 11$ with nonnegative integers.

```
> numbweakcomp(11, 3)
78
```

(6.108)

Listing Weak Compositions

The **composition** command is used to create a list of all compositions. The arguments are the same as **numbcomp**. For example, to list the 2-compositions of 4, enter the following.

```
> composition(4, 2)
{[1, 3], [2, 2], [3, 1]}
```

(6.109)

We can use **composition** to create a **weakcomposition** procedure. Given arguments n and r , we apply **composition** to $n + r$ and r . For each list in the result, we subtract 1 in each position.

The subtraction of 1 in each position can be done with the **map** command using the functional operator **x ->x - 1** as the first argument. For example,

```
> map(x → x - 1, [1, 2, 3, 4, 5, 6])
[0, 1, 2, 3, 4, 5]
```

(6.110)

The functional operator is applied to each element in the list given as the second argument.

We need to apply the above to each composition in the set produced by the **composition** function. To do this, we use a for loop over the result of **composition** and build a set of weak compositions. Here is the complete procedure.

```
1 weakcomposition := proc(n::posint, r::posint)
2   local minus1, strong, weak, C;
3   minus1 := x → x - 1;
4   strong := combinat[composition](n+r, r);
```

```

5   weak := {};
6   for C in strong do
7       weak := weak union {map(minus1, C)};
8   end do;
9   return weak;
10 end proc;

```

The weak 2-compositions of 4 can now be produced by this procedure.

> *weakcomposition*(4, 2)
{[0, 4], [1, 3], [2, 2], [3, 1], [4, 0]} (6.111)

Distinguishable Objects and Indistinguishable Boxes

As described in the text, the number of ways to place n distinguishable objects in k indistinguishable boxes is given by the Stirling numbers of the second kind, $S(n, k)$.

The command **Stirling2** computes the Stirling number of the second kind. This command requires two arguments, the number of objects and the number of boxes. For example, the statement below computes the number of ways to put seven different employees in four offices when each office must not be empty.

> *Stirling2*(7, 4)
350 (6.112)

In order to compute the number of ways to assign the seven employees to the four offices and allow empty offices, we must add the number of ways to assign all seven employees to one office, to two offices, to three offices, and to four offices.

> *add*(*Stirling2*(7, k), $k = 1..4$)
715 (6.113)

Generating Assignments of Employees to Offices

The **Stirling2** command tells us how many ways there are to place distinguishable objects in indistinguishable boxes, but Maple does not have a command to produce the assignments.

To make such a command, we rely on the following observations. First, as indicated in the text, a choice of distinguishable objects to indistinguishable boxes can be modeled as a set of subsets. For instance, $\{\{D\}, \{A, C\}, \{B, E\}\}$ represents the assignment of A and C to one box, D to a box of its own, and B and E to another box. The set of subsets must not contain the empty set and must be such that the union of the subsets be the entire collection of objects.

We can produce such assignments recursively. The basis step is that there is only one way to assign n objects to 1 box and there are no ways to assign n objects to $k > n$ (under the requirement that no box be empty). To assign n objects to k boxes with $k \leq n$, proceed as follows.

First, find all assignments of $n - 1$ objects to $k - 1$ boxes and update each assignment by placing object n in a box by itself. In terms of the set representation, given $\{B_1, B_2, \dots, B_{k-1}\}$, we produce $\{B_1, B_2, \dots, B_{k-1}, \{n\}\}$.

Second, find all assignments of $n - 1$ objects to k boxes. For each such assignment $\{B_1, B_2, \dots, B_k\}$, produce the following k assignments of n objects to the k boxes:

$$\{B_1 \cup \{n\}, B_2, \dots, B_k\}, \{B_1, B_2 \cup \{n\}, B_3, \dots, B_k\}, \dots, \{B_1, B_2, \dots, B_k \cup \{n\}\}.$$

The assignments produced by the two methods above produce all assignments. The following procedure implements this algorithm.

```

1 makeStirling2 := proc (n :: posint, k :: posint)
2   local A, k1boxes, kboxes, B, new, i;
3   if k = 1 then
4     return {{ ${\$1..n} } };
5   end if ;
6   if k > n then
7     return { };
8   end if ;
9   A := { };
10  # n-1 objects in k-1 boxes
11  k1boxes := makeStirling2 (n-1, k-1);
12  for B in k1boxes do
13    new := B union { {n} };
14    A := A union {new};
15  end do;
16  # n-1 objects in k boxes
17  kboxes := makeStirling2 (n-1, k);
18  for B in kboxes do
19    for i from 1 to k do
20      new := subsop (i=B[i] union {n}, B);
21      A := A union {new};
22    end do;
23  end do;
24  return A;
25 end proc;
```

Let us analyze the procedure. It accepts n and k as parameters and returns the set consisting of all possible assignments of distinguishable objects to indistinguishable boxes.

In the case that $k = 1$, there is only one possible assignment, all objects are assigned to the single box. This assignment is represented by $\{ \{1, 2, \dots, n\} \}$, since an assignment corresponds to a set of subsets. The procedure returns the set consisting of this single assignment when $k = 1$. Recall the $\$$ operator with no left argument and a range as its right argument produces the sequence defined by the range.

If $k > n$, then there are no valid assignments and the procedure returns the empty set.

Otherwise, we initialize **A**, which will store the set of assignments that is returned, to the empty set. Recall that there are two recursive steps. First expanding on the assignments of $n - 1$ objects to $k - 1$ boxes and then expanding on the assignments of $n - 1$ objects to k boxes.

For the first part, we assign the name **k1boxes** to the set of assignments of $n - 1$ objects to $k - 1$ boxes. For each such assignment, for example, $\{\{2\}, \{4, 6\}, \{1, 3, 5\}\}$, we add n in its own box:

$$\{\{1, 3, 5\}, \{2\}, \{4, 6\}\} \cup \{\{7\}\} = \{\{1, 3, 5\}, \{2\}, \{4, 6\}, \{7\}\}.$$

This **new** assignment is then added to **A**.

In the second part, we assign **kboxes** to the set of assignments of $n - 1$ objects to k boxes. For each such assignment, we consider each of the k boxes in turn and add n to that box. For instance, the assignment $\{\{2, 3\}, \{1\}, \{5\}, \{4, 6\}\}$ would generate the four assignments:

$$\begin{aligned} & \{\{2, 3, 7\}, \{1\}, \{5\}, \{4, 6\}\} \\ & \{\{2, 3\}, \{1, 7\}, \{5\}, \{4, 6\}\} \\ & \{\{2, 3\}, \{1\}, \{5, 7\}, \{4, 6\}\} \\ & \{\{2, 3\}, \{1\}, \{5\}, \{4, 6, 7\}\}. \end{aligned}$$

Recall that the **subsop** command is used to replace (substitute) elements in sets and lists. The syntax **subsop(*i=x,S*)** where **i** is an index of the set or list **S**, causes the expression **x** to replace whatever had been at index **i** in **S**.

Compare the result of our procedure to the solution of Example 10 in Section 6.5 of the text.

```
> makeStirling2(4, 3)
{\{{1}, {2}, {3, 4}\}, {{1}, {3}, {2, 4}\}, {{1}, {4}, {2, 3}\},
 {2}, {3}, {1, 4}\}, {{2}, {4}, {1, 3}\}, {{3}, {4}, {1, 2}\}} (6.114)
```

Except for using the integers 1 through 4 instead of the letters A through D, the output above is the same as the six ways listed in the text for placing the four employees in three offices.

To produce all 14 ways to assign the four employees to three offices with each office containing any number of employees, we need to loop over the different values of k .

```
> offices := {}:
for k from 1 to 3 do
    offices := offices union makeStirling2(4, k):
end do :
```

We print them out one at a time.

```
> for o in offices do
    print(o)
end do
{{1, 2, 3, 4}}
{{1}, {2, 3, 4}}
{{2}, {1, 3, 4}}
```

```

{{3},{1,2,4}}
{{4},{1,2,3}}
{{1,2},{3,4}}
{{1,3},{2,4}}
{{1,4},{2,3}}
{{1},{2},{3,4}}
{{1},{3},{2,4}}
{{1},{4},{2,3}}
{{2},{3},{1,4}}
{{2},{4},{1,3}}
{{3},{4},{1,2}}

```

(6.115)

Indistinguishable Objects and Indistinguishable Boxes

As described in the text, distributing n indistinguishable objects into k indistinguishable boxes is identical to forming a partition of n into k positive integers. A partition of n into j positive integers is a sum $n = a_1 + a_2 + \dots + a_j$ with $0 < a_1 \leq a_2 \leq \dots \leq a_j$. (Note that this is the reverse order from the definition in the text. Maple's commands for producing partitions list them in nondecreasing order, rather than in nonincreasing order as is done in the text.)

The Maple commands **numbpart** and **partition** are used to count and form partitions of integers.

With one argument, a nonnegative integer n , **numbpart** returns the total number of partitions of n into as many as n boxes. Likewise, **partition** applied to one argument returns a list containing lists representing partitions of the argument.

For example, the statements below compute the number of partitions of 7 and lists all the partitions of 7.

```

> numbpart(7)
15

```

(6.116)

```

> partition(7)
[[1,1,1,1,1,1,1], [1,1,1,1,1,2], [1,1,1,2,2], [1,2,2,2], [1,1,1,1,3],
 [1,1,2,3], [2,2,3], [1,3,3], [1,1,1,4], [1,2,4], [3,4], [1,1,5],
 [2,5], [1,6], [7]]

```

(6.117)

Both commands also accept a second argument, which specifies the maximum integer allowed to appear in a partition. For example, to answer the question: how many ways are there to distribute 7 indistinguishable balls in up to 7 identical boxes when each box can hold at most 4 objects, we would give 4 as the second argument to **numbpart**.

```

> numbpart(7,4)
11

```

(6.118)

To list them, we give the same arguments with **partition**.

```

> partition(7,4)
[[1,1,1,1,1,1,1], [1,1,1,1,1,2], [1,1,1,2,2], [1,2,2,2], [1,1,1,1,3],
 [1,1,2,3], [2,2,3], [1,3,3], [1,1,1,4], [1,2,4], [3,4]]

```

(6.119)

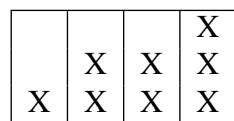
We will now see how to determine the number of ways to partition n into at most k boxes.

Limiting the Number of Boxes

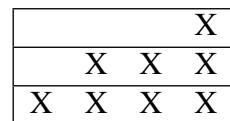
Example 11 asks how many ways there are to pack six copies of the same book into four boxes. The way that **numbpart** and **partition** are described, with the second argument controlling the maximum value in the partition rather than the number of entries in the list, suggests that they do not answer this question. However, as it turns out, they do.

In fact, the partitions of n with at most k objects in a box are in one-to-one correspondence with the partitions of n into at most k boxes. To understand why, consider the partition $[1, 2, 2, 3]$.

Think about stacking the 8 objects in boxes according to the partition $[1, 2, 2, 3]$. (The diagram shown below is called a Ferrers diagram.)



Instead of thinking about the columns as the boxes, we can instead consider the rows as boxes.



Now, the 8 objects are contained in three boxes. One box (the top row) has 1 object, another (the middle row) has 3 objects, and the last box (the bottom row) has 4 objects. In other words, we have partitioned 8 as $[1, 3, 4]$. These two partitions are said to be conjugate. (Note that you can also think about forming the conjugate by rotating the original diagram by 90 degrees or reflecting it across its diagonal while still interpreting the boxes as columns.)

We began with a partition whose maximum entry was 3 and found that its conjugate was a partition into 3 boxes. It is always the case that the conjugate of a partition with maximum n is a partition into n boxes. Moreover, this correspondence is one-to-one. We leave it to the reader to prove these facts.

The consequence of this discussion is that the number of ways to pack 6 copies of the same book into at most 4 identical boxes that can each hold all the books is the same as the number of ways to pack 6 copies of the same book into boxes that can hold at most 4 books each. That is, the solution to Example 11 is given by the following computation.

```
> numbpart(6, 4)  
9  
(6.120)
```

To produce these 9 partitions, we will use the **conjpart** command. This command accepts a partition as its only argument. It returns the conjugate of that partition. Using the example $[1, 2, 2, 3]$ from above,

```
> conjpart([1, 2, 2, 3])  
[1, 3, 4]  
(6.121)
```

The statement **partition(6,4)** will produce the list of all partitions of 6 with maximum value 4. Applying **conjpart** to each of those partitions produces the list of all partitions of 6 into at most 4 boxes. We use **map** with **conjpart** as the first argument and **partition(6,4)** as the second argument to apply **conjpart** to each partition.

```
> map (conjpart, partition (6, 4))
[[6], [1, 5], [2, 4], [3, 3], [1, 1, 4], [1, 2, 3], [2, 2, 2],
 [1, 1, 1, 3], [1, 1, 2, 2]] (6.122)
```

Generating Partitions

Here, we will describe how to generate partitions.

The first observation to make is that the partitions of n are identical to the partitions of n when each box can hold at most n objects. In terms of the **partition** command, the following are identical.

```
> partition (7)
[[1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 2], [1, 1, 1, 1, 2, 2], [1, 1, 1, 2, 2, 2], [1, 1, 1, 1, 3],
 [1, 1, 1, 2, 3], [1, 1, 1, 3, 3], [1, 1, 1, 4], [1, 1, 2, 4], [1, 1, 3, 4], [1, 1, 4, 4], [1, 2, 2, 2, 2],
 [1, 2, 2, 3, 2], [1, 2, 3, 3, 2], [1, 3, 3, 3, 2], [1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 2, 2],
 [1, 1, 1, 1, 2, 2, 2, 2], [1, 1, 1, 1, 3, 3, 3], [1, 1, 1, 1, 4, 4, 4], [1, 1, 1, 2, 2, 2, 2, 2],
 [1, 1, 1, 2, 2, 3, 3, 3], [1, 1, 1, 3, 3, 3, 3], [1, 1, 1, 4, 4, 4, 4], [1, 1, 2, 2, 2, 2, 2, 2],
 [1, 1, 2, 2, 3, 3, 3, 3], [1, 1, 2, 3, 3, 3, 3, 3], [1, 1, 3, 3, 3, 3, 3, 3], [1, 1, 4, 4, 4, 4, 4, 4], [1, 2, 2, 2, 2, 2, 2, 2, 2],
 [1, 2, 2, 2, 3, 3, 3, 3, 3], [1, 2, 2, 3, 3, 3, 3, 3, 3], [1, 2, 3, 3, 3, 3, 3, 3, 3], [1, 3, 3, 3, 3, 3, 3, 3, 3]] (6.123)
```

```
> partition (7, 7)
[[1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 2], [1, 1, 1, 1, 1, 2, 2], [1, 1, 1, 1, 1, 3, 3],
 [1, 1, 1, 1, 2, 2, 2], [1, 1, 1, 1, 3, 3, 3], [1, 1, 1, 2, 2, 2, 2], [1, 1, 1, 3, 3, 3, 3], [1, 1, 2, 2, 2, 2, 2],
 [1, 1, 2, 2, 3, 3, 3], [1, 1, 2, 3, 3, 3, 3], [1, 1, 3, 3, 3, 3, 3], [1, 1, 4, 4, 4, 4, 4], [1, 2, 2, 2, 2, 2, 2, 2],
 [1, 2, 2, 2, 3, 3, 3, 3], [1, 2, 2, 3, 3, 3, 3, 3], [1, 2, 3, 3, 3, 3, 3, 3], [1, 3, 3, 3, 3, 3, 3, 3], [1, 4, 4, 4, 4, 4, 4, 4], [1, 5, 5, 5, 5, 5, 5, 5]] (6.124)
```

We will describe how to form the partitions of n objects when each box can hold at most k objects recursively. The basis cases are: when $k = 1$, there is only one partition of n , the partition consisting of n 1s; when $n \leq 0$, there are no partitions.

To determine the partitions of n when each box can hold at most k objects, proceed as follows. First, determine all partitions of n when each box can hold at most $k - 1$ objects. These partitions are also partitions of n satisfying the requirement that each box holds at most k .

Second, provided that $n - k > 0$, determine all partitions of $n - k$ when each box can hold at most k objects, and append k to each partition. For example, with $n = 7$ and $k = 3$, we have $n - k = 4$ and the partitions of 4 with each box holding at most 3 are:

```
> partition (4, 3)
[[1, 1, 1, 1], [1, 1, 1, 2], [1, 1, 2, 2], [1, 2, 2, 2]] (6.125)
```

Therefore, $[1, 1, 1, 1, 3]$, $[1, 1, 2, 3]$, $[2, 2, 3]$, $[1, 3, 3]$ are partitions of 7 with each box having at most 3 objects.

Combining the partitions of n with each box holding at most $k - 1$ objects with the partitions of n formed by appending k to the partitions of $n - k$ with each box holding at most k objects produces all partitions of n with each box holding at most k objects. Setting $k = n$ produces all partitions of n .

It is an exercise to implement this algorithm.

Maple Commands for Generating Partitions Sequentially

We have seen that the **partition** command will produce all partitions of a given integer. Maple also has commands for producing partitions sequentially. Like **subsets** and **cartprod** for sequentially computing the subsets of a set and the Cartesian product of sets, producing partitions sequentially rather than all at once can save time and memory.

Maple's procedures are based on a canonical ordering of partitions. This ordering begins with the partition of all 1s and ends with the partition $[n]$. In this ordering, a partition $[x_k, x_{k-1}, \dots, x_2, x_1]$ precedes the partition $[y_j, y_{j-1}, \dots, y_2, y_1]$ if, for an index i ,

$$x_1 = y_1, x_2 = y_2, \dots, x_{i-1} = y_{i-1}, \text{ and } x_i < y_i.$$

In other words, partition x precedes partition y when the first (from the right) position at which they differ contains a smaller value in x than in y .

For example, the partition $[1, 1, 1, 3, 5]$ precedes the partition $[1, 2, 3, 5]$ because they agree in the two right-most positions, but in the third position from the right, the entry in $[1, 1, 1, 3, 5]$ is smaller than the third value from the right in $[1, 2, 3, 5]$. This is called reverse lexicographic order.

The commands **firstpart** and **lastpart** accept a positive integer n as their only argument. They return the partition consisting of n 1s and the partition $[n]$, respectively.

```
> firstpart(11)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] (6.126)
```

```
> lastpart(11)
[11] (6.127)
```

The command **nextpart** accepts as its sole argument a partition, that is, a nondecreasing list of positive integers. It returns the next partition in the ordering.

```
> nextpart([1, 1, 1, 2, 2, 4])
[1, 2, 2, 2, 4] (6.128)
```

Likewise, **prevpart** accepts a partition and returns the previous partition in the ordering.

```
> prevpart(%)
[1, 1, 1, 2, 2, 4] (6.129)
```

Using these commands, we can easily generate all partitions of a given number one at a time. We initialize a name with the **firstpart** command. Inside a while loop controlled by testing the current partition against the last partition, we update the partition using the **nextpart** command.

```
> p := firstpart(5)
p := [1, 1, 1, 1, 1]
> while p ≠ lastpart(5) do
    p := nextpart(p)
  end do (6.130)
```

```

p := [1,1,1,2]
p := [1,2,2]
p := [1,1,3]
p := [2,3]
p := [1,4]
p := [5] (6.131)

```

Maple also includes the **randpart** command for generating a random partition. This command accepts only one argument, the integer to be partitioned.

```

> randpart(5)
[1,4] (6.132)

```

6.6 Generating Permutations and Combinations

In this section, we will implement Algorithm 1 from Section 6.6 of the text for generating the next permutation in lexicographic order. Implementing Algorithms 2 and 3 will be left as exercises for the reader.

Interchange

Before implementing Algorithm 1, we will first write a procedure to interchange two elements in a list. This will be called by the procedure for generating permutations.

The **Interchange** procedure will require three parameters: the list and two integers representing the indices to be swapped.

The procedure has five statements. A copy of the list is made, since parameters cannot be assigned to. The element in the list at the first position to be swapped is assigned to a local variable. Then, the first position is assigned to the value in the second position. Finally, the second position is assigned to the value stored in the temporary name and the list is returned.

Here is the implementation and an example of applying it.

1	Interchange := proc(L :: list, i :: integer, j :: integer)
2	local l, temp;
3	l := L;
4	temp := l[i];
5	l[i] := l[j];
6	l[j] := temp;
7	return l;
8	end proc:

```

> Interchange(["a","b","c","d","e","d"],2,5)
["a","e","c","d","b","d"] (6.133)

```

nextpermutation

The input to the procedure will be a permutation $[a_1, a_2, \dots, a_n]$ of the set $\{1, 2, \dots, n\}$. Algorithm 1 consists of three steps: finding the largest j such that $a_j < a_{j+1}$; finding the smallest a_k to the right of

a_j and interchanging a_k and a_j ; and putting the elements in positions $j + 1$ and beyond in increasing order.

The first step comprises the first four lines of the body of Algorithm 1, through the comment. The index j is initialized to the next to last index in the permutation. A while loop is used to conduct the search. The body of the while loop decreases the value of j by one, and it is controlled by the condition $a_{j+1} < a_j$. When the while loop terminates, it will be the case that $a_j < a_{j+1}$ and j is the largest index for which that is true. Consequently, $a_{j+1} > a_{j+2} > \dots > a_n$.

The second step is to find the smallest a_k to the right of and larger than a_j and interchange the two. Since we are guaranteed that the elements to the right of a_j are in increasing order, a_n is the smallest element to the right of a_j , a_{n-1} is the next smallest, and so on. We are again searching from the right. Initialize k to n . A while loop is used to decrease k by one so long as $a_k < a_j$. When the while loop terminates, k will be such that $a_j < a_k$. Note that the loop is guaranteed to stop with $j < k$ since $a_j < a_{j+1}$. Once j has been identified, we interchange a_j and a_k using the **Interchange** procedure.

The third step is to put the elements of the permutation to the right of position j in increasing order. Note that before the interchange, a_{j+1} through a_n were in decreasing order. After the interchange of a_j with a_k , the tail end of the permutation remains in decreasing order. This is because a_j was smaller than a_k , but a_k was the smallest of the entries bigger than a_j . Thus all of a_{k+1}, \dots, a_n are smaller than a_j , and all of a_{j+1}, \dots, a_{k-1} are larger than a_k which is larger than a_j . Therefore,

$$a_{j+1}, \dots, a_{k-1}, a_j, a_{k+1}, \dots, a_n$$

is in decreasing order.

To put the tail in increasing order, we follow the instructions in the pseudocode that follow the interchange of a_j and a_k . Variables r and s are initialized to n and $j + 1$, respectively. Provided that r remains larger than s , we interchange a_r and a_s and then decrease r by 1 and increase s by 1. This has the effect of swapping a_{j+1} with a_n , then a_{j+2} with a_{n-1} , then a_{j+3} with a_{n-2} , etc.

Once the tail is in increasing order, the result is the new permutation and it is returned.

We need to add to the procedure two tests to ensure that the input is valid. We will declare the input to be a list of positive integers. Within the body of the procedure, we will check first to ensure that all of the elements in the list are no greater than n , the length of the list. Otherwise, the procedure will generate an error. We will also compare the input to the final permutation $[n, n - 1, \dots, 1]$ and return **FAIL** if they are equal.

Here is the implementation.

```

1 nextpermutation := proc (A::list(posint))
2   local a, n, i, j, k, r, s;
3   a := A;
4   n := nops(a);
5   for i from 1 to n do
6     if a[i] > n then
7       error "Input must be a permutation of {1,2,...,n}.";
8     end if;

```

```

9   end do;
10  if a = [seq(n-i, i=0..(n-1))] then
11    return FAIL;
12  end if;
13  # step 1: find j
14  j := n - 1;
15  while a[j] > a[j+1] do
16    j := j - 1;
17  end do;
18  # step 2: find k and interchange aj and ak
19  k := n;
20  while a[j] > a[k] do
21    k := k - 1;
22  end do;
23  a := Interchange(a, j, k);
24  # step 3: sort the tail
25  r := n;
26  s := j + 1;
27  while r > s do
28    a := Interchange(a, r, s);
29    r := r - 1;
30    s := s + 1;
31  end do;
32  return a;
33 end proc:

```

Example 2 of Section 6.6 finds that the permutation after 362 541 is 364 125. We use that example to confirm that our procedure is working.

```
> nextpermutation([3, 6, 2, 5, 4, 1])
[3, 6, 4, 1, 2, 5] (6.134)
```

To generate all permutations of a set $\{1, 2, \dots, n\}$, we use a while loop.

```
> aperm := [$1..4]:
while aperm ≠ FAIL do
  print(aperm);
  aperm := nextpermutation(aperm)
end do:
[1, 2, 3, 4]
[1, 2, 4, 3]
[1, 3, 2, 4]
[1, 3, 4, 2]
[1, 4, 2, 3]
[1, 4, 3, 2]
[2, 1, 3, 4]
[2, 1, 4, 3]
```

[2, 3, 1, 4]
 [2, 3, 4, 1]
 [2, 4, 1, 3]
 [2, 4, 3, 1]
 [3, 1, 2, 4]
 [3, 1, 4, 2]
 [3, 2, 1, 4]
 [3, 2, 4, 1]
 [3, 4, 1, 2]
 [3, 4, 2, 1]
 [4, 1, 2, 3]
 [4, 1, 3, 2]
 [4, 2, 1, 3]
 [4, 2, 3, 1]
 [4, 3, 1, 2]
 [4, 3, 2, 1]

(6.135)

Finding Permutations of Other Sets

As mentioned in the text, any set with n elements can be put in one-to-one correspondence with $\{1, 2, \dots, n\}$. Consequently, any permutation of a set with n elements can be obtained from a permutation of $\{1, 2, \dots, n\}$ and the correspondence.

In Maple, we can use the **subs** command to transform a permutation of $\{1, 2, \dots, n\}$ into a permutation of another set with n elements.

The **subs** command is used to perform substitutions in an expression. It has several forms, but we will use it with two arguments. The second argument will be the permutation of $\{1, 2, \dots, n\}$. The first argument will be a list of equations of the form $i = x$ where i is an integer in $\{1, 2, \dots, n\}$ and x is the corresponding element of the set being permuted.

For example, consider the set $\{a, b, c\}$ and the permutation $[3, 1, 2]$ of $\{1, 2, 3\}$. We establish the correspondence that 1, 2, and 3 are identified with a , b , and c , respectively. To produce the corresponding partition, we use the **subs** command with first argument $[1 = a, 2 = b, 3 = c]$ to specify the correspondence. The second argument is the permutation $[3, 1, 2]$.

```
> subs ([1 = "a", 2 = "b", 3 = "c"], [3, 1, 2])
["c", "a", "b"]
```

(6.136)

As another example, consider the set $\{2, 10, 13, 19\}$ and the permutation $[4, 2, 3, 1]$. Identify 1 with 2, 2 with 10, 3 with 13, and 4 with 19. Then, the following command produces the desired permutation.

```
> subs ([1 = 2, 2 = 10, 3 = 13, 4 = 19], [4, 2, 3, 1])
[19, 10, 13, 2]
```

(6.137)

Note that **subs** will allow you to provide the equations without enclosing them in a list. However, putting them in a list ensures that the substitutions are done simultaneously rather than sequentially. In this example, sequential assignment would not have produced the correct result.

A Procedure to Apply Permutations

We conclude by writing a procedure that, given a set and a permutation of $\{1, 2, \dots, n\}$ with n equal to the size of the set, will return the corresponding permutation of the set.

The key is automating the creation of the equations defining the correspondence. This can be done with the **zip** command. Recall that **zip** requires three arguments. The second and third arguments are lists while the first argument is a procedure on two parameters. The result is the list formed by applying the procedure to corresponding pairs of elements from the two lists.

We create a functional operator that takes two expressions and forms an equation from them.

```
> makeEqn := (a,b) → a = b  
makeEqn := (a,b) ↪ a = b
```

(6.138)

```
> makeEqn(5,a)  
5 = a
```

(6.139)

Using the list [1, 2, 3] and the list formed from the elements of the set $\{a, b, c\}$ as the second and third arguments, **zip** will produce the list of equations used in (6.136).

```
> zip(makeEqn,[1,2,3],[“a”, “b”, “c”])  
[1 = “a”, 2 = “b”, 3 = “c”]
```

(6.140)

This is all we need to write a procedure that accepts a set and a permutation of $\{1, 2, \dots, n\}$ and returns the corresponding permutation of the elements of the set.

```
1 permuteSet := proc(S::set,P::list(posint))  
2   local L, M, makeeqn, eqns;  
3   L := [op(S)];  
4   M := [$1..nops(S)];  
5   makeeqn := (a,b) -> a=b;  
6   eqns := zip(makeeqn,M,L);  
7   return subs(eqns,P);  
8 end proc;
```

Using the procedure, we can compute the permutation of $\{a, b, c, d, e\}$ corresponding to [3, 5, 2, 1, 4].

```
> permuteSet({“a”, “b”, “c”, “d”, “e”}, [3,5,2,1,4])  
[“c”, “e”, “b”, “a”, “d”]
```

(6.141)

Solutions to Computer Projects and Computations and Explorations

Computer Projects 10

Given positive integers n and r , list all the r -combinations, with repetition allowed, of the set $\{1, 2, 3, \dots, n\}$.

Solution: In Section 6.5 of this manual, we showed that the **choose** command could be used to generate combinations with repetition by repeating the elements in the list given as the first argument to **choose**.

To generate the 2-combinations of $\{1, 2, 3\}$, for example, we apply the **choose** command to the list consisting of 1, 2, and 3, each repeated twice. Note that the number of repetitions must be the same as r in order to choose r all of the same object.

```
> choose([1 $ 2, 2 $ 2, 3 $ 2], 2)
[[1, 1], [1, 2], [1, 3], [2, 2], [2, 3], [3, 3]]
```

(6.142)

Recall that **x\$n** generates the sequence of n x 's.

The first argument to **choose** can be generated using **seq** as follows.

```
> [seq(i $ 2, i = 1 .. 3)]
[1, 1, 2, 2, 3, 3]
```

(6.143)

We now write a procedure that accepts n and r as input and produces all r -combinations with repetition allowed.

```
1| chooseRepetition := proc(n :: posint, r :: posint)
2|   local L, i;
3|   L := [seq(i $ r, i=1..n)];
4|   return combinat[choose](L, r);
5| end proc;
```

We can obtain all of the 3-combinations of $\{1, 2, 3, 4, 5\}$ by

```
> chooseRepetition(5, 3)
[[1, 1, 1], [1, 1, 2], [1, 1, 3], [1, 1, 4], [1, 1, 5], [1, 2, 2], [1, 2, 3], [1, 2, 4], [1, 2, 5],
 [1, 3, 3], [1, 3, 4], [1, 3, 5], [1, 4, 4], [1, 4, 5], [1, 5, 5], [2, 2, 2], [2, 2, 3], [2, 2, 4],
 [2, 2, 5], [2, 3, 3], [2, 3, 4], [2, 3, 5], [2, 4, 4], [2, 4, 5], [2, 5, 5], [3, 3, 3], [3, 3, 4],
 [3, 3, 5], [3, 4, 4], [3, 4, 5], [3, 5, 5], [4, 4, 4], [4, 4, 5], [4, 5, 5], [5, 5, 5]]
```

(6.144)

Computations and Explorations 1

Find the number of possible outcomes in a two-team playoff when the winner is the first team to win 5 out of 9, 6 out of 11, 7 out of 13, and 8 out of 15.

Solution: We will designate the two teams as 1 and 2 and model a playoff as a list of 1s and 2s. For example, $[1, 2, 2, 1, 1, 1, 2, 1]$ is a playoff in which team 1 wins the first game, team 2 wins games 2 and 3, team 1 wins games 4, 5, and 6, team 2 wins game 7, and team 1 wins game 8. If the winner is the first team to win 5 out of 9 games, then team 1 has won the tournament after 8 games.

We will write a procedure to produce all of the possible outcomes in a playoff where the winner is the first team to win n out of $2n - 1$ games.

First, we need to be able to determine, given a list of the outcomes of individual games and the number of games needed to win, whether a team has won the playoff or not. To do this, we will use

the **Collect** command from the **ListTools** package. When applied to a list, **Collect** returns a list of two-element lists. Each sublist contains a unique element of the input list followed by the number of times that object appeared in the input. For example,

```
> ListTools[Collect]([“a”, “b”, “b”, “c”, “c”, “c”, “d”, “d”, “d”, “d”])
[[“d”, 4], [“a”, 1], [“b”, 2], [“c”, 3]]
```

(6.145)

We will use this function to create a small procedure to determine whether or not the playoff tournament is over. The procedure will return true if one of the teams has won or false if neither team has reached the threshold for winning. Note that since we do not care which team won, we will take the output of **Collect** and immediately extract the count value by using **map** with a function that extracts the second member of each sublist. (An alternative approach to extracting the second elements of a list of sublists is illustrated in the solution to Computations and Explorations 3.)

```
> map (L → L[2], (6.145))
[4, 1, 2, 3]
```

(6.146)

We conclude the procedure by using **max** to compare the number of wins of whichever team has won more games with the threshold for winning the playoff.

```
1  playoffWon := proc (L : : list ( {1, 2} ) , n : : posint)
2    local collected;
3    collected := ListTools[Collect] (L);
4    evalb(max (map (L->L[2], collected) )>=n);
5  end proc;
```

For instance, in [1, 2, 2, 1, 1, 1, 2, 1], the procedure recognizes that the playoff has been won.

```
> playoffWon ([1, 2, 2, 1, 1, 1, 2, 1], 5)
true
```

(6.147)

With this helper procedure in place, we are ready to construct all possible outcomes. Begin with a set **outcomes** and a list **S**. Initialize **outcomes** to the empty set and **S** to the list [[1], [2]].

Consider the first element of **S**, say **p**. Remove **p** from the list. Then, construct the two lists formed by adding 1 and 2, respectively, to **p**. For each of these, use **playoffWon** to determine whether or not they are outcomes. If so, they are added to the **outcomes** set, and if not, they are added to the end of **S**. When **S** is empty, then **outcomes** consists of all possible outcomes of the playoff.

Here is the procedure.

```
1  allPlayoffs := proc (n : : posint)
2    local outcomes, S, p, p1, p2;
3    outcomes := {};
4    S := [[1], [2]];
5    while S <> [] do
6      p := S[1];
7      S := S[2..-1];
8      p1 := [op(p), 1];
9      p2 := [op(p), 2];
```

```

10      if playoffWon(p1, n) then
11          outcomes := outcomes union {p1};
12      else
13          S := [op(S), p1];
14      end if;
15      if playoffWon(p2, n) then
16          outcomes := outcomes union {p2};
17      else
18          S := [op(S), p2];
19      end if;
20  end do;
21  return outcomes;
22 end proc;

```

We now apply this procedure to playoffs that are best 3 out of 5.

```

> best3of5 := allPlayoffs(3)
best3of5 := {[1, 1, 1], [2, 2, 2], [1, 1, 2, 1], [1, 2, 1, 1], [1, 2, 2, 2],
[2, 1, 1, 1], [2, 1, 2, 2], [2, 2, 1, 2], [1, 1, 2, 2, 1], [1, 1, 2, 2, 2],
[1, 2, 1, 2, 1], [1, 2, 1, 2, 2], [1, 2, 2, 1, 1], [1, 2, 2, 1, 2], [2, 1, 1, 2, 1],
[2, 1, 1, 2, 2], [2, 1, 2, 1, 1], [2, 1, 2, 1, 2], [2, 2, 1, 1, 1], [2, 2, 1, 1, 2]}   (6.148)

```

```

> nops(best3of5)
20                                         (6.149)

```

The reader is left to apply the procedure to the cases called for in the problem and to conjecture a general formula.

Computations and Explorations 3

Verify that $C(2n, n)$ is divisible by the square of a prime, when $n \neq 1, 2$, or 4 , for as many positive integers n as you can. [The theorem that tells that $C(2n, n)$ is divisible by the square of a prime for $n \neq 1, 2$, or 4 was proved in 1996 by Andrew Granville and Olivier Ramaré. Their proof settled a conjecture made in 1980 by Paul Erdős and Ron Graham.]

Solution: We will first consider one example to see exactly what we need to do. Then, we will write a general procedure. Consider $n = 3$, the smallest n for which the theorem is true.

First, compute $C(2n, n)$ for $n = 3$.

```

> c := binomial(6, 3)
c := 20                                         (6.150)

```

To determine whether or not $C(2n, n)$ is divisible by the square of a prime, we look at its prime factorization. If any of the exponents in the prime factorization are 2 or greater, then we know the number is divisible by the square of the corresponding prime.

We use the command **ifactors** (first discussed in Section 4.3 of this manual). The **ifactors** command requires one argument, an integer. Its output is a list of the form $[sign, [[p_1, e_1], [p_2, e_2], \dots, [p_m, e_m]]]$

where $sign$ is positive or negative 1; p_1, p_2, \dots, p_m are the primes in the prime factorization; and e_1, e_2, \dots, e_m are the corresponding exponents.

Apply **ifactors** to $C(6, 3)$.

```
> ifactors(c)
[1, [[2, 2], [5, 1]]] (6.151)
```

The result tells us that $C(6, 3) = 1 \cdot 2^2 \cdot 5^1$.

We are interested in the exponents of the primes. We take the second element of the list to obtain the list of pairs of primes and their exponents without the sign component.

```
> cFacts := ifactors(c)[2]
cFacts := [[2, 2], [5, 1]] (6.152)
```

This gives us a list of lists, where the first element in each internal list is the prime factor and the second element in each pair is the exponent associated to that prime. Since we are only interested in the exponents, we separate them out. (An alternative approach to extracting the second elements of a list of sublists is illustrated in the solution to Computations and Explorations 1.)

```
> cPowers := seq(x[2], x=cFacts)
cPowers := 2, 1 (6.153)
```

Provided that there is an exponent of 2 or larger, we know that the square of a prime divides the number. In this case, the first exponent we obtained is 2, so in fact $C(6, 3)$ is divisible by the square of a prime.

Combining these steps, we write a procedure, **HasPrimeSquare**, that, given n , returns true if $C(2n, n)$ is divisible by the square of a prime and false if not.

```
1 HasPrimeSquare := proc(n::posint)
2   local c, facts, powers, x, e;
3   c := binomial(2*n, n);
4   facts := ifactors(c)[2];
5   powers := seq(x[2], x=facts);
6   for e in powers do
7     if e >= 2 then
8       return true;
9     end if;
10    end do;
11    return false;
12  end proc;
```

As in the solution of Computations and Explorations 2 in Chapter 5, we will use **timelimit** to test as many values of n as we can in a given amount of time. The **timelimit** command accepts an amount of time as its first argument and a procedure call as its second. If the execution of the command takes longer than the amount of time allowed, a “time expired” error will be raised. Since we will

be using **timelimit** to control an otherwise infinite loop, we are expecting the error and so will use a try...catch block to catch the time limit error.

First, we need to write the procedure that will execute the infinite loop. Note that the name storing the integer being checked is made global so that it can be printed once time has expired.

```
1 CheckPrimeSquares := proc()
2   global n;
3   for n from 1 do
4     if not HasPrimeSquare(n) then
5       print(cat("Found counterexample: ", n));
6     end if;
7   end do;
8 end proc;
```

Now we execute this procedure for 1 second and verify that the only small values of n for which $C(2n, n)$ is not divisible by the square of a prime are 1, 2, and 4, as expected.

```
> try
  timelimit(1, CheckPrimeSquares());
  catch "time expired":
    print(cat("Checked through n = ", n - 1));
  end try;
  "Found counterexample: 1"
  "Found counterexample: 2"
  "Found counterexample: 4"
  "Checked through n = 442" (6.154)
```

Exercises

Exercise 1. Build a recursive version of **SubsetSumCount**, using the ideas of **FindBitStrings**.

Your procedure should determine all subsets of a given set whose sum is less than a target value. Rather than considering all sets, it should build potential sets recursively using the fact that once a set has sum larger than the target no larger set of positive integers can have smaller sum. Compare the performance of your procedure with **SubsetSumCount**.

Exercise 2. Create a procedure **FindDecreasing** by modifying **FindIncreasing** in order to determine a strictly decreasing subsequence of maximal length.

Exercise 3. Modify the Patience algorithm to find *all* of the strictly increasing subsequences of maximal length.

Exercise 4. Use Maple to find an example demonstrating that n positive integers not exceeding $2n$ are not sufficient to guarantee that one integer divides one of the others (see Example 11 of Section 6.2).

Exercise 5. The functions for comparing and generating circular permutations, **CPEquals** and **allCP**, are inefficient. Using the idea that explains the formula for the number of circular r -permutations of n people, write more efficient procedures.

Exercise 6. Use Maple to determine how many different strings can be made from the word “PAPARAZZI” when all the letters are used, when any number of letters are used, when all the letters are used and the string begins and ends with the letter “Z”, and when all the letters are used and the three “A’s are consecutive.

Exercise 7. Suppose that a certain Department of Mathematics and Statistics has m mathematics faculty and s statistics faculty. Write a Maple procedure to find all committees with $2k$ members in which both mathematicians and statisticians are represented equally.

Exercise 8. Use Maple to prove the identity

$$\binom{n+1}{k} = \frac{n+1}{k} \binom{n}{k-1}$$

for positive integers n and k with $k \leq n$.

Exercise 9. Use Maple to prove Pascal’s identity:

$$\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}$$

for all positive integers n and k with $k \leq n$.

Exercise 10. Use Maple to generate many rows of Pascal’s triangle. See if you can formulate any conjectures involving identities satisfied by the binomial coefficients. Use Maple to help you verify that your conjecture is true by using the techniques at the end of Section 6.4 of this manual.

Exercise 11. Write a procedure that mixes the techniques used in **BinomialF** and **BinomialR** to generate the rows of Pascal’s triangle from row a to row b for $b > a > 0$.

Exercise 12. Use Maple to count and list all solutions to the equation

$$x_1 + x_2 + x_3 + x_4 = 25,$$

where x_1, x_2, x_3 , and x_4 are nonnegative integers. Also count and list all solutions such that $x_1 \geq 1, x_2 \geq 2, x_3 \geq 3$, and $x_4 \geq 4$.

Exercise 13. Generate a large triangle of Stirling numbers of the second kind and look for patterns that suggest identities among the Stirling numbers. In addition, see if you can make any conjectures about the relationship between Stirling numbers and the binomial coefficients.

Exercise 14. Implement the algorithm described in the “Generating Partitions” subsection of Section 6.5 of this manual.

Exercise 15. Write a procedure that generates all possible schedules for airplane pilots who must fly d days in a month with m days with the restrictions that they cannot work on consecutive days (see Exercise 22 in Section 6.5).

Exercise 16. Write a Maple procedure that takes as input three positive integers n , k , and i , and returns the i th multinomial, in lexicographic order, of the polynomial $(x_1 + x_2 + \cdots + x_k)^n$. Write its inverse; that is, given a multinomial, the inverse should return its index (position) in the sorted polynomial.

Exercise 17. Implement Algorithm 2 of Section 6.6 for generating the next largest bit string.

Exercise 18. Implement Algorithm 3 of Section 6.6 for generating the next r -combination.

Exercise 19. Write a Maple procedure to compute the Cantor expansion of an integer. (See the preamble to Exercise 14 of Section 6.6 of the text.)

Exercise 20. Implement the algorithm for generating the set of all permutations of the first n integers using the bijection from the collection of all permutations of the set $\{1, 2, \dots, n\}$ to the set $\{1, 2, \dots, n!\}$ described prior to Exercise 14 of Section 6.6 of the textbook.

7 Discrete Probability

Introduction

In this chapter, we will see how to use Maple to perform computations in discrete probability and how to use Maple's capabilities to explore concepts of discrete probability. We will continue to make use of the **combinat** package described in the previous chapter. We will also introduce the **Statistics** package. While the **Statistics** package has the support for descriptive statistics and visualization of data that you would expect, it also provides functionality that will help us explore probability distributions and random variables. This includes the ability to create a random variable with a defined distribution and then perform computations with that variable. We load both packages now.

```
> with(combinat):  
> with(Statistics):
```

In this chapter, we will make use of simulations to help explore concepts in discrete probability. In this context, a simulation refers to a computer program, that models a real physical system. For example, instead of flipping an actual coin 100 times and recording whether each flip resulted in “heads” or “tails,” we could write a program that uses random numbers to generate a sequence of one hundred “heads” or “tails.”

Simulations are useful in discrete probability from two different perspectives. First, they can help analyze and/or confirm probabilities for systems that are difficult to compute deductively. For example, Computations and Explorations 7 from the text asks you to simulate the odd-person-out procedure in order to confirm your deductive calculations. Second, simulations can be very helpful as a way to better understand a problem and how to arrive at a solution. For example, in the Computer Projects 10 exercise from the text, you are asked to build a simulation of the famous Monty Hall Three-Door problem. Building the simulation and analyzing the results can help improve your understanding of the reasons why the strategy described in the text is the best possible.

7.1 An Introduction to Discrete Probability

To find the probability of an event in a finite sample space, you calculate the number of times the event occurs and divide by the total number of possible outcomes (the size of the sample space).

As in Example 4 of Section 7.1, we calculate the probability of winning a lottery, where we need to choose 6 numbers correctly out of 40 possible numbers. The total number of ways to choose 6 numbers is:

```
> numbcomb(40, 6)  
3 838 380
```

(7.1)

Since there is one winning combination, the probability is

$$> \frac{1}{\text{numbcomb}(50, 6)}$$
$$\frac{1}{3838380} \quad (7.2)$$

We can find a real number approximation by using the **evalf** function—evaluate as a floating point number.

$$> \text{evalf}((7.2))$$
$$2.605265763 \cdot 10^{-7} \quad (7.3)$$

We could also force a decimal approximation of the result by using **1.0** or simply **1.**, to show that we wish to work with decimals instead of the exact rational representation. For example, if we needed to choose from 50 numbers, the probability is

$$> \frac{1.0}{\text{numbcomb}(50, 6)}$$
$$6.292988981 \cdot 10^{-8} \quad (7.4)$$

Continuing with this type of example, we define a functional operator that computes the probability of winning a lottery where 6 numbers must be matched out of n possible numbers.

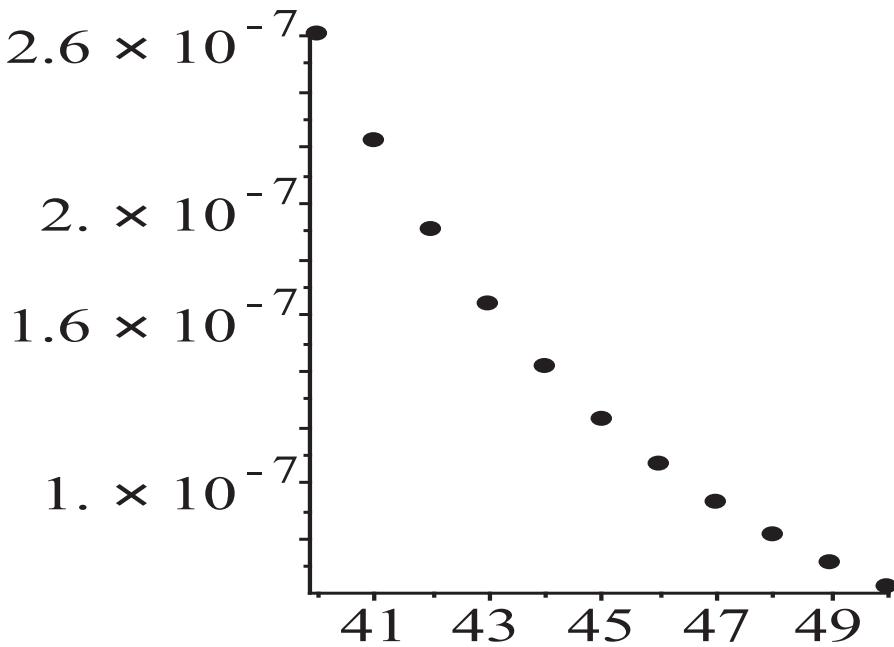
$$> \text{Lottery} := n \rightarrow \frac{1.0}{\text{numbcomb}(n, 6)} :$$

The probabilities above can then be computed with the function.

$$> \text{Lottery}(40), \text{Lottery}(50)$$
$$2.605265763 \cdot 10^{-7}, 6.292988981 \cdot 10^{-8} \quad (7.5)$$

We can use the sequence function **seq** to look at a list of probabilities for a range of values of n and graph these values to visualize how the number of possible values in the lottery affects the probability of choosing the correct values.

$$> \text{LotteryVals} := [\text{seq}(\text{Lottery}(n), n = 40 .. 50)]$$
$$\text{LotteryVals} := [2.605265763 \cdot 10^{-7}, 2.224007359 \cdot 10^{-7},$$
$$1.906292022 \cdot 10^{-7}, 1.640297786 \cdot 10^{-7}, 1.416620815 \cdot 10^{-7},$$
$$1.227738040 \cdot 10^{-7}, 1.067598296 \cdot 10^{-7}, 9.313091515 \cdot 10^{-8},$$
$$8.148955076 \cdot 10^{-8}, 7.151123842 \cdot 10^{-8}, 6.292988981 \cdot 10^{-8}] \quad (7.6)$$
$$> \text{plot}([\$40 .. 50], \text{LotteryVals}, \text{style} = \text{point}, \text{symbol} = \text{solidcircle}, \text{symbolsize} = 15)$$



Recall that the dollar operator with no left operand and a range as the right operand, as in **\$40.0.50**, produces the sequence described by the range. Refer to Section 2.3 of this manual for detailed information on the use of **plot**.

7.2 Probability Theory

We can use Maple to perform a variety of calculations of probabilities.

Random Variables

Consider Example 9 in Section 7.2, which asks us to calculate the probability that eight of the bits in a string of 10 bits are 0s if the probability of a 0 bit is 0.9, the probability of a 1 bit is 0.1, and the bits are generated independently. To perform this calculation, we can input the formula directly:

$$\begin{aligned} > \text{numbcomb}(10, 8) 0.9^8 0.1^2 \\ & 0.1937102445 \end{aligned} \tag{7.7}$$

We can think of this same question in terms of a random variable. Specifically, consider a random variable X that assigns to each string of 10 bits the number of the bits that are 0s. Then, the probability that eight of the bits were 0 is $P(X = 8)$.

To define a random variable in Maple, we use the **RandomVariable** command in the **Statistics** package. The **RandomVariable** command takes one parameter: a probability distribution which serves to specify the probabilities of the possible values of the random variable. The distribution can be one of Maple's built-in distributions or it can be a distribution you define.

Discrete Distributions

Maple provides many different probability distributions, including several discrete distributions. All of the commands described here are inert, that is, they do nothing on their own. Instead, they are used as the argument to the **RandomVariable** command, which is able to interpret them to create a random variable with the desired distribution.

Theorem 2 defines the binomial distribution, implemented in Maple as **Binomial**. The **Binomial** distribution takes two parameters: the number of independent Bernoulli trials and the probability of a success. For the bit string example that began this chapter, there are 10 trials, and we interpret success to be a 0 bit, so the probability of success is 0.9.

```
> RandomVariable(Binomial(10, 0.9))  
_R
```

(7.8)

Note that the output is a symbol consisting of an underscore followed by the letter R. After the first random variable, a number follows the R. This merely indicates that the result is a random variable and the number indicates how many random variables have been defined previously in this Maple session. In the future, we will suppress this output.

Related to the binomial distribution is the probability distribution of a single Bernoulli trial. The **Bernoulli** distribution takes only one parameter, the probability of success. The following creates the random variable associated to a single trial with probability of success 0.9.

```
> RandomVariable(Bernoulli(0.9)) :
```

Definition 1 in Section 7.1 defines the uniform distribution. Maple includes the distribution **DiscreteUniform**, which is different from the **Uniform** distribution (for continuous random variables). The **DiscreteUniform** distribution requires two arguments, the lower and upper bounds of the distribution. For example, the following produces a random variable distributed uniformly on {1, 2, 3, 4, 5}.

```
> RandomVariable(DiscreteUniform(1, 5)) :
```

Definition 2 in Section 7.4 defines the geometric distribution. The **Geometric** distribution in Maple requires one argument, the probability of success. The following produces the random variable associated to Example 10 from that section.

```
> RandomVariable(Geometric(0.5))
```

Computing Probabilities from a Random Variable

Once a random variable is defined, we use the **Probability** command to calculate probabilities. The probability command's required argument is an event, described as a relation involving a random variable.

Earlier, we described Example 9 in Section 7.2. The random variable associated to that example is defined by the following command.

```
> Ex9 := RandomVariable(Binomial(10, 0.9)):
```

We compute the probability that there are eight 1 bits, that is, $P(X = 8)$, by applying the **Probability** command to the equation that sets the name of the random variable, **Ex9**, equal to 8.

```
> Probability(Ex9 = 8)  
0.1937102445  
(7.9)
```

The probability of at least eight 1 bits is found with an inequality. Recall that in 2-D input mode, the \geq symbol is obtained by pressing the greater than key followed by the equals key.

```
> Probability(Ex9 >= 8)  
0.9298091736  
(7.10)
```

The probability of an intersection of events is found by passing a set of relations to **Probability**. For example, the probability that the number of 1s is between 5 and 8 can be thought of as the intersection $\{X \geq 5\} \cap \{X \leq 8\}$. This is computed in Maple as follows:

```
> Probability({Ex9 >= 5, Ex9 <= 8})  
0.2637541683  
(7.11)
```

To find probabilities of unions of events, you must add the results of the **Probability** command applied to each event separately. $P(\{X \leq 5\} \cup \{8 \leq X\})$.

```
> Probability(Ex9 <= 5) + Probability(Ex9 >= 8)  
0.9314441110  
(7.12)
```

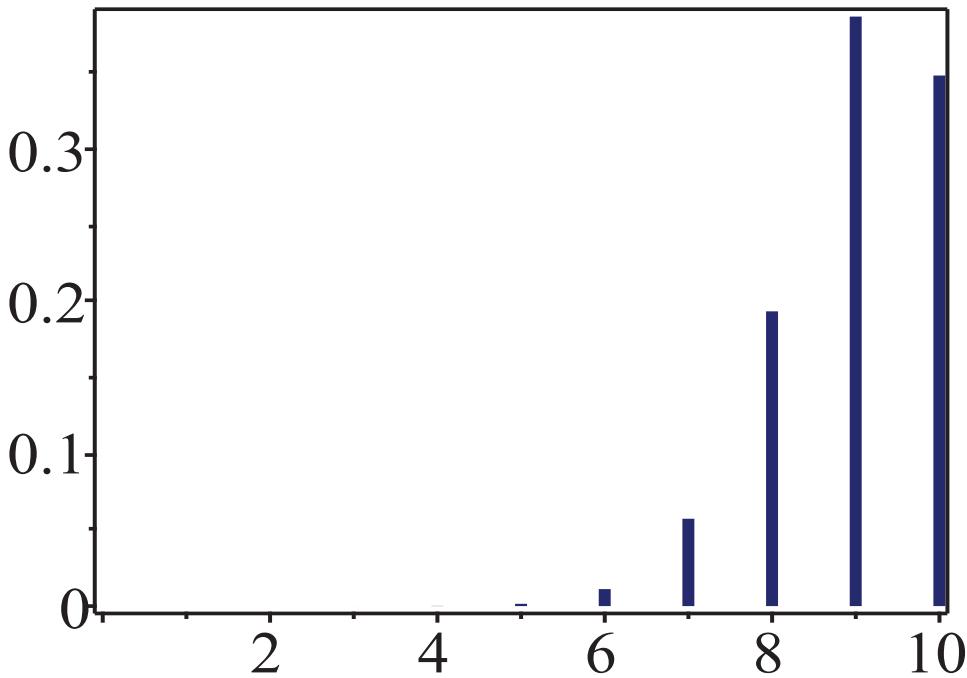
Of course, you must be careful that the events are disjoint in order to add the probabilities.

Graphing Probabilities

It is often useful to graph the probabilities associated to the values of a random variable. To do this, Maple provides the **DensityPlot** command.

The required argument is the name of a random variable. You can also specify the range of values of the random variable to be displayed, as illustrated below. Omitting the range will cause Maple to determine the size of the plot for itself.

```
> DensityPlot(Ex9, range = 0 .. 10)
```



The **DensityPlot** command also accepts most of the options that are available for the **plot** command.

Defining Random Variables from Your Own Data

Maple provides two convenient ways for you to define probability distributions, and thereby random variables, of your own.

First, to define specific probabilities associated to the integers 1 through n , you can use the **ProbabilityTable** command. The argument to **ProbabilityTable** is a list of real numbers between 0 and 1 that sum to 1. For example, to create a random variable with probability of 1 as $1/2$, probability of 2 equal to $1/8$ and probability of 3 is $3/8$, you enter the following statement.

```
> ProbT := RandomVariable(ProbabilityTable([1/2, 1/8, 3/8])):
```

The resulting random variable can be used as any other.

```
> Probability(ProbT = 2)
1/8
(7.13)
```

If, instead of a list of probabilities, you have a list representing the results of experiments, you can use the **EmpiricalDistribution**. This requires one argument, which can be a list or Array.

For example, suppose you manually roll a die 20 times and obtain the following results:

```
> dieRolls := [3, 2, 1, 1, 5, 2, 3, 6, 5, 1, 2, 5, 6, 4, 4, 3, 1, 3, 1, 1]
dieRolls := [3, 2, 1, 1, 5, 2, 3, 6, 5, 1, 2, 5, 6, 4, 4, 3, 1, 3, 1, 1]
(7.14)
```

Form a random variable by applying **EmpiricalDistribution** to the list.

```
> dieRV := RandomVariable(EmpiricalDistribution(dieRolls)):
```

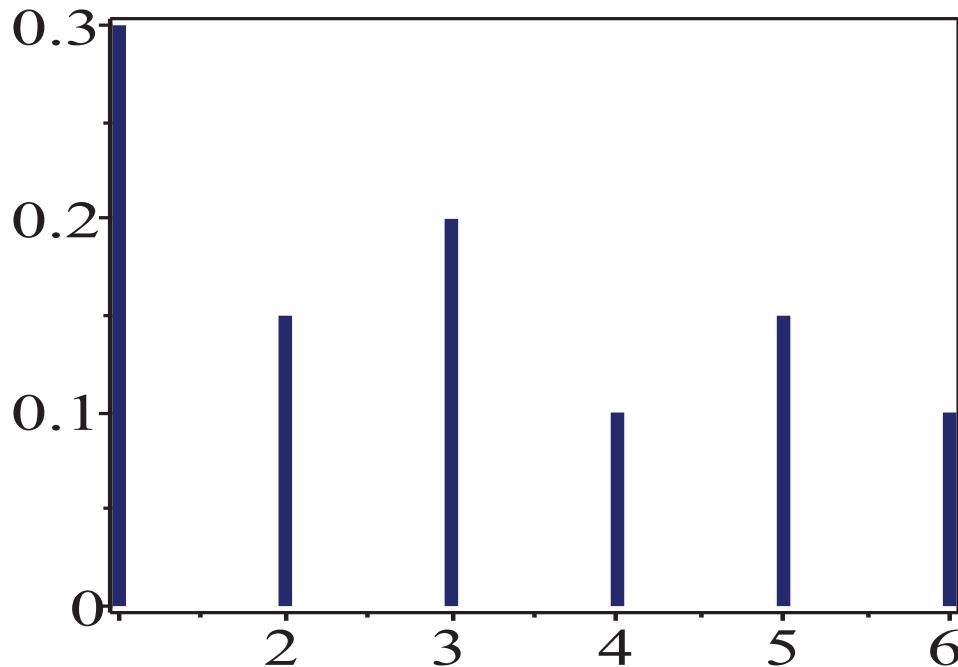
Now, you can compute probabilities and graph the distribution.

```
> Probability(dieRV ≥ 4)
```

$$\frac{7}{20}$$

(7.15)

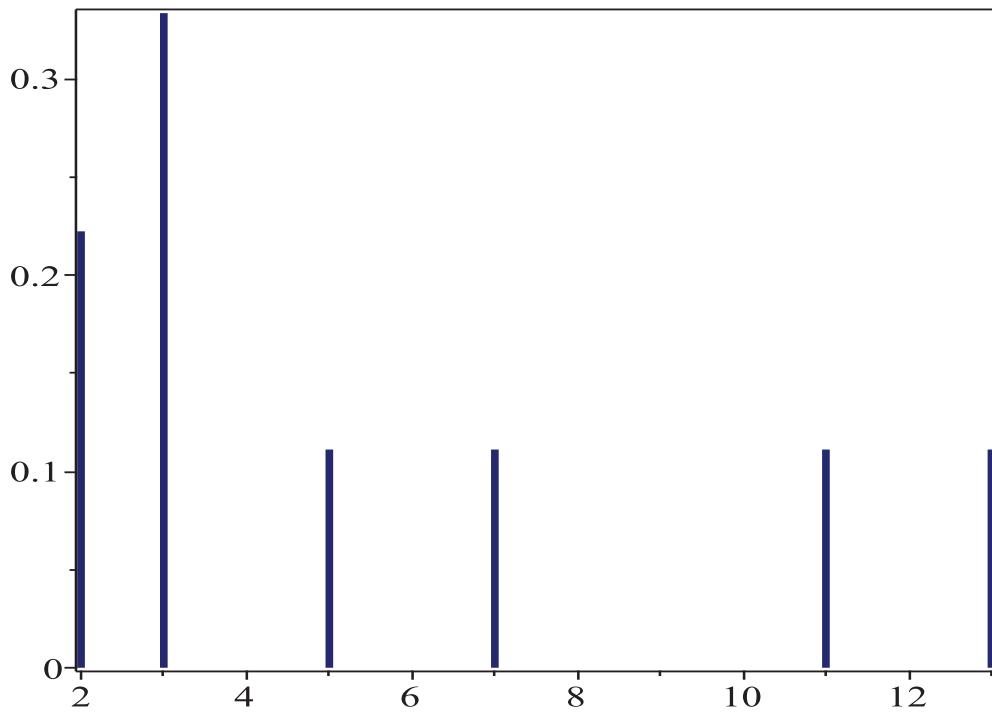
```
> DensityPlot(dieRV)
```



The **EmpiricalDistribution** command also allows you to specify the probabilities of each outcome by using the **probabilities** option. This is like **ProbabilityTable**, but more flexible in that **ProbabilityTable** insists that the outcomes be the positive integers. With **EmpiricalDistribution**, you can provide a list of values for the outcomes as the first argument and a list of probabilities as the value associated to the **probabilities** option. For example, the following creates a random variable whose values are the first six prime numbers.

```
> primeRV := RandomVariable(EmpiricalDistribution([2, 3, 5, 7, 11, 13],  
probabilities = [2/9, 1/3, 1/9, 1/9, 1/9, 1/9])):
```

```
> DensityPlot(primeRV)
```



Combining Random Variables

Most interesting questions in probability arise from combining random variables. For example, consider two loaded dice. One is weighted so that the probability that a 1 appears is $2/7$, and the probabilities of all other values are $1/7$. The other is weighted so that the probability of a 4 appearing is $3/8$, and the probabilities of all other values are $1/8$. What is the probability that the sum is 7 when the two dice are rolled?

To answer this question, we first define two random variables. Since the values of each die is 1 through 6, we can use the **ProbabilityTable** function to define the distribution.

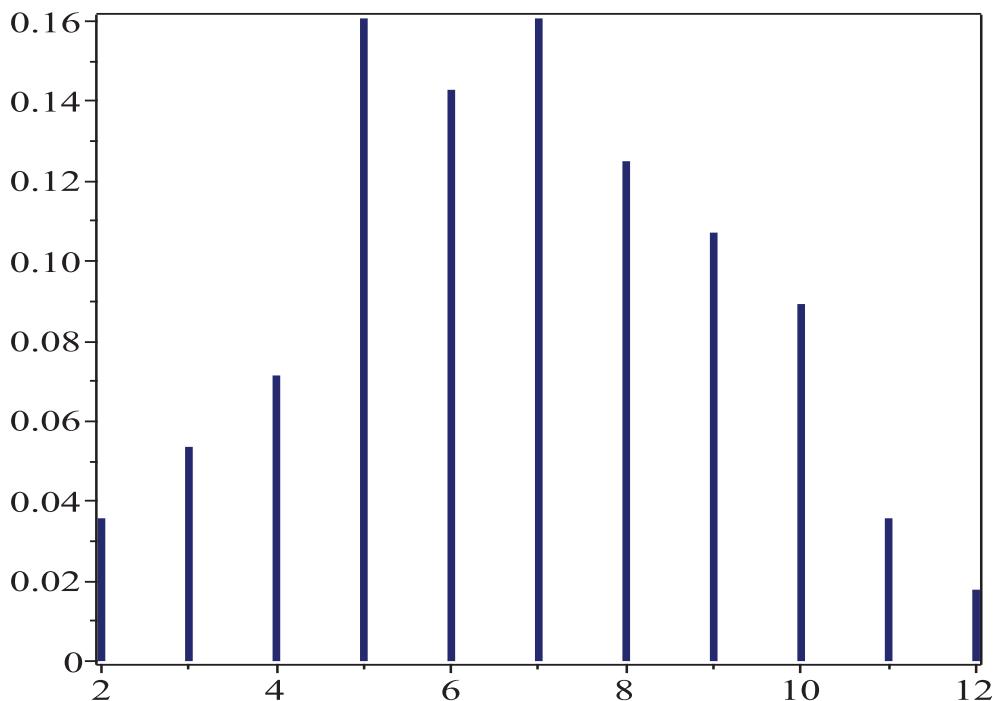
```
> die1 := RandomVariable(ProbabilityTable([2/7, 1/7, 1/7, 1/7, 1/7, 1/7]):  
> die2 := RandomVariable(ProbabilityTable([1/8, 1/8, 1/8, 3/8, 1/8, 1/8])):
```

The question is about the sum of the values on the dice, so we apply **Probability** to the sum of the random variables.

```
> Probability(die1 + die2 = 7)  
9  
56  
(7.16)
```

We can also use the sum of the random variables as the argument to **DensityPlot**.

```
> DensityPlot(die1 + die2)
```



Sampling

Once you have defined a random variable, you may wish to use it to conduct experiments or simulations. To do this, you use the **Sample** command.

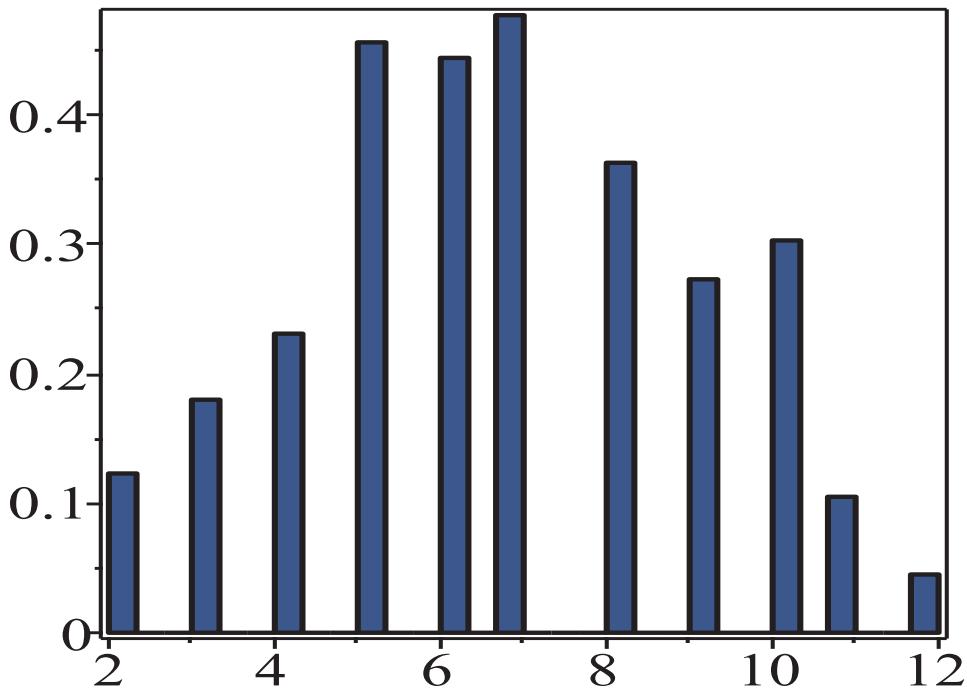
Sample requires two arguments: a random variable or a distribution as the first argument, and a positive integer indicating the sample size as the second argument. It produces a row vector containing results obtained by randomly choosing values in accordance with the random variable.

The following simulates rolling the pair of weighted dice **die1** and **die2** 1000 times.

```
> dieSample := Sample(die1 + die2, 1000);
dieSample := 
$$\begin{bmatrix} 1..1000 \text{ Vector}_{\text{row}} \\ \text{Data Type: } \text{float}_8 \\ \text{Storage: } \text{rectangular} \\ \text{Order: } \text{Fortran\_order} \end{bmatrix}$$
 (7.17)
```

We can use the **Histogram** command to draw a histogram of the data. You see that the data produced has approximately the same distribution as the density plot of the sum of the variables.

```
> Histogram(dieSample)
```



Monte Carlo Methods

We can also implement Monte Carlo algorithms using Maple. Miller's test for base b is described in the preamble to Exercise 44 in Section 4.4 of the textbook. In that description, it is mentioned that a composite integer n passes Miller's test for base b for fewer than $n/4$ bases less than n , and Exercise 44 asked you to show that primes pass Miller's test for all bases that they do not divide. In other words, Miller's test is a probabilistic primality test that fails less than one-fourth of the time. In this section, we use Miller's test to create a Monte Carlo primality testing algorithm.

Miller's Test

First, we must implement Miller's test for base b . Recall the description preceding Exercise 44 in Section 4.4. Let n and b be positive integers. Assume s is a nonnegative integer and t is an odd positive integer such that $n - 1 = 2^s t$. If $b^t \equiv 1 \pmod{n}$ or if there is a j with $0 \leq j \leq s - 1$ such that $b^{2^j t} \equiv -1 \pmod{n}$, then n is said to pass Miller's test for base b .

To implement Miller's test, we first must calculate s and t . Initialize s to 0 and set t equal to $n - 1$. If t is even, we add 1 to s and divide t by 2. When t is no longer even, then s and t are the correct values.

Once s and t have been calculated, we check the congruence $b^t \equiv 1 \pmod{n}$. If that congruence is satisfied, then n passes Miller's test and we return true. Otherwise, we begin testing the congruences $b^{2^j t} \equiv -1 \pmod{n}$. A for loop assigns j to each integer from 0 to $s - 1$ and inside the for loop, the congruence is tested. If any congruence holds, the procedure returns true. (Recall from Section 4.1 of this manual that **modp** returns the smallest positive integer congruent to its first argument modulo its second argument. Thus we test for congruence to -1 modulo n by comparing the result to $n - 1$.) If the procedure completes without having returned true, then it returns false.

```

1 | Miller := proc (n :: posint, b :: posint)
2 |   local s, t, j;
```

```

3   s := 0;
4   t := n-1;
5   while modp(t, 2) = 0 do
6       t := t/2;
7       s := s + 1;
8   end do;
9   if modp(b^t, n) = 1 then
10      return true;
11   end if;
12   for j from 0 to s-1 do
13       if modp(b^(2^j*t), n) = n-1 then
14           return true;
15       end if;
16   end do;
17   return false;
18 end proc;

```

Monte Carlo Primality Test

Now, we use Miller's test to implement a Monte Carlo primality testing algorithm, as described in Example 16 in Section 7.2 of the text. The question the Monte Carlo algorithm is going to answer is “Is n composite?” for an integer n . For each iteration, the algorithm will select a random base b with $1 < b < n$ and check to see if n passes Miller's test for base b . If Miller's test returns false, then we know that n is composite and the Monte Carlo algorithm will return true, indicating that yes, n is composite. If Miller's test returns true, then the iteration results in “unknown” and the next iteration is started. After 30 iterations, if Miller's test has only resulted in true, then the algorithm will return false, indicating that it is very likely that the number is prime. Since Miller's test falsely identifies a composite as prime less than one-fourth of the time, the probability that the Monte Carlo algorithm will incorrectly identify a composite number as prime is

$$> \left(\frac{1}{4}\right)^{30} \\ 8.673617380 \cdot 10^{-19} \quad (7.18)$$

Here is the Miller Monte Carlo test:

```

1 MillerMC := proc (n :: integer) :: string;
2   local gen, b;
3   gen := rand(2..n-1);
4   from 1 to 30 do
5     b := gen();
6     if (not Miller(n, b)) then return "composite" end if;
7   end do;
8   return "prime";
9 end proc;

```

Note the use of the **rand** command. This command, when passed a range like **2..n-1**, does *not* produce a random number between 2 and $n - 1$. Rather, when used this way, the **rand** command

returns a procedure that generates a random number in the specified range. In our algorithm, we assign the variable **gen** to the random number generator created by **rand(2..n-1)**, and the assignment **b := gen()** produces a random number between 2 and $n - 1$ and stores that value in **b**.

We use **MillerMC** to test a random integer to see if it is prime. We can use Maple's **ithprime** function to find the 40 000th prime and then check that our algorithm confirms that it is prime.

> *ithprime*(40 000)
 479 909 (7.19)

> *MillerMC*(479 909)
 "prime" (7.20)

7.3 Bayes' Theorem

Section 7.3 focuses on applications of Bayes' theorem, which asserts that for events E and F from a sample space S with $p(E) \neq 0$ and $p(F) \neq 0$, one has

$$p(F | E) = \frac{p(E | F)p(F)}{p(E | F)p(F) + p(E | \bar{F})p(\bar{F})}.$$

The text describes how to use this theorem to create a Bayesian spam filter. We will use Maple to implement such a filter.

Recall the notation from the text. A message is received containing the word w . The event S will be the event that the message is spam and the event E is the event that the message contains the word w . If we assume that a message is as likely to be spam as not, so that $p(S) = p(\bar{S}) = 1/2$, then Bayes' theorem tells us that the probability that the incoming message is spam given that it contains the word w is:

$$p(S | E) = \frac{p(E | S)}{p(E | S) + p(E | \bar{S})}.$$

By estimating the conditional probabilities with empirical data, we can compute an estimate that the given message is spam.

Before building the spam filter, we will first need messages to serve as spam and nonspam. For the spam messages, we will use the sonnets of William Shakespeare, and for the nonspam messages, we will use sonnets written by Shakespeare's contemporaries, Michael Drayton, Bartholomew Griffin, and William Smith, published in the book *Elizabethan Sonnet Cycles*.

It may seem strange to consider Shakespeare's sonnets to be spam, but consider the goals and methods of a Bayesian spam filter. The goal of a spam filter is to filter out the "junk mail." In the case of the Bayesian filter described by the text, these filters work by comparing the specific words used by authors of spam in contrast to authors of nonspam messages. Think about email messages you receive from your classmates versus messages your professors may send you. Chances are good

that you and your peers use more slang and generally less formal English when writing to each other than you and your professor use when communicating. This applies to kinds of message writers like peers versus professors, but it also can apply to individual message writers, like a mathematics professor versus a literature professor. A literature professor, for example, is not likely to use words like “Bayes’ theorem” in an email to you. A Bayesian spam filter can pick up on these differences in word choice and effectively filter messages based on the assumption that different authors generally use different words. We will see, by comparing Shakespeare with other Elizabethan sonnet writers, that a Bayesian filter can even distinguish one author from others writing at the same time, for the same audience, and in a very similar style.

Obtaining Data

On the website for this manual, you will find these three files: “ShakespeareData.txt”, “ElizabethanData.txt” and “testMessages.txt”. The first two contain the sonnets of Shakespeare and the other authors, respectively. Five of Shakespeare’s poems and five of the other authors’ poems were randomly selected and moved to the “testMessages.txt” file. We will use our “spam” filter on the poems in this file to determine which of them were written by Shakespeare and which were not.

Begin by downloading the three files and storing them in the same directory as this Maple Worksheet. Then, load the three files and store the text in variables using the **ReadFile** command. Note that the commands below are set up to obtain the directory in which this Worksheet is saved.

```
> shakespeare := FileTools[Text][ReadFile](  
    FileTools[JoinPath]([“ShakespeareData.txt”), base = worksheetdir)) :  
  
> elizabethan := FileTools[Text][ReadFile](  
    FileTools[JoinPath]([“ElizabethanData.txt”), base = worksheetdir)) :  
  
> test := FileTools[Text][ReadFile](  
    FileTools[JoinPath]([“testMessages.txt”), base = worksheetdir)) :
```

If your worksheet is not in the same location as the text files, you may need to replace the argument of **ReadFile** with the full path. The command below will certainly not work on your computer, but illustrates the command.

```
> test := FileTools[Text][ReadFile](  
    “/Users/danieljordan/DiscreteMath/testMessages.txt”)
```

The following illustrates how to have Maple produce a file dialog window in order to load the file. Note that this input has been made nonexecutable, so that it will not create the dialog if you execute the entire worksheet.

```
> test := FileTools[Text][ReadFile](Maplets[Utilities][GetFile])(  
    ‘title’ = “Open Text File”, ‘directory’ = currentdir(homedir),  
    ‘filefilter’ = “txt”)) :
```

If you inspect the files in a text editor, you will see that the sonnets are separated by three ampersands (“&&&”). The three commands above store each of the files as a single string. It would be more useful to store them as lists of strings, with each sonnet being one element of a list. To separate the files into lists, we use **RegSplit** from the **StringTools** package.

```

> SPoems := [StringTools[RegSplit](“&&&”, shakespeare)]:
> EPoems := [StringTools[RegSplit](“&&&”, elizabethan)]:
> testPoems := [StringTools[RegSplit](“&&&”, test)]:

```

The **RegSplit** command splits the string in the second argument based on the pattern given in the first argument. In this case, the pattern is the string “ $\&\&\&$ ” so the **RegSplit** command uses that pattern as a delimiter in the **shakespeare**, **elizabethan**, and **test** strings to separate them into lists. The pattern can be a string, as we use it here, or it can be a regular expression, hence the name **RegSplit**. Now that the “messages” are prepared, we begin building the filter.

Estimating the Probabilities

The spam filter relies on two computations: first, the probability that a message contains a word given that it is spam, and second, the probability that a message contains the word given that it is not spam. That is, we will need empirical estimates for $p(E | S)$ and $p(E | \bar{S})$.

Following the notation of the textbook, for a word w , let $p(w)$ be the estimate of $p(E | S)$, the probability that a message contains w given that it is spam. Therefore, $p(w)$ is the number of spam messages containing the word w divided by the number of spam messages. Likewise, let $q(w)$ be the estimate for $p(E | \bar{S})$, the probability that a message contains w given that it is not spam. This is computed as the number of nonspam messages containing w divided by the number of nonspam messages.

Counting the number of messages (i.e., poems) in each list can be done with **nops**.

```

> nops(SPoems)
149
(7.21)

```

(This is five less than the 154 sonnets that Shakespeare published, because five of them were moved to the “testMessages.txt” file as “unknown” messages.)

To count the number of messages that contain a particular word, we make use of two functions. First, we use the **Words** command in the **StringTools** package to separate a string into its component words and remove punctuation. For example:

```

> exampleWords := StringTools[Words](“To count the number of
messages that contain a particular word we’ll make use of two functions”)
exampleWords := [“To”, “count”, “the”, “number”, “of”, “messages”,
“that”, “contain”, “a”, “particular”, “word”, “we’ll”, “make”, “use”,
“of”, “two”, “functions”]
(7.22)

```

Second, we use the **ListTools Search** function to determine if a message contains a particular word. **Search(element, L)** returns the index of the first occurrence of **element** in the list **L**. If the element is not in the list, then it returns 0. For example:

```

> ListTools[Search](“of”, exampleWords)
5
(7.23)

```

> *ListTools[Search]*(“elephant”, *exampleWords*)

0

(7.24)

Putting these together, we can create a procedure for counting the number of times a word appears in a list of messages as follows. For each message in the list, we use the **Words** function to separate the message into words. Then, we use the **Search** function to see if the word we are looking for is in the sonnet. If the **Search** function returns a value greater than 0, then we know the word is in the message and we increment a counter. Here is the procedure:

```

1 countMessages := proc (w : :string, L : :list)
2   local count, m, P;
3   count := 0;
4   for m in L do
5     P := StringTools[Words](m);
6     if (ListTools[Search](w, P) > 0) then
7       count := count + 1;
8     end if;
9   end do;
10  return count;
11 end proc;
```

For instance, we can see in how many sonnets Shakespeare uses the word “fairest”:

> *countMessages*(“fairest”, *SPoems*)

4

(7.25)

The empirical probability that a sonnet contains the word “fairest” given that it was written by Shakespeare is:

> $\frac{\text{countMessages}(\text{“fairest”}, \text{SPoems})}{\text{nops}(\text{SPoems})}$

$\frac{4}{149}$

(7.26)

The probability that a sonnet contains the word “fairest” given that it was written by one of our other authors is:

> $\frac{\text{countMessages}(\text{“fairest”}, \text{EPoems})}{\text{nops}(\text{EPoems})}$

$\frac{10}{173}$

(7.27)

Applying Bayes’ theorem, we can compute the probability that a sonnet was written by Shakespeare given that it contains the word “fairest”:

> *evalf* $\left(\frac{(7.26)}{(7.26) + (7.27)}\right)$

0.3171402383

(7.28)

The above computation illustrates how to write a procedure to compute the probability that a sonnet is spam (i.e., “written by Shakespeare”) given that it contains a specific word:

```

1 PShakespeareGivenWord := proc (w::string)
2   local SCount, ECount, PWordGivenS, PWordGivenNotS;
3   global SPoems, EPoems;
4   SCount := nops(SPoems);
5   ECount := nops(EPoems);
6   PWordGivenS := countMessages(w, SPoems) / SCount;
7   PWordGivenNotS := countMessages(w, EPoems) / ECount;
8   return evalf(PWordGivenS / (PWordGivenS + PWordGivenNotS));
9 end proc;
```

For example, the probability that a sonnet is Shakespearean given that it contains the word “beauty” is:

```

> PShakespeareGivenWord("beauty")
0.5601371298
(7.29)
```

Using Multiple Words

We can improve the filter by using multiple words, rather than just one. Using the notation of the text, let $p(w_i)$ and $q(w_i)$ be the probability that a message contains word w_i given that it is spam and that it is not spam, respectively. Then, the probability that a message is spam given that it contains all of the words w_1, w_2, \dots, w_k is:

$$r(w_1, w_2, \dots, w_k) = \frac{\prod_{i=1}^k p(w_i)}{\prod_{i=1}^k p(w_i) + \prod_{i=1}^k q(w_i)}.$$

The **mul** command is useful here. Recall that we compute $\prod_{i \in S} i^2$ for $S = \{1, 3, 5, 7, 9\}$, with:

```

> S := [1, 3, 5, 7, 9]:
> mul(i^2, i in S)
893 025
(7.30)
```

For instance, to compute the probability that a message contains the words “from”, “fairest”, and “creatures”,

```

> S := ["from", "fairest", "creatures"]:
> mul((countMessages(w, SPoems) / nops(SPoems)), w in S)
416
3 307 949
(7.31)
```

We can modify our **PShakespeareGivenWord** procedure to work on lists of words instead of single words by putting the probability computations inside of **mul** commands. It is also a good idea to protect against division by zero errors, so we will put the division inside of an if statement. This is needed in case one or more of the selected words appears in none of the sonnets by either Shakespeare or the other authors. In this case, we default to a probability of 0.5.

```

1 PShakespeareGivenList := proc (L :: list)
2   local SCount, ECount, PGivenS, w, PGivenNotS;
3   global SPoems, EPoems;
4   SCount := nops (SPoems) ;
5   ECount := nops (EPoems) ;
6   PGivenS := mul (countMessages (w, SPoems) / SCount, w in L) ;
7   PGivenNotS := mul (countMessages (w, EPoems) / ECount, w in L) ;
8   if (PGivenS + PGivenNotS <> 0) then
9     return evalf (PGivenS / (PGivenS + PGivenNotS)) ;
10  else
11    return 0.5 ;
12  end if ;
13 end proc;
```

Therefore, the probability that a sonnet is by Shakespeare given that it contains the words “from”, “fairest”, and “creatures” is:

```
> PShakespeareGivenList(["from", "fairest", "creatures"])
0.5559873068
(7.32)
```

Selecting Test Words Randomly

Finally, we can use the **randcomb** command to randomly select words from a test message, and then use those randomly selected words to compute the probability that the message was written by Shakespeare. Here is the first test message in “testMessages.txt”:

```
> testPoems[1]
“When to the sessions of sweet silent thought
I summon up remembrance of things past,
I sigh the lack of many a thing I sought,
And with old woes new wail my dear time’s waste:
Then can I drown an eye, unused to flow,
For precious friends hid in death’s dateless night,
And weep afresh love’s long since cancell’d woe,
And moan the expense of many a vanish’d sight:
Then can I grieve at grievances foregone,
And heavily from woe to woe tell o’er
The sad account of fore-bemoaned moan,
Which I new pay as if not paid before.
But if the while I think on thee, dear friend,
All losses are restor’d and sorrows end.
”
```

(7.33)

We use the **Words** command to separate the poem into individual words:

```
> exampleTestWords := StringTools[Words](testPoems[1])  
exampleTestWords := [“When”, “to”, “the”, “sessions”, “of”, “sweet”,  
“silent”, “thought”, “I”, “summon”, “up”, “remembrance”, “of”,  
“things”, “past”, “I”, “sigh”, “the”, “lack”, “of”, “many”, “a”,  
“thing”, “I”, “sought”, “And”, “with”, “old”, “woes”, “new”, “wail”,  
“my”, “dear”, “time’s”, “waste”, “Then”, “can”, “I”, “drown”,  
“an”, “eye”, “unused”, “to”, “flow”, “For”, “precious”, “friends”, “hid”,  
“in”, “death’s”, “dateless”, “night”, “And”, “weep”, “afresh”, “love’s”,  
“long”, “since”, “cancell’d”, “woe”, “And”, “moan”, “the”, “expense”,  
“of”, “many”, “a”, “vanish’d”, “sight”, “Then”, “can”, “I”, “grieve”,  
“at”, “grievances”, “foregone”, “And”, “heavily”, “from”, “woe”, “to”,  
“woe”, “tell”, “o’er”, “The”, “sad”, “account”, “of”, “fore”,  
“bemoaned”, “moan”, “Which”, “I”, “new”, “pay”, “as”, “if”, “not”,  
“paid”, “before”, “But”, “if”, “the”, “while”, “I”, “think”, “on”,  
“thee”, “dear”, “friend”, “All”, “losses”, “are”, “restor’d”, “and”,  
“sorrows”, “end”] (7.34)
```

Randomly select four of those words:

```
> exampleTestList := combinat[randcomb](exampleTestWords, 4)  
exampleTestList := [“lack”, “love’s”, “think”, “All”] (7.35)
```

Now, use our procedure to find the probability that a message with these four words was written by Shakespeare:

```
> PShakespeareGivenList(exampleTestList)  
0.7329839295 (7.36)
```

Putting this all together:

```
1 PShakespeare := proc (testMessage::string, testSize::integer)  
2   local testWordList;  
3   testWordList :=  
4     combinat[randcomb](StringTools[Words](testMessage),  
5       testSize);  
6   return PShakespeareGivenList(testWordList);  
7 end proc;
```

As an example, we run the filter on the second test message with a test size of 3.

```
> PShakespeare(testPoems[2], 3)  
0.3805889954 (7.37)
```

7.4 Expected Value and Variance

In Section 7.2 of this manual, we introduced Maple's commands for using random variables. In this section, we explore Maples's **Statistics** package more closely and use random variables to explore the concepts of expected value and variance.

As mentioned earlier, the **Statistics** package provides the distribution **Geometric**, which takes one parameter, the probability of a "success."

$$\begin{aligned} > X := \text{RandomVariable}(\text{Geometric}(1/4)) \\ X := \text{_R9} \end{aligned} \tag{7.38}$$

We use the **Probability** command to compute probabilities of events. For example, the probability $p(X = 5)$ is computed by:

$$\begin{aligned} > \text{Probability}(X = 5) \\ \frac{243}{4096} \end{aligned} \tag{7.39}$$

Note that Maple's definition of a geometric random variable differs slightly from the textbook's. The textbook defines the value of the geometric random variable, in terms of coin flips, to be the number of flips it takes to get a tails, where the parameter is the probability of tails. Maple's definition is that the value of the random variable is the number of heads that appear before tails comes up. Thus, the probability $p(X = k)$ is

$$\begin{aligned} > \text{Probability}(X = k) \\ \begin{cases} 0 & k < 0 \\ \frac{\left(\frac{3}{4}\right)^k}{4} & \text{otherwise} \end{cases} \end{aligned} \tag{7.40}$$

Contrast this with the formula given in the text.

The **Statistics** package also includes commands for computing the expected value, variance, and standard deviation of a random variable:

$$\begin{aligned} > \text{ExpectedValue}(X) \\ 3 \end{aligned} \tag{7.41}$$

$$\begin{aligned} > \text{Variance}(X) \\ 12 \end{aligned} \tag{7.42}$$

Maple can also compute these symbolically, if we use an unassigned name for the argument to **Geometric**.

$$\begin{aligned} > Y := \text{RandomVariable}(\text{Geometric}(p)) : \\ > \text{ExpectedValue}(Y) \\ \frac{1-p}{p} \end{aligned} \tag{7.43}$$

> *Variance*(Y)

$$\frac{1-p}{p^2} \quad (7.44)$$

> *StandardDeviation*(Y)

$$\frac{\sqrt{1-p}}{p} \quad (7.45)$$

Notice that the expected value differs from the expected value of a geometric distribution given in the text. This is because of the difference in definitions mentioned previously. Since the difference between Maple's definition and the textbook's definition is that Maple's geometric random variables have values one less than the textbooks, we can create a random variable Z that has the geometric distribution defined by the textbook by adding one to a random variable created by Maple:

> *Z := RandomVariable(Geometric(p)) + 1 :*

We check that Z agrees with the formula given by the text, namely $p(Z = k) = (1 - p)^{k-1}$ for $k = 1, 2, 3, \dots$

> *Probability(Z = k)*

$$\begin{cases} 0 & k < 1 \\ p(1-p)^{k-1} & \text{otherwise} \end{cases} \quad (7.46)$$

We also confirm that the expected value agrees with the text:

> *ExpectedValue(Z)*

$$-\frac{-1+p}{p} + 1 \quad (7.47)$$

Finally, you can also use these functions with combinations of random variables, as illustrated below.

> *Rvariable1 := RandomVariable(Geometric(1/4)) :*

> *Rvariable2 := RandomVariable(Binomial(20, 0.3)) :*

> *ExpectedValue(Rvariable1 + 2 · Rvariable2)*

$$15.00000000 \quad (7.48)$$

> *Variance(Rvariable1 + 2 · Rvariable2)*

$$28.80000000 \quad (7.49)$$

Solutions to Computer Projects and Computations and Explorations

Computer Projects 7

Given a positive integer m , simulate the collection of cards that come with the purchase of products to find the number of products that must be purchased to obtain a full set of m different collector cards. (See Supplementary Exercise 33.)

Solution: We will define a procedure called **CardSimulate** that will simulate the process of choosing random collectible cards until all the possible cards have been obtained. This procedure needs to do three things: (1) keep track of which cards have been obtained; (2) keep selecting random cards until the complete set is obtained; and (3) keep track of how many cards have been purchased.

Think of the cards as numbered 1 through m . To keep track of which cards have been obtained and which have not, we will use a list that we call **currCollection**, for current collection. The entries in this list will be 0s and 1s, with a 0 representing the fact that the card corresponding to that position is not owned and 1 that it is. To initialize **currCollection**, we use the **\$** operator.

```
> [0 $ 10]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0] (7.50)
```

Recall that when the dollar operator is given a positive integer as its right operand produces that number of copies of its left operand.

Second, random selection of cards can be accomplished by the **rand** command. The optional argument to **rand** specifies the range of values that it will return. Since there are m possible collectible cards, we use **rand(1..m)**. With no argument, **rand()** returns a number, but with an argument, it creates a procedure. Therefore, we assign a name to the procedure that the **rand** command creates and then use that name to make random cards. For example:

```
> exampleRand := rand(1 .. 100)
exampleRand := proc()
proc() option builtin = RandNumberInterface; end proc(6, 100, 7) + 1
end proc (7.51)
```

```
> exampleRand()
6 (7.52)
```

Our procedure will generate a random card and set the entry in **currCollection** at that card's position equal to 1. This needs to keep happening until all the cards are owned. Therefore, we need to know when all of the entries of the list are 1s. We can do this by adding up the entries in the list. Since the entries are always 0 or 1, when the list is all 1s, the sum will be equal to m and that is the only way the sum can be m . To add the entries in the list, we can use the **add** command as follows:

```
> exampleList := [1, 0, 0, 1, 1, 1]:
> add(exampleList)
4 (7.53)
```

Third, we keep track of how many cards have been purchased with a counter that we increment each time a random card is generated.

Putting all of these pieces together, here is the procedure:

1	CardSimulate := proc(m: :integer)
2	local currCollection, i, count, tempCard, cardGen;
3	currCollection := [0\$m];
4	count := 0;

```

5   cardGen := rand(1..m);
6   while add(currCollection) < m do
7       tempCard := cardGen();
8       count := count + 1;
9       currCollection[tempCard] := 1;
10  end do;
11  return count;
12 end proc;

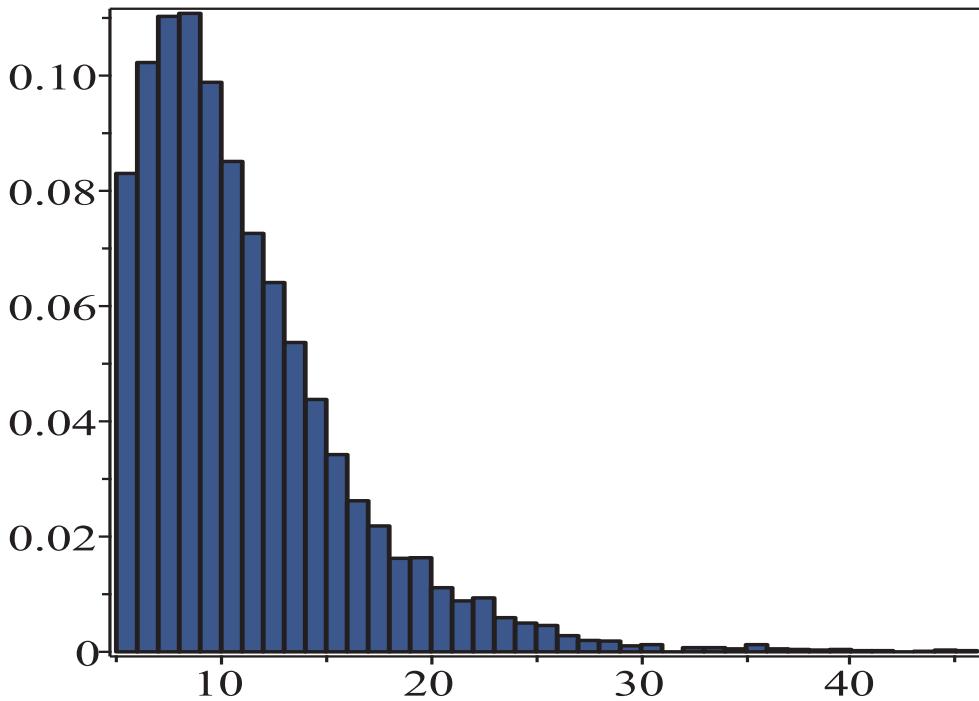
```

Run the simulation 10 000 times for $m = 5$ and draw a histogram of the resulting data:

```

> simulations := [seq(CardSimulate(5), i = 1 .. 10 000)]:
> Statistics[Histogram](simulations, binwidth = 1)

```



Computer Projects 9

Given a positive integer n , find the probability of selecting the six integers from the set $\{1, 2, n, \dots\}$ that were mechanically selected in a lottery.

Solution: We will follow Example 4 in Section 7.1 of the main text. The total number of ways of choosing 6 numbers from n numbers is $C(n, 6)$, which is found with the procedure **numbcomb** in the **combinat** package. This gives us the total number of possibilities, only one of which will win.

```

1 lottery := proc(n :: posint)
2     local total;
3     total := combinat[numbcomb](n, 6);
4     1.0 / total;
5 end proc;

```

> *lottery*(49)
 7.151123842 10⁻⁸

(7.54)

If the rules of the lottery change, so that the number of numbers chosen is something other than 6, then we must modify the procedure above. We can easily modify our program to allow us to specify how many numbers we want to choose, by adding another parameter.

```

1 lottery2 := proc(n :: posint, k :: posint)
2   local total;
3   total := combinat[numbcomb](n, k);
4   1.0 / total;
5 end proc;
```

> *lottery2*(49, 6)
 7.151123842 10⁻⁸

(7.55)

> *lottery2*(30, 3)
 0.0002463054187

(7.56)

Computations and Explorations 3

Estimate the probability that two integers selected at random are relatively prime by testing a large number of randomly selected pairs of integers. Look up the theorem that gives this probability and compare your results with the correct probability.

Solution: To solve this problem, three things must be done:

1. Devise a method for generating pairs of random integers.
2. Produce a large number of these pairs, test whether they are relatively prime, and note the probability estimate based on this sample.
3. Look up the theorem mentioned in the question.

Naturally, part 3 is left to the reader.

A simple approach is to use the Maple procedure **rand** to generate a list of random integers. Then, having generated such a list we can test whether the pairs of its members are coprime using the Maple procedure **igcd** in a second loop. We implement these two loops in a new Maple procedure called **RandPairs**:

```

1 RandPairs := proc(numberPairs :: integer)
2   local listSize, i, tmp, randnums, count;
3   listSize := 2 * numberPairs;
4   randnums := [];
5   # Generate list of random integers
6   for i from 1 to listSize do
7     tmp := rand();
8     randnums := [op(randnums), tmp];
9   end do;
10  # Count the number of pairs that are coprime
11  count := 0;
12  for i from 1 to listSize-1 by 2 do
```

```

13      if igcd(randnums[i], randnums[i+1]) = 1 then
14          count := count + 1;
15      end if;
16  end do;
17  evalf(count / numberPairs);
18 end proc:
```

We can now execute this procedure on 100 pairs of integers, as follows:

> *RandPairs*(100)
0.60000000000 (7.57)

Note that repeating the computation may very well lead to a somewhat different result since the list of integers we used was generated randomly. You should try this with a much larger sample size, say 10 000 pairs of integers.

Computations and Explorations 4

Determine the number of people needed to ensure that the probability at least two of them have the same day of the year as their birthday is at least 70%, at least 80%, at least 90%, at least 95%, at least 98%, and at least 99%.

Solution: Given that we know the formula for the probability of two people having the same birthday, we can use Maple to loop over a range of possible numbers of people until we reach a probability greater than the desired probability. Example 13 in Section 7.2 of the text shows that the probability that n people in a room have different birthdays is

$$p_n = \frac{365}{366} \frac{364}{366} \frac{363}{366} \dots \frac{367-n}{366} = \frac{P(366, n)}{366^n}.$$

Our task is to find n such that $1 - p_n$ is greater than the values specified in the problem. We can do this using the Maple procedure below.

```

1 Birthdays := proc(percentage :: float)
2   local numPeople, curProb;
3   # Initialize
4   curProb := 0;
5   numPeople := 0;
6   # loop until there are enough people
7   while curProb < percentage do
8     numPeople := numPeople + 1;
9     curProb := 1 -
10        (combinat[numbperm](366, numPeople) / 366^numPeople);
11   end do;
12   return numPeople;
end proc:
```

This procedure returns the number of people required to attain the given probability that two have the same birthday. We now execute our procedure for probabilities of 0.70 and 0.95.

> *Birthdays*(0.70)
30
(7.58)

> *Birthdays*(0.95)
47
(7.59)

Exercises

Exercise 1. Use Maple to determine the integer k such that the chances of picking six numbers correctly in a lottery from the first k positive integers is less than

- a) 1 in 100 million (10^{-8}),
- b) 1 in a billion (10^{-9}),
- c) 1 in 10 billion (10^{-10}),
- d) 1 in 100 billion (10^{-11}), and
- e) 1 in a trillion (10^{-12}).

Exercise 2. Implement a Monte Carlo algorithm that determines whether a permutation of the integers 1 through n has already been sorted or is a random permutation. (See Exercise 40 in Section 7.2 of the textbook.)

Exercise 3. Modify the implementation of the collector card simulator given in the solution to Computer Projects 7 to model the situation in which the cards do not appear with equal probabilities. For instance, there could be five possible cards all of which appear with probability $2/9$ except for card number 5 which appears with probability $1/9$.

Exercise 4. Modify the implementation of the collector card simulator given in the solution to Computer Projects 7 to model the situation in which cards are purchased in packs. For example, there could be 10 possible cards and they are purchased three to a pack. Assume the cards in a pack are always different from each other. The procedure should return the number of packs necessary to collect all of the cards.

Exercise 5. Compute the average of the probabilities returned by running the Bayesian filter **PShakespeare** 100 times with the **testMessage** argument equal to **testPoems[10]** and a **testSize** of 1, that is, on the tenth of the test poems and using one word. Repeat this with a **testSize** of 2, 3, ..., 10. Graph the average probabilities for the different numbers of test words. Is there a trend in the average probabilities as the number of words increases? Explain why.

Exercise 6. The textbook describes how a Bayesian filter can be improved by considering pairs of words. Implement a Bayesian spam filter that uses this idea. Using the Shakespearean and Elizabethan sonnets as messages, compare the performance of your filter with **PShakespeare**.

Exercise 7. As described in the textbook, spam filters are most effective when the words being used as the basis of comparison are not chosen randomly, as they are in the implementation of **PShakespeare** above, but instead are chosen more carefully. Specifically, choosing words which have very high or very low probability of appearing in spam messages can improve the performance of the filter. Implement a Bayesian spam filter that uses this idea. Using the Shakespearean and Elizabethan sonnets as messages, compare the performance of your filter with **PShakespeare**.

8 Advanced Counting Techniques

Introduction

In this chapter, we will describe how to apply Maple to three important topics in counting: recurrence relations, generating functions, and inclusion–exclusion. We begin by describing how Maple can be used to solve recurrence relations, including the recurrence relations that describe the complexity of divide-and-conquer algorithms. After studying recurrence relations, we show how to use Maple to manipulate generating functions using the package **powseries** and how these capabilities can help solve counting problems. We conclude the chapter with a discussion of the principle of inclusion and exclusion.

8.1 Recurrence Relations

A recurrence relation describes a relationship between the members of a sequence and their predecessors. For example, the famous Fibonacci sequence $\{f_n\}$ satisfies the recurrence relation

$$f_n = f_{n-1} + f_{n-2}.$$

Together with the initial conditions $f_1 = 1$ and $f_2 = 1$, this relation is sufficient to define the entire sequence $\{f_n\}$.

To understand how we can work with recurrence relations in Maple, we have to remember that a sequence $\{a_n\}$ is a function whose domain is a subset of the integers (usually the positive integers or nonnegative integers, depending on the context) and whose codomain contains the terms of the sequence (which can be numbers, matrices, circles, functions, etc.). (See the definition of sequence given in Section 2.4 of the textbook.)

With this point of view, the sequence $\{a_n\}$ is a function a and the n th term of the sequence is the value of the function evaluated at the integer n , that is, $a_n = a(n)$. This is only a change in notation, but it makes it easier to see that a recurrence relation can be represented in Maple as a procedure taking integer arguments.

We can represent the Fibonacci sequence by the procedure below, which we use to compute the first 20 terms of the Fibonacci sequence. This procedure takes one argument, a positive integer, and returns the appropriate term in the Fibonacci sequence.

```
1 Fibonacci := proc(n::posint)
2   option remember;
3   if n = 1 or n = 2 then
4     return 1;
5   end if;
6   Fibonacci(n-1) + Fibonacci(n-2);
7 end proc;
```

> *seq(Fibonacci(n), n = 1..20)*

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
2584, 4181, 6765

(8.1)

Note that the **remember** option given in the first line instructs Maple to “remember” the values of the procedure that have already been calculated. (To be precise, the **remember** option causes Maple to store the output of the procedure in a “remember table.” If it is called with that same input again, Maple simply looks up the result from the table rather than recomputing. You can force Maple to delete the remember table for a procedure with the **forget** command.)

Sometimes, a recursive implementation of an algorithm may be too costly (specifically, in time and memory) due to the nature of the algorithm, regardless of steps you may take to improve efficiency. A recursive implementation of a recurrence relation can be avoided if we can find an explicit formula for the general term of the sequence. The process of finding such a formula is referred to as “solving” the recurrence relation. In the next section, we will see how to use Maple to solve certain kinds of recurrence relations.

Tower of Hanoi Problem

In Example 2 of Section 8.1 of the textbook, the author describes the famous “Tower of Hanoi” puzzle and derives the recurrence relation

$$H_n = 2H_{n-1} + 1, \quad H_1 = 1,$$

where H_n represents the number of moves required to solve the puzzle for n disks. As discussed in the text, this has the solution

$$H_n = 2^n - 1.$$

Later, we will see how to use Maple to derive this result.

Rather than just computing the values, we can illustrate the solution to the Tower of Hanoi puzzle by writing a Maple program to compute the moves needed and to describe them to us. We will write a small program consisting of three Maple procedures: the main procedure **Hanoi**, a utility routine **PrintMove**, and **TransferDisk**, which does most of the work.

The easiest part to write is the function **PrintMove**, which merely displays the move to make at a given step.

```

1 | PrintMove := proc(src::string, dest::string)
2 |   printf("Move disk from peg %s to peg %s\n", src, dest);
3 | end proc;
```

In the above, we call the Maple command **printf**, which is used for formatted output. The first argument to **printf** is the “format string,” that is, a string that contains both regular characters, format specifications (in this case “%s”), and escape characters (such as “\n”). The format specification “%s” tells the **printf** command to put the next argument in that location in the output and format it as a string. (The first %s refers to the second argument, **src** and the second %s refers to the third argument, **dest**.) The escape character “\n” tells **printf** to insert a new line. The **printf** command is very flexible with many available options. You should refer to Maple’s help pages for more detailed information. (Note: Maple’s **printf** command is very similar to the command of the same name in C and other programming languages.)

Next, we write the recursive procedure **TransferDisk**, which does most of the work. This function models the idea of transferring a stack of **ndisks** disks from the source peg, which is given as the argument **src**, to the destination peg, **dest**, via the intermediate peg, **via**. As described in the text, in order to move a stack of n disks, you first move the top $n - 1$ pegs to the intermediate peg (using the destination as the intermediary), then move the bottom disk to the destination, and then move the smaller stack from the intermediate peg to the destination. Unless, of course, there is only 1 disk, in which you just move that disk to the destination. This is coded as follows.

```

1 TransferDisk := proc (src::string, via:: string, dest::string,
2   ndisks::posint)
3   if ndisks = 1 then
4     PrintMove (src, dest);
5   else
6     TransferDisk (src, dest, via, ndisks - 1);
7     PrintMove (src, dest);
8     TransferDisk (via, src, dest, ndisks - 1);
9   end if;
end proc:
```

Finally, we package the recursive procedure in a top-level procedure, **Hanoi**, providing an interface to the recursive engine.

```

1 Hanoi := proc (ndisks::posint)
2   TransferDisk ("A", "B", "C", ndisks);
3 end proc:
```

Our **Hanoi** program can exhibit a specific solution to the Tower of Hanoi puzzle for any number of disks:

> *Hanoi(2)*
 Move disk from peg A to peg B
 Move disk from peg A to peg C
 Move disk from peg B to peg C

> *Hanoi(3)*
 Move disk from peg A to peg C
 Move disk from peg A to peg B
 Move disk from peg C to peg B
 Move disk from peg A to peg C
 Move disk from peg B to peg A
 Move disk from peg B to peg C
 Move disk from peg A to peg C

Try experimenting with different values of **ndisk** to get a feel for how large the problem becomes for even moderately large numbers of disks.

Dynamic Programming

We conclude this section with an implementation of Algorithm 1 from Section 8.1 of the text. Recall that the goal of this algorithm is to find the maximum number of attendees that can be achieved by a schedule of talks.

We will represent each talk as a list of three elements, with the start time being the first element, the end time in the second position, and the weight, or attendance, will be last. Time of day will be represented by a single number with whole part equal to the hour in the 24-hour system and with fractional part equal to the part of an hour that corresponds to the number of minutes. For instance, 2:30 P.M. would be represented as 14.5.

As an example, consider the following eight talks.

Start Time	End Time	Attendance
9:00 AM	11:00 AM	17
9:00 AM	10:30 AM	15
10:00 AM	11:30 AM	22
10:30 AM	12:00 PM	11
11:30 AM	1:30 PM	18
12:00 PM	1:00 PM	12
1:30 PM	3:00 PM	21
2:00 PM	4:00 PM	17

We create the following list of lists to represent the talks.

```
> talks := [[9, 11, 17], [9, 10.5, 15], [10, 11.5, 22], [10.5, 12, 11], [11.5, 13.5, 18],  
[12, 13, 12], [13.5, 15, 21], [14, 16, 17]]  
talks := [[9, 11, 17], [9, 10.5, 15], [10, 11.5, 22], [10.5, 12, 11],  
[11.5, 13.5, 18], [12, 13, 12], [13.5, 15, 21], [14, 16, 17]] (8.2)
```

Recall the description of Algorithm 1 from the text. We summarize the general outline of the algorithm below.

1. Sort the talks in order of increasing end time.
2. For each index j , compute $p(j)$ —the maximum index i less than j such that talk i is compatible with talk j .
3. For each index j , compute $T(j)$, which is computed by the recurrence relation
$$T(j) = \max(w_j + T(p(j)), T(j - 1))$$
 and with initial condition $T(0) = 0$.
4. The maximum total number of attendees is $T(n)$, where n is the number of talks.

For step 1, we will make use of the **sort** command with a custom ordering procedure. Recall that **sort** can accept an optional argument in the form of a Boolean procedure of two arguments. This procedure should return true if the first argument precedes the second and false otherwise. Since we must sort the talks in increasing order of end time (which is stored in position 2 in the lists representing the talks), we use the following procedure.

```
1  sortEnd := proc (a, b)  
2      return a[2] < b[2];  
3  end proc;
```

For step 2, we must compute $p(j)$. For this, we create a procedure that accepts the sorted list of talks and returns a table that represents the function p . Recall that the value of $p(j)$ is the largest index among talks compatible with the talk with index j , so we call this procedure **compatible**.

After declaring local variables and initializing **p** to the empty table, we will loop through all the indices, **j**, from 1 to the number of talks in the list. We use a local variable, **jstart**, to store the start time of the current talk being analyzed and we set the value of **p** for **j** to 0. We then consider all the talks earlier in the list beginning with the talk with index **j-1** and working backward to talk **1**. For each talk, we check to see if it ends before talk **j** starts. When we find such a talk, we set its index to the value of **p[j]** (since we are working backward, the first one found is the talk with the largest index) and terminate the loop. If no compatible talk is found, then **p[j]** was already set to **0**. Here is the procedure.

```

1 compatible := proc(talkList)
2   local p, j, jstart, i;
3   p := table();
4   for j from 1 to nops(talkList) do
5     jstart := talkList[j][1];
6     p[j] := 0;
7     for i from j-1 to 1 by -1 do
8       if talkList[i][2] <= jstart then
9         p[j] := i;
10        break;
11      end if;
12    end do;
13  end do;
14  return p;
15 end proc;
```

For step 3, we must compute $T(j)$. To do this, we create a procedure that accepts as input the sorted list of talks and the table representing the function p . Initialize **T** to the empty table and set its value at 0 to 0. Then, consider each integer j from 1 to the number of talks and apply the formula: $T(j) = \max(w_j + T(p(j)), T(j - 1))$.

```

1 totalAttendance := proc(talkList, p)
2   local j, T;
3   T := table();
4   T[0] := 0;
5   for j from 1 to nops(talkList) do
6     T[j] := max(talkList[j][3] + T[p[j]], T[j-1]);
7   end do;
8   return T;
9 end proc;
```

We can now put the pieces together as outlined at the start of this subsection.

```

1 maximumAttendees := proc(talkList)
2   local L, p, T;
3   L := sort(talkList, sortEnd);
```

```

4   p := compatible(L);
5   T := totalAttendance(L, p);
6   return T[nops(L)];
7 end proc;

```

And thus, the maximum attendance for the talks described above is:

> *maximumAttendees(talks)*
 61 (8.3)

8.2 Solving Linear Recurrence Relations

Maple has a very powerful recurrence solver, **rsolve**. Its use, however, can obscure some of the important ideas that are involved. Therefore, we will first use some of Maple's more fundamental facilities to solve certain kinds of recurrence relations one step at a time.

Given a recursively defined sequence $\{a_n\}$, we would like to find a formula, involving only the index n (and, perhaps, other fixed constants and known functions) which does *not* depend on knowing the value of any prior elements of the sequence.

Linear Homogeneous Recurrence Relations with Constant Coefficients

We will begin by considering recurrence relations that are *linear, homogeneous*, and which have *constant coefficients*; that is, they have the form

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$$

where c_1, c_2, \dots, c_k are real constants and c_k is nonzero. Recall that the integer k is called the *degree* of this recurrence relation. To have a unique solution, at least k initial conditions must be specified.

The general method for solving such a recurrence relation involves finding the roots of its characteristic polynomial

$$r^k - c_1 r^{k-1} - c_2 r^{k-2} - \cdots - c_{k-1} r - c_k.$$

When this polynomial has distinct roots, all solutions are linear combinations of the n th powers of these roots. When there are repeated roots, the situation is a little more complicated, as we will see.

A First Example

Consider the linear homogeneous recurrence relation with constant coefficients of degree two

$$a_n = 2 a_{n-1} + 3 a_{n-2},$$

subject to the initial conditions

$$a_1 = 4 \text{ and } a_2 = 2.$$

Its characteristic equation is

$$r^2 - 2r - 3 = 0.$$

To solve the recurrence relation, we must solve for the roots of this equation. Using Maple makes this very easy; we use the **solve** function.

```
> solve (r^2 - 2r - 3 = 0, r)
3, -1
```

(8.4)

The **solve** command computes the values of the variable r , given as the second argument, that satisfy the equation in the first argument. Note that you may omit **=0** from the command above and Maple will interpret it the same.

```
> solve (r^2 - 2r - 3, r)
3, -1
```

(8.5)

Now that Maple has determined that the solutions are $r = 3$ and $r = -1$, we can write down the form of the solution to the recurrence as

$$a_n = \alpha 3^n + \beta (-1)^n,$$

where α and β are constants that we have yet to determine.

Since the initial conditions are $a_1 = 4$ and $a_2 = 2$, we know that our recurrence relation must satisfy the following pair of equations.

$$\begin{cases} 3\alpha - \beta = 4 \\ 3^2\alpha + \beta = 2 \end{cases}$$

To find the solution to this system of linear equations, we again use Maple's **solve** command:

```
> solve ({3α - β = 4, 9α + β = 2}, {α, β})
{α = 1/2, β = -5/2}
```

(8.6)

This time, we are telling Maple to solve the set of equations. Likewise, the variables to be solved for form a set.

Now that we have the values for α and β , we see that the complete solution to the recurrence relation is

$$a_n = \frac{1}{2} \cdot 3^n - \frac{5}{2} \cdot (-1)^n.$$

This formula allows us to write a Maple function for finding the terms of the sequence $\{a_n\}$, which can be more efficient than a recursive procedure.

```
> a := n :: posint →  $\frac{3^n}{2} - \frac{5(-1)^n}{2}$  :  
> seq(a(n), n = 1..10)  
4, 2, 16, 38, 124, 362, 1096, 3278, 9844, 29522
```

(8.7)

A Second Example

Let us try another example. We will solve the recurrence relation

$$a_n = -\frac{5}{3}a_{n-1} + \frac{2}{3}a_{n-2}$$

with initial conditions

$$a_1 = \frac{1}{2} \text{ and } a_2 = 4.$$

To do this, we ask Maple to solve the characteristic equation of the recurrence relation, and then solve the system of linear equations obtained from the roots of the characteristic equation and the initial conditions. Note that this method works because this recurrence relation is linear, homogeneous, and has constant coefficients.

```
> CharEqnRoots := solve  $\left(r^2 + \frac{5}{3}r - \frac{2}{3}, r\right)$   
CharEqnRoots :=  $\frac{1}{3}, -2$ 
```

(8.8)

```
> solve  $\left\{ \begin{array}{l} \alpha \cdot CharEqnRoots[1] + \beta \cdot CharEqnRoots[2] = \frac{1}{2}, \\ \alpha \cdot CharEqnRoots[1]^2 + \beta \cdot CharEqnRoots[2]^2 = 4 \end{array} \right\},$   
 $\{\alpha, \beta\}$   
 $\left\{ \alpha = \frac{45}{7}, \beta = \frac{23}{28} \right\}$ 
```

(8.9)

Thus, we see that the solution to the recurrence relation is

$$a_n = \frac{45}{7} \left(\frac{1}{3}\right)^n + \frac{23}{28} (-2)^n.$$

The Fibonacci Sequence

We can derive an explicit formula for the Fibonacci sequence this way as well. The characteristic polynomial for the Fibonacci sequence is

$$r^2 - r - 1.$$

We find the roots of the characteristic equation.

$$> \text{CEqnRoots} := \text{solve}(r^2 - r - 1, r)$$

$$\text{CEqnRoots} := \frac{\sqrt{5}}{2} + \frac{1}{2}, -\frac{\sqrt{5}}{2} + \frac{1}{2} \quad (8.10)$$

Therefore, the formula for the n th Fibonacci number is of the form

$$> \text{Fn} := \text{alpha} \cdot \text{CEqnRoots}[1]^n + \text{beta} \cdot \text{CEqnRoots}[2]^n$$

$$\text{Fn} := \alpha \left(\frac{\sqrt{5}}{2} + \frac{1}{2} \right)^n + \beta \left(-\frac{\sqrt{5}}{2} + \frac{1}{2} \right)^n \quad (8.11)$$

We find the coefficients α and β in the formula by using the initial conditions.

$$> \text{alphas} := \text{solve}(\left\{ \text{alpha} \cdot \text{CEqnRoots}[1] + \text{beta} \cdot \text{CEqnRoots}[2] = 1, \text{alpha} \cdot \text{CEqnRoots}[1]^2 + \text{beta} \cdot \text{CEqnRoots}[2]^2 = 1 \right\}, \{\text{alpha}, \text{beta}\})$$

$$\text{alphas} := \left\{ \alpha = \frac{\sqrt{5}}{5}, \beta = -\frac{\sqrt{5}}{5} \right\} \quad (8.12)$$

We use the **subs** command to substitute the values for α and β into the formula **Fn**.

$$> \text{Fn} := \text{subs}(\text{alphas}, \text{Fn})$$

$$\text{Fn} := \frac{\sqrt{5} \left(\frac{\sqrt{5}}{2} + \frac{1}{2} \right)^n}{5} - \frac{\sqrt{5} \left(-\frac{\sqrt{5}}{2} + \frac{1}{2} \right)^n}{5} \quad (8.13)$$

If we are to use such a formula to repeatedly compute values, then we should use it to define a function. You can enter a new function definition manually, but a more convenient way is to use the **unapply** command. It takes two arguments: an expression and the variable that is to be the argument of the function. It produces a functional operator.

$$> \text{Fibonacci2} := \text{unapply}(\text{Fn}, n)$$

$$\text{Fibonacci2} := n \mapsto \frac{\sqrt{5} \left(\frac{\sqrt{5}}{2} + \frac{1}{2} \right)^n}{5} - \frac{\sqrt{5} \left(-\frac{\sqrt{5}}{2} + \frac{1}{2} \right)^n}{5} \quad (8.14)$$

The procedure **Fibonacci2** is much more efficient than even the optimized recursive procedure **Fibonacci**. To see this, we record the accumulated time for computing the first 100 000 Fibonacci numbers (after, of course, clearing the remember table for the **Fibonacci** procedure with the **forget** command).

$$> \text{forget}(\text{Fibonacci})$$

```

> st := time():
for i to 100 000 do
    Fibonacci(i)
end do :
time( ) - st
1.452

```

(8.15)

```

> st := time():
for i to 100 000 do
    Fibonacci2(i)
end do :
time( ) - st
0.417

```

(8.16)

A Solver

Now, we will generalize what we have been doing and write a Maple procedure to solve a degree two linear, homogeneous recurrence relation with constant coefficients, provided that the roots of the characteristic polynomial are distinct. We will write a procedure **RecSol2Distinct** which solves the recurrence

$$a_n = ca_{n-1} + da_{n-2}$$

subject to the initial conditions

$$a_1 = u \text{ and } a_2 = v$$

and then returns a procedure that can be used to compute terms of the sequence.

For the moment, we assume that the characteristic polynomial $r^2 - cr - d$ has two distinct roots. Later, we will modify the procedure to relax that restriction. With the assumption that the roots of the characteristic polynomial are distinct, all our procedure needs to do is to repeat the steps we did manually in the examples above.

```

1 RecSol2Distinct := proc(c, d, u, v)
2   local CERoots, alphas, alpha, beta, f, n, r;
3   # First solve the characteristic equation
4   CERoots := solve(r^2 - c*r - d, r);
5   # Next solve the equations derived from initial conditions
6   alphas := solve({
7     alpha * CERoots[1] + beta * CERoots[2] = u,
8     alpha * CERoots[1]^2 + beta * CERoots[2]^2 = v
9   }, {alpha, beta});
10  # Finally substitute the answers into the general form
11  f := subs(alphas, alpha * CERoots[1]^n + beta * CERoots[2]^n);
12  return unapply(f, n);
13 end proc:

```

To see how it works, we check that it gives the same result for the Fibonacci sequence that we obtained by hand. To construct a function for computing the Fibonacci sequence, invoke the new procedure as:

$$> f := \text{RecSol2Distinct}(1, 1, 1, 1)$$

$$f := n \mapsto \frac{\sqrt{5} \left(\frac{\sqrt{5}}{2} + \frac{1}{2} \right)^n}{5} - \frac{\sqrt{5} \left(-\frac{\sqrt{5}}{2} + \frac{1}{2} \right)^n}{5} \quad (8.17)$$

The procedure **f** can be used to compute the general term of the Fibonacci sequence.

$$> f(n)$$

$$\frac{\sqrt{5} \left(\frac{\sqrt{5}}{2} + \frac{1}{2} \right)^n}{5} - \frac{\sqrt{5} \left(-\frac{\sqrt{5}}{2} + \frac{1}{2} \right)^n}{5} \quad (8.18)$$

You can see that is the same formula that we derived above and correctly produces the first 10 Fibonacci numbers.

$$> \text{seq}(\text{simplify}(f(n)), n = 1 .. 10)$$

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55 \quad (8.19)$$

A Recurrence with Repeated Roots

We will next create a procedure that can handle the case of repeated roots. First, let us look at an example of a recurrence relation whose characteristic polynomial has a double root. The recurrence relation

$$a_n = 4a_{n-1} - 4r_{n-2}$$

has the characteristic equation

$$> \text{charEqn} := r^2 - 4r + 4 = 0$$

$$\text{charEqn} := r^2 - 4r + 4 = 0 \quad (8.20)$$

which has roots

$$> \text{CERoots} := \text{solve}(\text{charEqn}, r)$$

$$\text{CERoots} := 2, 2 \quad (8.21)$$

We can clearly see that in this case the root is repeated, but for Maple to recognize it, we need to use the following test.

$$> \text{evalb}(\text{CERoots}[1] = \text{CERoots}[2])$$

$$\text{true} \quad (8.22)$$

Note that the **evalb** command forces evaluation as a Boolean value. This is not necessary in conditional statements (like **if** statements) because the context of those statements causes Maple to evaluate the expression as a Boolean. If we call the double root (2 in this case) r_0 , then the recurrence relation has the explicit solution

$$a_n = \alpha r_0^n + n\beta r_0^n,$$

for all positive integers n , and for some constants α and β . The initial conditions of $a_1 = 1$ and $a_2 = 4$ produce the system of equations

$$\begin{cases} \alpha \cdot 2^1 + 1 \cdot \beta \cdot 2^1 = 1 \\ \alpha \cdot 2^2 + 2 \cdot \beta \cdot 2^2 = 4 \end{cases}$$

As before, we solve this system for α and β .

$$\begin{aligned} > \text{Alphas} := \text{solve} \left(\left\{ \text{alpha} \cdot \text{CERoots}[1] + \text{beta} \cdot \text{CERoots}[1] = 1, \right. \right. \\ & \quad \left. \left. \text{alpha} \cdot \text{CERoots}[1]^2 + 2 \text{beta} \cdot \text{CERoots}[1]^2 = 4 \right\}, \{\text{alpha}, \text{beta}\} \right) \\ & \text{Alphas} := \{\alpha = 0, \beta = 1/2\} \end{aligned} \tag{8.23}$$

And finally, substitute these values into the general form $a_n = \alpha r_0^n + n\beta r_0^n$,

$$\begin{aligned} > \text{subs}(\text{Alphas}, \text{alpha} \cdot \text{CERoots}[1]^n + n \cdot \text{beta} \cdot \text{CERoots}[1]^n) \\ & \frac{n 2^n}{2} \end{aligned} \tag{8.24}$$

A More General Recurrence Solver

The steps carried out above are quite general and we can write a procedure, **RecSolver2**, which solves a recurrence relation (degree two, linear, homogeneous, with constant coefficients) regardless of whether the characteristic polynomial has distinct roots or not. The following procedure solves the recurrence

$$a_n = ca_{n-1} + da_{n-2},$$

with initial conditions

$$a_1 = u \text{ and } a_2 = v.$$

```

1 RecSolver2 := proc(c, d, u, v)
2   local CERoots, alphas, alpha, beta, f, n, r;
3   # First solve the characteristic equation
4   CERoots := solve(r^2 - c*r - d, r);
5   # Then test if the roots are the same
6   if (CERoots[1] = CERoots[2]) then
7     # the roots are the same so follow the last example
8     alphas := solve({
9       alpha * CERoots[1] + beta * CERoots[1] = u,
10      alpha * CERoots[1]^2 + 2 * beta * CERoots[1]^2 = v
11    }, {alpha, beta});
12    f := subs(alphas, alpha * CERoots[1]^n + n * beta *
13      CERoots[1]^n);
14  else

```

```

14      # otherwise, use the RecSol2 method
15      alphas := solve ({
16          alpha * CERoots [1] + beta * CERoots [2] = u,
17          alpha * CERoots [1]^2 + beta * CERoots [2]^2 = v
18      }, {alpha, beta});
19      f := subs (alphas, alpha * CERoots [1]^n + beta * CERoots [2]^n);
20  end if;
21  # Finally, use unapply
22  return unapply (f, n);
23 end proc:

```

RecSolver2 first tests for a repeated root and then does the appropriate computation. We test this procedure on the examples we did by hand, such as the Fibonacci sequence:

$$> \text{RecSolver2}(1, 1, 1, 1)$$

$$n \mapsto \frac{\sqrt{5} \left(\frac{\sqrt{5}}{2} + \frac{1}{2}\right)^n - \sqrt{5} \left(-\frac{\sqrt{5}}{2} + \frac{1}{2}\right)^n}{5} \quad (8.25)$$

For the example with a double root:

$$> \text{RecSolver2}(4, -4, 1, 4)$$

$$n \mapsto \frac{n 2^n}{2} \quad (8.26)$$

In both of those examples, the result is consistent with what we had obtained before. We will now use the solver to find the first 10 terms of the sequence defined by the following recurrence relation and initial conditions.

$$a_n = 4a_{n-1} - 3a_{n-2}$$

$$a_1 = 1 \text{ and } a_2 = 2.$$

$$> g := \text{RecSolver2}(4, -3, 1, 2)$$

$$g := n \mapsto \frac{3^n}{6} + \frac{1}{2} \quad (8.27)$$

$$> \text{seq}(\text{simplify}(g(n)), n = 1 .. 10)$$

$$1, 2, 5, 14, 41, 122, 365, 1094, 3281, 9842 \quad (8.28)$$

As another example, consider the following recurrence relation

$$a_n = -a_{n-1} - a_{n-2},$$

with initial conditions

$$a_1 = 1 \text{ and } a_2 = 2.$$

$$\begin{aligned}
> h := & \text{RecSolver2}(-1, -1, 1, 2) \\
h := & n \mapsto \frac{\left(\sqrt{3} - 5I\right)\sqrt{3}\left(-\frac{1}{2} + \frac{I\sqrt{3}}{2}\right)^n}{3(-1 + I\sqrt{3})} \\
& + \frac{I\sqrt{3}\left(-2 + 6I\sqrt{3}\right)\left(-\frac{1}{2} - \frac{I\sqrt{3}}{2}\right)^n}{12}
\end{aligned} \tag{8.29}$$

Notice that the solution to this recurrence is very complicated and requires the use of complex numbers. (Maple uses the name **I** to represent the imaginary unit $\sqrt{-1}$.) However, if we compute the first 10 terms, we notice a very simple pattern emerges.

$$\begin{aligned}
> & \text{seq}(\text{simplify}(h(n)), n = 1 .. 10) \\
& 1, 2, -3, 1, 2, -3, 1, 2, -3, 1
\end{aligned} \tag{8.30}$$

Maple can make this pattern explicit if we replace the numerical initial conditions with symbolic constants.

$$\begin{aligned}
> k := & \text{RecSolver2}(-1, -1, \lambda, \mu) \\
k := & n \mapsto \frac{\left(\sqrt{3}\lambda - I\lambda - 2I\mu\right)\sqrt{3}\left(-\frac{1}{2} + \frac{I\sqrt{3}}{2}\right)^n}{3(-1 + I\sqrt{3})} \\
& + \frac{I\sqrt{3}\left(2\lambda + 2I\sqrt{3}\lambda + 2I\sqrt{3}\mu - 2\mu\right)\left(-\frac{1}{2} - \frac{I\sqrt{3}}{2}\right)^n}{12}
\end{aligned} \tag{8.31}$$

$$\begin{aligned}
> & \text{seq}(\text{simplify}(k(n)), n = 1 .. 10) \\
& \lambda, \mu, -\lambda - \mu, \lambda, \mu, -\lambda - \mu, \lambda, \mu, -\lambda - \mu, \lambda
\end{aligned} \tag{8.32}$$

Nonhomogeneous Recurrence Relations

So far, we have been restricted to homogeneous linear recurrence relations with constant coefficients. However, the techniques used in solving them may be extended to provide solutions to *nonhomogeneous* linear recurrence relations with constant coefficients. That is, recurrence relations of the form

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} + F(n)$$

with c_1, c_2, \dots, c_k real numbers and $F(n)$ a function depending only on n . To solve this more general form of a recurrence relation, we do two things: (1) find the solutions of the associated homogeneous recurrence relation (the relation obtained by removing $F(n)$); (2) find a particular solution for the nonhomogeneous equation.

Consider the following example:

$$a_n = 6a_{n-1} - 9a_{n-2} + n3^n,$$

from Example 12 of Section 8.2 in the text.

The first step is to find the solutions to the associated homogeneous recurrence relation

$$a_n = 6a_{n-1} - 9a_{n-2}.$$

To do this, we can use our **RecSolver2**. We will use α and β for the initial conditions so that we get all the solutions.

> *HSolution* := *RecSolver2*(6, -9, alpha, beta)

$$\text{HSolution} := n \mapsto \left(-\frac{\beta}{9} + \frac{2\alpha}{3} \right) 3^n + n \left(\frac{\beta}{9} - \frac{\alpha}{3} \right) 3^n \quad (8.33)$$

The second step is to find a particular solution. Theorem 6 in Section 8.2 of the text tells us how to find the form of the particular solution. Note that $F(n) = n3^n$ and 3 is a root of the characteristic polynomial with multiplicity 2 (you can verify this by solving the characteristic equation of the associated homogeneous relation; it is also made apparent by the form of **HSolution**). Thus, the theorem tells us that there is a particular solution of the form

$$n^2(pn + q)3^n.$$

We will define a functional operator for the form of the particular solution.

> *PForm* := $n \rightarrow n^2(p \cdot n + q)3^n$

$$\text{PForm} := n \mapsto n^2(pn + q)3^n \quad (8.34)$$

To find a particular solution, we need to find the values of p and q . To find these values, we substitute the terms of **PForm** into the recurrence relation. This gives us an equation in terms of p and q (and n).

> *Peqn* := *PForm*(n) = $6 \cdot \text{PForm}(n-1) - 9 \cdot \text{PForm}(n-2) + n \cdot 3^n$

$$\begin{aligned} \text{Peqn} := n^2(pn + q)3^n &= 6(n-1)^2(p(n-1) + q)3^{n-1} \\ &\quad - 9(n-2)^2(p(n-2) + q)3^{n-2} + n3^n \end{aligned} \quad (8.35)$$

> *Peqn* := *simplify*(*Peqn*)

$$\text{Peqn} := n^2(pn + q)3^n = 3^n(n^3p + n^2q - 6pn + n + 6p - 2q) \quad (8.36)$$

The second line in the pair of commands above replaces the equation **Peqn** with its simplified form. This explicit use of the **simplify** command is necessary for Maple to be able to solve the equation.

Next, we have Maple solve that equation for p and q . In order to indicate that we want Maple to find the values of p and q that satisfy the equation for all values of n , we will make use of the **identity** command within the **solve** command as follows.

The **identity** command can only be used within the first argument of **solve**. It requires two arguments. The first is an equation, such as **Peqn**, or an expression which is assumed to be equated to 0. The second argument is the name of the variable that the identity is in terms of.

$$> \text{Pvals} := \text{solve}(\text{identity}(\text{Peqn}, n), \{p, q\})$$

$$\text{Pvals} := \left\{ p = \frac{1}{6}, q = \frac{1}{2} \right\} \quad (8.37)$$

Thus, the particular solution is

$$> \text{subs}(\text{Pvals}, \text{PForm}(n))$$

$$n^2 \left(\frac{n}{6} + \frac{1}{2} \right) 3^n \quad (8.38)$$

Putting it all together, we see that all solutions to the recurrence relation $a_n = 6a_{n-1} - 9a_{n-2} + n3^n$ are of the form

$$> \text{HSolution}(n) + \text{subs}(\text{Pvals}, \text{PForm}(n))$$

$$\left(-\frac{\beta}{9} + \frac{2\alpha}{3} \right) 3^n + n \left(\frac{\beta}{9} - \frac{\alpha}{3} \right) 3^n + n^2 \left(\frac{n}{6} + \frac{1}{2} \right) 3^n \quad (8.39)$$

Maple's Recurrence Solver

Now that we have seen how to use Maple to implement an algorithm to solve simple recurrence relations, it is time to introduce Maple's command for solving recurrence relations.

We have already seen the Maple command **solve** for working with polynomial equations and systems of equations. Similarly, there is a Maple command **rsolve**, which is specially engineered for dealing with recurrence relations. It is a much more sophisticated version of our **RecSolver2** procedure and can deal with recurrence relations of arbitrary degree, repeated roots, and nonlinear recurrence relations. To use **rsolve**, you need to tell it what the recurrence relation is and some initial conditions. You must also specify the name of the recursive function to solve for.

For example, to solve the Fibonacci recurrence, you enter the following statement.

$$> \text{unassign}(F):$$

$$\text{rsolve}(\{F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)\}, F(n))$$

$$\frac{\sqrt{5} \left(\frac{\sqrt{5}}{2} + \frac{1}{2} \right)^n}{5} - \frac{\sqrt{5} \left(-\frac{\sqrt{5}}{2} + \frac{1}{2} \right)^n}{5} \quad (8.40)$$

(We use the **unassign** command to clear the name **F** of anything that may have been stored in it before. Using **unassign**, or a command like **F := 'F'**, is a good idea before using **rsolve**, since if the function name is already storing a value, **rsolve** may return an error.)

The **rsolve** command will let us solve nonhomogeneous recurrence relations like the Tower of Hanoi problem very easily. Recall that the Tower of Hanoi problem has the recurrence relation

$$H_n = 2H_{n-1} + 1,$$

with initial condition $H_1 = 1$.

$$> \text{unassign}(H):$$

$$\text{rsolve}(\{H(1) = 1, H(n) = 2H(n-1) + 1\}, H(n))$$

$$2^n - 1 \quad (8.41)$$

It is not actually necessary to specify the initial conditions for a recurrence relation. If they are not present, Maple will still solve the equation, inserting symbolic constants (e.g., **G(0)** and **G(1)**) in place of numeric values, as the following example illustrates.

$$\begin{aligned}
 > \text{unassign}(G) : \\
 & \text{rsolve}\left(G(n) = 2G(n-1) - 6G(n-2), G(n)\right) \\
 & \quad - \frac{I \left(-G(1)\sqrt{5} + G(0)\sqrt{5} + 5IG(0)\right) \left(-I\sqrt{5} + 1\right)^n}{10} \\
 & \quad - \frac{I \left(G(1)\sqrt{5} - G(0)\sqrt{5} + 5IG(0)\right) \left(I\sqrt{5} + 1\right)^n}{10}
 \end{aligned} \tag{8.42}$$

The function **rsolve** can handle different kinds of recurrence relations, including:

- linear recurrence relations with constant coefficients,
- systems of linear recurrence relations with constant coefficients,
- divide-and-conquer recurrence relations with constant coefficients,
- many first order linear recurrence relations, and
- some nonlinear first order recurrence relations.

The capabilities of **rsolve**, like other Maple functions, are constantly being enhanced and extended. However, **rsolve** is not a panacea—you can easily find recurrence relations that it is incapable of solving. When **rsolve** is unable to solve a recurrence relation, it simply returns unevaluated, as below.

$$\begin{aligned}
 > \text{unassign}(u) : \\
 & \text{rsolve}\left(u(n) = (u(n-1))^2 - e^{2u(n-2)}, u(n)\right) \\
 & \text{rsolve}\left(u(n) = (u(n-1))^2 - e^{2u(n-2)}, u(n)\right)
 \end{aligned} \tag{8.43}$$

Problem Solving with Maple and Recurrence Relations

It is often the case that a problem, as presented, gives no clue that a solution may be found using recurrence. Let us see how we can use Maple to solve a problem that is not explicitly expressed as one requiring the use of recurrence for its solution.

Here is our problem: into how many regions is the plane divided by 1000 lines, assuming that no two of the lines are parallel, and no three are coincident? Such a situation may arise in an attempt to model fissures in the ocean floor.

To start, we might try to discover the answer for smaller numbers of lines. To generalize the problem, we may ask for the number of regions produced by n lines, where n is some positive integer. It is fairly obvious that a single line (corresponding to the case $n = 1$) divides the plane into two regions. Two lines, if they are not parallel, can easily be seen to divide the plane into four regions. (Two parallel lines produce only three regions.) If we call the number of regions produced by n lines, no two of which are parallel and no three of which are coincident, R_n , then $R_1 = 2$ and $R_2 = 4$.

What does the situation look like when $n = 3$? Figure 8.1 is representative of this situation. In this case, the number of regions is 7, so $R_3 = 7$. To find R_4 we must add a fourth line to the diagram. This suggests trying to compute R_4 in terms of R_3 so that we begin to think of $\{R_n\}$ as a recurrence relation. Figure 8.2 shows what the situation looks like when a fourth line is added to the three

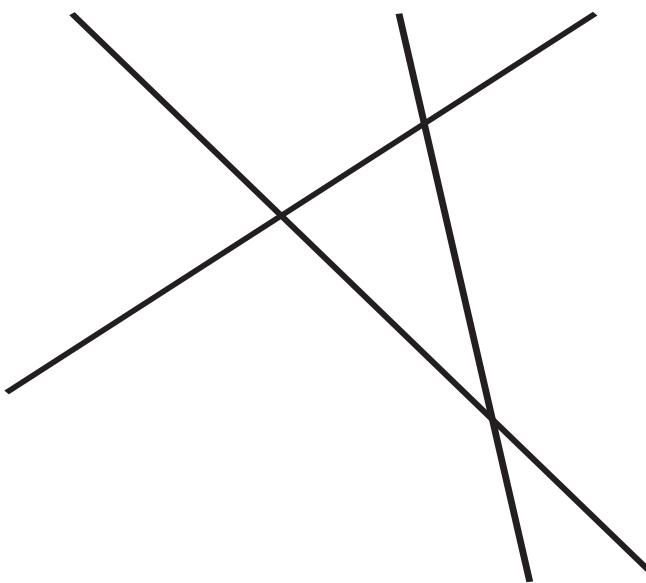


Figure 8.1: Three lines dividing the plane

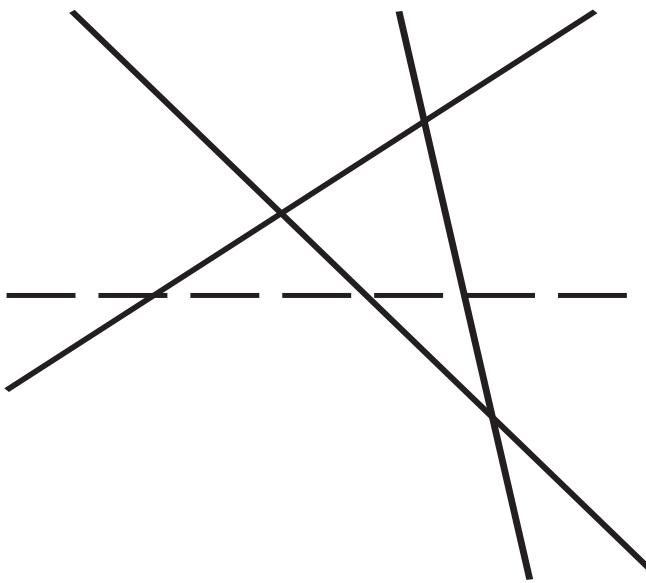


Figure 8.2: Four lines dividing the plane

existing lines. From the assumptions that no two lines are parallel and no three pass through a single point, it follows that the new line must intersect each of the existing three lines in exactly one point. This means that the new line passes through exactly four of the regions formed by the original three lines. Each region that it passes through is divided into two regions, so the total number of new regions added by the fourth point is 4. Thus, $R_4 = R_3 + 4$. Similar arguments for a general configuration of lines reveals that R_n satisfies the recurrence relation $R_n = R_{n-1} + n$.

Furthermore, we have already computed the initial condition $R_1 = 2$. This is enough to solve the recurrence.

$$\begin{aligned} > \text{unassign}(R) : \\ & \text{rsolve}(\{R(1) = 2, R(n) = R(n-1) + n\}, R(n)) \\ & (n+1)\left(\frac{n}{2} + 1\right) - n \end{aligned} \tag{8.44}$$

$$\begin{aligned} > \text{simplify}((8.44)) \\ & \frac{1}{2}n^2 + \frac{n}{2} + 1 \end{aligned} \tag{8.45}$$

To answer the question: how many regions is the plane divided by 1000 lines with no two parallel and no three coincident?

$$\begin{aligned} > R := \text{unapply}((8.44), n) \\ & R := n \mapsto (n+1)\left(\frac{n}{2} + 1\right) - n \end{aligned} \tag{8.46}$$

$$\begin{aligned} > R(1000) \\ & 500\,501 \end{aligned} \tag{8.47}$$

8.3 Divide-and-Conquer Algorithms and Recurrence Relations

A very good example of divide-and-conquer relations is the one provided by the binary search algorithm. Here, we consider a practical application of this algorithm in an implementation of a binary search on a sorted list of integers. This is an implementation of the algorithm described in Algorithm 3 in Section 3.1 of the text and first presented in Section 3.1 of this manual.

```

1  binarysearch := proc (x: :integer, A: :list (integer) )
2      local n, i, j, m, location;
3      n := nops (A) ;
4      i := 1;
5      j := n;
6      while i < j do
7          m := floor ((i+j)/2);
8          if x > A[m] then
9              i := m + 1;
10         else
11             j := m;
12         end if ;
13     end do;
14     if x = A[i] then
15         location := i;
16     else
17         location := 0;
18     end if ;
19     return location;
20 end proc;
```

The variable **A** is the list of integers to search, which is assumed to be sorted in increasing order, and **x** is the integer to search for. The local variables **j** and **i** are initialized to the number of elements in the list and 1, respectively. The while loop continues as long as **i** and **j** are different from each other. Each step of the loop serves to narrow the difference between them by calculating the middle of the list, represented by **m**, and determining which half **x** is in. Eventually, the search will focus in on one location in the list, which is either **x** or, if not, the search has failed and the algorithm returns 0.

Let us now do an analysis of the algorithm to see how divide-and-conquer recurrence relations are generated. In general, a divide-and-conquer type recurrence relation has the form

$$f(n) = af(n/b) + g(n).$$

Each iteration of the while loop of **binarysearch** produces a single list half the size of the original. Therefore, $a = 1$ and $b = 2$. The function $g(n)$, which measures the comparisons added in implementing the reduction, is identically 2. This is because one comparison is added to see which half of the list the key is on, and one is added to see if the while loop needs to continue. Hence, for the **binarysearch** algorithm, the recurrence relation is

$$f(n) = f(n/2) + 2.$$

Additionally, we can see that $f(1) = 2$, because if the list is of length 1, then the algorithm will do one comparison to determine that the while loop is unnecessary and one comparison to make sure that the element being searched for is the one element in the list. We can now use **rsolve** to solve this recurrence.

$$\begin{aligned} > \text{unassign}(bin) : \\ & \text{rsolve}\left(\left\{bin(1) = 2, bin(n) = bin\left(\frac{n}{2}\right) + 2\right\}, bin(n)\right) \\ & 2 + \frac{2 \ln(n)}{\ln(2)} \end{aligned} \tag{8.48}$$

8.4 Generating Functions

Generating functions are a powerful tool for manipulating sequences of numbers and for solving a variety of counting problems. In this section, we will see how Maple can be used to represent and manipulate generating functions.

The generating function $G(x)$ for a sequence $\{a_k\}$ is the formal power series

$$\sum_{k=0}^{\infty} a_k x^k = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_k x^k + \cdots.$$

It is called *formal* because we are not interested in evaluating it as a function of x . Our focus is on finding a formula for its coefficients. In particular, this means that there are no convergence issues to be considered.

Power Series Tools

Maple provides extensive facilities for manipulating formal power series. They belong to the Maple package **powseries**.

$$> \text{with}(powseries) :$$

The first thing we need to do is to learn how to create a power series with Maple, which is done with the command **powcreate**. This command can be used in two ways: either by specifying a closed form for the general coefficient or with a recurrence relation and initial conditions.

For example, to create the generating function for the sequence $\{3^k\}$, we use the following code.

```
> unassign(e):
powcreate (e(k) = 3^k)
```

Note that we use the **unassign** command with **powcreate**, just as we did with **rsolve**. While it is not necessary in most situations, the **unassign** command can help avoid errors that may occur if these commands are reexecuted.

Note that the **powcreate** command does not have a return value but has made **e** into a procedure which is now Maple's representation of the power series. We can confirm this by having Maple display the first few terms of the series using the **tpsform** command (for truncated power series). This command takes three arguments: the name of the power series, the variable to be used in the expression, and the order. Maple displays the terms of the series whose degree is smaller than the given order.

```
> tpsform(e,x,5)
1 + 3x + 9x^2 + 27x^3 + 81x^4 + O(x^5) (8.49)
```

We can also define a power series using a recurrence relation together with initial conditions. (Note: If you do not provide sufficiently many initial conditions to guarantee a unique solution to the recurrence, an error will be raised.)

For example, to create the generating function for the Fibonacci sequence, which is defined by the recurrence relation

$$F_n = F_{n-1} + F_{n-2},$$

with initial conditions

$$F_0 = 1 \text{ and } F_1 = 1,$$

we enter the following command.

```
> unassign(Fibonacci3):
powcreate(Fibonacci3(n) = Fibonacci3(n - 1) + Fibonacci3(n - 2),
Fibonacci3(0) = 1, Fibonacci3(1) = 1)
```

We can then have Maple display the first few terms of the generating function for the Fibonacci numbers.

```
> tpsform(Fibonacci3,x,7)
1 + x + 2x^2 + 3x^3 + 5x^4 + 8x^5 + 13x^6 + O(x^7) (8.50)
```

Maple also provides a way to access an arbitrary coefficient in a formal power series. To Maple, each formal power series is, in fact, a procedure that takes integer arguments. The value returned

when given the integer n as an argument is the coefficient of the x^n term. For example, the 50th Fibonacci number can be computed as follows.

```
> Fibonacci3(50)
20365011074
```

(8.51)

Solving Problems with Generating Functions

Generating functions are more than just a convenient way to represent numerical sequences. They are a powerful tool for solving recurrence relations, as well as other kinds of counting problems. This power stems from our ability to manipulate them like ordinary power series from Calculus and to interpret those manipulations. To illustrate Maple's facilities for manipulating generating functions, consider Example 12 from Section 8.4 of the text.

Use generating functions to determine the number of ways to insert tokens worth \$1, \$2, and \$5 into a vending machine to pay for an item that costs r dollars in both the cases when the order in which the tokens are inserted does not matter and when the order does matter.

Following the text, the solution to the problem when order does matter is the coefficient of x^r in the generating function

$$(1 + x + x^2 + x^3 + \dots)(1 + x^2 + x^4 + x^6 + \dots)(1 + x^5 + x^{10} + x^{15} + \dots).$$

To solve the problem, we need to create the three power series and multiply them together. To create the three series, we could use the **powcreate** command as above, but the **evalpow** command provides an approach that is often easier.

The **evalpow** command accepts one argument, an algebraic expression which may include formal power series, polynomials, the usual arithmetic operators, and some functions compatible with the power series package (such as **powdiff** and **powint** for differentiation and integration). From Table 1 of Section 8.4 of the text, we see that our three generating functions can be written as rational expressions:

$$\frac{1}{1 - x^r} = 1 + x^r + x^{2r} + x^{3r} + \dots$$

The three generating functions that we are interested in all share this same form with $r = 1$, $r = 2$, and $r = 5$, respectively. Therefore, we can use **evalpow** to create them.

```
> Token1D := evalpow(1/(1-x))
Token1D := proc (powparm) ...end proc
```

(8.52)

```
> Token2D := evalpow(1/(1-x^2))
Token2D := proc (powparm) ...end proc
```

(8.53)

```
> Token5D := evalpow(1/(1-x^5))
Token5D := proc (powparm) ...end proc
```

(8.54)

(It is worth noting that the **evalpow** command returns a procedure, which is how Maple represents power series.)

We use **tpsform** to confirm that these are the generating functions that we were trying to create.

$$> \text{tpsform}(Token1D, x, 5) \\ 1 + x + x^2 + x^3 + x^4 + O(x^5) \quad (8.55)$$

$$> \text{tpsform}(Token2D, x, 10) \\ 1 + x^2 + x^4 + x^6 + x^8 + O(x^{10}) \quad (8.56)$$

$$> \text{tpsform}(Token5D, x, 25) \\ 1 + x^5 + x^{10} + x^{15} + x^{20} + O(x^{25}) \quad (8.57)$$

Now, we can use **evalpow** again to multiply the three series together.

$$> Tokens := evalpow(Token1D \cdot Token2D \cdot Token5D) \\ Tokens := \text{proc}(powparm) \dots \text{end proc} \quad (8.58)$$

$$> \text{tpsform}(Tokens, x, 8) \\ 1 + x + 2x^2 + 2x^3 + 3x^4 + 4x^5 + 5x^6 + 6x^7 + O(x^8) \quad (8.59)$$

We can see from the above that there are six ways to pay for a \$7 item (since the coefficient of x^7 is 6), just as was computed in the text. If we wanted to know the number of ways to pay for an item costing \$234, all we would need to do is find the coefficient of x^{234} .

$$> Tokens(234) \\ 2832 \quad (8.60)$$

For the second part, the case where the order does matter, the text explains that the generating function we need is

$$1 + (x + x^2 + x^5) + (x + x^2 + x^5)^2 + \dots = \frac{1}{1 - (x + x^2 + x^5)}.$$

We can create this power series in Maple just like we did above.

$$> Tokens2 := evalpow\left(\frac{1}{1 - (x + x^2 + x^5)}\right) \\ Tokens2 := \text{proc}(powparm) \dots \text{end proc} \quad (8.61)$$

$$> \text{tpsform}(Tokens2, x, 8) \\ 1 + x + 2x^2 + 3x^3 + 5x^4 + 9x^5 + 15x^6 + 26x^7 + O(x^8) \quad (8.62)$$

We see that the coefficient of x^7 is 26, so there are 26 ways to pay for a \$7 item when order does matter.

8.5 Inclusion–Exclusion

In this section, we will apply the principle of inclusion and exclusion. At the heart of the principle of inclusion and exclusion is the formula

$$|A \cup B| = |A| + |B| - |A \cap B|,$$

which says that, for two finite sets A and B , the number of elements in the union $A \cup B$ of the two sets may be found by adding the sizes of A and B and then subtracting the number of elements common to both A and B , which would otherwise be counted twice. This formula can be generalized to count the number of elements in the union of any finite number of finite sets.

Recall that we define a set in Maple by enclosing the comma-separated list of elements in braces.

```
> A := {1,2,3}  
A := {1,2,3} (8.63)
```

To find the cardinality, or size of a set, we can use the command **nops** or **numelems**.

```
> nops(A)  
3 (8.64)
```

```
> numelems(A)  
3 (8.65)
```

The set operations of **union** and **intersect** are represented in Maple by writing out their names:

```
> X := {1,2,3,4,5}:  
Y := {4,5,6,7,8}:  
> X union Y  
{1,2,3,4,5,6,7,8} (8.66)
```

```
> X intersect Y  
{4,5} (8.67)
```

The set theoretic difference is computed by the Maple operator **minus**.

```
> X minus Y  
{1,2,3} (8.68)
```

Let us use the operations to verify the principle of inclusion and exclusion in a particular example.

```
> Flintstones := {"Fred", "Pebbles", "Wilma"}  
Flintstones := {"Fred", "Pebbles", "Wilma"} (8.69)
```

```
> Rubbles := {"BamBam", "Barney", "Betty"}  
Rubbles := {"BamBam", "Barney", "Betty"} (8.70)
```

```
> Husbands := {"Barney", "Fred"}  
Husbands := {"Barney", "Fred"} (8.71)
```

```
> Wives := {"Betty", "Wilma"}  
Wives := {"Betty", "Wilma"}
```

(8.72)

```
> Kids := {"BamBam", "Pebbles"}  
Kids := {"BamBam", "Pebbles"}
```

(8.73)

If this were a complete census, then the number of children living in Bedrock would be

```
> nops(Kids)  
2
```

(8.74)

while the number of Bedrock inhabitants who are either Flintstones or children is

```
> nops(Flintstones union Kids)  
4
```

(8.75)

According to the principle of inclusion and exclusion, this number should be the same as

```
> nops(Flintstones) + nops(Kids) - nops(Flintstones intersect Kids)  
4
```

(8.76)

which, of course, it is.

As another example, consider the problem of determining the number of positive integers less than or equal to 100 that are not divisible by either 2 or 11. First, we generate the set of positive integers less than or equal to 100.

```
> hundred := {$1..100}  
hundred := {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,  
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,  
41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,  
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80,  
81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}
```

(8.77)

Next, we remove those elements that are divisible by 2:

```
> DivBy2 := hundred minus {seq(2 · i, i = 1 .. 100)}  
DivBy2 := {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33,  
35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71,  
73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99}
```

(8.78)

and those that are divisible by 11:

```
> DivBy11 := hundred minus {seq(11 · i, i = 1 .. 100)}  
DivBy11 := {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 17, 18, 19,  
20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 34, 35, 36, 37, 38, 39, 40,  
41, 42, 43, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 56, 57, 58, 59, 60, 61,  
62, 63, 64, 65, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 78, 79, 80, 81, 82,  
83, 84, 85, 86, 87, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 100}
```

(8.79)

(Note the combined use of the **minus** and **seq** operators; they work very conveniently together in this example.)

We are looking for integers that belong to either or both of A and B , that is, to their union, so we want the size of $A \cup B$, which is

```
> nops(DivBy2 union DivBy11)  
96
```

(8.80)

According to the principle of inclusion and exclusion, this value could also be computed as

```
> nops(DivBy2) + nops(DivBy11) - nops(DivBy2 intersect DivBy11)  
96
```

(8.81)

8.6 Applications of Inclusion–Exclusion

In this section, we will explore the following problem: Three sets of twins, Ashley and Amanda Abel; Brandon and Benjamin Bernoulli; and Christopher and Courtney Cartan (none of whom bear any relation to the mathematicians with the same surname), are to be seated in a row. List the ways in which they can be seated so that no person sits next to his or her twin.

The principle of inclusion–exclusion gives us insight into how we might accomplish this task. Rather than attempting to generate the seating arrangements in which no person sits next to his or her twin, it will be easier to consider all the possible arrangements of the twins and then exclude those that do not satisfy the condition. To begin, we define lists to store the names of the twins.

```
> Abels := ["Ashley", "Amanda"]:  
Bernoullis := ["Brandon", "Benjamin"]:  
Cartans := ["Christopher", "Courtney"]:  
Abels := ["Ashley", "Amanda"]  
Bernoullis := ["Brandon", "Benjamin"]  
Cartans := ["Christopher", "Courtney"]
```

(8.82)

```
> twins := [Abels, Bernoullis, Cartans]  
twins := [[["Ashley", "Amanda"], ["Brandon", "Benjamin"],  
["Christopher", "Courtney"]]]
```

(8.83)

We solve the problem using two procedures. The main procedure will consider each possible arrangement in turn and populate the list of those which satisfy the condition. The main procedure will rely on a second procedure to test whether an arrangement satisfies the condition of having no twins seated next to each other. We begin by writing the second procedure.

To test whether a given arrangement has a pair of twins seated next to one another, we will consider the five pairs of seats, 1 and 2, 2 and 3, ..., 5 and 6, and check to see if the people seated in those positions are twins.

```
1 testSeating := proc(seating :: list)  
2   global twins;  
3   local i, twinpair;
```

```

4   for i from 1 to 5 do
5     for twinpair in twins do
6       if ((seating[i] in twinpair) and (seating[i+1] in twinpair))
7         then
8           return false;
9         end if;
10        end do;
11      end do;
12      return true;
end proc:

```

The procedure is passed a list of the six people's names, representing a seating, so that the person in the third seat is **seating[3]**. The **for** loop indexed by **i** goes through the five pairs of seats. The inner **for** loop avoids having to duplicate the **if** statement. We could have written one **if** statement checking to see if the people in seats **i** and **i+1** are both Abels, and then a second if statement to see if they are both Bernoullis, and then a third to see if they are both Cartans. Instead, the loop sets the **twinpair** variable to each of the lists in **twins** in turn. Thus, **twinpair** represents, at each step in the loop, one of the families. And then the **if** statement checks to see whether the people in the seats **i** and **i+1** are members of that family, using the **in** operator. If any of these **if** statements are true, that the people in the pair of consecutive seats are from the same family, then the procedure immediately returns false, indicating that the seating is not acceptable. If the seating survives all of the **if** statements, then the procedure returns true.

To check the **TestSeating** procedure, consider the following potential seatings.

```

> seating1 := [“Ashley”, “Amanda”, “Brandon”, “Benjamin”,
  “Christopher”, “Courtney”]
  seating1 := [“Ashley”, “Amanda”, “Brandon”, “Benjamin”,
  “Christopher”, “Courtney”] (8.84)

```

```

> seating2 := [“Ashley”, “Brandon”, “Christopher”, “Amanda”,
  “Benjamin”, “Courtney”]
  seating2 := [“Ashley”, “Brandon”, “Christopher”, “Amanda”,
  “Benjamin”, “Courtney”] (8.85)

```

We see that the first seating fails but the second passes, as they should.

```

> testSeating(seating1)
false (8.86)

```

```

> testSeating(seating2)
true (8.87)

```

We now have a procedure to test a potential seating for the condition of not having twins seated next to each other. To generate a list of all of such seatings, we use Maple's **permute** command (from the **combinat** package) to generate all the possible permutations of the people and then test to see which

are valid and which should be discarded. The **permute** command takes a list and returns all the possible permutations of the objects. (Refer to Chapter 6 of this manual for more information about the use of the commands in the **combinat** package.)

Note that the **permute** command expects a list of objects, so we will need a list of all the people. We use the **Flatten** command to turn our **twins** list into a list of all the names. **Flatten** takes a list and removes any nesting of lists so that the result is a list of the objects.

```
> ListTools[Flatten](twins)
["Ashley", "Amanda", "Brandon", "Benjamin", "Christopher",
 "Courtney"]
```

(8.88)

Here is the procedure to generate all of the valid seating arrangements.

```
1 ListSeatings := proc()
2   global twins;
3   local possibles, seating, OKseatings;
4   OKseatings := [];
5   possibles := combinat[permute](ListTools[Flatten](twins));
6   for seating in possibles do
7     if testSeating(seating) then
8       OKseatings := [op(OKseatings), seating];
9     end if;
10    end do;
11    return OKseatings;
12  end proc;
```

This procedure initializes **OKseatings**, which is what will be returned, to the empty list, and it initializes the **possibles** variable to all the permutations of the six people. It then checks each possible permutation using the **testSeating** procedure, and the valid arrangements are added to **OKseatings**. Finally, the procedure returns **OKseatings**.

We run this procedure and store its output in the variable **twinSeatings** but suppress the output. Then, we use **nops** to check how many possible seatings there are. (It is generally a good idea to suppress the output of a procedure that is listing what may be a very large number of possibilities until you know how many there are. Otherwise you may have to wait a very long time for Maple to print all of them out.)

```
> twinSeatings := ListSeatings():
> nops(twinSeatings)
240
```

(8.89)

```
> twinSeatings[123]
["Benjamin", "Ashley", "Christopher", "Amanda", "Brandon",
 "Courtney"]
```

(8.90)

We see that there are 240 possible seatings and have displayed the 123rd seating.

Solutions to Computer Projects and Computations and Explorations

Computer Projects 12

Given positive integers m and n , find the number of onto functions from a set with m elements to a set with n elements.

Solution: We have a very convenient formula:

$$\sum_{k=0}^{n-1} (-1)^k C(n, k) (n - k)^m.$$

This is the number of onto functions from a set of m elements to a set of n elements, assuming $m \geq n$. This formula is derived in the textbook from the principle of inclusion–exclusion (see Theorem 1 of Section 8.6). The only input required in this formula are the integer parameters m and n , which represent the sizes of the domain and codomain, respectively. Maple has a command that corresponds to summations like the one above called **add**. The **add** command can take a few different forms, but the one similar to standard summation notation has two arguments: an expression in terms of an index of summation, for example, **k**, and an argument of the form **k=a..b** indicating the bounds of the summation. For example, to compute

$$\sum_{k=3}^8 \frac{1}{k},$$

you would enter the following command.

$$\begin{aligned} > \text{add}\left(\frac{1}{k}, k = 3 .. 8\right) \\ &\frac{341}{280} \end{aligned} \tag{8.91}$$

The **add** command will form the heart of our procedure, which will take the sizes of the domain and codomain as input, check to make sure that the assumption $m \geq n$ is satisfied, and then apply the formula.

```
1 OntoFunctions := proc (m::posint, n::posint)
2   local k;
3   if m < n then
4     return 0;
5   end if;
6   return add ( (-1)^k * combinat[numbcomb](n, k) * (n-k)^m, k=0..n-1);
7 end proc;
```

The if statement is necessary—and makes sense mathematically—because there are no onto functions from a set to a larger set. For example:

$$\begin{aligned} > \text{OntoFunctions}(4, 9) \\ 0 \end{aligned} \tag{8.92}$$

As an example, we can use our function to compute the number of onto functions from a set with 100 elements to a set with 20 elements.

```
> OntoFunctions(100, 20)
11 238 195 910 319 657 928 539 447 038 143 170 285 517 894 975 095
769 496 294 319 007 413 091 913 959 828 334 936 464 196 298 192
508 890 182 316 163 261 067 934 269 440 000
```

(8.93)

Computations and Explorations 2

Find the smallest Fibonacci number greater than 1 000 000, greater than 1 000 000 000, and greater than 1 000 000 000 000.

Solution: We can solve this quite easily with Maple using a simple **while** loop. In this chapter, we have seen several ways to compute Fibonacci numbers, including the procedure **Fibonacci** in Section 8.1, the formula **Fibonacci2** in Section 8.2, and the generating function **Fibonacci3** in Section 8.4. For this exercise, we will use Maple's built-in function **fibonacci** in the **combinat** package. (Caution: there are other commands also called "fibonacci" in Maple, within the **StringTools** package and the (deprecated) **linalg** package.)

The idea is to compute Fibonacci numbers until the value exceeds the target. The **while** loop is well-suited to this sort of problem. We will create a procedure that takes the target values as input and prints out the desired Fibonacci number and its index.

```
1 FindFib := proc(target : :integer)
2   local n;
3   n := 1;
4   while combinat[fibonacci](n) < target do
5     n := n + 1;
6   end do;
7   printf("The %dth Fibonacci number is %d", n, combinat[fibonacci](n));
8 end proc;
```

As long as the n th Fibonacci number is smaller than the target value, the index n is increased. Once the target has been exceeded, the **printf** statement displays the index and the value of the Fibonacci number.

The numbers called for by the question are:

```
> FindFib(1 000 000)
The 31th Fibonacci number is 1 346 269

> FindFib(1 000 000 000)
The 45th Fibonacci number is 1 134 903 170

> FindFib(1 000 000 000 000)
The 60th Fibonacci number is 1 548 008 755 920
```

Computations and Explorations 3

Find as many prime Fibonacci numbers as you can. It is unknown whether there are infinitely many of these.

Solution: Using Maple, this sort of problem becomes fairly straightforward. We can simply use the Maple procedure **fibonacci** from the **combinat** package to generate Fibonacci numbers and use the **isprime** function to test each for primality. We will wrap this in a procedure so that we can use it in conjunction with the **timelimit** function. The number of Fibonacci numbers tested will depend on your computer and patience.

```
1 PrimeFib := proc ()
2   global primefibs;
3   local i, temp;
4   primefibs := NULL;
5   for i from 1 do
6     temp := combinat[fibonacci](i);
7     if isprime(temp) then
8       primefibs := primefibs, temp;
9     end if;
10    end do;
11    return primefibs;
12  end proc;
```

This produces fairly large values relatively quickly, so we limit it to a tenth of a second.

```
> try
  timelimit(0.1, PrimeFib());
catch "time expired":
  [primefibs];
end try;
[2, 3, 5, 13, 89, 233, 1597, 28657, 514229, 433494437, 2971215073,
 99194853094755497, 1066340417491710595814572169,
 19134702400093278081449423917,
 475420437734698220747368027166749382927701417016557193662268716\
 376935476241,
 529892711006095621792039556787784670197112759029534506620905162\
 834769955134424689676262369,
 1387277127804783827114186103186246392258450358171783690079918\
 032136025225954602593712568353,
 3061719992484545030554313848083717208111285432353738497131674\
 799321571238149015933442805665949,
 10597999265301490732599643671505003412515860435409421932560009\
 680142974347195483140293254396195769876129909,
 36684474316080978061473613646275630451100586901195229815270242\
 868417768061193560857904335017879540515228143777781065869,
 96041200618922553823942883360924865026104917411877067816822264\
 789029014378308478864192589084185254331637646183008074629]
```

(8.94)

Computations and Explorations 11

Compute the probability that a permutation of n objects is a derangement for all positive integers not exceeding 20 and determine how quickly these probabilities approach the number $1/e$.

Solution: To solve this problem, we will make use of the formula which gives the number of derangements of n objects, namely,

$$D_n = n! \left[1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \cdots + (-1)^n \frac{1}{n!} \right].$$

The total number of permutations of n objects is, of course, $n!$, so the probability that one of them is a derangement is the ratio $\frac{D_n}{n!}$, which is given by the expression

$$1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \cdots + (-1)^n \frac{1}{n!}.$$

A very simple Maple function will compute these values for us.

```

1 | DerProb := proc (n :: posint)
2 |   local k;
3 |   return add ( (-1) ^k * (1/k!), k=0..n);
4 | end proc;
```

The probabilities that a permutation of n objects is a derangement for $n \leq 20$ are:

$$\begin{aligned}
&> \text{seq}(\text{DerProb}(n), n = 1 .. 20) \\
&0, \frac{1}{2}, \frac{1}{3}, \frac{3}{8}, \frac{11}{30}, \frac{53}{144}, \frac{103}{280}, \frac{2119}{5760}, \frac{16687}{45360}, \frac{16481}{44800}, \frac{1468457}{3991680}, \\
&\frac{16019531}{43545600}, \frac{63633137}{172972800}, \frac{2467007773}{6706022400}, \frac{34361893981}{93405312000}, \\
&\frac{15549624751}{42268262400}, \frac{8178130767479}{22230464256000}, \frac{138547156531409}{376610217984000}, \\
&\frac{92079694567171}{250298560512000}, \frac{4282366656425369}{11640679464960000} \tag{8.95}
\end{aligned}$$

To see how these probabilities differ from $1/e$, we will multiply them by e and subtract 1. To represent the number e in Maple, we use the **exp** command with argument 1, that is, e^1 .

$$\begin{aligned}
&> \text{seq}(\text{evalf}[25](\text{exp}(1) \cdot \text{DerProb}(n) - 1), n = 1 .. 20) \\
&-1.0, 0.359140914229522617680144, -0.0939060571803182548799044, \\
&0.019355685672141963260108, -0.0032966628983500803678947, \\
&0.000478728530065260236772, -0.0000606131025655027067517, \\
&6.804601513342661189 \cdot 10^{-6}, -6.862544954179352489 \cdot 10^{-7}, \\
&6.2831105458124395 \cdot 10^{-8}, -5.2675855306083001 \cdot 10^{-9}, \\
&4.07305385119424 \cdot 10^{-10}, -2.92246853211696 \cdot 10^{-11}, \\
&1.956033996016 \cdot 10^{-12}, -1.226806251301 \cdot 10^{-13}, \\
&7.239038692 \cdot 10^{-15}, -4.032944743 \cdot 10^{-16}, 2.127959000 \cdot 10^{-17}, \\
&-1.066413000 \cdot 10^{-18}, 5.08870000010 \cdot 10^{-20} \tag{8.96}
\end{aligned}$$

(Note: the optional form **evalf[n](expr)** of **evalf** specifies that Maple should use **n** digits of precision when evaluating the expression **expr**. The default is 10 digits of precision.)

Exercises

Exercise 1. Implement a procedure to find the optimal schedule that maximizes total attendance.

Exercise 2. Implement a dynamic programming algorithm for finding the maximum sum of consecutive terms of a sequence of real numbers. (See Exercise 56 in Section 8.1.)

Exercise 3. Implement a dynamic programming algorithm for optimally computing matrix-chain multiplication. (See Exercise 57 in Section 8.1.)

Exercise 4. Use Maple to solve the following recurrence relations.

- $a_n = a_{n-1} - a_{n-2}$, $a_1 = 1$, $a_2 = 1$;
- $a_n = 15a_{n-1} + \frac{1}{2}a_{n-2}$, $a_1 = \frac{23}{22}$, $a_2 = \frac{7}{2}$.

Exercise 5. Use Maple to solve each of the recurrence relations in Exercise 1 in Section 8.2 of the textbook. (Solve even those that are *not* linear homogeneous recurrence relations with constant coefficients.)

Exercise 6. Write a general solver in Maple for linear homogeneous recurrence relations with constant coefficients of degree three with distinct roots. Your solver should check that the roots are in fact distinct and, if they are not, should return **FAIL**, which is the standard return value for a Maple function when it cannot complete a computation for some reason.

Exercise 7. Use Maple to investigate the behavior of the limit

$$\lim_{n \rightarrow \infty} \frac{\varphi_n}{\psi_n},$$

where φ_n is defined to be the number of prime Fibonacci numbers less than or equal to n , and ψ_n is defined to be the number of Fermat numbers less than or equal to n .

Exercise 8. Implement the recursive algorithm described in Example 12 of Section 8.3 of the text for solving the closest-pair problem.

Exercise 9. Use Maple to find the number of square-free integers less than 100 000 000.

Exercise 10. Use Maple to find the number of onto functions from a set with 1 000 000 elements to a set with 1000 elements.

Exercise 11. It is probably obvious that the number of onto functions from one set to another increases with the sizes of either the domain or the range. Using Maple to experiment, explore whether an increase in the size of the domain or the size of the range has the greater impact on the number of onto functions.

Exercise 12. To generate the *lucky numbers* start with the positive integers and delete every second integer in the list, starting the count with 1 (e.g., delete 2, 4, 6, etc., leaving 1, 3, 5, 7, ...). Other than 1, the smallest integer left is 3. Continue by deleting every third integer from those that remain,

starting the counting with 1 (since 1, 3, 5, 7, 9, ... remain, 1 is the first number left, 3 is the second one left, 5 is the third left and gets deleted, and so on). Continue the process where at each stage, every k th integer is deleted, where k is the smallest integer left, other than the previous values of k . The integers that remain are the lucky numbers. Develop a Maple procedure that generates the lucky numbers up to n .

Exercise 13. Can you make any conjectures about lucky numbers by looking at a list of the first 1000 of them? For example, what sort of conjectures can you make about twin lucky numbers? What evidence do you have for your conjectures?

Exercise 14. Generalize the **ListSeatings** procedure to accept one argument, a list of lists (the same structure as the **twins** list), and determines the arrangements such that no two from the same sublist are seated next to one another.

Exercise 15. Further generalize the **ListSeatings** procedure so that it takes two arguments: a list of lists as before and a number n . The procedure should determine the arrangements of the people such that no n from the same sublist are seated together.

9 Relations

Introduction

In this chapter, we will learn how to use Maple to work with relations. We explain how to use Maple to represent binary relations using sets of ordered pairs, zero-one matrices, and directed graphs. We show how to use Maple to determine whether a relation has various properties using these different representations.

We also describe how to compute closures of relations. In particular, we show how to find the transitive closure of a relation using two different algorithms and we compare the time performance of these algorithms. After explaining how to use Maple to work with equivalence relations, we show how to use Maple to work with partial orderings and how to use Maple to draw Hasse diagrams and how to implement topological sorting.

9.1 Relations and Their Properties

The first step in understanding and manipulating relations in Maple is to determine how to represent them. There is no specific package in Maple designed to handle relations. We will implement relations in Maple using the most convenient form for the question at hand. In this chapter, we will make use of sets of ordered pairs, zero-one matrices, and directed graphs in order to explore relations in Maple.

Relations as Ordered Pairs

In this section, we will represent relations as sets of ordered pairs. We define our own type for relations, which we will call **rel**. Our reason for defining a type is that it gives us a way to ensure that when arguments are passed to procedures we write, the arguments are valid for that procedure. In the past, we have made use of Maple's existing types for this purpose. As an illustration of the utility of types, consider the procedure below.

```
1 myFactorial := proc(n :: posint)
2   if n = 1 then
3     return 1;
4   else
5     return n * myFactorial(n-1);
6   end if;
7 end proc;
```

In this simple example, when we declare **myFactorial** to be a procedure that takes **n** as an input, we also specify that **n** is supposed to be of type **posint**, that is, a positive integer. That way, if we try to compute the factorial of -3 :

```
> myFactorial(-3)
```

Error, invalid input: myFactorial expects its 1st argument, n, to be of type posint, but received -3

an error is generated. It is usually better for a program to generate an error when it receives invalid input than to attempt to operate on the bad input value. In the case of **myFactorial**, omitting the type check from the procedure definition would result in an infinite recursion.

We could also deal with the problem of potentially invalid arguments by checking “by hand.” That is, putting the type checking in the body of the procedure rather than as part of its declaration. However, Maple’s automatic type checking results in more readable, and often faster, code.

As mentioned, we are going to represent relations as sets of ordered pairs. We will define two types. First, an “ordered pair” type that we call **pair**. Then the relation type, which will be called **rel**, will be defined to be a set of objects of type **pair**. We define the **pair** type as follows:

```
| 'type/pair' := [anything, anything] ;
```

type/pair := [anything, anything]

On the left side of the assignment, we have ‘**type**/name‘ enclosed in left single quotes. The right side of the assignment uses Maple’s structured type syntax to indicate that the type is a list (indicated by the brackets) of two objects which may be of any type (indicated by the use of the **anything** keyword).

The following defines the relation type.

```
| 'type/rel' := set(pair) ;
```

type/rel := set(pair)

In this case, we indicate that an object of type **rel** is a set of objects of type **pair**. (Note: You may have thought braces would be used to indicate a set, but braces have a different meaning in a structured type. Also note: we cannot use the word **relation** for this type, as that type is already used by the Maple library.)

Creating Relations

Now that we have established the relation type, we create an actual relation.

The Divides Relation

Example 4 in Section 9.1 describes the “divides relation,” that is, $R = \{(a, b) \mid a \text{ divides } b\}$. We will write a procedure to construct this relation. The procedure will consider every possible ordered pair of elements and will include them in the relation if they satisfy the condition that the remainder obtained by dividing b by a is 0 (which is equivalent to saying that the first element of the pair divides the second).

```
1 DividesRelation := proc (A::set(integer))  
2   local a, b, R;  
3   R := {};  
4   for a in A do  
5     for b in A do
```

```

6      if (irem(b,a) = 0) then
7          R := R union { [a,b] } ;
8      end if ;
9      end do ;
10     end do ;
11     return R ;
12 end proc;

```

Note that in defining this procedure, we use an unnamed structured type, **set(integer)**, to ensure that only sets of integers may be passed to the procedure.

We use the procedure to construct the relation on the integers 1 through 4.

```

> DividesRelation ({1,2,3,4})
{[1,1],[1,2],[1,3],[1,4],[2,2],[2,4],[3,3],[4,4]}          (9.1)

```

We can check that this procedure has really produced a relation (i.e., an object of type **rel**) by using the **type** command.

```

> type (9.1),rel
true                                         (9.2)

```

The **type** command takes two arguments. The first is any expression and the second is the name of a type. The command returns true if the expression is of the specified type and false otherwise.

For convenience, we also define the following procedure for constructing the “divides relation” on the set $\{1, 2, \dots, n\}$ given any positive integer n .

```

1 DivRel := proc (n :: posint)
2     DividesRelation ({\$1..n}) ;
3 end proc;

```

```

> Div6 := DivRel(6)
Div6 := {[1,1],[1,2],[1,3],[1,4],[1,5],[1,6],[2,2],[2,4],[2,6],[3,3],
[3,6],[4,4],[5,5],[6,6]}                                     (9.3)

```

The Inverse of a Relation

Now that we have seen an example of a procedure that creates a relation, we look at a simple example of a procedure that manipulates a relation.

For any relation R , its inverse relation, denoted R^{-1} , is defined by $R^{-1} = \{(b, a) \mid (a, b) \in R\}$. The following procedure computes the inverse of a relation.

```

1 InverseRelation := proc (R :: rel)
2     map (u -> [u[2], u[1]], R) ;
3 end proc;

```

The **map** command, which we use above, applies the function specified in the first argument to each operand of the second argument. In this case, the second argument is a **rel** object, that is, a set of

ordered pairs. Thus, the operands of **R** are the ordered pairs. The function, **u ->[u[2],u[1]]**, takes a pair **u**, and reverses its elements.

Since we have defined the “divides” relation, we can use the **InverseRelation** procedure to create the “multiple of” relation.

```
> Mul6 := InverseRelation(Div6)
Mul6 := {[1, 1], [2, 1], [2, 2], [3, 1], [3, 3], [4, 1], [4, 2], [4, 4], [5, 1],
[5, 5], [6, 1], [6, 2], [6, 3], [6, 6]}(9.4)
```

Properties of Relations

Maple can be used to determine if a relation has a particular property, such as reflexivity, symmetry, antisymmetry, or transitivity. This can be accomplished by creating Maple procedures that take as input the given relation, examine the elements of the relation, and return true or false based on whether the relation has the property or not.

Before writing procedures to test for properties of relations, it will be convenient to have a routine that extracts the domain of a given relation. This procedure works by simply collecting all of the elements that appear as either entry in a pair of the relation.

Note that, strictly speaking, this need not equal the domain of the relation, since there may exist elements in the domain that are not related to any object in the domain. It might be better to call this the “effective domain” of the relation. Also note that this function presumes that the relation is a relation on a set, as distinguished from a relation from one set to a different set.

```
1 FindDomain := proc (R::rel)
2   map (op, R);
3 end proc;
```

We again use the **map** command in this procedure. In this case, **map** is being used to apply the function **op** to each pair of **R**. This has the effect of removing the list structure from the pairs in the relation and leaving just the elements. Since **R** is a set, the result of the **map** is a set, and so duplicates are automatically removed. An alternative approach is to use **Flatten** from the **ListTools** package. The downside of that approach is that it requires converting the set of pairs into a list and then converting back to a set after applying the **Flatten** command.

Reflexivity

We are now ready to begin testing relations for various properties. The first property we consider is reflexivity. Remember that a relation *R* is reflexive if $(a, a) \in R$ for every *a* in the domain.

To check to see if a relation is reflexive, we will compute the domain of the relation and then check each element *a* of the domain to see if (a, a) is in the relation. If the procedure finds an element of the domain with $(a, a) \notin R$, then it returns false immediately. If it checks all of the members of the domain with no failures, then it returns true.

```
1 IsReflexive := proc (R::rel)
2   local a;
3   for a in FindDomain(R) do
4     if not ([a, a] in R) then
```

```

5      return false;
6  end if;
7 end do;
8 return true;
9 end proc:
```

We use this on the “divides” relation.

> *IsReflexive (Div6)*
true (9.5)

Symmetry

Next, we will examine the symmetric and antisymmetric properties. To determine whether a relation is symmetric, we simply use the definition. That is, we check, for every member (a, b) of R , whether (b, a) is also a member of the relation. If we discover a pair in the relation for which the reverse pair is not in the relation, then we know that the relation is not symmetric. Otherwise, it must be symmetric. This is the logic employed by the following procedure.

```

1 IsSymmetric := proc (R::rel)
2   local u;
3   for u in R do
4     if not ([u[2], u[1]] in R) then
5       return false;
6     end if;
7   end do;
8   return true;
9 end proc:
```

For example, we can see that the “divides” relation is not symmetric.

> *IsSymmetric (Div6)*
false (9.6)

However, the union of “divides” and “multiple of” is symmetric.

> *DivOrMul6 := Div6 union Mul6*
DivOrMul6 := {[1, 1], [1, 2], [1, 3], [1, 4], [1, 5], [1, 6], [2, 1], [2, 2],
[2, 4], [2, 6], [3, 1], [3, 3], [3, 6], [4, 1], [4, 2], [4, 4], [5, 1], [5, 5], [6, 1],
[6, 2], [6, 3], [6, 6]} (9.7)

> *IsSymmetric (DivOrMul6)*
true (9.8)

To determine whether a given relation R is antisymmetric, we again use the definition. Remember that a relation is antisymmetric when it has the property that whenever a pair (a, b) and its reverse (b, a) both belong to R , then it must be that $a = b$. To check this, we simply loop over all members u of R and see if the opposite pair belongs to R and whether the members of the pair are different.

```

1 IsAntisymmetric := proc (R::rel)
2   local u;
3   for u in R do
4     if (([u[2], u[1]] in R) and (u[1] <> u[2])) then
5       return false;
6     end if;
7   end do;
8   return true;
9 end proc;

```

We use this procedure to check to see if the “divides” and “multiple of” relations defined earlier are antisymmetric.

> *IsAntisymmetric (Div6)*
true (9.9)

> *IsAntisymmetric (Mul6)*
true (9.10)

Transitivity

The transitive property is the most difficult to check. Recall the definition of transitive relations: a relation R is transitive if, whenever (a, b) and (b, c) are in R , then (a, c) must be as well.

To check transitivity, we will consider all possible a, b , and c in the domain of R . Then, if $(a, b) \in R$, $(b, c) \in R$, and $(a, c) \notin R$, we know that the relation is not transitive. If there is no such triple a, b, c to contradict transitivity, then we conclude that the relation is transitive.

Here is the procedure.

```

1 IsTransitive := proc (R::rel)
2   local DomR, a, b, c;
3   DomR := FindDomain(R);
4   for a in DomR do
5     for b in DomR do
6       for c in DomR do
7         if (([a, b] in R) and ([b, c] in R) and not ([a, c] in R)) then
8           return false;
9         end if;
10        end do;
11      end do;
12    end do;
13    return true;
14 end proc;

```

We see that the “divisible” relation is transitive. However, we can cause it to fail to be transitive by removing the $(1, 6)$ pair, since $(1, 2)$ and $(2, 6)$ are in R .

> *IsTransitive (Div6)*
true (9.11)

```

> R2 := Div6 \ {[1,6]}
R2 := {[1,1],[1,2],[1,3],[1,4],[1,5],[2,2],[2,4],[2,6],[3,3],[3,6],
[4,4],[5,5],[6,6]}(9.12)

```

```

> IsTransitive(R2)
false(9.13)

```

9.2 n -ary Relations and Their Applications

Using Maple, we can construct an n -ary relation where n is a positive integer. As in the previous section, we will begin by defining types both for the elements of the relation (**tuple**) and for the n -ary relation (**nrel**). The only difference here, as compared to the types we defined in the previous section, is that we do not know the length of the list that makes up a **tuple**. Therefore, we make a generic list with the structured type syntax.

1	‘type/tuple’ := list (anything) :
2	‘type/nrel’ := set (tuple) :

Consider the following 4-ary relation that represents student records.

```

> R3 := {[“Adams”, 9 012 345, “Politics”, 2.98],
[“Woo”, 9 100 055, “Film Studies”, 4.99],
[“Warshall”, 9 354 321, “Mathematics”, 3.66]}

R3 := {[“Adams”, 9 012 345, “Politics”, 2.98],
[“Woo”, 9 100 055, “Film Studies”, 4.99],
[“Warshall”, 9 354 321, “Mathematics”, 3.66]}(9.14)

```

The first field represents the name of the student, the second field is the student ID number, the third field is the student’s home department, and the last field stores the student’s grade point average. Note that this relation is of type **nrel**.

```

> type(R3,nrel)
true(9.15)

```

While we created a very generic n -ary relation type, you can also create more specific types for particular situations. For instance, the tuples in the relation above will always consist of a string, integer, string, and a floating point number. Therefore, we could make the following type specifically for that kind of relation.

1	‘type/studentrecord’ := [string, integer, string, float] :
2	‘type/studentrel’ := set (studentrecord) :

```

> type(R3,studentrel)
true(9.16)

```

Operations on n -ary Relations

Now, we will create procedures that act on **nrels** to compute projections and the join of relations.

Projection

We will construct a procedure for computing a projection of a relation. The procedure takes as input a **nrel** along with a list of integers representing the indices of the fields that are to remain. The output will be another **nrel**.

```
1 ProjectRelation := proc (R::nrel, P::list(posint) )
2   local u, S;
3   S := {};
4   for u in R do
5     S := S union {u[P]};
6   end do;
7   return S;
8 end proc;
```

The expression **u[P]** returns the sublist of **u** consisting of the elements corresponding to the indices in the list **P**.

We can use this procedure with the relation we created earlier.

> *ProjectRelation(R3,[2,4])*
{[9 012 345, 2.98], [9 100 055, 4.99], [9 354 321, 3.66]} (9.17)

> *ProjectRelation(R3,[3,4,1])*
{{“Film Studies”, 4.99, “Woo”}, {“Mathematics”, 3.66, “Warshall”},
{“Politics”, 2.98, “Adams”}} (9.18)

Join

Next, consider joins of relations. The join operation has applications to databases when tables of information need to be combined in a meaningful manner.

The join procedure that we will implement here follows the following outline.

1. Input two relations *R* and *S* and a positive integer *p*, representing the overlap between the relations.
2. Examine each element *u* of *R* and determine the last *p* fields of *u*.
3. Examine all elements *v* of *S* to determine if the first *p* fields of *v* match the last *p* fields of *u*.
4. Upon finding a match, we combine the elements and place the result in a relation *T*, which is returned as the output of the procedure.

```
1 JoinRelation := proc (R::nrel, S::nrel, p::posint)
2   local overlapR, overlapS, degreeR, i, u, isDone, x, joinElement,
3       T;
4   T := {};
5   degreeR := nops(R[1]);
# overlapR = indices for the last p entries in R elements
6   overlapR := [seq(degreeR - p + i, i=1..p)];
# overlapS = indices for the first p entries in S elements
7   overlapS := [seq(i, i=1..p)];
8   for u in R do
# x = the part of u that is supposed to overlap
```

```

11      x := u[overlapR];
12      i := 1;
13      # examine elements of S until either a match is found
14      # (indicated by isDone = true) or all elements are
15      # examined (tested by i > nops(S))
16      isDone := false;
17      while i <= nops(S) and isDone = false do
18          if S[i][overlapS] = x then
19              joinElement := [op(u), op(S[i][(p+1)..-1])];
20              T := T union {joinElement};
21              isDone := true;
22          end if;
23          i := i + 1;
24      end do;
25  end do;
26  return T;
27 end proc:
```

This procedure will be easier to understand with a simple example in hand.

```
> exampleR := {[“a”, “A”, 1], [“b”, “B”, 2], [“c”, “C”, 3], [“d”, “D”, 4]}
exampleR := {[“a”, “A”, 1], [“b”, “B”, 2], [“c”, “C”, 3],
[“d”, “D”, 4]} (9.19)
```

```
> exampleS := {[“A”, 1, “A1”, “a1”], [“B”, 2, “B2”, “b2”],
[“D”, 4, “D4”, “d4”]}
exampleS := {[“A”, 1, “A1”, “a1”], [“B”, 2, “B2”, “b2”],
[“D”, 4, “D4”, “d4”]} (9.20)
```

The **R** relation elements consist of the lower case and upper case versions of letters and their position in the alphabet (up to “D”). The **S** relation consists of the capital letter, its position in the alphabet, and the two character strings combining letter and position for both upper and lower cases, but omitting the relation for “C”.

The **JoinRelation** procedure begins by initializing the return relation, **T**, to the empty set. It then computes **degreeR**, the degree of the relation **R**, by accessing the first element of **R** and applying the **nops** command. In our example, this line would apply **nops** to the list **[“a”, “A”, 1]** and thus set **degreeR** to **3**.

The next two lines set the lists **overlapS** and **overlapR**. These lists represent the indices of the overlapping elements in the **S** and **R** relations. The indices of the elements of **S** that overlap are merely **1** through **p**. For **R**, we need the last **p** indices, which are obtained by computing **degreeR - p + i** for **i** from **1** to **p**. For our example, the relations overlap in two fields, so **p** is **2**.

```
> overlapS := [1, 2];
overlapR := [2, 3]
overlapS := [1, 2]
overlapR := [2, 3] (9.21)
```

The **for** loop begins the heart of the procedure. For each element **u** of **R**, the variable **x** is set to the portion of **u** that may overlap with elements of **S**. The variable **i** is a counter that tracks the progress as it compares **x** to the elements of **S**. The **isDone** variable is used to track whether a match has been found from **S**. It is initialized to false and is set to true if a match is found, thereby short-circuiting the **while** loop.

The **while** loop is used to check the elements of **S** one at a time until all of the elements of **S** have been checked or until a match has been found. The condition in the if statement takes the **i**th element of **S** (i.e., **S[i]**) and looks at the part of that list that may overlap (namely, **S[i][overlapS]**). For example,

```
> exampleS[3]
["D", 4, "D4", "d4"]
```

(9.22)

```
> exampleS[3][overlapS]
["D", 4]
```

(9.23)

If this segment of **S** is equal to **x**, then a match has been found. The variable **joinElement** is set to the result of combining **u** (from **R**) with the element from **S** that matches. This is accomplished by taking all of **u**, using **op(u)**, together with the portion of the matching element from **S** beyond the overlap, with **op(S[i][(p+1)..-1])**.

The resulting **joinElement** is added to the **T** relation. In addition, **isDone** is set to true which ends the while loop, so that the procedure goes on to the next element of **R**.

We conclude this section by applying the **JoinRelation** procedure to Example 11 of Section 9.2.

```
> TeachingAssignments := {[["Cruz", "Zoology", 335],
  ["Cruz", "Zoology", 412], ["Farber", "Psychology", 501],
  ["Farber", "Psychology", 617], ["Grammer", "Physics", 544],
  ["Grammer", "Physics", 551], ["Rosen", "Mathematics", 575],
  ["Rosen", "Computer Science", 518]}

TeachingAssignments := {[["Cruz", "Zoology", 335],
  ["Cruz", "Zoology", 412], ["Farber", "Psychology", 501],
  ["Farber", "Psychology", 617], ["Grammer", "Physics", 544],
  ["Grammer", "Physics", 551], ["Rosen", "Mathematics", 575],
  ["Rosen", "Computer Science", 518]}

> ClassSchedule := {[["Physics", 544, "B505", "4:00 P.M."],
  ["Zoology", 335, "A100", "9:00 A.M."],
  ["Zoology", 412, "A100", "8:00 A.M."],
  ["Mathematics", 575, "N502", "3:00 P.M."],
  ["Mathematics", 611, "N521", "4:00 P.M."],
  ["Psychology", 501, "A100", "3:00 P.M."],
  ["Psychology", 617, "A110", "11:00 A.M."],
  ["Computer Science", 518, "N521", "2:00 P.M."]}
```

(9.24)

```

ClassSchedule := {[["Physics", 544, "B505", "4:00 P.M."],
  ["Zoology", 335, "A100", "9:00 A.M."],
  ["Zoology", 412, "A100", "8:00 A.M."],
  ["Mathematics", 575, "N502", "3:00 P.M."],
  ["Mathematics", 611, "N521", "4:00 P.M."],
  ["Psychology", 501, "A100", "3:00 P.M."],
  ["Psychology", 617, "A110", "11:00 A.M."],
  ["Computer Science", 518, "N521", "2:00 P.M."]}(9.25)

```

> *TeachingSchedule* := *JoinRelation*(*TeachingAssignments*, *ClassSchedule*, 2)

```

TeachingSchedule := {[["Cruz", "Zoology", 335, "A100", "9:00 A.M."],
  ["Cruz", "Zoology", 412, "A100", "8:00 A.M."],
  ["Farber", "Psychology", 501, "A100", "3:00 P.M."],
  ["Farber", "Psychology", 617, "A110", "11:00 A.M."],
  ["Grammer", "Physics", 544, "B505", "4:00 P.M."],
  ["Rosen", "Mathematics", 575, "N502", "3:00 P.M."],
  ["Rosen", "Computer Science", 518, "N521", "2:00 P.M."]}(9.26)

```

We can make the result a little more readable with a simple loop.

> **for** *u* **in** *TeachingSchedule* **do**
 print(*u*)
end do

```

  ["Cruz", "Zoology", 335, "A100", "9:00 A.M."]
  ["Cruz", "Zoology", 412, "A100", "8:00 A.M."]
  ["Farber", "Psychology", 501, "A100", "3:00 P.M."]
  ["Farber", "Psychology", 617, "A110", "11:00 A.M."]
  ["Grammer", "Physics", 544, "B505", "4:00 P.M."]
  ["Rosen", "Mathematics", 575, "N502", "3:00 P.M."]
  ["Rosen", "Computer Science", 518, "N521", "2:00 P.M."](9.27)

```

9.3 Representing Relations

As was stated earlier in this chapter, Maple allows us to represent and manipulate relations in a variety of ways. From this point forward, we will consider exclusively binary relations. This gives us additional options for how we represent relations. In this section, we will see how to represent binary relations with zero-one matrices and digraphs.

Representing Relations Using Matrices

We begin with representations of relations with zero-one matrices.

A First Example

We create a matrix with the **Matrix** command. To produce a square matrix, this is as simple as giving the Matrix command with a single argument—the dimension of the matrix.

> *exampleMatrix* := *Matrix*(4)

$$exampleMatrix := \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad (9.28)$$

(For a nonsquare matrix, you would need two arguments for the number of rows and the number of columns.) Observe that Maple creates the matrix and puts zeros in all the entries by default.

Right now, this matrix does not represent a very interesting relation. We need to change entries to 1 to represent elements of the domain that are related to each other. For instance, if $(1, 2) \in R$, then we need to change the $(1, 2)$ entry to a 1. To do this, we enter the following.

```
> exampleMatrix[1,2] := 1
exampleMatrix1,2 := 1
```

(9.29)

We can see that it modified the matrix:

```
> exampleMatrix
[ 0 1 0 0 ]
[ 0 0 0 0 ]
[ 0 0 0 0 ]
[ 0 0 0 0 ]
```

(9.30)

Let us make this matrix represent the relation “is one less than” on $\{1, 2, 3, 4\}$, as in, “1 is one less than 2.”

```
> exampleMatrix[2,3] := 1 :
> exampleMatrix[3,4] := 1 :
> exampleMatrix
[ 0 1 0 0 ]
[ 0 0 1 0 ]
[ 0 0 0 1 ]
[ 0 0 0 0 ]
```

(9.31)

Transforming a Set of Pairs Representation into a Matrix

Now, we create a procedure to turn a relation of type **rel** (defined in the first section) into a matrix representation. Doing so is fairly straightforward. Given a relation R , whose domain consists of integers, we can use **FindDomain** from above to extract the domain. We then create a square matrix whose size is equal to the largest integer in the domain, which we can obtain with the **max** function. Then, we simply have to loop through the elements of the relation and set the value of the corresponding entry in the matrix to 1.

```

1 RelToMatrix := proc(R::rel)
2   local u, M, domain;
3   domain := FindDomain(R);
4   M := Matrix(max(domain));
5   for u in R do
6     M[op(u)] := 1;
7   end do;
8   return M;
9 end proc;

```

We use this procedure to convert the relations we defined earlier, specifically **Div6** and **DivOrMul6** into matrices.

> *Div6M* := *RelToMatrix*(*Div6*)

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (9.32)$$

> *DivOrMul6M* := *RelToMatrix*(*DivOrMul6*)

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \quad (9.33)$$

Now that we have zero-one matrix representations of relations to work with, we can use these matrices to determine which properties apply to them. In this form, it is sometimes easier to determine whether a relation is reflexive, symmetric, or antisymmetric.

Checking Properties

For example, to determine whether or not a relation is reflexive from its zero-one matrix representation, we only need to check the diagonal entries. If any diagonal entry is 0, then the relation is not reflexive.

We begin the procedure by checking to make sure the matrix is square. For if not, there is no chance that the relation is reflexive, and, moreover, code later in the procedure will result in an error for nonsquare matrices. We use the **RowDimension** and **ColumnDimension** commands from the **LinearAlgebra** package to determine number of rows and columns of the matrix.

```

1 IsReflexiveM := proc (M::Matrix)
2   local i, numrows, numcols;
3   numrows := LinearAlgebra[RowDimension](M);
4   numcols := LinearAlgebra[ColumnDimension](M);
5   if numrows <> numcols then
6     return false;
7   end if;
8   for i from 1 to numrows do
9     if M[i,i] = 0 then
10       return false;
11     end if;
12   end do;
13   return true;
14 end proc;

```

We can now use this to test some of the relations above.

> *IsReflexiveM(exampleMatrix)*
false (9.34)

> *IsReflexiveM(Div6M)*
true (9.35)

Symmetry is particularly easy to test, because of the fact that a relation is symmetric if and only if its matrix representation is symmetric. Recall from Chapter 2 of the textbook that a matrix is symmetric when it is equal to its transpose. So, all we need to do is compute the transpose of the matrix, using the **Transpose** command in the **LinearAlgebra** package, and check to see if the matrices are equal. Checking equality of matrices requires using the **Equal** command.

```

1 IsSymmetricM := proc (M::Matrix)
2   return LinearAlgebra[Equal](M, LinearAlgebra[Transpose](M));
3 end proc;

```

> *IsSymmetricM(Div6M)*
false (9.36)

> *IsSymmetricM(DivOrMul6M)*
true (9.37)

Representing Relations Using Digraphs

Now, we turn to representing relations with directed graphs, commonly called digraphs. Maple provides a package called **GraphTheory** that contains the functions we will need to create and view graphs.

> *with(GraphTheory):*

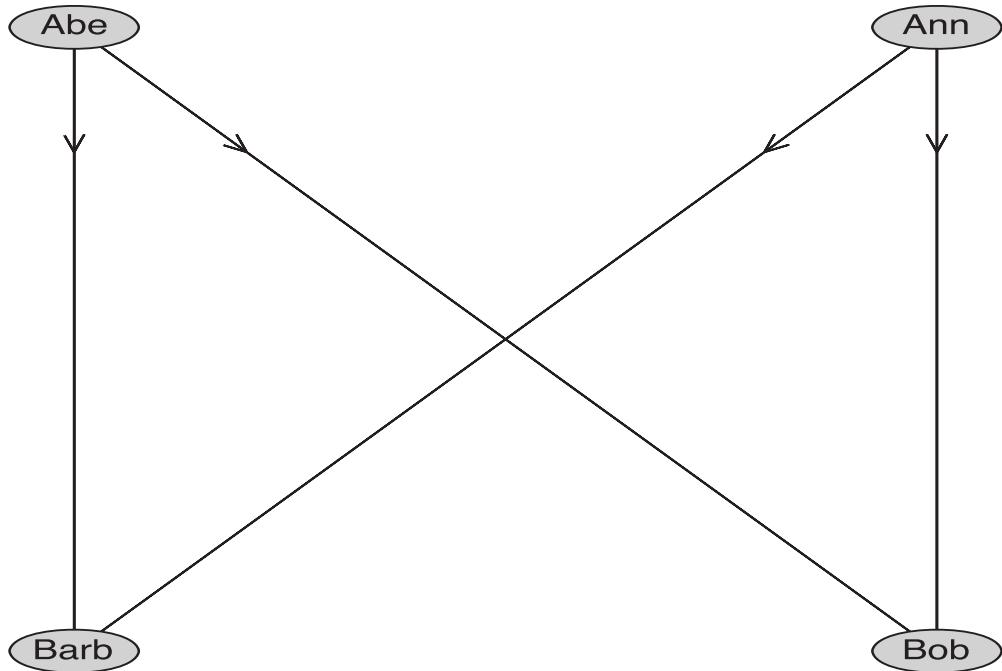
You create a digraph in Maple using the **Digraph** command. This command has a variety of possible calling sequences, including the ability to give the command only one argument: a set of

2-element lists representing the edges of the graph. For example, consider Bob and his sister Barb, whose parents are Ann and Abe. We can make a directed graph representing the relation “parent of” as follows.

```
> familyTree := Digraph({["Abe", "Barb"], ["Abe", "Bob"],
["Ann", "Barb"], ["Ann", "Bob"]})
familyTree := Graph 1: a directed unweighted graph with 4 vertices and 4 arc(s) (9.38)
```

The command **DrawNetwork** provides a visual representation of the graph.

```
> DrawNetwork(familyTree)
```



Note that we use the **DrawNetwork** command rather than Maple’s **DrawGraph** command. In this context, a network is a connected directed graph with at least one “source” and at least one “sink.” A “source” is an element with no edges into it and a “sink” is an element with no edges out of it. In the above, Abe and Ann are sources and Barb and Bob are sinks. In the language of posets, described in Section 9.6, sources and sinks are equivalent to minimal and maximal elements. If the digraph associated to a relation is a network, then **DrawNetwork** has a distinct advantage over **DrawGraph**, namely, **DrawNetwork** uses the source and sink to arrange the elements in a natural way producing a much more illustrative graph. If the digraph is not a network, then we can still use **DrawGraph**.

Many, but not all, of the relations we may wish to visualize are networks. Therefore, we will create a procedure, **DrawRelation**, that will check to see if the digraph associated to the given relation is a network and then use the appropriate draw command. In this procedure, we apply the **IsNetwork** test to the digraph associated to the relation. With this syntax, **IsNetwork** does not return a Boolean

value. Instead, it returns two sets, the first being the set of all sources and the second the set of all sinks. If either of these is empty, then the digraph is not a network.

Before creating the **DrawRelation** procedure, there are two more issues we must address. First, Maple does not allow for self-loops in a graph object. For example, if we tried to make Abe his own parent in the example above, Maple would raise an error.

```
> Digraph({["Abe","Abe"], ["Abe","Barb"], ["Abe","Bob"],
           ["Ann","Barb"], ["Ann","Bob"]})
```

Error, (in GraphTheory:-Graph) invalid edge/arc: ["Abe", "Abe"]

In other words, in order for a relation to be passed to the **Digraph** command, the relation must be *irreflexive* (i.e., no element can be related to itself). We can still use Maple to visualize relations as graphs, we just have to keep this limitation in mind. We write a procedure that removes any pair of the form **[a,a]** from the relation before constructing the graph.

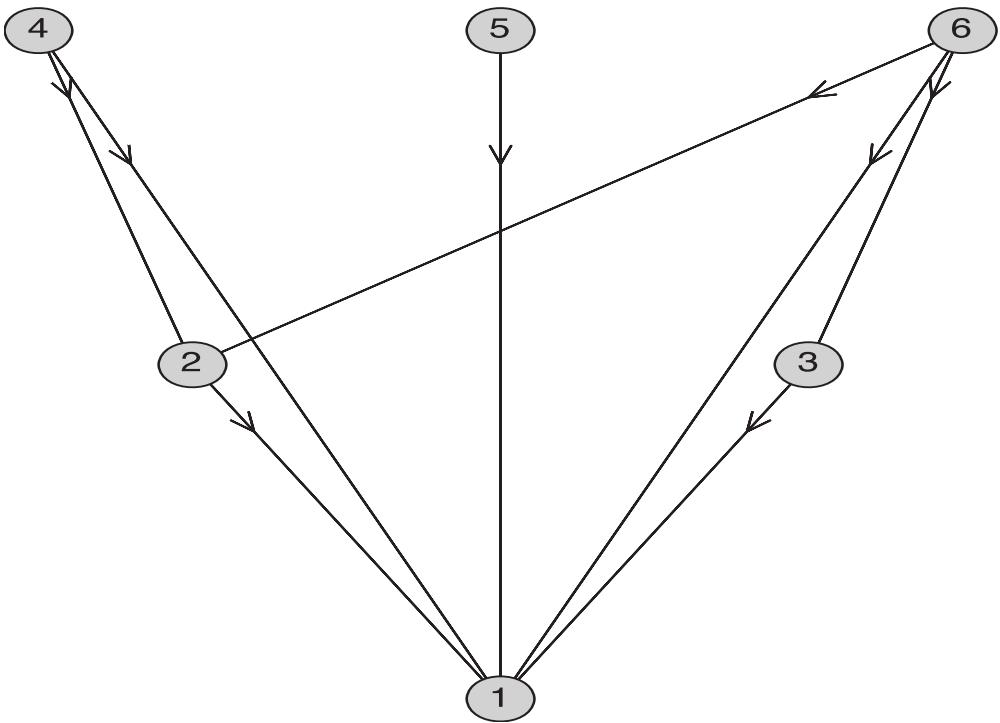
```
1 MakeIrreflexive := proc (R::rel)
2   local E, u;
3   E := {};
4   for u in R do
5     if u[1] <> u[2] then
6       E := E union {u};
7     end if;
8   end do;
9   return E;
10 end proc:
```

Finally, Maple draws networks with the sources at the top, which, in this context, yields graphs that are upside down in comparison to the usual way we draw them. Thus, we reverse the relation using our **InverseRelation** procedure. We are ready to define the **DrawRelation** procedure.

```
1 DrawRelation := proc (R::rel)
2   local IrrR, RevR, G, S;
3   IrrR := MakeIrreflexive(R);
4   RevR := InverseRelation(IrrR);
5   G := GraphTheory[Digraph](RevR);
6   S := GraphTheory[IsNetwork](G);
7   if S[1] = {} or S[2] = {} then
8     GraphTheory[DrawGraph](G);
9   else
10    GraphTheory[DrawNetwork](G);
11  end if;
12 end proc:
```

Now, we can convert our relations to graphs and visualize them.

```
> DrawRelation(Div6)
```



We can use this representation to determine whether or not the relation is transitive. To do this, we use Maple's implementation of the Floyd–Warshall all-pairs shortest path algorithm called **AllPairsDistance**. This procedure returns a matrix whose (i, j) entry represents the shortest path from vertex i to vertex j . For example, the all-pairs distance matrix for the **Div6** relation is:

> *AllPairsDistance (Digraph (MakeIrreflexive (Div6)))*

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ \infty & 0 & \infty & 1 & \infty & 1 \\ \infty & \infty & 0 & \infty & \infty & 1 \\ \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 \end{bmatrix} \quad (9.39)$$

In a graph of a transitive relation, the distance between any two distinct elements must be either 1 or infinite (meaning there is no path between them). To see this, assume that you have a transitive relation and suppose there are elements A and Z that the all-pairs algorithm has determined have a distance of 3. That means there must be two elements, say M and N , such that A is connected to M is connected to N is connected to Z . From the point of view of the relation, then, (A, M) and (M, N) and (N, Z) are all members of the relation. But if the relation is transitive, the fact that (A, M) and (M, N) are in the relation means that (A, N) is in the relation. So, A to N to Z is a shorter path (of length 2). Applying transitivity again shows that A and Z are adjacent. While this does not amount to a proof, it should be convincing that we can check for transitivity by making sure that no two vertices in the graph of a relation have distance which is finite and greater than 1.

```

1 IsTransitiveG := proc (R::rel)
2   uses GraphTheory;
3   local G, D, i, j;
4   G := Digraph(MakeIrreflexive(R));
5   D := AllPairsDistance(G);
6   for i from 1 to LinearAlgebra[RowDimension](D) do
7     for j from 1 to LinearAlgebra[ColumnDimension](D) do
8       if D[i, j] > 1 and D[i, j] < infinity then
9         return false;
10      end if;
11    end do;
12  end do;
13  return true;
14 end proc;

```

> *IsTransitiveG*(Div6)

true

(9.40)

> *IsTransitiveG*(R2)

false

(9.41)

9.4 Closures of Relations

In this section, we will develop algorithms to compute the reflexive, symmetric, and transitive closures of binary relations. We begin with the reflexive closure.

Reflexive Closure

The algorithm for computing the reflexive closure of a relation, with the matrix representation, is very simple. We simply set each diagonal entry equal to 1. The resulting matrix represents the reflexive closure of the relation.

There is one technical consideration to bear in mind: in order to avoid changing the matrix that we pass to these procedures, we begin by assigning a temporary variable, **ans**, to

LinearAlgebra[Copy](M). It is common in programming languages, and Maple is no exception, that variables assigned to high-level objects like matrices are stored as references. This means that if you have a matrix named **M**, and you enter **N := M**; you do not get two different matrices, you get two names for the same matrix. Moreover, changing one changes both. The **Copy** command forces Maple to make a distinct matrix so that the original is preserved.

Here is the procedure for computing the reflexive closure.

```

1 reflexiveClosure := proc (M::Matrix)
2   local i, ans;
3   ans := LinearAlgebra[Copy](M);
4   for i from 1 to LinearAlgebra[ColumnDimension](M) do
5     ans[i, i] := 1;
6   end do;
7   return ans;
8 end proc;

```

(Note that all the closure operations only apply to a relation on a set and are generally not valid for a relation from one set to a different set. This means we may assume that the matrix representation of the relation is square. If the matrix passed to this or the following procedures is not square, an error will likely occur.)

We use this procedure to find the reflexive closure of the example relation we introduced earlier in the chapter.

> *reflexiveClosure (exampleMatrix)*

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (9.42)$$

Recall that **exampleMatrix** represented the “is one less than” relation. Looking at the matrix above, you can see that the reflexive closure includes equality.

Symmetric Closure

Next, we write a procedure for constructing the symmetric closure of a relation R . We use the observation that if (a, b) is a member of R then (b, a) must be included in the symmetric closure, so we can simply add it to the relation.

```

1 symmetricClosure := proc (M::Matrix)
2   local i, j, ans;
3   ans := LinearAlgebra[Copy] (M);
4   for i from 1 to LinearAlgebra[RowDimension] (M) do
5     for j from 1 to LinearAlgebra[ColumnDimension] (M) do
6       if ans[i, j] = 1 then
7         ans[j, i] := 1;
8       end if;
9     end do;
10    end do;
11    return ans;
12  end proc;
```

Applying this to our **exampleMatrix** yields the “different by 1” relation. Applying it to the “is a divisor of” relation yields the “is a divisor or multiple of” relation.

> *symmetricClosure (exampleMatrix)*

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad (9.43)$$

```
> symmetricClosure(Div6M)
```

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \quad (9.44)$$

Transitive Closure

Having created the reflexive and symmetric closures, we turn to implementing the transitive closure in Maple. This is a more difficult problem than the earlier cases, both in terms of computational complexity and implementation. In the text, there are two algorithms outlined (a generic transitive closure and Warshall's algorithm) and both will be covered in this section.

A Transitive Closure Procedure

First, we will implement the transitive closure algorithm presented as Algorithm 1 in Section 9.4 of the main text. This will require the Boolean join and Boolean product operations on zero-one matrices that were introduced in Chapter 2. Recall from Section 2.6 of this manual that the **Bits** package includes Maple's functionality for bit-wise operations. In particular, remember that the **And** and **Or** functions correspond to the Boolean operations **and** and **or**. Here are some examples.

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

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

$$1 \quad (9.45)$$

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

$$0 \quad (9.46)$$

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

$$1 \quad (9.47)$$

```
> Or(1, 1)
```

$$1 \quad (9.48)$$

Now, we can recreate the Boolean join matrix operation. Recall that for zero-one matrices A and B of the same size, the join of A and B is the matrix $A \vee B$ whose (i, j) entry is $A_{ij} \vee B_{ij}$. In Section 2.6 of this manual, we provided one version of this procedure. Here, we will make greater use of the functionality provided by the **LinearAlgebra** package. In particular, the **Zip** command (note, this is different from **zip**) requires three arguments. The first argument is the name of a procedure on two arguments or a functional operator on two inputs. The other two arguments are two **Matrix** objects. The result is the matrix obtained by applying the first argument coordinate-wise. For example, we can multiply corresponding entries of two matrices as illustrated below.

```
> matrix1 := <1,2,3;4,5,6>
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad (9.49)$$

> *matrix2* := $\langle 2, 2, 2; 3, 3, 3 \rangle$

$$\begin{bmatrix} 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} \quad (9.50)$$

> *LinearAlgebra[Zip]* ((*a*, *b*) \mapsto *ba*, *matrix1*, *matrix2*)

$$\begin{bmatrix} 2 & 4 & 6 \\ 12 & 15 & 18 \end{bmatrix} \quad (9.51)$$

We use **Zip** with the **Or** function from the **Bits** package to create a Boolean join function.

```

1  boolJoin := proc (A::'Matrix'( {0,1}), B::'Matrix'( {0,1}) )
2      return LinearAlgebra[Zip](Bits[Or], A, B);
3  end proc;
```

For example,

> *joinA* := *Matrix*([[1, 0], [0, 1]])

$$joinA := \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (9.52)$$

> *joinB* := *Matrix*([[1, 1], [0, 0]])

$$joinB := \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \quad (9.53)$$

> *boolJoin*(*joinA*, *joinB*)

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad (9.54)$$

Next, recall that for appropriately sized zero-one matrices, the Boolean product $A \odot B$ is the matrix whose (i, j) entry is obtained by the formula

$$\bigvee_{k=1}^n (a_{ik} \wedge b_{kj})$$

where n is the number of columns of A , which is also the number of rows of B . The **BoolProduct** procedure, written in Section 2.6, illustrated how to apply this formula directly.

```

1  BoolProduct := proc (A::'Matrix'( {0,1}), B::'Matrix'( {0,1}) )
2      local m, k, n, C, i, j, c, p;
3      uses LinearAlgebra, Bits;
4      m := RowDimension(A);
5      k := ColumnDimension(A);
6      if k <> RowDimension(B) then
7          error "Dimension mismatch.";
8      end if;
```

```

9   n := ColumnDimension(B);
10  C := Matrix(m, n);
11  for i from 1 to m do
12    for j from 1 to n do
13      c := And(A[i, 1], B[1, j]);
14      for p from 2 to k do
15          c := Or(c, And(A[i, p], B[p, j]));
16      end do;
17      C[i, j] := c;
18    end do;
19  end do;
20  return C;
21 end proc;

```

As an example,

> *productA* := *Matrix*([[1, 0], [0, 1], [1, 0]])

$$\text{productA} := \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \quad (9.55)$$

> *productB* := *Matrix*([[1, 1, 0], [0, 1, 1]])

$$\text{productB} := \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \quad (9.56)$$

> *BoolProduct*(*productA*, *productB*)

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} \quad (9.57)$$

We are now ready to implement Algorithm 1 from Section 9.4 for calculating the transitive closure. Recall that the idea of this algorithm is that we compute Boolean powers of the matrix of the relation, up to the size of the domain. At each step, we use the Boolean join on $A = M^{[i]}$ and the result matrix B .

```

1 transitiveClosure := proc (M::Matrix({0, 1}))
2   local A, B;
3   uses LinearAlgebra, Bits;
4   A := Copy(M);
5   B := Copy(M);
6   from 2 to RowDimension(M) do
7     A := BoolProduct(A, M);
8     B := boolJoin(B, A);
9   end do;
10  return B;
11 end proc;

```

We test our transitive closure procedure on Example 7 from Section 9.4, where it was found that the relation with matrix representation

$$M_R = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

has transitive closure

$$M_{R^*} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

> *example7 := Matrix([[1, 0, 1], [0, 1, 0], [1, 1, 0]])*

$$\text{example7 := } \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \quad (9.58)$$

> *transitiveClosure(example7)*

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad (9.59)$$

Warshall's Algorithm

Next, we consider Warshall's algorithm, as presented as Algorithm 2 in Section 9.4. This algorithm is straightforward to implement.

```

1 Warshall := proc (M::Matrix)
2   local i, j, k, W, n;
3   uses LinearAlgebra, Bits;
4   n := ColumnDimension (M);
5   W := Copy (M);
6   for k from 1 to n do
7     for i from 1 to n do
8       for j from 1 to n do
9         W[i, j] := Or(W[i, j], And(W[i, k], W[k, j]));
10        end do;
11      end do;
12    end do;
13    return W;
14  end proc;
```

Applying this to the same example as before, we see that the result is correct.

> *Warshall(example7)*

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

(9.60)

We can compare these two procedures in terms of execution time using Maple's **time** command. But, we must point out that this comparison for a single example does not prove anything about the complexity or relative performance of the two algorithms. Rather, it serves as a demonstration that, even for relations on small domains, the difference in the computational complexity of the algorithms is noticeable. We shall consider the following zero-one matrix that represents a relation on the set $\{1, 2, 3, 4, 5, 6\}$.

> *transitiveCompare := Matrix([[0, 0, 0, 0, 0, 1], [1, 0, 1, 0, 0, 0], [1, 0, 0, 1, 0, 0], [1, 0, 0, 0, 1, 0], [1, 0, 0, 0, 0, 1], [0, 1, 0, 0, 0, 0]])*

$$\text{transitiveCompare} := \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(9.61)

> *st := time()* :
Warshall(transitiveCompare) :
time () - st
0.019

(9.62)

> *st := time()* :
transitiveClosure(transitiveCompare) :
time () - st
0.021

(9.63)

From this example, we can see that Warshall's algorithm can be an improvement over the alternative, at least on this specific example. The reader is encouraged to explore this further.

9.5 Equivalence Relations

In this section, we will examine how we can use Maple to compute with equivalence relations. There are three specific problems that we will address here: given an equivalence relation on a set, how to compute the equivalence class of an element; how to determine the number of equivalence relations on a finite set; and how to compute the smallest equivalence relation that contains a given relation on some finite set. Note that in this section, relations are assumed to be represented as in the start of this chapter, as objects of type **rel**.

First, we provide a test that determines whether or not a relation is an equivalence relation. Using the work that we have already done and recalling that an equivalence relation is simply a relation that is reflexive, symmetric, and transitive, this task is a simple one.

```

1  IsEquivalenceRelation := proc (R::rel);
2      evalb(IsReflexive(R) and IsSymmetric(R) and IsTransitive(R));
3  end proc;
```

As an example, we define the equivalence relation “congruent mod 4” on the integers from 0 to n .

```

> makeMod4Rel(8)
{[0, 0], [0, 4], [0, 8], [1, 1], [1, 5], [2, 2], [2, 6], [3, 3], [3, 7], [4, 0], [4, 4], [4, 8],
 [5, 1], [5, 5], [6, 2], [6, 6], [7, 3], [7, 7], [8, 0], [8, 4], [8, 8]} (9.64)

> IsEquivalenceRelation (9.64)
true (9.65)
```

Equivalence Classes

Recall that, given an equivalence relation R and a member a of the domain of R , the equivalence class of a is the set of all members b of the domain for which the pair (a, b) belongs to R . In other words, it is the set of all elements in the domain that are R -equivalent to a . To determine the equivalence class of a particular element of the domain, the algorithm is fairly simple. We just search through R looking for all pairs of the form (a, b) , adding each such second element b to the class. We do not have to search for pairs of the form (b, a) because equivalence relations are symmetric. The following procedure returns the equivalence class for a given equivalence relation and a point in the domain.

```

1  EquivalenceClass := proc (R::rel, a::anything)
2      local u, S;
3      S := {};
4      for u in R do
5          if u[1] = a then
6              S := S union {u[2]};
7          end if;
8      end do;
9      return S;
10 end proc;
```

As an example, we compute the equivalence class of 3 in the modulo 4 relation on the domain $\{1, 2, \dots, 30\}$.

```

> EquivalenceClass (makeMod4Rel(30), 3)
{3, 7, 11, 15, 19, 23, 27} (9.66)
```

Number of Equivalence Relations on a Set

Next, we consider how to construct all of the equivalence relations on a given (finite) set. The straightforward way to do this is to construct all relations on the given domain and then check them to see if they are equivalence relations. Since a relation on a set A is merely a subset of $A \times A$, generating all relations is the same as generating all subsets of $A \times A$.

To implement this, we begin by creating the set $A \times A$ using Maple's **cartprod** function. The **cartprod** function takes a list of sets and returns a table with two entries: **finished** and **nextvalue**. The **finished** entry is a Boolean variable indicating whether the last element of the product has been reached. The **nextvalue** entry is a function which returns the next element of the Cartesian product. This may seem a strange approach, but it allows for the conceptual creation of the Cartesian product of two sets without the need to actually store the entire product. In this case, we do want to create and store the entire product, so we will use the following structure.

```

> Cprod := combinat[cartprod]([{1,2,3}, {8,9}]) :
> CprodSet := {} :
  while not Cprod[finished] do
    CprodSet := CprodSet union {Cprod[nextvalue]()}
  end do :
CprodSet
{[1,8], [1,9], [2,8], [2,9], [3,8], [3,9]}(9.67)
```

To complete our equivalence relation procedure, we use the **powerset** command from the **combinat** package. Recall that the powerset of a set is the set of all subsets. Hence, the powerset of $A \times A$ is the set of all relations on A . We simply check them one by one to see which are equivalence relations.

```

1 AllEquivalenceRelations := proc (A::set)
2   local C, AA, P, R, E;
3   C := combinat[cartprod] ([A, A]) ;
4   AA := {} ;
5   while not C[finished] do
6     AA := AA union {C[nextvalue]()} ;
7   end do ;
8   P := combinat[powerset](AA) ;
9   E := {} ;
10  for R in P do
11    if IsEquivalenceRelation(R) then
12      E := E union {R} ;
13    end if ;
14  end do ;
15  return E ;
16 end proc:(9.68)
```

For example, there are 15 equivalence relations on $\{1, 2, 3\}$.

```

> AllEquivalenceRelations({1,2,3})
{[], {[1,1]}, {[2,2]}, {[3,3]}, {[1,1],[2,2]}, {[1,1],[3,3]}, {[2,2],[3,3]}, {[1,1],[2,2],[3,3]}, {[1,1],[1,2],[2,1],[2,2]}, {[1,1],[1,3],[3,1],[3,3]}, {[2,2],[2,3],[3,2],[3,3]}, {[1,1],[1,2],[2,1],[2,2],[3,3]}, {[1,1],[1,3],[2,2],[3,1],[3,3]}, {[1,1],[2,2],[2,3],[3,2],[3,3]}, {[1,1],[1,2],[1,3],[2,1],[2,2],[2,3],[3,1],[3,2],[3,3]}}(9.68)
```

```

> nops(9.68)
15(9.69)
```

Closure

The last question to be considered in this section is the problem of finding the smallest equivalence relation containing a relation R .

The key idea is that we need to find the smallest relation containing R that is reflexive, symmetric, and transitive. Recalling the previous section on closures, it is natural to think that we may compute the reflexive closure, the symmetric closure, and then the transitive closure, one after the other. The only concern would be that one closure would no longer have one of the previous properties. The following outlines why this is not the case.

1. First, create the reflexive closure of R , call it P .
2. Compute the symmetric closure of P and call this Q . Note that Q is still reflexive since no pairs were removed from the relation and no elements were added to the domain. So Q is both symmetric and reflexive.
3. Compute the transitive closure of Q and name this S . Note that S is still reflexive for the same reason as above. Also note that S is still symmetric since, if (a, b) and (b, c) are in Q to force the addition of (a, c) , then since Q is symmetric, (c, b) and (b, a) must also be in Q forcing (c, a) to also be included in S . Hence, S is an equivalence relation.

We implement this method as the composition of the four methods **RelToMatrix**, **reflexiveClosure**, **symmetricClosure**, and then **transitiveClosure**. We use Maple's composition operator, $@$.

```
| equivalenceClosure := transitiveClosure @ symmetricClosure @  
|   reflexiveClosure @ RelToMatrix:
```

As an example, recall the **Div6** relation representing the “is a divisor of” on $\{1, 2, 3, 4, 5, 6\}$. We can see that the smallest equivalence relation that contains **Div6** is the relation in which every number is related to every other number.

```
> equivalenceClosure(Div6)
```

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \quad (9.70)$$

This is unsurprising, since 1 is a divisor of every number meaning that, in any equivalence relation containing the “divides” relation, 1 is related to every number. We can make this example slightly more interesting by removing 1.

```
> Div17minus1 := DividesRelation({$2..17})
```

$$\begin{aligned} \text{Div17minus1} := & \{[2, 2], [2, 4], [2, 6], [2, 8], [2, 10], [2, 12], [2, 14], [2, 16], \\ & [3, 3], [3, 6], [3, 9], [3, 12], [3, 15], [4, 4], [4, 8], [4, 12], [4, 16], [5, 5], [5, 10], \\ & [5, 15], [6, 6], [6, 12], [7, 7], [7, 14], [8, 8], [8, 16], [9, 9], [10, 10], [11, 11], \\ & [12, 12], [13, 13], [14, 14], [15, 15], [16, 16], [17, 17]\} \end{aligned} \quad (9.71)$$

```
> interface(rtablesize = 17):
equivalenceClosure(Div17minus1)


$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (9.72)$$

```

(Note the first row and column still correspond to 1 because of the way the matrix is constructed in **RelToMatrix**.) In this example, you see that 11, 13, and 17 become isolated, being the three primes in the set which do not have multiples of themselves included.

9.6 Partial Orderings

In this section, we consider partial orderings (or partial orders) and related topics, including maximal and minimal elements, Hasse diagrams, and lattices.

Partial Orders and Examples

First, we will define a new Maple type for partial orders and create some examples of them.

Recall that a partial order is a binary relation on a set that satisfies the three conditions of being reflexive, antisymmetric, and transitive. Earlier in this section, we defined various types (e.g., **rel**, our relation type) via the structured type syntax. In this case, though, it is not enough to know the structure of the type's data, we also want to insist on certain properties. We define the type to be a procedure that tests an object against the definition of a partial order. It is very similar to the **IsEquivalenceRelation** procedure we created in the previous section.

Here is the definition of the partial order type.

```

1  'type/po' := proc(obj)
2      type(obj, rel) and IsReflexive(obj) and IsAntisymmetric(obj) and
        IsTransitive(obj);
3  end proc;

```

Unlike the structured type syntax, we explicitly define a procedure. The main rules for defining types in this way are that the procedure must take one argument and it must return true or false.

We can now use the name **po** as the second argument to the **type** command for checking to see if an object is of the type. For example, we can check that the **Div6** relation we defined earlier (recall that this is the “divides” relation on the set 1 through 6) is a partial order.

```

> type(Div6,po)
true

```

(9.73)

We create some additional examples of partial orderings that we can use in the remainder of the section. The **Div17minus1** relation (this was the “divides” relation on the set 2 through 17) is a partial order.

```

> type(Div17minus1,po)
true

```

(9.74)

Indeed, all the relations created via the **DividesRelation** or **DivRel** procedures will be partial orders.

Next, we create a procedure to produce examples of a class of lattices (we will discuss lattices more below, for now it is enough that these examples are partial orders). The **DivisorLattice** procedure will create the partial order whose domain is the set of positive divisors of a given number and whose order is defined by the “divides” relation. We only need to apply the **DividesRelation** procedure to the divisors of the given number.

```

1  DivisorLattice := proc(n::posint)
2      DividesRelation(NumberTheory[Divisors](n));
3  end proc;

```

The **Divisors** command in the **NumberTheory** package produces all of the positive divisors of the given integer.

```

> DivisorLattice(10)
{[1, 1], [1, 2], [1, 5], [1, 10], [2, 2], [2, 10], [5, 5], [5, 10], [10, 10]}

```

(9.75)

Finally, for a bit of variety, we create the posets whose Hasse diagrams are shown in Figures 8(a) and 10 in Section 9.6.

```

> Fig8A := {[["a", "a"], ["a", "b"], ["a", "c"], ["a", "d"], ["a", "e"],
    ["a", "f"], ["b", "b"], ["b", "c"], ["b", "d"], ["b", "e"], ["b", "f"],
    ["c", "c"], ["c", "e"], ["c", "f"], ["d", "d"], ["d", "e"], ["d", "f"],
    ["e", "e"], ["e", "f"], ["f", "f"]]}

```

```

Fig8A := {[["a", "a"], ["a", "b"], ["a", "c"], ["a", "d"], ["a", "e"],
           ["a", "f"], ["b", "b"], ["b", "c"], ["b", "d"], ["b", "e"], ["b", "f"],
           ["c", "c"], ["c", "e"], ["c", "f"], ["d", "d"], ["d", "e"], ["d", "f"],
           ["e", "e"], ["e", "f"], ["f", "f"]}] (9.76)

```

```

> type(Fig8A, po)
true (9.77)

```

```

> Fig10 := {[["A", "A"], ["A", "B"], ["A", "D"], ["A", "F"], ["A", "G"],
             ["B", "B"], ["B", "D"], ["B", "F"], ["B", "G"], ["C", "B"], ["C", "C"],
             ["C", "D"], ["C", "F"], ["C", "G"], ["D", "D"], ["D", "G"], ["E", "E"],
             ["E", "F"], ["E", "G"], ["F", "F"], ["F", "G"], ["G", "G"]}

Fig10 := {[["A", "A"], ["A", "B"], ["A", "D"], ["A", "F"], ["A", "G"],
           ["B", "B"], ["B", "D"], ["B", "F"], ["B", "G"], ["C", "B"], ["C", "C"],
           ["C", "D"], ["C", "F"], ["C", "G"], ["D", "D"], ["D", "G"], ["E", "E"],
           ["E", "F"], ["E", "G"], ["F", "F"], ["F", "G"], ["G", "G"]}] (9.78)

```

```

> type(Fig10, po)
true (9.79)

```

Hasse Diagrams

Now that we have defined a type and have examples at our disposal, we turn to the problem of having Maple draw Hasse diagrams of partial orders. As demonstrated in the textbook, a Hasse diagram is a very useful tool for visualizing and understanding posets. Drawing the Hasse diagram for a poset is not as simple as drawing all of the elements of the set and then connecting all related pairs with an edge. Doing so would create an extremely messy, and not very useful, diagram. Instead, a Hasse diagram contains only those edges that are absolutely necessary to reveal the structure of the poset.

Covering Relations

The covering relation for a partial order is a minimal representation of the partial order, from which the partial order can be reconstructed via transitive and reflexive closure.

Let \leq be a partial order on a set S . Recall that an element y in S *covers* an element x in S if $x < y$, $x \neq y$, and there is no element z of S , different from x and y , such that $x < z < y$. In other words, y covers x if y is greater than x and there is no intermediary element. The set of pairs (x, y) for which y covers x is the covering relation of \leq .

As a simple example, consider the set $\{1, 2, 3, 4\}$ ordered by magnitude, that is, the usual “less than or equal to.” This relation consists of 10 ordered pairs:

$$\{(1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4), (3, 3), (3, 4), (4, 4)\}.$$

Its covering relation is the set

$$\{(1, 2), (2, 3), (3, 4)\},$$

which consists of only 3 pairs. All the other pairs of the partial order can be inferred from the covering relation using transitivity and reflexivity. For instance, $(1, 3)$ can be recovered from $(1, 2)$ and

(2, 3) via transitivity. Note that the covering relation involves many fewer pairs and thus is a much more efficient way to represent the partial order, at least in terms of storage.

Our goal in this subsection is to write a procedure that will have Maple draw the Hasse diagram of a given partial order. Since a Hasse diagram is, in fact, the graph of the associated covering relation, we will create a procedure to find the covering relation of the partial order.

First, we need a test to check whether a given element covers another.

```

1 Covers := proc (R::po, x, y)
2   local z;
3   if x = y then
4     return false;
5   end if;
6   if not [x,y] in R then
7     return false;
8   end if;
9   for z in FindDomain(R) minus {x,y} do
10    if ([x,z] in R) and ([z,y] in R) then
11      return false;
12    end if;
13  end do;
14  return true;
15 end proc:
```

This procedure works by first checking to see if the two elements x and y are equal to each other or if the pair (x, y) fails to be in the partial order. In either of these situations, y does not cover x . Assuming the pair of elements passes these basic hurdles, the procedure then checks every other element of the domain. If it can find an element that sits between x and y , then we know they do not cover. If no element sits between them, then in fact y does cover x .

Now, we can construct the covering relation of a partial order using the following Maple procedure. This procedure simply checks every element of the given relation to see if one covers the other and adds those that do to the output relation C .

```

1 CoveringRelation := proc (R::po)
2   local C, u;
3   C := {};
4   for u in R do
5     if Covers (R, u[1], u[2]) then
6       C := C union { [u[1], u[2]] };
7     end if;
8   end do;
9   return C;
10 end proc:
```

Let us look at a couple of examples. First, the example described above, of the set $\{1, 2, 3, 4\}$ ordered by magnitude.

```
> CoveringRelation ({[1,1],[1,2],[1,3],[1,4],[2,2],[2,3],[2,4],[3,3],
[3,4],[4,4]})  

{[1,2],[2,3],[3,4]} (9.80)
```

As a second example, we consider a lattice.

```
> CoveringRelation (DivisorLattice (30))  

{[1,2],[1,3],[1,5],[2,6],[2,10],[3,6],[3,15],[5,10],[5,15],[6,30],
[10,30],[15,30]} (9.81)
```

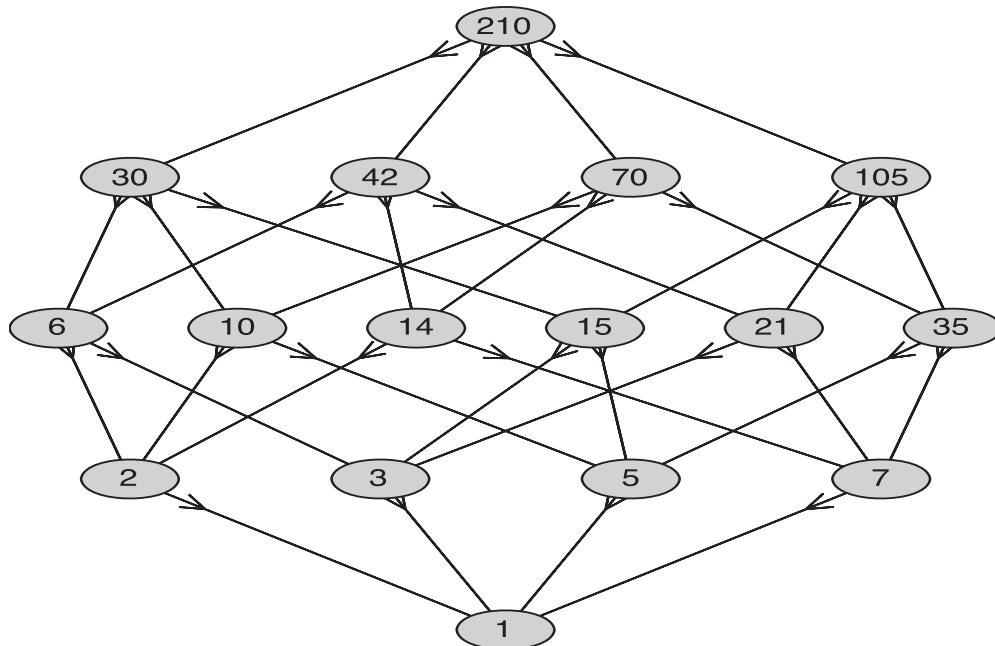
Drawing Hasse Diagrams

Now, we will use the covering relation in order to write a program to draw the Hasse diagram for partial orders. By using the **CoveringRelation** procedure that we just completed and the **Digraph** and **DrawNetwork** commands from the **GraphTheory** package, we can create the graph associated to a partial order. As we did previously in this chapter when graphing relations, we reverse the relation in order to have the smallest elements at the bottom, as is typical.

```
1 HasseDiagram := proc (R::po)
2   local C, D;
3   C := InverseRelation (CoveringRelation (R));
4   D := GraphTheory [Digraph] (C);
5   GraphTheory [DrawNetwork] (D);
6 end proc;
```

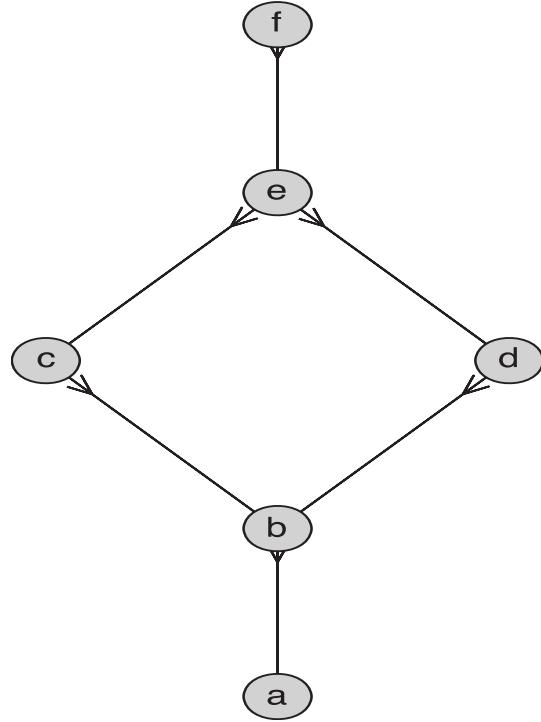
As an example, here is a diagram representing the divisor lattice of 210.

```
> HasseDiagram (DivisorLattice (210), 1, 210)
```

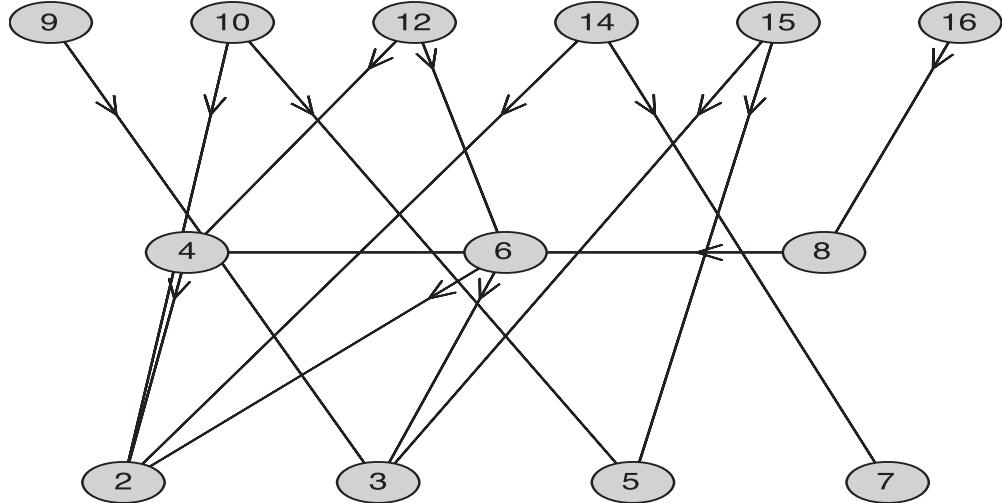


And here are the Hasse diagrams for some of the other examples we discussed in this section.

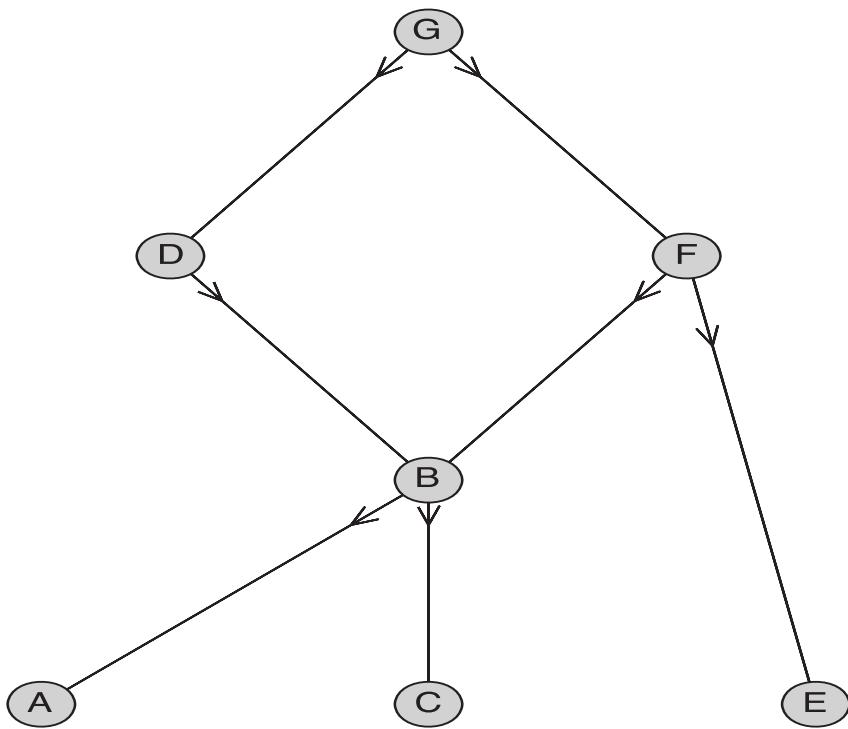
> *HasseDiagram (Fig8A)*



> *HasseDiagram (Div17minus1)*



> *HasseDiagram (Fig10)*



Comparing this last example to the diagram given in the textbook illustrates that, while using Maple's **DrawNetwork** command does not result in quite as appealing graphs as those that are created by hand, it still provides a fairly useful graph.

Maximal and Minimal Elements

We will construct a procedure that determines the set of minimal elements of a partially ordered set.

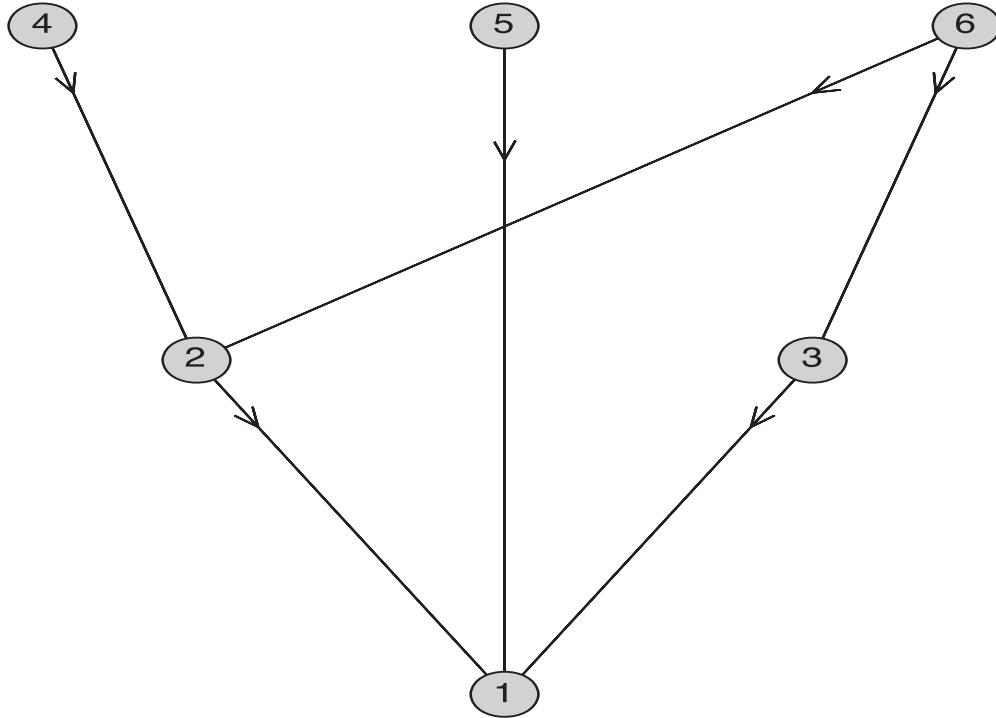
The procedure takes two arguments: a partial order R and a subset S of the domain of R . It returns the set of minimal elements of S with respect to R . It first initializes the set of minimal elements to all of S and then removes those that are not minimal.

```

1 MinimalElements := proc (R::po, S::set)
2   local M, s, t;
3   if S minus FindDomain(R) <> {} then
4     error "Set must be in the domain of the relation ";
5   end if ;
6   M := S;
7   for s in S do
8     for t in S minus {s} do
9       if [t,s] in R then
10         M := M minus {s};
11       end if ;
12     end do;
13   end do;
14   return M;
15 end proc:
```

We can see this work on our **Div6** partial order. Since we will be using the **Div6** partial order for many examples in this section, its Hasse diagram may also be useful.

> *HasseDiagram(Div6)*



> *MinimalElements(Div6, {1, 2, 3, 4, 5, 6})*

{1}

(9.82)

> *MinimalElements(Div6, {2, 3, 4, 5, 6})*

{2, 3, 5}

(9.83)

Note that, by reversing the relation and thus the order, we can compute maximal elements very easily.

1	MaximalElements := proc (R::po, S::set)
2	MinimalElements (InverseRelation(R), S);
3	end proc:

> *MaximalElements(Div6, {1, 2, 3, 4, 5, 6})*

{4, 5, 6}

(9.84)

Least Upper Bound

Next, we will write a procedure for computing the least upper bound of a set with respect to a partial order, if it exists. Our procedure will return the value **NULL** in the case that the set has no least upper bound.

First, we create a procedure **IsUpperBound** that determines whether a given element is an upper bound of a set with respect to a relation. It accomplishes this by checking to make sure that the given element is greater than every element of the set.

```

1  IsUpperBound := proc (R::po, S::set, u::anything)
2    local s;
3    if not u in FindDomain(R) then
4      error "Element is not in the domain of the relation ."
5    end if;
6    for s in S do
7      if not [s,u] in R then
8        return false;
9      end if;
10    end do;
11    return true;
12  end proc;
```

For example, under the **Div6** relation, 6 is an upper bound of $\{1, 2, 3\}$, but not of $\{1, 2, 3, 4\}$.

> *IsUpperBound(Div6, {1, 2, 3}, 6)*
true (9.85)

> *IsUpperBound(Div6, {1, 2, 3, 4}, 6)*
false (9.86)

Next, we write a procedure to find all of the upper bounds for a given set. We do this by considering every element of the domain of the relation and checking to see which are upper bounds, using the **IsUpperBound** procedure.

```

1  UpperBounds := proc (R::po, S::set)
2    local U, DomR, d;
3    DomR := FindDomain(R);
4    if S minus DomR <> {} then
5      error "set must be contained in the domain of the relation ."
6    end if;
7    U := {};
8    for d in DomR do
9      if IsUpperBound(R, S, d) then
10        U := U union {d};
11      end if;
12    end do;
13    return U;
14  end proc;
```

For instance, the upper bounds of the set $\{1, 2\}$ under **Div6** are:

> *UpperBounds(Div6, {1, 2})*
 $\{2, 4, 6\}$ (9.87)

To complete the task of finding the least upper bound of a set, we merely use **UpperBounds** to compute all of the upper bounds for the set, use **MinimalElements** to see which of the upper bounds are minimal, and then check to see how many minimal upper bounds are found. If there is exactly one minimal upper bound, then this is the least upper bound. Otherwise, the set has no least upper bound.

```

1 LeastUpperBound := proc (R::po, S::set)
2   local U, M;
3   U := UpperBounds (R, S);
4   M := MinimalElements (R, U);
5   if nops (M) <> 1 then
6     return NULL;
7   else
8     return op (M);
9   end if;
10 end proc;
```

For example, the least upper bounds of $\{1, 2\}$ and $\{1, 2, 3\}$ are found below, while $\{4, 5\}$ has no least upper bound in the domain of **Div6** and so does not return a value.

> *LeastUpperBound(Div6, {1, 2})*
2 (9.88)

> *LeastUpperBound(Div6, {1, 2, 3})*
6 (9.89)

> *LeastUpperBound(Div6, {4, 5})*

Lattices

As the last topic in this section, we will consider the problem of determining whether a partial order is a lattice. The approach we will take is a good example of “top down programming.” The test we design here will confirm that the procedure **DivisorLattice** written at the beginning of this section does indeed produce lattices.

Recall that a partial order is a lattice if every pair of elements has both a least upper bound and a greatest lower bound (in lattices, these are also referred to as the supremum and infimum of the pair or as their meet and join). With this in mind, we can write the following procedure (with the understanding that the helper procedures still need to be written).

```

1 IsLattice := proc (R::po)
2   HasLUBs (R) and HasGLBs (R);
3 end proc;
```

We need to write the two helper relations: **HasLUBs** to determine if the partial order satisfies the property that every pair of elements has a least upper bound, and **HasGLBs** to determine if every pair has a greatest lower bound. Just as we did above with the **MaximalElements** procedure, we really only need to write one procedure if we recognize that a partial order satisfies the greatest lower bound property if the inverse relation satisfies the least upper bound property. Thus, we compose **HasLUBs** with the **InverseRelation** procedure to create **HasGLBs**.

```

1 HasGLBs := HasLUBs @ InverseRelation;

```

Now, we complete the work by coding the **HasLUBs** procedure. We must test whether, for a given relation R , each pair a and b in the domain of R has a least upper bound with respect to R .

```

1 HasLUBs := proc (R::po)
2   local DomR, a, b;
3   DomR := FindDomain(R);
4   for a in DomR do
5     for b in DomR do
6       if LeastUpperBound(R, {a, b}) = NULL then
7         return false;
8       end if;
9     end do;
10    end do;
11    return true;
12  end proc;

```

Finally, all of the subroutines that go into making up the **IsLattice** program are complete, and we can test it on some examples. Contrast the relations constructed by the **DivRel** procedure versus those made by **DivisorLattice**.

```

> IsLattice(DivRel(20))
false
(9.90)

```

```

> IsLattice(DivisorLattice(20))
true
(9.91)

```

Solutions to Computer Projects and Computations and Explorations Computer Projects 15

Given a partial ordering on a finite set, find a total ordering compatible with it using topological sorting.

Solution: The textbook contains a detailed explanation of topological sorting and summarizes it as Algorithm 1 of Section 9.6.

The set S is initialized to the domain of the given relation. At each step, find a minimal element (using the **MinimalElements** procedure we created above) of S . This minimal element is removed from S and added as the next largest element of the total ordering. This repeats until S is empty and consequently all elements are in the total order.

```

1 TopSort := proc (R::po)
2   local S, a, T;
3   T := [];
4   S := FindDomain(R);
5   while S <> {} do
6     a := MinimalElements(R, S)[1];

```

```

7      S := S minus {a} ;
8      T := [op(T), a];
9  end do;
10 return T;
11 end proc:
```

We apply this procedure to **Fig10**.

> *TopSort(Fig10)*
 [“A”, “C”, “B”, “D”, “E”, “F”, “G”] (9.92)

Computations and Explorations 1

Display all the different relations on a set with four elements.

Solution: As usual, Maple is much too powerful to solve only the single instance of the general problem suggested by this question. We provide a very simple procedure that will compute all relations on any finite set. This procedure merely constructs the Cartesian product $C = S \times S$ and then makes use of the **powerset** command to obtain all of the relations on the set.

```

1 AllRelations := proc (S :: set)
2   local s, t, C;
3   C := {};
4   for s in S do
5     for t in S do
6       C := C union {[s, t]};
7     end do;
8   end do;
9   return combinat[powerset](C);
10 end proc:
```

We now test our procedure on a set with two elements. (This keeps the output to a reasonable length.)

> *AllRelations({1, 2})*
{ \emptyset , {[1, 1]}, {[1, 2]}, {[2, 1]}, {[2, 2]}, {[1, 1], [1, 2]}, {[1, 1], [2, 1]}, {[1, 1], [2, 2]}, {[1, 2], [2, 1]}, {[1, 2], [2, 2]}, {[2, 1], [2, 2]}, {[1, 1], [1, 2], [2, 1]}, {[1, 1], [1, 2], [2, 2]}, {[1, 1], [2, 1], [2, 2]}, {[1, 2], [2, 1], [2, 2]}, {[1, 1], [1, 2], [2, 1], [2, 2]}} (9.93)

The reader is encouraged to determine the running time and output length for the procedure when the input set has cardinality 4 or 5. Keep in mind that there are 2^{n^2} relations on a set with n members.

Computations and Explorations 4

Determine how many transitive relations there are on a set with n elements for all positive integers n with $n \leq 7$.

Solution: We will construct each possible $n \times n$ zero-one matrix using an algorithm similar to binary counting. The approach is as follows:

1. For each number from 0 to $2^n - 1$, we create a list of 0s and 1s that is the base 2 representation of that integer. We can do this with the **convert** command. The syntax **convert(i,base,2)** returns a list whose entries are the base 2 representation of the integer, with the first entry being the 1s place, the second entry the 2s place, the third entry the 4s place, etc.
2. Then create a matrix M whose elements are that list of values. These are all possible 2^n zero-one matrices (the reader is encouraged to prove this statement). The **Matrix** command with the syntax **Matrix(r,c,L)** produces a matrix of dimension $r \times c$ whose entries are specified by the values in the list L .
3. Finally, evaluate the transitive closure of each of those matrices, using the **Warshall** procedure from Section 9.4. We test to see if the matrix is transitive by checking to see if it is equal to its transitive closure. If so, it is counted as a transitive relation.

The implementation is as follows:

```

1 CountTransitive := proc (n::posint)
2   local i, T, M, count;
3   count := 0;
4   for i from 0 to (2^(n^2) - 1) do
5     T := convert(i,base,2);
6     M := Matrix(n,n,T);
7     if LinearAlgebra[Equal](M,Warshall(M)) then
8       count := count + 1;
9     end if;
10    end do;
11    return count;
12  end proc;
```

We use our procedure on a relatively small value and leave further computations to the reader.

> *CountTransitive(3)*

171

(9.94)

Computations and Explorations 5

Find the transitive closure of a relation of your choice on a set with at least 20 elements. Either use a relation that corresponds to direct links in a particular transportation or communications network or use a randomly generated relation.

Solution: We will generate a random zero-one matrix with dimension 100, and then apply Warshall's algorithm to compute the transitive closure.

To generate a random zero-one matrix, we use the **RandomMatrix** command from the **LinearAlgebra** package. We will use the form of the command with four arguments. The first two arguments are the row and column dimensions of the matrix. The third argument will be the equation **generator = 0..1**. This indicates that the generated entries should be random integers from the range **0..1** (i.e., they will be 0s and 1s). The final argument is the equation **density = .25**. This tells Maple to only fill about 25% of the entries in the matrix. The other entries are left 0. (Note that those entries that are randomly selected to be filled are filled with 1 or 0 with equal likelihood.)

This produces a fairly sparse matrix and increases the chance that the transitive closure will have entries that are not 1.

> *LinearAlgebra[RandomMatrix](10, 10, generator = 0..1, density = 0.25)*

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad (9.95)$$

> *Warshall(9.95)*

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \end{bmatrix} \quad (9.96)$$

Exercises

Exercise 1. The **RelToMatrix** procedure converts a relation of type **rel** to a zero-one matrix representation while the **DrawRelation** procedure displays a digraph representation of a **rel**. Write procedures to (a) convert a zero-one matrix representation of a relation to a **rel** and to (b) display a digraph representation of a matrix representation.

Exercise 2. Write a Maple procedure with the signature

mkRelation(S::set(integer), e::expression)

that creates the relation $\{(a, b) \in S \times S : e(a, b) \text{ is true}\}$. That is, **mkRelation** should return the set of all ordered pairs (a, b) of elements of S for which the expression evaluates to true when a and b

are substituted for the variables in the expression e . The input expression e should be a Boolean-valued Maple expression involving two integer variables x and y as well as operators that take integer operands. For example, your procedure should accept an expression such as

$$\mathbf{x + y < x * y}.$$

Exercise 3. Write a Maple procedure to generate a random relation on a given finite set of integers.

Exercise 4. Use the procedure you wrote in the preceding exercise to investigate the probability that an arbitrary relation has each of the following properties: (a) reflexivity; (b) symmetry; (c) anti-symmetry; and (d) transitivity.

Exercise 5. Write Maple procedures to determine whether a given relation is irreflexive or asymmetric. (See the text for definitions of these properties.)

Exercise 6. Write functions to compute the count and support of an itemset. Then write a function to test whether an itemset is frequent relative to a given threshold; your function should accept the itemset, the set of transactions, and the threshold as arguments.

Exercise 7. Investigate the Apriori algorithm (see Exercise 40 from Section 9.2 and Writing Project 3) and then implement this algorithm in Maple.

Exercise 8. Investigate the ratio of the size of an arbitrary relation to the size of its transitive closure. How much does the transitive closure make a relation “grow” on average?

Exercise 9. Examine the function φ defined as follows. For a positive integer n , we define $\varphi(n)$ to be the number of relations on a set of n elements whose transitive closure is the “all” relation. (If A is a set, then the “all” relation on A is the relation $A \times A$ with respect to which every member of A is related to every other member of A , including itself.)

Exercise 10. Write a Maple procedure that finds the antichain with the greatest number of elements in a partial ordering. (See the text for the definition of antichain.)

Exercise 11. The transitive reduction of a relation G is the smallest relation H such that the transitive closure of H is equal to the transitive closure of G . Use Maple to generate some random relations on a set with 10 elements and find the transitive reduction of each of these random relations.

Exercise 12. Write a Maple procedure that computes a partial order, given its covering relation.

Exercise 13. Write a Maple procedure to determine whether a given lattice is a Boolean algebra, by checking whether it is distributive and complemented. (See the text for definitions.)

10 Graphs

Introduction

In this chapter, we consider ways in which Maple can help you explore and understand graph theory. In particular, we describe how to do computations on graphs using Maple and how Maple can be used to visualize graphs.

Throughout the first half of this chapter, pseudographs are a recurring theme. Recall that pseudographs are graphs that may contain loops and may contain multiple edges between vertices. Maple includes numerous and powerful commands for representing, manipulating, and calculating with simple graphs, both undirected and directed. Each section in what follows will introduce you to these useful tools so that you can more easily explore the concepts described in the textbook. However, Maple does not support pseudographs (or their directed counterparts). Therefore, parts of several sections in this chapter are devoted to extending Maple's existing functionality to pseudographs. This will serve to give you tools that you can use to explore these kinds of graphs. More than that, seeing how to create the procedures for pseudographs will also help you to better understand how the procedures work in the slightly "simpler" case of simple graphs.

10.1 Graphs and Graph Models

Recall that a simple graph, as defined in Section 10.1 of the textbook, is a set V of vertices and a set E of unordered pairs of elements of V , called the edges of the graph, and where each edge connects two different vertices and no two edges connect the same pair of vertices. That is, the edges are undirected, there are no loops, and there are no multiple edges.

Maple has a large collection of commands related to graph theory contained in the **GraphTheory** package. In order to access the short forms of these commands, we use the **with** command.

```
> with(GraphTheory):
```

Creating and Modifying Graphs

The **GraphTheory** package includes commands that allow us to create new graphs and then add or delete edges and vertices or even contract edges. Subsets of the vertices can be used to induce subgraphs. Some of the commands are used to create special kinds of graphs such as complete graphs, hyper-cubes, the Petersen graph, and random graphs. Other commands compute some of the important characteristics of a given graph, such as its maximum degree, its diameter, or its planarity.

To create a new graph, we use the **Graph** command. There are a variety of forms of the **Graph** command, but the most natural uses two arguments: a list of vertices and a set of edges. The edges are given as either sets or lists (e.g., **{1,2}** or **[1,2]**) depending on whether the graph is undirected or directed. We demonstrate the creation of a graph by constructing the graph in Exercise 3 in Section 10.1.

```
> Exercise3 := Graph(["a", "b", "c", "d"],  
    {{{"a", "b"}, {"a", "c"}, {"b", "c"}, {"b", "d"}}})  
Exercise3 := Graph 1: an undirected unweighted graph with 4 vertices and 4 edge(s)  
(10.1)
```

Note that Maple always expects the vertices to be given in a list and the edges in a set. This is how Maple differentiates between them in alternate forms of the command.

The **Vertices** and **Edges** commands can be used to recover the vertices and edges of a graph.

```
> Vertices(Exercise3)
["a", "b", "c", "d"]
```

(10.2)

```
> Edges(Exercise3)
{{{"a", "b"}, {"a", "c"}, {"b", "c"}, {"b", "d}}}
```

(10.3)

Note that explicitly specifying the vertices when defining a graph is often unnecessary, as Maple can determine the vertices from the definition of the edges. The easiest way to define a graph is to call **Graph** with only one argument, the set of edges.

```
> Ex3again := Graph({{"a", "b"}, {"a", "c"}, {"b", "c"}, {"b", "d}})
Ex3again := Graph 2: an undirected unweighted graph with 4 vertices and 4 edge(s)
```

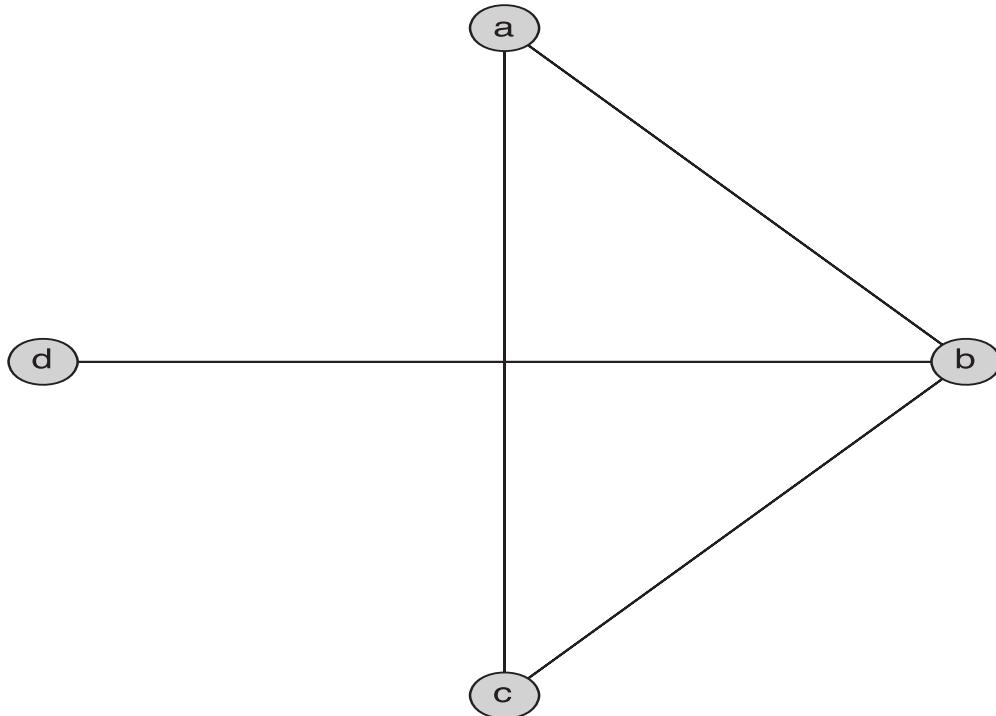
(10.4)

```
> Vertices(Ex3again);
Edges(Ex3again)
["a", "b", "c", "d"]
{{{"a", "b"}, {"a", "c"}, {"b", "c"}, {"b", "d}}}
```

(10.5)

Graphs can be visualized in Maple by applying the command **DrawGraph** to the name assigned to the graph.

```
> DrawGraph(Ex3again)
```



We will explore this command and options for changing the appearance of graphs in more detail shortly.

We can modify existing graphs by adding and deleting edges and vertices using the commands **AddEdge**, **AddArc**, **AddVertex**, **DeleteEdge**, **DeleteArc**, and **DeleteVertex**. (Note: Maple refers to a directed edge as an arc, so the edge commands are used in the case of an undirected graph, and the arc commands are used for directed graphs.)

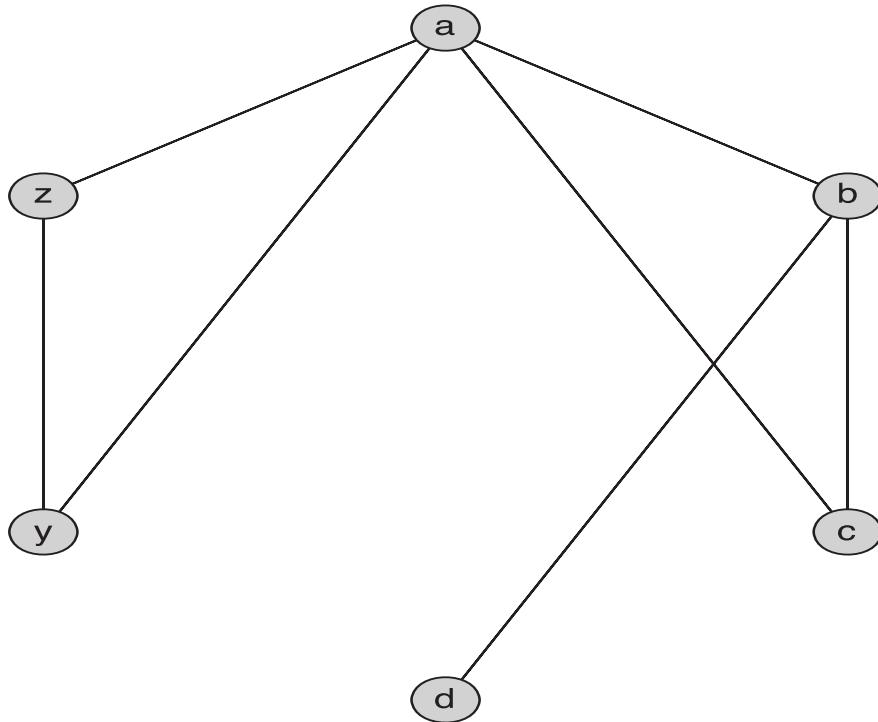
First, we add two vertices to **Ex3again**.

```
> Ex3plus := AddVertex(Ex3again, ["y", "z"])
Ex3plus := Graph 3: an undirected unweighted graph with 6 vertices and 4 edge(s) (10.6)
```

Now, we add edges to connect the new vertices with the rest of the graph.

```
> AddEdge(Ex3plus, {{“a”, “y”}, {“a”, “z”}, {“y”, “z”}})
Graph 3: an undirected unweighted graph with 6 vertices and 7 edge(s) (10.7)

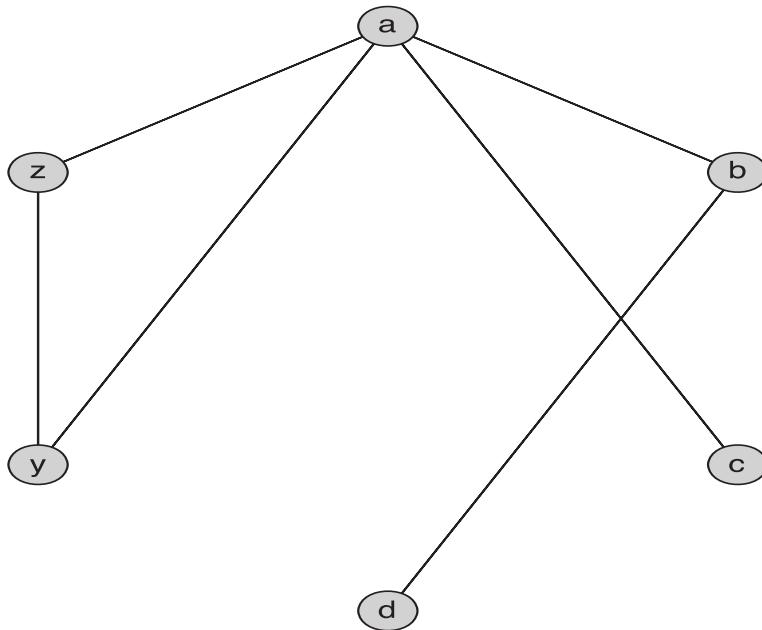
> DrawGraph(Ex3plus)
```



Finally, we delete one of the old edges before once again drawing the graph.

```
> DeleteEdge(Ex3plus, {{“b”, “c”}})
Graph 3: an undirected unweighted graph with 6 vertices and 6 edge(s) (10.8)

> DrawGraph(Ex3plus)
```

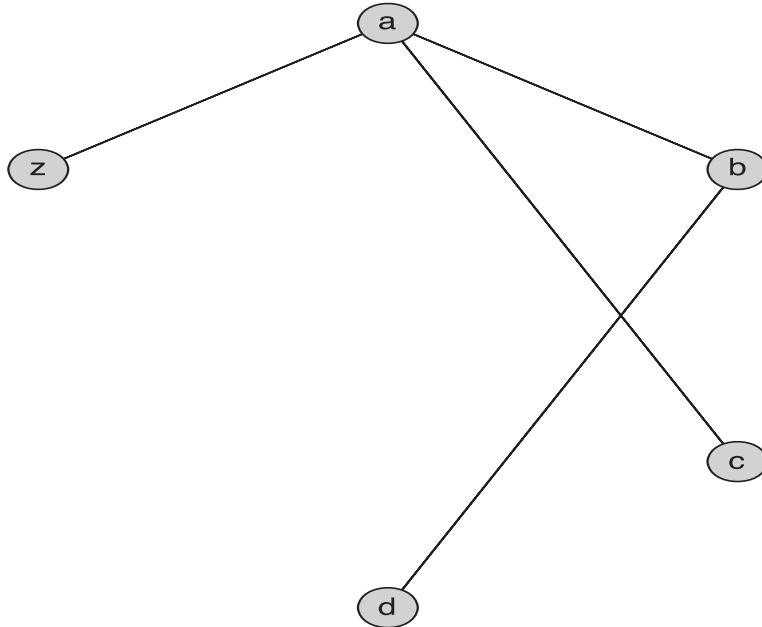


It is important to be aware of a slight inconsistency in the operation of these commands. The **AddVertex** and **DeleteVertex** commands do not modify the original graph, while the edge and arc commands, by default, do modify the original. The edge commands can be made to not modify the original by giving the equation **inplace=false** as an argument. The vertex commands cannot be made to replace the original except through the usual method of reassignment.

Finally, note that deleting a vertex also deletes all the edges incident with that vertex.

```
> Ex3plus := DeleteVertex(Ex3plus, ["y"])
Ex3plus := Graph 4: an undirected unweighted graph with 5 vertices and 4 edge(s)
(10.9)
```

> *DrawGraph(Ex3plus)*

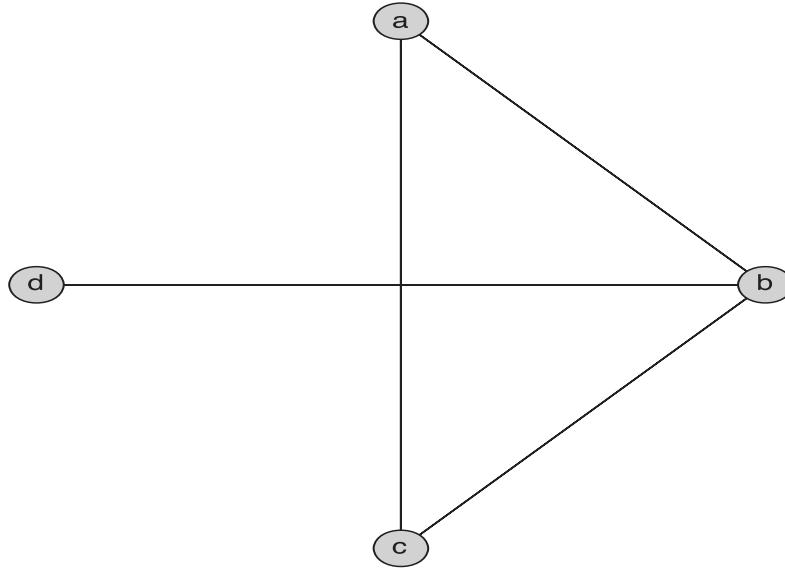


Visualizing Graphs in Maple

The usefulness of graphs is realized partly through our ability to draw diagrams representing them. Visual representations of graphs sometimes lead to a clearer understanding of the underlying relationships represented by the graphs. The beauty of some of the resulting diagrams is also one of the things that helps to make this such a popular subject.

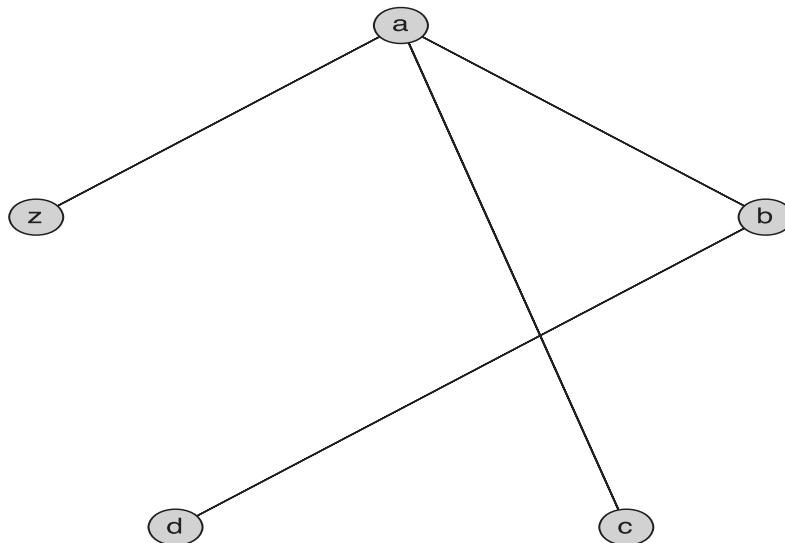
In Maple, we present a graph visually using the **DrawGraph** command. We have already seen that this command can simply be applied to the graph to be displayed.

> *DrawGraph(Exercise3)*



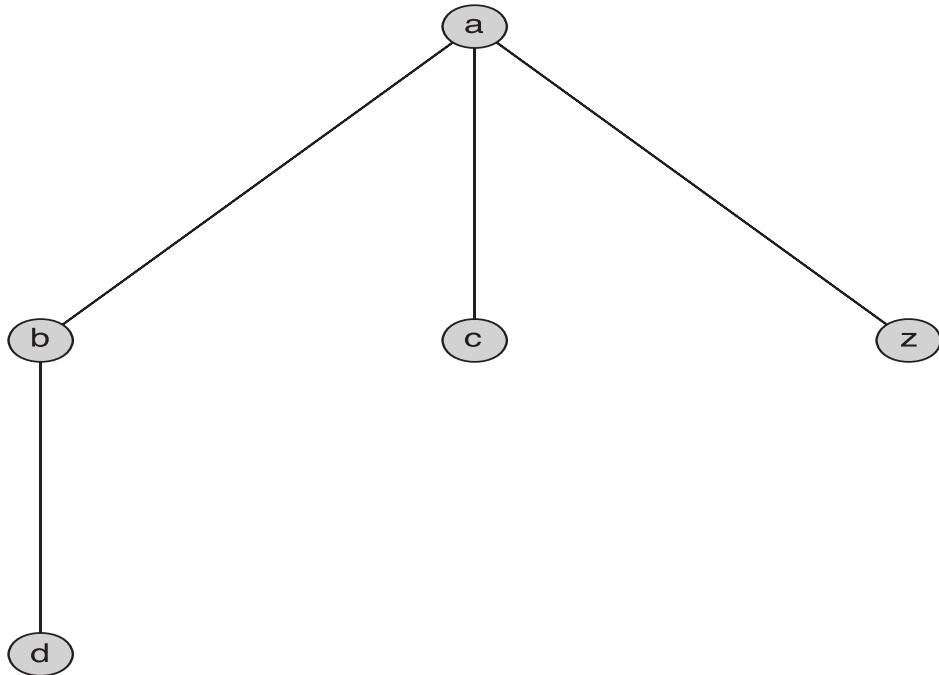
Without any other arguments, Maple does its best to arrange the vertices in reasonable positions. The **style** option allows you to explicitly select a method of arranging the vertices. There are five possible styles: **circle**, **tree**, **bipartite**, **spring**, and **planar**. The **circle** style places the vertices on a circle, equally spaced. This is the style Maple automatically selected for the graph of Exercise 3 and our modifications of it, so specifying the option appears to have no effect.

> *DrawGraph(Ex3plus, style = circle)*



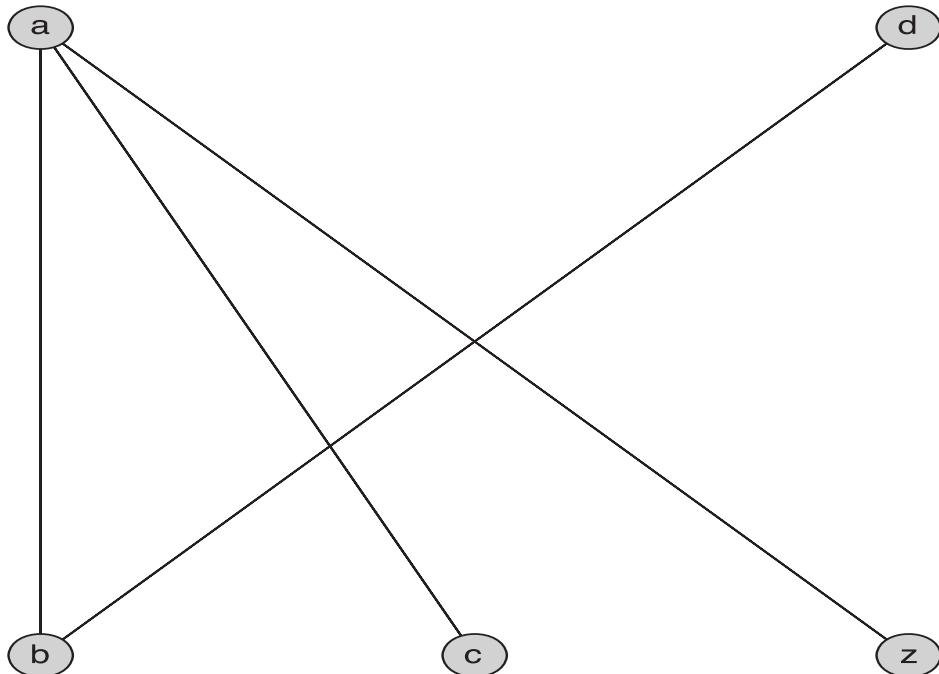
The **tree** style is available only when the graph is, in fact, a tree—connected and with no circuits (refer to Chapter 11 of the text for more information on trees). **Ex3plus** is a tree.

```
> DrawGraph(Ex3plus, style = tree)
```



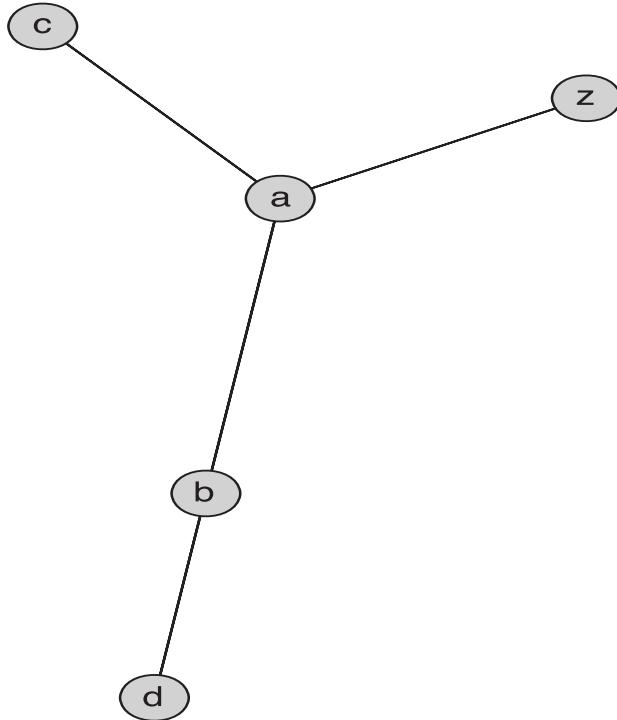
The **bipartite** style can be used when the graph is bipartite, that is, the vertices can be separated into two sets such that every edge has one end in one set and the other end in the other (see Section 10.2.4 of the textbook). Maple places the vertices of a bipartite set in two rows indicating the two sets.

```
> DrawGraph(Ex3plus, style = bipartite)
```



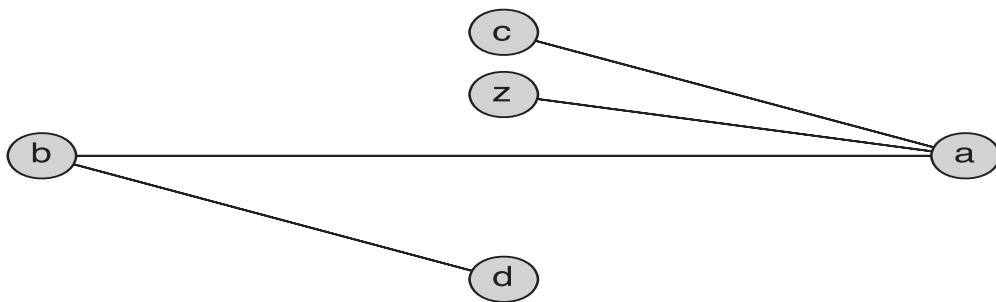
The **spring** style simulates a model where the vertices are taken to be particles repelling each other and the edges are springs pulling vertices together.

```
> DrawGraph(Ex3plus, style = spring)
```



Finally, the **planar** style attempts to draw the graph as a planar graph, that is, with no edges crossing each other (see Section 10.7 of the main text for more on planar graphs). If the graph is nonplanar, then the command will result in an error.

```
> DrawGraph(Ex3plus, style = planar)
```



Pseudographs: Loops and Multiple Edges

As mentioned above, Maple's **GraphTheory** package does not support graphs that have loops or multiple edges. For example, if we try to add a loop to our **Ex3plus** graph, we get an error.

```
> AddEdge(Ex3plus, {"c"})
```

Error, (in GraphTheory:-AddEdge) invalid edge {"c"}

We also get an error if we try to create a directed graph with a loop.

```
> Graph({["a","a"], ["a","b"]})
```

Error, (in GraphTheory:-Graph) invalid edge/arc: ["a", "a"]

On the other hand, multiple edges are merely ignored.

```
> Edges(Ex3plus)
```

```
{ {"a", "b"}, {"a", "c"}, {"a", "z"}, {"b", "d"} } (10.10)
```

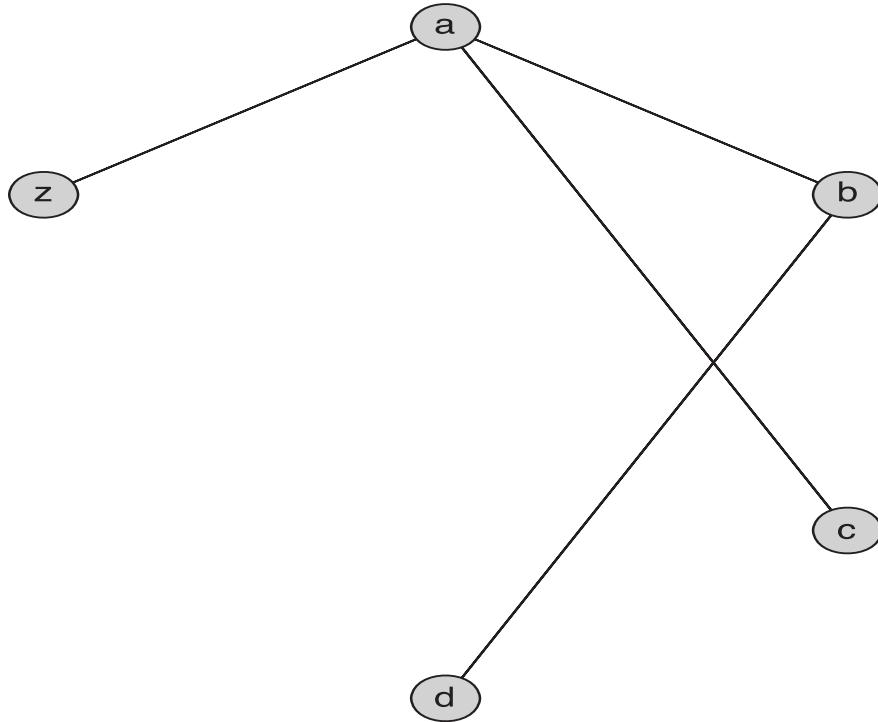
```
> Ex3plus := AddEdge(Ex3plus, { {"a", "c"} })
```

```
Ex3plus := Graph 4: an undirected unweighted graph with 5 vertices and 4 edge(s) (10.11)
```

```
> Edges(Ex3plus)
```

```
{ {"a", "b"}, {"a", "c"}, {"a", "z"}, {"b", "d"} } (10.12)
```

```
> DrawGraph(Ex3plus)
```

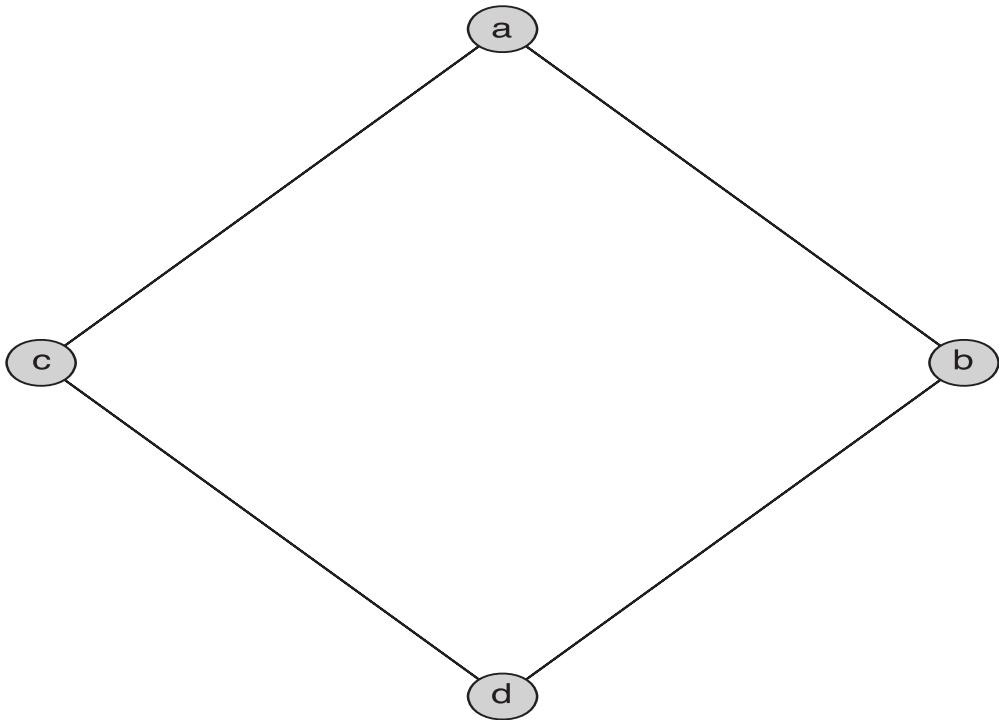


There are ways to, at least partially, get around these two limitations. By way of illustration, we will attempt to create Exercise 5 in Section 10.1. We begin with the simple version of the graph, that is, the graph with the loops and multiple edges omitted.

```
> Exercise5 := Graph({ {"a", "b"}, {"a", "c"}, {"b", "d"}, {"c", "d"} })
```

```
Exercise5 := Graph 5: an undirected unweighted graph with 4 vertices and 4 edge(s) (10.13)
```

```
> DrawGraph(Exercise5, style = planar)
```



Loops

With regards to loops, we can mark vertices as having a loop by setting an attribute. An attribute can be used to store arbitrary information about a vertex (or an edge, or for the graph as a whole) in the form **tag=value**. The tag and value can be nearly anything at all. In this case, we will use the tag “loop” and the value will be true or false.

The **SetVertexAttribute** command takes three arguments: the name of the graph, the name of the vertex, and the attribute in the **tag=value** format.

```
> SetVertexAttribute(Exercise5, "a", "loop" = true)
> SetVertexAttribute(Exercise5, "b", "loop" = true)
> SetVertexAttribute(Exercise5, "d", "loop" = true)
```

We check the value of an attribute with the **GetVertexAttribute** command, which accepts a graph, a vertex, and the tag whose value is desired. Note that if the attribute has not been set, this returns **FAIL**. You can also list all the tags set for a given vertex using the **ListVertexAttributes** command. Note that this does not display the values, only the tags.

```
> GetVertexAttribute(Exercise5, "a", "loop")
true
(10.14)
```

```
> GetVertexAttribute(Exercise5, "c", "loop")
FAIL
(10.15)
```

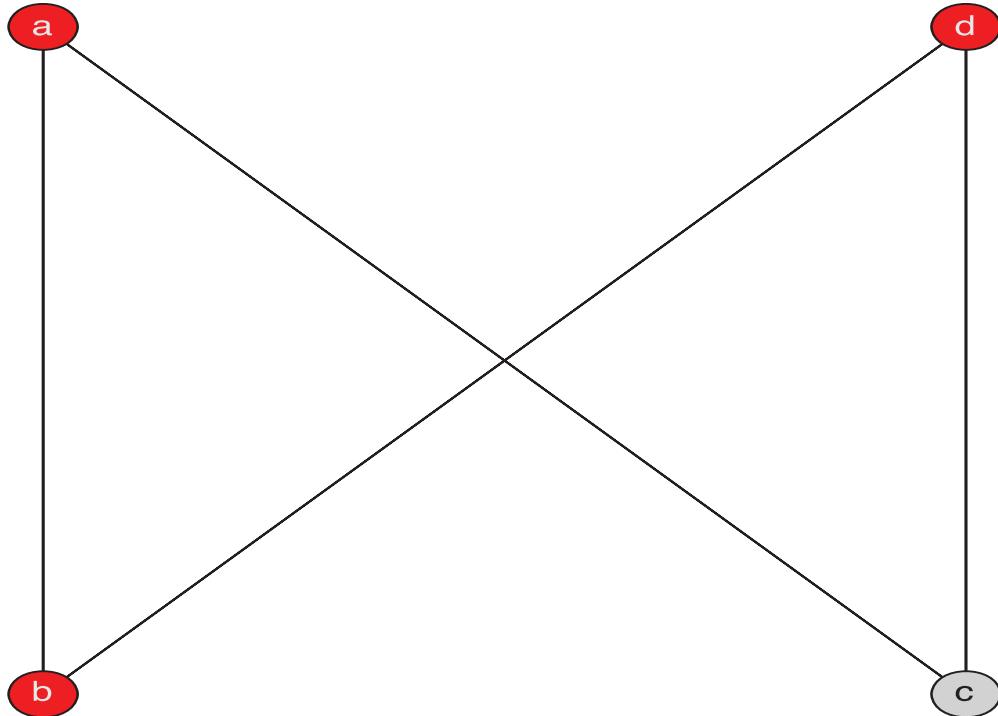
```
> ListVertexAttributes(Exercise5, "b")
["draw-pos-planar", "loop"]
(10.16)
```

We will use attributes to write a program that marks the vertices that have loops by changing their color to red. The color change will be done with the **HighlightVertex** command. The **HighlightVertex** command requires two arguments: the name of the graph and a vertex or list or set of vertices to be highlighted. It optionally accepts a color to use as the highlight.

Here is the procedure to highlight loops.

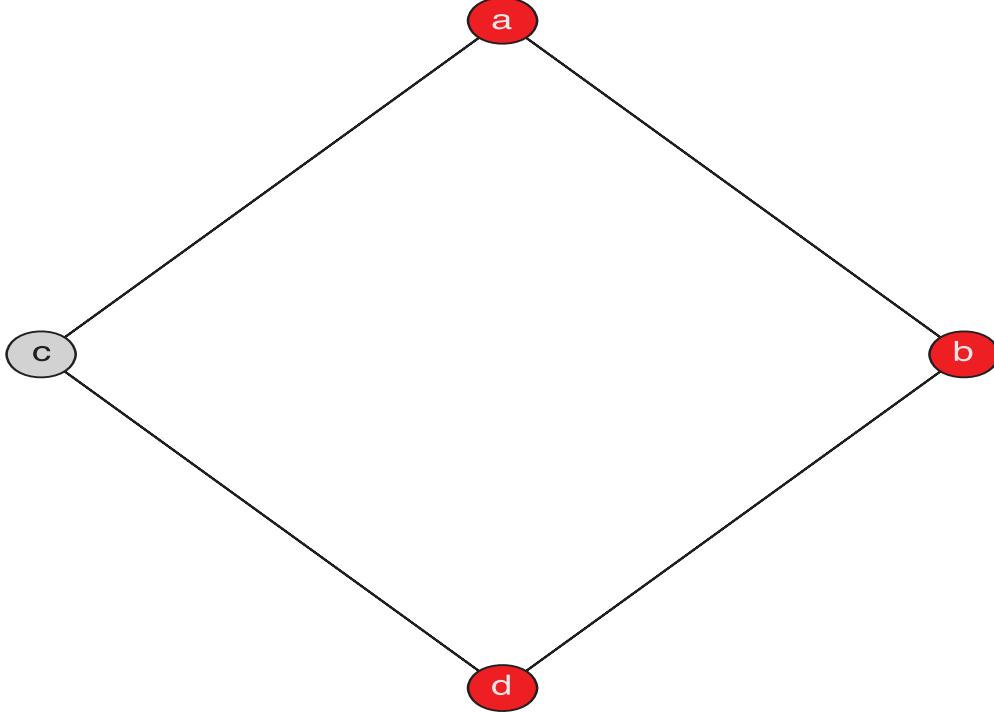
```
1 DrawLoops := proc (G :: Graph)
2   local v;
3   uses GraphTheory;
4   for v in Vertices(G) do
5     if GetVertexAttribute(G, v, "loop") then
6       HighlightVertex(G, v, "Red");
7     end if;
8   end do;
9   DrawGraph(G);
10 end proc;
```

> *DrawLoops (Exercise5)*



Note that the color changes remain in effect until they are changed again. For instance, if we draw the graph in the planar style, the vertices with loops remain red.

```
> DrawGraph(Exercise5, style = planar)
```



You might not be surprised to discover that, in fact, **HighlightVertex** sets a vertex attribute.

```
> ListVertexAttributes(Exercise5, "a")
["draw-pos-planar", "loop", "draw-vertex-color",
 "draw-pos-default"]
```

(10.17)

```
> GetVertexAttribute(Exercise5, "a", "draw-vertex-color")
COLOR(RGB, 1.00000000, 0., 0.)
```

(10.18)

Multiple Edges

We now turn our attention to the representation of multiple edges, which we will do with edge weights. Here is a graph with one weighted edge.

```
> Graph({[{"a", "b"}], 2})
Graph 6: an undirected weighted graph with 2 vertices and 1 edge(s)
```

(10.19)

```
> DrawGraph((10.19))
```



Note that the edge above is specified as the list `[[“a”, “b”], 2]`. This list consists of two elements: first is the set consisting of the endpoints of the edge, and second is the weight of the edge. Maple displays the edge weight next to the edge.

For an existing graph, we can assign weights to edges with the **SetEdgeWeight** command, which takes as arguments the name of the graph, the edge to be weighted, and the weight. It returns the previous weight of the edge. The **SetEdgeWeight** command can only be used with a graph that Maple considers to be weighted. To add weights to an unweighted graph, we first must use the **MakeWeighted** command. (Note that **MakeWeighted** does not change the original graph, it creates a weighted copy of the graph.)

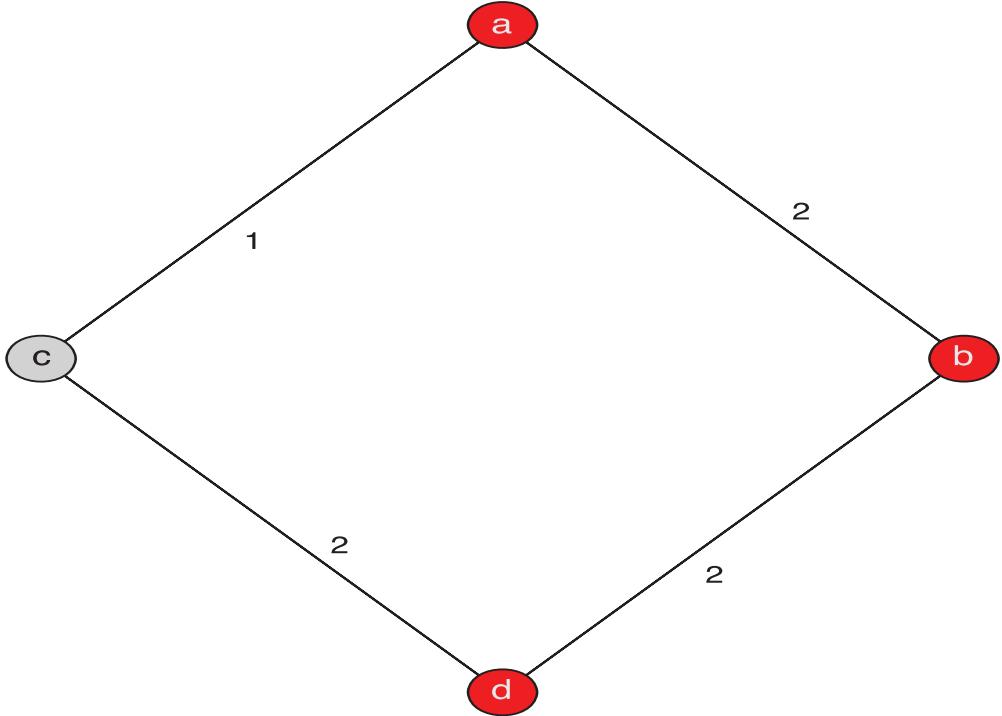
```
> Exercise5 := MakeWeighted(Exercise5)
Exercise5 := Graph 7: an undirected weighted graph with 4 vertices and 4 edge(s) (10.20)
```

```
> SetEdgeWeight(Exercise5, {“a”, “b”}, 2)
1 (10.21)
```

```
> SetEdgeWeight(Exercise5, {“b”, “d”}, 2)
1 (10.22)
```

```
> SetEdgeWeight(Exercise5, {“c”, “d”}, 2)
1 (10.23)
```

```
> DrawGraph(Exercise5, style = planar)
```



While we cannot easily draw multiple edges with multiple lines in Maple, we can represent edge multiplicity as the thickness of the lines. The “draw-edge-thickness” edge attribute changes the thickness of the line representing an edge. We expand on our **DrawLoops** procedure above to not only set the color of vertices which have loops but to also give a visual representation of multiple edges by thickening them. We set the thickness of edges to $3n - 2$ where n is the weight of the edge (i.e., the number of edges) so that a single edge has thickness 1, and each additional edge increases the thickness by 3.

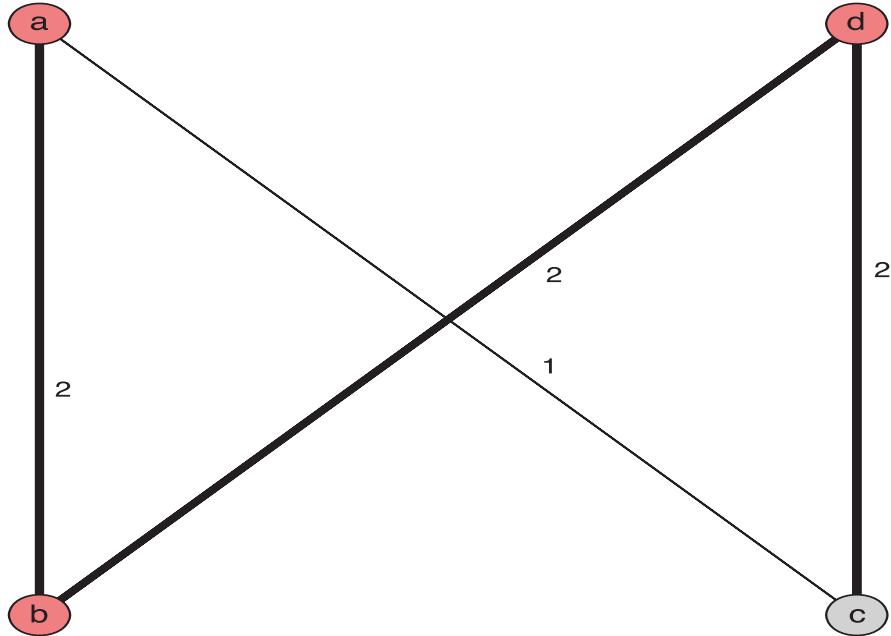
```

1 DrawPseudograph := proc (G::Graph)
2   local vertexcolor, v, e, w;
3   uses GraphTheory;
4   vertexcolor := ToPlotColor(ColorTools[Color] ("LightCoral"));
5   for v in Vertices(G) do
6     if GetVertexAttribute(G, v, "loop") then
7       SetVertexAttribute(G, v, "draw-vertex-color"=vertexcolor);
8     end if;
9   end do;
10  if IsWeighted(G) then
11    for e in Edges(G) do
12      if GetEdgeWeight(G, e) > 1 then
13        w := 3*GetEdgeWeight(G, e) - 2;
14        SetEdgeAttribute(G, e, "draw-edge-thickness"=w);
15      end if;
16    end do;
17  end if;
18  DrawGraph(G);
19 end proc;

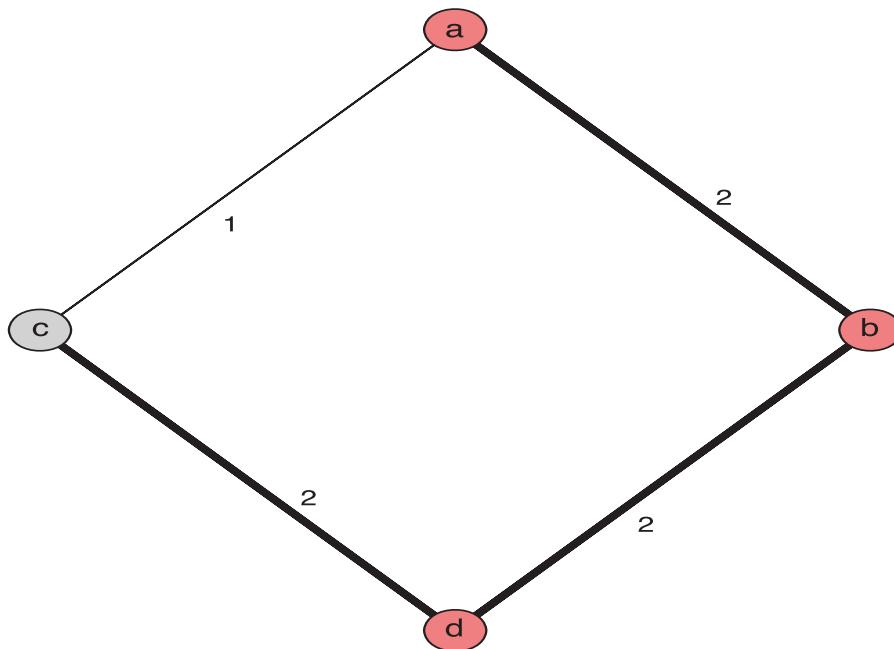
```

Note that **HighlightEdges** does not allow us to set edge thickness, only color. For consistency, we replaced the use of **HighlightVertex** with a call to **SetVertexAttribute**. We also use the **ColorTools** package to select a different color for the vertices with loops. Note that when using attributes to set colors of vertices or edges, the value of the attribute must be in a format understood by the plot commands. In particular, while "Red" is an acceptable argument to **HighlightVertex**, "Red" cannot be used directly as the value of the vertex attribute. You can obtain a list of defined colors by executing **ColorTools[GetColorNames]()**.

> *DrawPseudograph(Exercise5)*



> *DrawGraph(Exercise5, style = planar)*

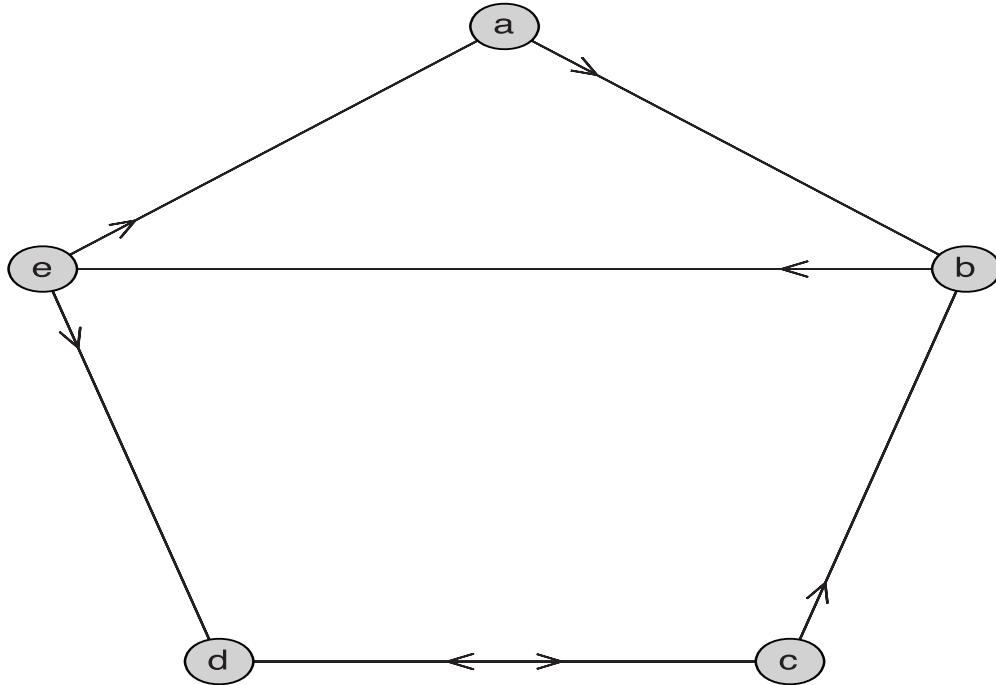


Directed Graphs

Next, we consider an example of a directed graph. Specifically, we will reproduce, as far as possible, Exercise 7 in Section 10.1. We will create this graph with the **Digraph** command, which works like **Graph** but emphasizes that the graph is directed.

We also use the **Trail** command in this example. This command is used to specify a sequence of edges. For instance, **Trail(1,2,3,1)** is a shorter way to specify the edges [1,2], [2,3], and [3,1]. The **Digraph** and **Graph** commands allow us to include applications of **Trail** alongside a set containing additional edges.

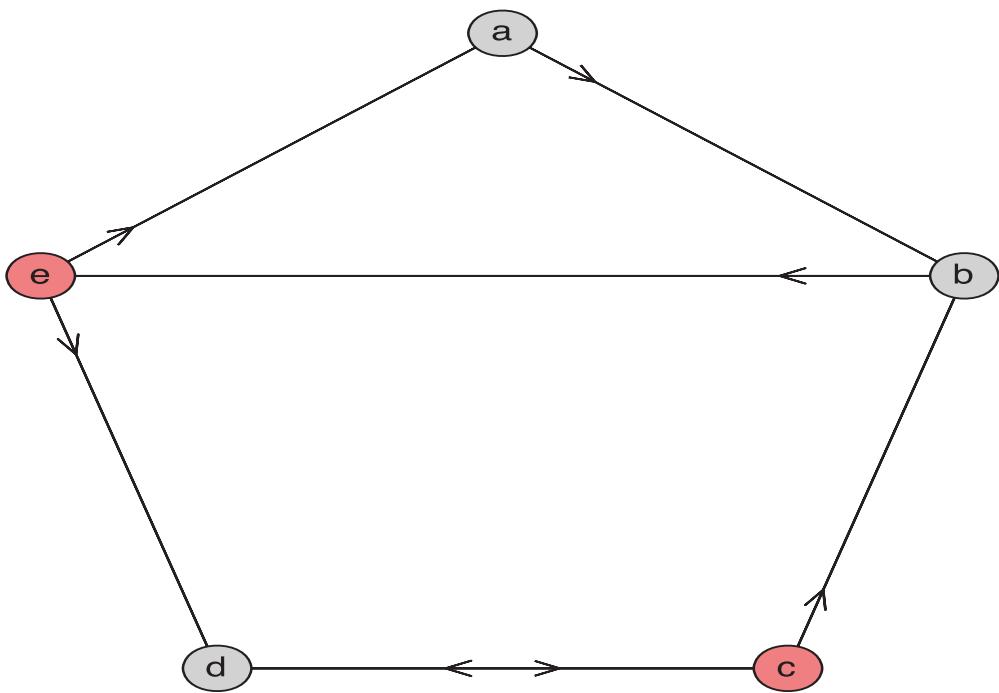
```
> Exercise7 := Digraph([“a”, “b”, “c”, “d”, “e”],  
    Trail(“e”, “a”, “b”, “e”, “d”, “c”, “b”), {[“c”, “d”]})  
Exercise7 := Graph 8: a directed unweighted graph with 5 vertices and 7 arc(s) (10.24)  
  
> DrawGraph(Exercise7)
```



Notice that the edge between *c* and *d* has two arrows representing the pair of edges between them.

Now, we add the loops. Note that **DrawPseudograph** works perfectly well even though there are not multiple edges.

```
> SetVertexAttribute(Exercise7, “c”, “loop” = true)  
  
> SetVertexAttribute(Exercise7, “e”, “loop” = true)  
  
> DrawPseudograph(Exercise7)
```

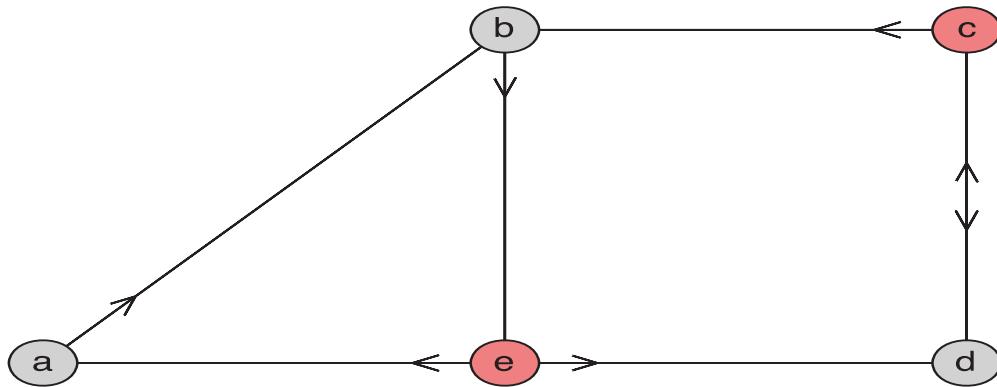


While this image displays all of the information contained in the graph, the positions of the vertices makes it look very different from the drawing in the textbook. We can override Maple's choice of vertex positions with the **SetVertexPositions** command. This command takes two arguments: the graph and a list of pairs specifying the x and y coordinates of each vertex. The first pair specifies the location of the first vertex, the second pair the second vertex, etc.

```

> SetVertexPositions(Exercise7, [[0,0], [1,1], [2,1], [2,0], [1,0]])
> DrawGraph(Exercise7)

```



Semantic Networks

The textbook defines semantic networks at the end of Section 10.1. While Maple cannot understand the meanings of words and thus cannot decide whether two terms should or should not be connected, representing a semantic network in Maple can be useful in a variety of ways, including being able to easily draw the graph, make modifications to it, and perform calculations and check properties like those described throughout Chapter 10.

As an example, we will construct a word graph around the word “program,” which can refer to a computer program, or more generally to instructions for accomplishing a task, or an ordering of elements in an event (as in a program for a performance). Some of the words related to program are: agenda, code, design, docket, malware, package, plan, policy, program, routine, scheme, system, arrangement, function, methodology, procedure, schedule, software.

Our first step will be to assign a name for the list of words. This will be the vertex list for our graph.

```
> wordList := ["agenda", "code", "design", "docket", "malware", "package",
  "plan", "policy", "program", "routine", "scheme", "system",
  "arrangement", "function", "methodology", "procedure", "schedule",
  "software"] :
```

While it is not necessary to store the words in a list, it can be useful to refer to them by their position in the list. For example, the words that most clearly refer to a set of instructions followed by a computer are:

```
> wordList[[2, 6, 9, 10, 14, 16, 18]]
["code", "package", "program", "routine", "function", "procedure",
 "software"]
```

(10.25)

Next, we create the graph on this set of vertices and with no edges. As we determine what pairs of words should be considered related, we will add edges to the graph.

```
> wordGraph := Graph(wordList)
wordGraph := Graph 9: an undirected unweighted graph with 18 vertices and 0 edge(s)
```

(10.26)

First, consider the seven words mentioned above that refer to computer instructions. These words all have a similar meaning, and so each should be considered closely related to each of the others. This is referred to as a clique in graph theory (see preamble to Exercise 19 in the Supplementary Exercises for Chapter 10). We will need to add $C(7, 2) = 21$ edges. We would rather not enter all these edges by hand, so instead we will use the **choose** command from the **combinat** package. Recall that **choose** will accept a set or list as its first argument and, given an integer as the second argument, produces all subsets or sublists of that size. Since edges in a word graph are undirected, we convert our list of words into a set in the first argument.

```
> combinat[choose]({op(wordList[[2, 6, 9, 10, 14, 16, 18]])}, 2)
{{"code", "package"}, {"code", "program"}, {"code", "routine"}, {"code", "function"}, {"code", "procedure"}, {"code", "software"}, {"package", "program"}, {"package", "routine"}, {"package", "function"}, {"package", "procedure"}, {"package", "software"}, {"program", "routine"}, {"program", "function"}, {"program", "procedure"}, {"program", "software"}, {"routine", "function"}, {"routine", "procedure"}, {"routine", "software"}, {"function", "procedure"}, {"function", "software"}, {"procedure", "software"}}
```

(10.27)

These are the first edges we add to our graph. Recall that **AddEdge** can accept a single edge or a set of edges as its second argument and that it modifies the graph given as the first argument.

```
> AddEdge (wordGraph, { {"code", "package"}, {"code", "program"},  
  {"code", "routine"}, {"code", "function"}, {"code", "procedure"},  
  {"code", "software"}, {"package", "program"},  
  {"package", "routine"}, {"package", "function"},  
  {"package", "procedure"}, {"package", "software"},  
  {"program", "routine"}, {"program", "function"},  
  {"program", "procedure"}, {"program", "software"},  
  {"routine", "function"}, {"routine", "procedure"},  
  {"routine", "software"}, {"function", "procedure"},  
  {"function", "software"}, {"procedure", "software"} })
```

Graph 9: an undirected unweighted graph with 18 vertices and 21 edge(s) (10.28)

Likewise, agenda, docket, program, and schedule all refer to a list or timeline of events or items (e.g., a concert or theater program).

```
> AddEdge (wordGraph,  
  combinat[choose] ({op (wordList[[1, 4, 9, -2]])}, 2))
```

Graph 9: an undirected unweighted graph with 18 vertices and 27 edge(s) (10.29)

The words arrangement, design, plan, program, scheme, and system all have meanings related to a method for solving a certain kind of problem or for accomplishing a goal.

```
> AddEdge (wordGraph,  
  combinat[choose] ({op (wordList[[13, 3, 7, 9, 11, 12]])}, 2))
```

Graph 9: an undirected unweighted graph with 18 vertices and 42 edge(s) (10.30)

Similarly, methodology, policy, program, and procedure all refer to a proscribed behavior (e.g., standard operating procedure). Observe that procedure was also in the list of words referring to computer code, creating a link between the two cliques.

```
> AddEdge (wordGraph,  
  combinat[choose] ({op (wordList[[-4, 8, -3, 9]])}, 2))
```

Graph 9: an undirected unweighted graph with 18 vertices and 47 edge(s) (10.31)

Now that we have taken care of the major cliques, we should look back at the original list of words and see what other connections might exist. Here is the list of words again.

```
> wordList  
["agenda", "code", "design", "docket", "malware", "package", "plan",  
 "policy", "program", "routine", "scheme", "system", "arrangement",  
 "function", "methodology", "procedure", "schedule", "software"]
```

(10.32)

First, note that malware has not yet been used, but is closely related to software, so we add that edge.

```
> AddEdge (wordGraph, {"malware", "software"})
```

Graph 9: an undirected unweighted graph with 18 vertices and 48 edge(s) (10.33)

Second, the word system brings to mind the term operating system, suggesting a link between system and software.

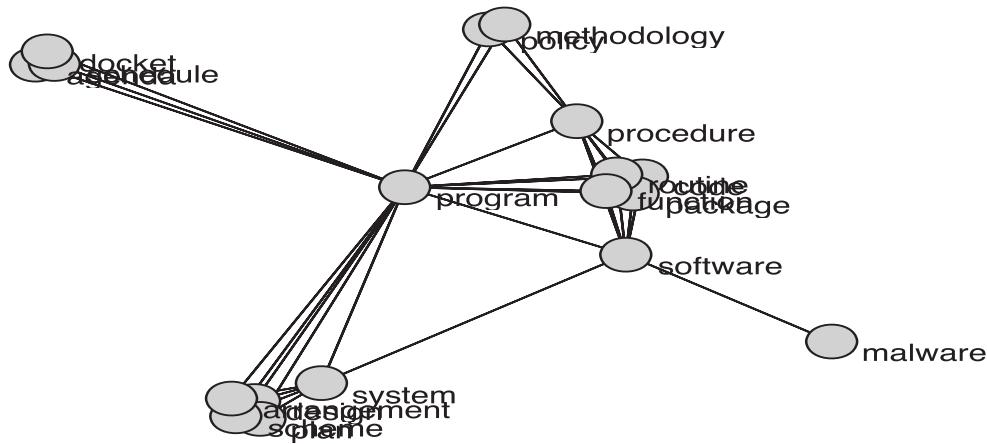
> *AddEdge (wordGraph, {"system", "software"})*

Graph 9: an undirected unweighted graph with 18 vertices and 49 edge(s)

(10.34)

Now we draw the graph we have developed. Note that the **spring** style is a natural choice for drawing word graphs. This will make the image better represent the clique structure. The **spring** style can be combined with the **redraw** option. When using the **spring** style, vertex locations are chosen based on a simulation of the graph modeled as a collection of vertices repelling each other connected by springs that attract adjacent vertices. The image is partially dependent on random starting positions for the vertices which then stabilize. Giving the option **redraw=true** means that reexecuting the command will choose different starting positions and thus produce different graphs. You might also wish to experiment with the **animate=true** option which allows you to see the process that leads to the final vertex positions. We use the **label-style=offset** option so that the labels will appear next to the vertices, rather than inside them, so that the vertices will be smaller. If you wish to turn off display of the words, you can use the **showlabels=false** option.

> *DrawGraph (wordGraph, style = spring, redraw = true, labelstyle = offset)*



Of course, you may recognize other connections that you may wish to add to the graph. Unless you choose an external authority, such as a dictionary or thesaurus, to make the decision of whether or not two words are related rigorous, some element of opinion is involved. Since the goal of this example is to illustrate some of the ways that Maple can be used in this process, we have not chosen an authority.

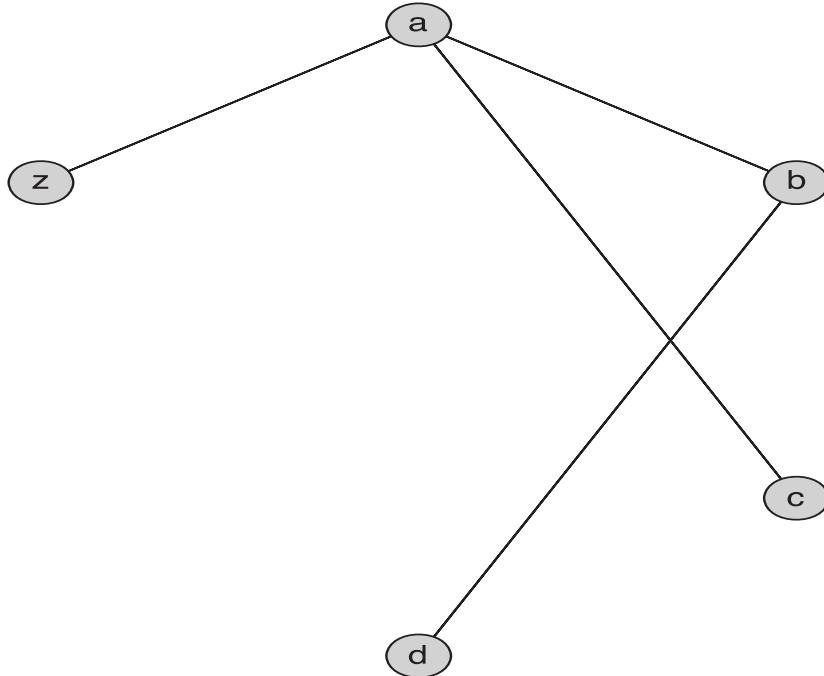
10.2 Graph Terminology and Special Types of Graphs

In this section, we will see how to use Maple to perform computations related to some of the basic terminology of graphs, such as calculating degree and checking for isolated vertices. We will also look at some of the special families of graphs that Maple has built-in support for. In addition, we discuss subgraphs and unions of graphs in Maple.

Degree

Maple's **GraphTheory** package has a **Degree** command for determining the degree of a vertex. Given a graph and one of the graph's vertices, the function returns the number of edges incident to that vertex. For example, we can check the degrees of vertices *a* and *z* of **Ex3plus** from above.

> *DrawGraph*(*Ex3plus*)



> *Degree*(*Ex3plus*, "a")

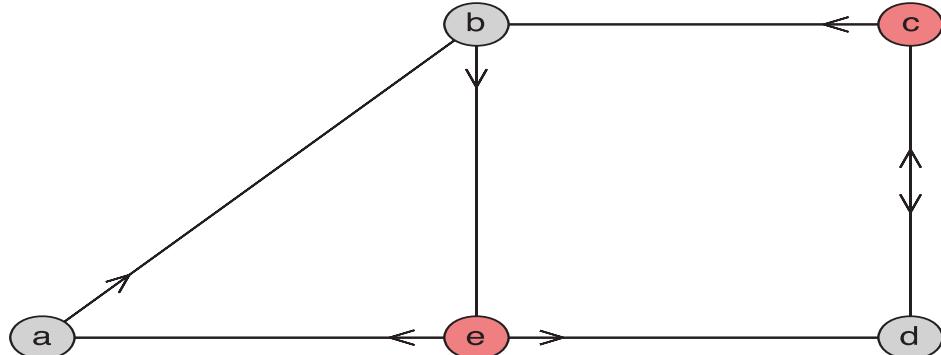
$$3 \tag{10.35}$$

> *Degree*(*Ex3plus*, "z")

$$1 \tag{10.36}$$

For a directed graph, the **Degree** command calculates the number of edges incident to the given vertex without regard for their direction. For calculating degrees in directed graphs, Maple also provides **InDegree** and **OutDegree** commands. As an example, consider vertex *d* in the **Exercise7** graph from the previous section.

> *DrawGraph*(*Exercise7*)



> *Degree*(Exercise7, "d")
 3
 (10.37)

> *InDegree*(Exercise7, "d")
 2
 (10.38)

> *OutDegree*(Exercise7, "d")
 1
 (10.39)

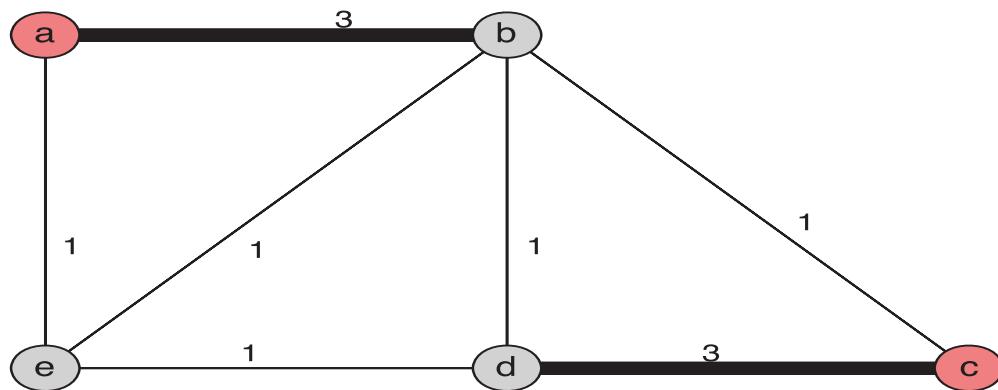
Degree in Pseudographs

Note that Maple's built-in commands for degree cannot take into account loops or multiple edges. We will write a procedure to rectify this, at least for pseudographs (undirected graphs which may have loops and multiple edges). The procedures for directed graphs are left to the reader.

First, we reproduce Exercise 2 in Section 10.2 to use as an example. Note that for the multiple edges, we use the weighted-edge format **[{v1,v2},w]**, which indicates an undirected edge between **v1** and **v2** with weight **w**. For the single edges, we use the usual **{v1,v2}** format. The presence of weighted edges tells Maple that the graph is weighted and causes it to assign weight 1 to edges that are not given a specific weight.

> *Exercise2* := *Graph*({{{"a", "b"}, 3}, {"c", "d"}, 3}, {"a", "e"}, {"b", "c"}, {"b", "d"}, {"b", "e"}, {"d", "e"}))
Exercise2 := Graph 10: an undirected weighted graph with 5 vertices and 7 edge(s)
 (10.40)

> *SetVertexPositions*(Exercise2, [[0, 1], [1, 1], [2, 0], [1, 0], [0, 0]])
 > *SetVertexAttribute*(Exercise2, "a", "loop" = true)
 > *SetVertexAttribute*(Exercise2, "c", "loop" = true)
 > *DrawPseudograph*(Exercise2)



To calculate the degree of a vertex, we first check to see if the graph is weighted or not using the **IsWeighted** command. If it is not weighted (i.e., there are no multiple edges), then we just use the built-in **Degree** function. If it is weighted, then we use the command **IncidentEdges** to determine the edges incident to the given vertex. The **add** command adds up the weights for us. Then, we just have to check to see if there is a loop and, if so, add 2 to the degree.

```

1 PseudoDegree := proc (G::Graph, v)
2   local e, d;
3   uses GraphTheory;
4   if IsWeighted(G) then
5     d := add (GetEdgeWeight (G, e), e in IncidentEdges (G, v));
6   else
7     d := Degree (G, v);
8   end if;
9   if GetVertexAttribute (G, v, "loop") then
10    d := d + 2;
11  end if;
12  return d;
13 end proc;

```

> *PseudoDegree(Exercise2, “a”)*

6 (10.41)

> *PseudoDegree(Exercise2, “d”)*

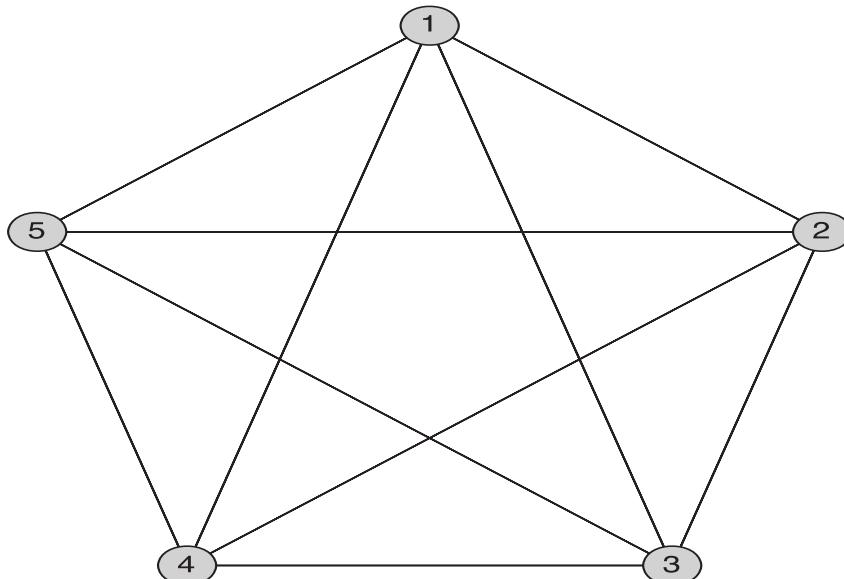
5 (10.42)

Some Special Simple Graphs

The textbook discusses several families of graphs, including complete graphs, cycles, wheels, and n -cubes. Maple provides commands for easily creating these and other special simple graphs.

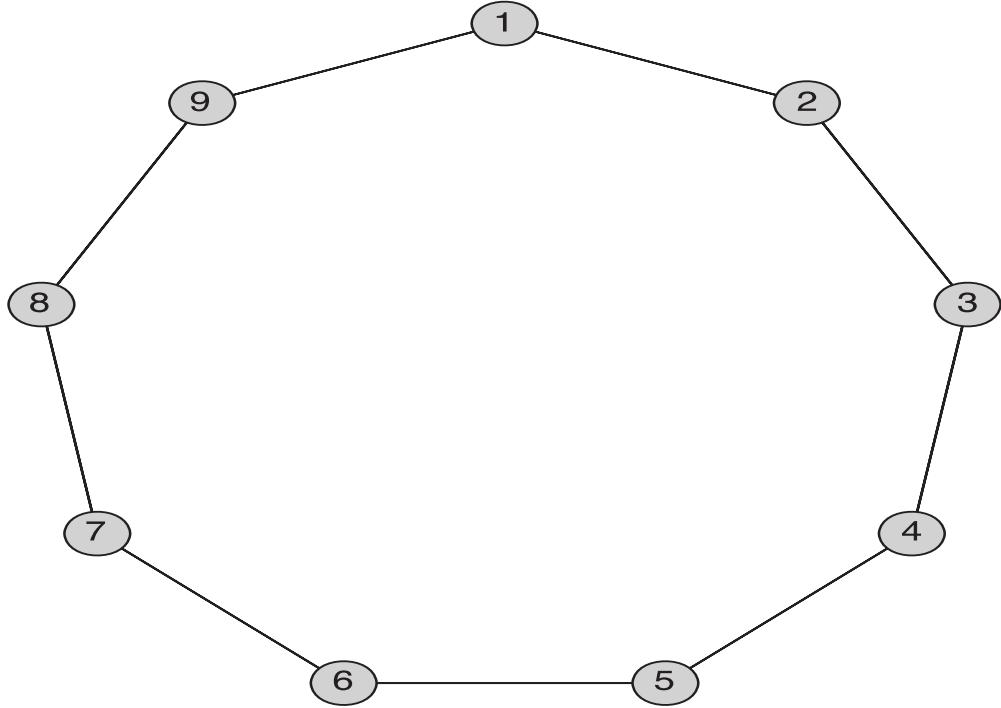
We begin with complete graphs. Recall that a complete graph is a simple, undirected graph on a given number of vertices that has all possible edges between those vertices. The complete graph on n vertices is denoted K_n . The function **CompleteGraph** generates complete graphs. For example, we can generate and display K_5 , the complete graph on 5 vertices.

> *DrawGraph(CompleteGraph(5))*



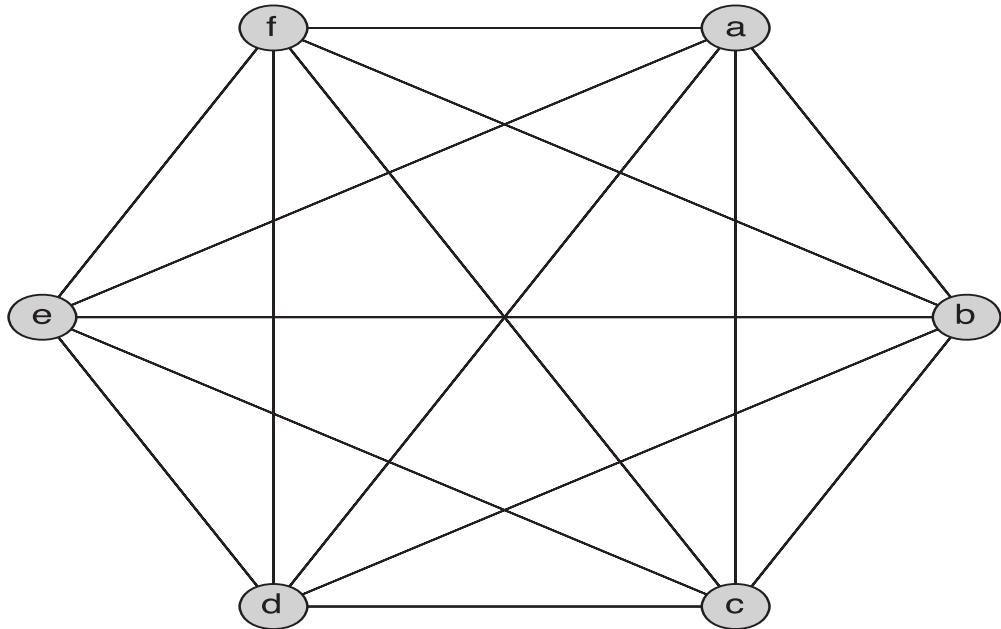
Similarly, we may construct a cycle C_n with the **CycleGraph** command.

> *DrawGraph(CycleGraph(9))*



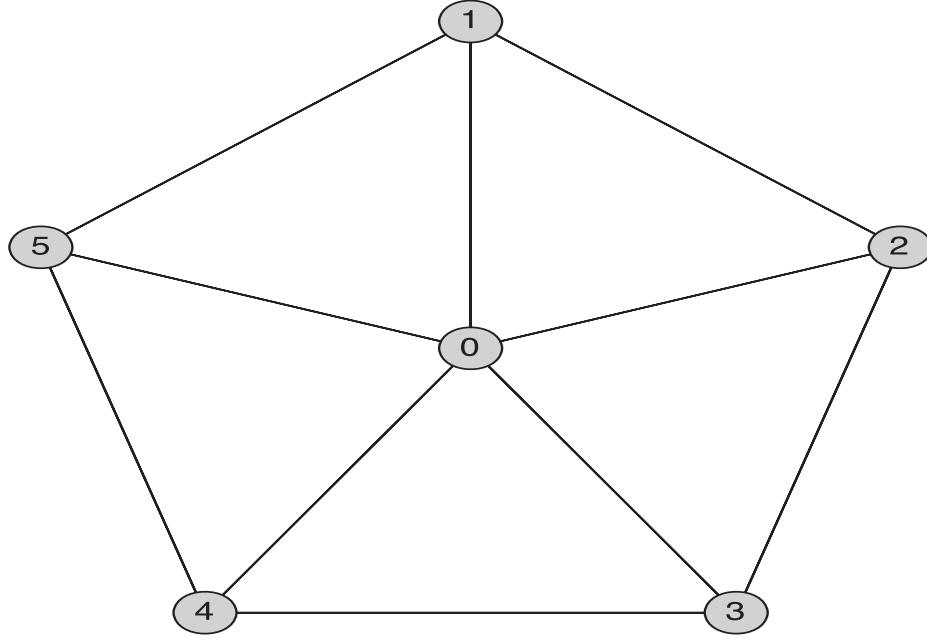
For both the complete and cycle graphs, if you prefer the vertices to be labeled with something other than the integers 1 through n , you can call the commands with a list of vertices instead.

> *DrawGraph(CompleteGraph(["a", "b", "c", "d", "e", "f"]))*



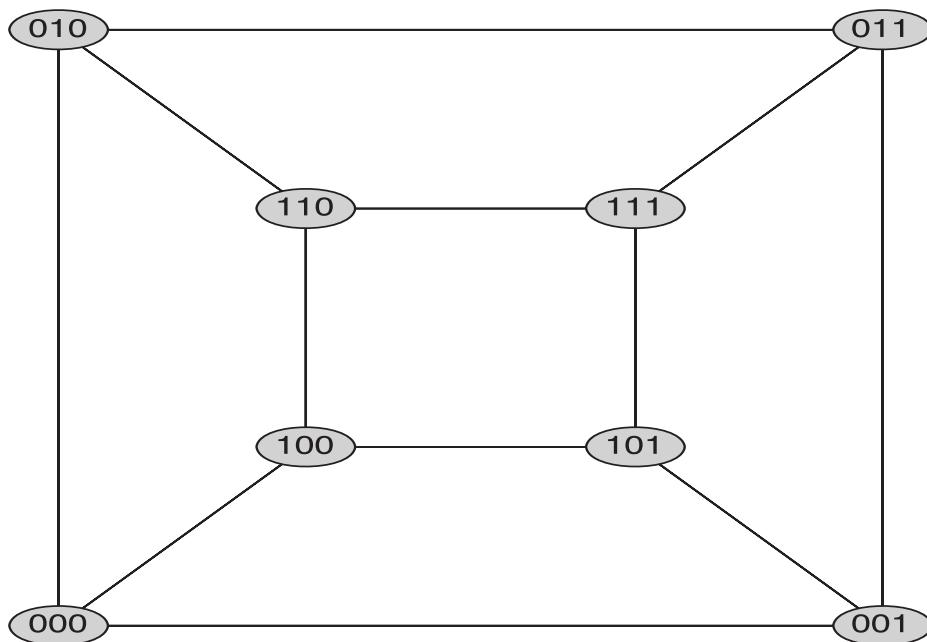
A wheel W_n is obtained from the cycle graph C_n by adding one additional vertex adjacent to all n of the original vertices. In Maple, the **WheelGraph** command is part of the **SpecialGraphs** package.

```
> DrawGraph(SpecialGraphs[WheelGraph](5))
```



To construct the n -cube Q_n , we use the **HypercubeGraph** command. Recall the definition of the hypercube graph given in Example 8 in Section 10.2. There are 2^n vertices labeled with the binary representations of the numbers 0 through $2^n - 1$. Two vertices are adjacent if their binary representations differ in only one digit.

```
> DrawGraph(SpecialGraphs[HypercubeGraph](3))
```



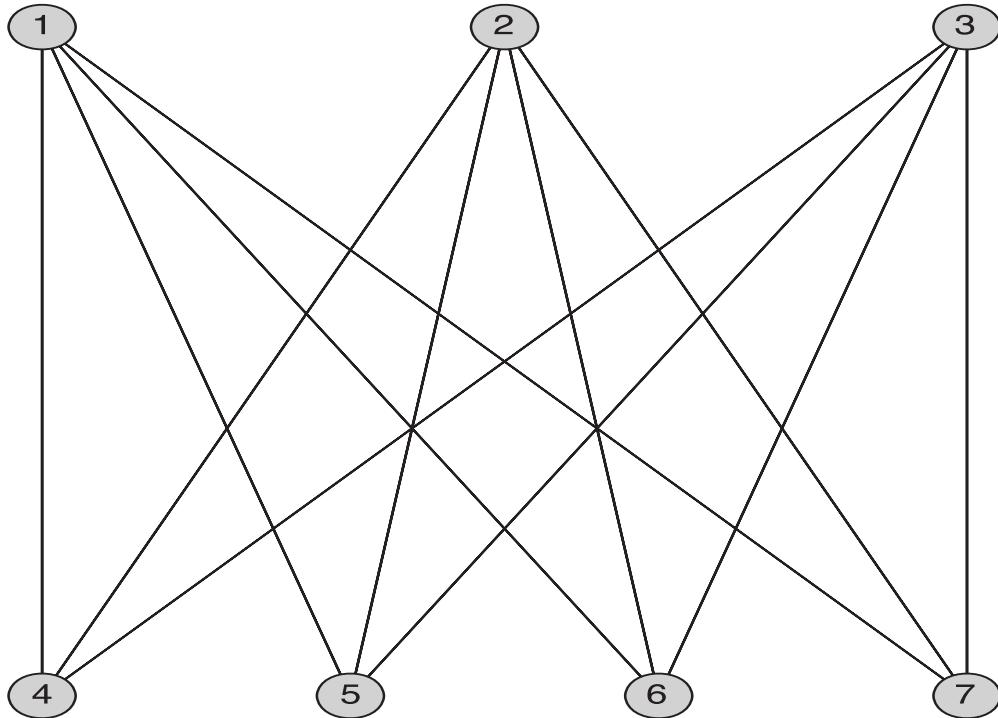
The help page for the **SpecialGraphs** package lists all of the available graphs. The reader is encouraged to spend some time exploring that package.

Bipartite Graphs

Another important class of graphs is the bipartite graphs. A bipartite graph is one whose vertex set can be partitioned into two disjoint sets such that every edge has one vertex in each of the partitioning sets. In other words, no two vertices in the same partitioning set are adjacent. We write $V = (A, B)$ to indicate that the vertex set V is partitioned into the sets A and B .

The complete bipartite graph $K_{m,n}$ is a bipartite graph with bipartition $V = (A, B)$ such that there are m vertices in A and n in B and such that there is an edge for every pair of vertices $a \in A$ and $b \in B$. The **CompleteGraph** command that was discussed earlier can be used to create complete bipartite graphs by entering the two integers m and n .

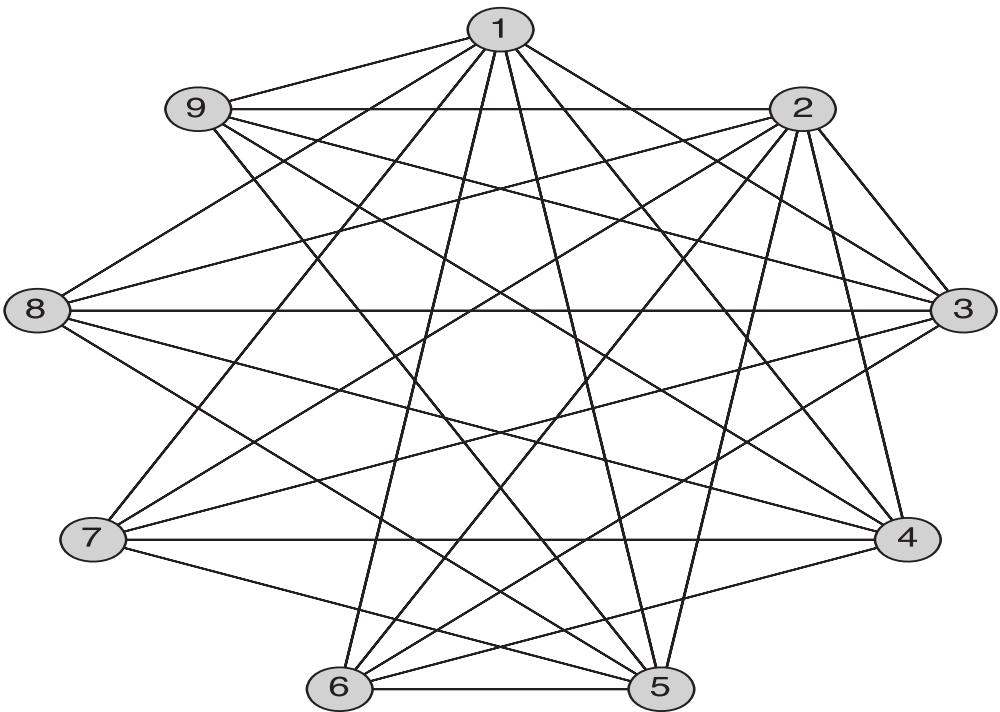
```
> DrawGraph(CompleteGraph(3,4))
```



Notice that Maple draws the complete bipartite graph with the two partitioning sets $\{1, 2, 3\}$ and $\{4, 5, 6, 7\}$ along the top and bottom of the graph, respectively, to make the partition visually clear.

Maple can also produce complete multipartite graphs. A k -partite graph is a graph in which the vertices can be partitioned into k disjoint sets so that no two vertices in any one of the partitioning sets are adjacent.

```
> DrawGraph(CompleteGraph(2,3,4))
```



Maple has a built in command for determining whether a graph is bipartite: **IsBipartite**. This command takes two arguments: the name of a graph and an optional unused name in which the bipartition is stored in the case the graph is bipartite.

```
> IsBipartite(SpecialGraphs[HypercubeGraph](3), 'examplePart')
true
```

(10.43)

```
> examplePart
[[“000”, “011”, “101”, “110”], [“001”, “010”, “100”, “111”]]
```

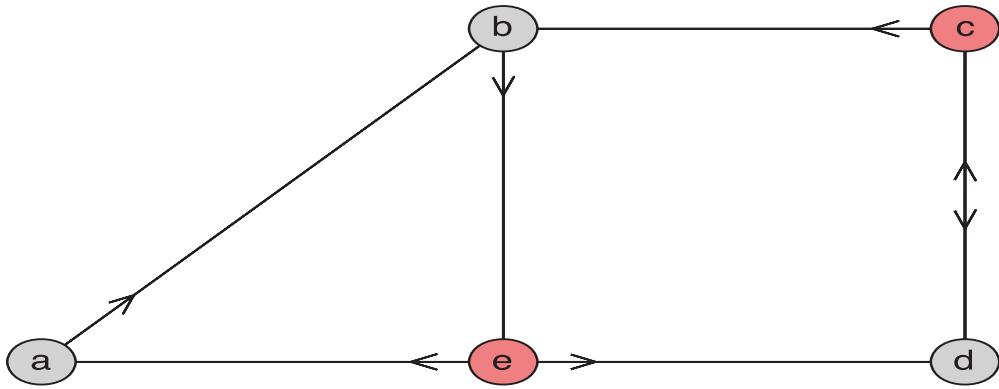
(10.44)

Visualizing Bipartite Graphs

It is worthwhile, however, recreating a version of **IsBipartite** from scratch in order to better understand the algorithm that determines whether the graph is bipartite and finds a bipartition. Instead of just returning true, our procedure will, if the graph is bipartite, display the graph with the vertices colored red and blue to represent the partitioning. Of course, if the graph is not bipartite, the procedure will return false.

This procedure will make use of the **Neighbors** command. Given a graph and a vertex, this command returns the list of vertices adjacent to the given vertex. For a directed graph, like **Degree**, the output of **Neighbors** is irrespective of the direction of edges. The **Arrivals** and **Departures** commands return the lists of vertices with edges towards and away from, respectively, the given vertex. We illustrate with the graph from Exercise 7.

```
> DrawGraph(Exercise7)
```



> *Neighbors*(Exercise7, "b")
 ["a", "c", "e"] (10.45)

> *Arrivals*(Exercise7, "b")
 ["a", "c"] (10.46)

> *Departures*(Exercise7, "b")
 ["e"] (10.47)

The idea of our procedure for coloring the vertices of bipartite graphs is as follows. (Note that this method is based on forming a spanning tree of the graph, a concept discussed in Section 11.4 of the textbook.)

1. Pick a vertex v from the vertex set and place it in the set A .
2. Place all of v 's neighbors in set B .
3. For each vertex w in the set B that has not already been processed, place all of w 's neighbors that are not already in either set into the set A .
4. Repeat step 3, reversing A and B until no more vertices remain to be processed.
5. Once step 4 is complete, we have formed a disjoint partition of the vertices. We then examine each edge of the graph and ensure that no edge has both ends in A or both ends in B . If some edge fails that test, then the graph is not bipartite. If all of the edges do pass the test, then the graph is bipartite and (A, B) is a bipartition.

Here is the implementation of our procedure **DrawBipartite**.

```

1 DrawBipartite := proc (G::Graph)
2   local V, E, AB, i, T, w, e;
3   uses GraphTheory;
4   V := {op(Vertices(G))};
5   E := Edges(G);
6   w := V[1];
7   AB[0] := {w};
8   AB[1] := {};
9   i := 0;
10  while V <> {} do
11    T := V intersect AB[i];
12    i := i + 1 mod 2;
13    for w in T do

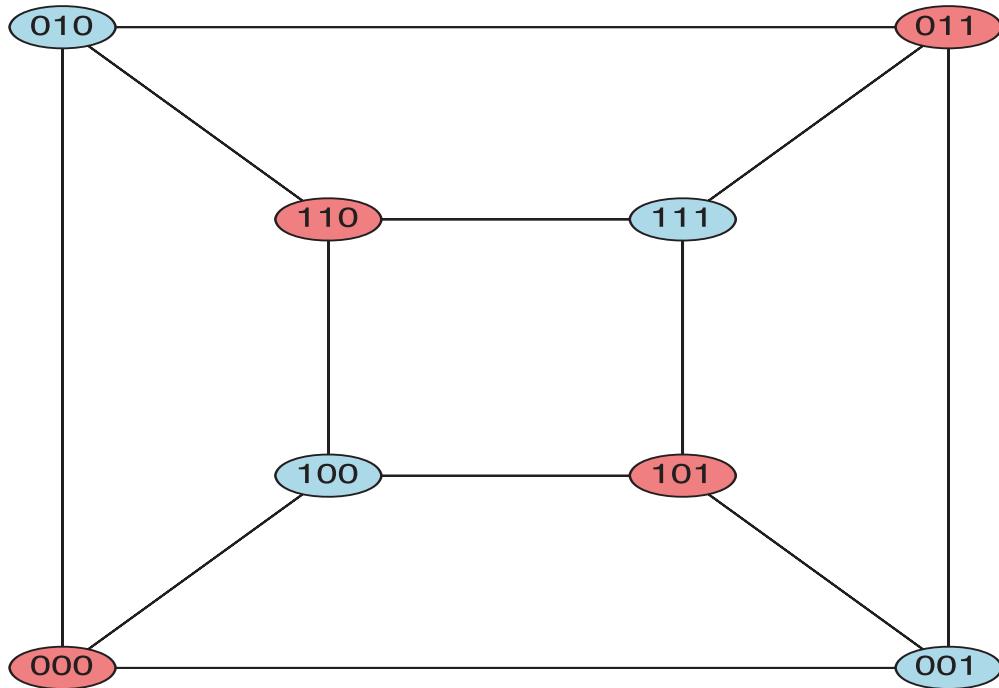
```

```

14      AB[i] := AB[i] union ( {op(Neighbors(G, w))} minus (AB[0] union
15          AB[1]) );
16      end do;
17      V := V minus T;
18  end do;
19  for e in E do
20      if ((e[1] in AB[0]) and (e[2] in AB[0])) or ((e[1] in AB[1]) and
21          (e[2] in AB[1])) then
22          return false;
23      end if;
24  end do;
25  HighlightVertex(G, AB[0], "LightCoral");
26  HighlightVertex(G, AB[1], "LightBlue");
27  DrawGraph(G);
28 end proc;

```

> *DrawBipartite(SpecialGraphs[HypercubeGraph])(3)*

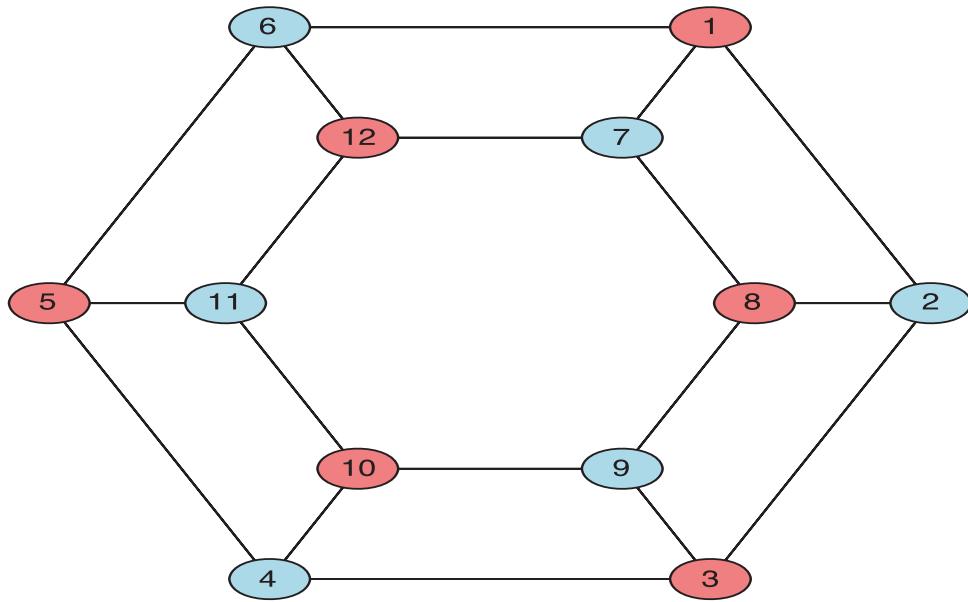


> *DrawBipartite(CompleteGraph)(6)*

false

(10.48)

> *DrawBipartite(SpecialGraphs[PrismGraph])(6)*



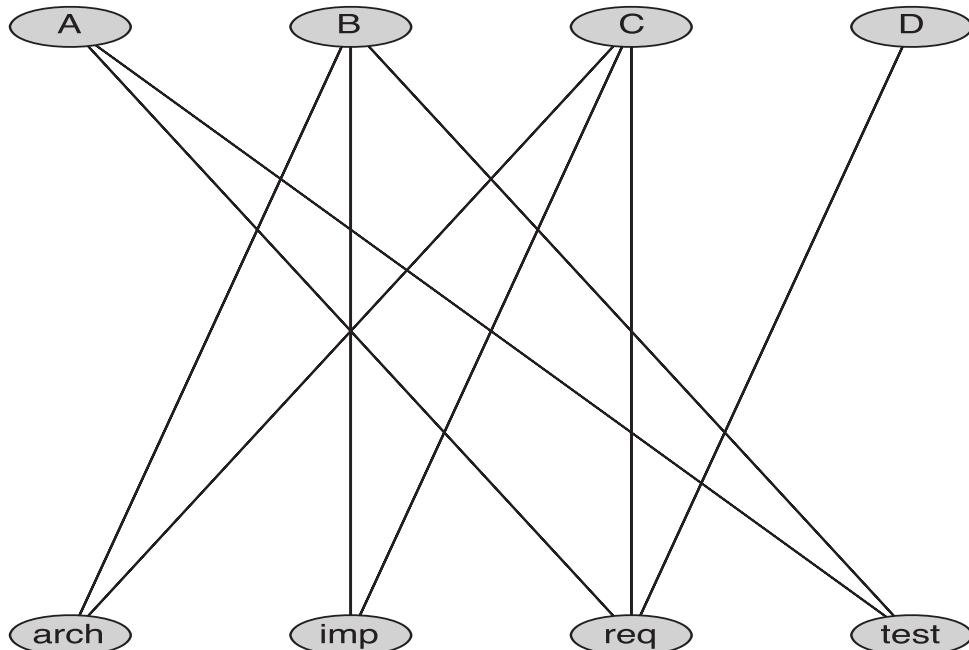
Bipartite Graphs and Matchings

Maple can help us find maximal matchings in a bipartite graph. We will use Figure 10a in Section 10.2 of the text as an example. To improve readability, we have abbreviated the names to their first letter and shortened the descriptions of the jobs.

```
> Figure10a := Graph({{"A", "req"}, {"A", "test"}, {"B", "arch"},  
  {"B", "imp"}, {"B", "test"}, {"C", "arch"}, {"C", "imp"},  
  {"C", "req"}, {"D", "req"}})
```

Figure10a := Graph 11: an undirected unweighted graph with 8 vertices and 9 edge(s)
(10.49)

> *DrawGraph(Figure10a)*



To find a maximal matching, we use the command **BipartiteMatching**. The only allowed argument to this command is the name of the graph. It returns a sequence with two elements: (1) the size of a maximal matching, that is, the largest possible number of edges in a matching; and (2) a set of edges which forms one maximal matching.

```
> Figure10aResult := BipartiteMatching(Figure10a)
Figure10aResult := 4, {{“A”, “test”}, {“B”, “arch”}, {“C”, “imp”},
{“D”, “req”}} (10.50)
```

The above output indicates that one maximal matching has Alvarez assigned to testing, Berkowitz to architecture, Chen to implementation, and Davis to requirements. (Note that Maple has produced a different matching than the one given in the text.)

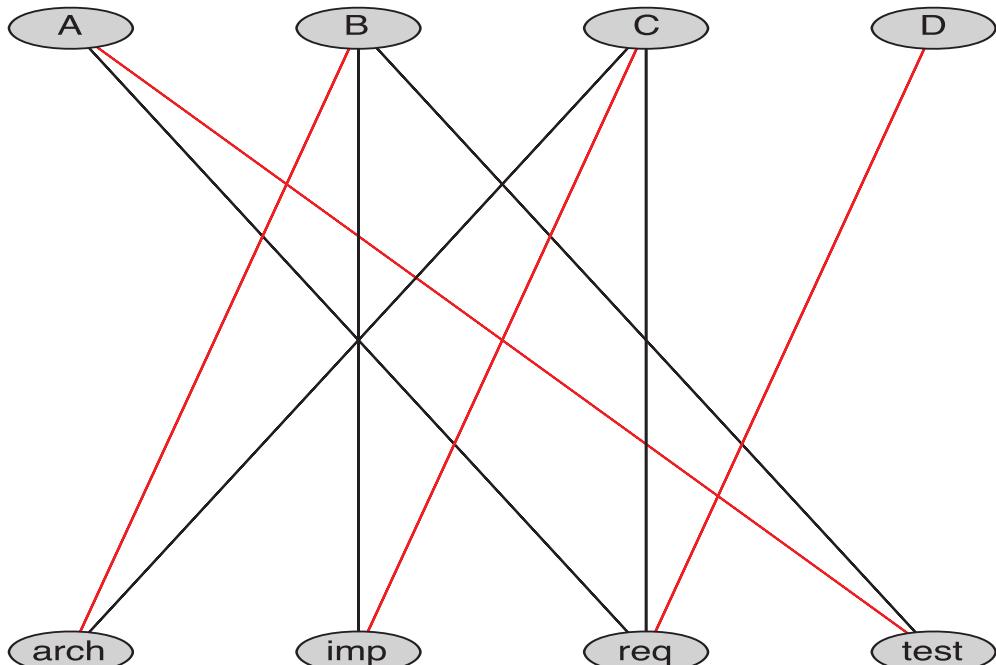
We can visualize this matching by having Maple highlight the edges that form the matching. The edges of the matching are the second element in the output, so we access the set of edges as follows.

```
> Figure10aResult[2]
{{“A”, “test”}, {“B”, “arch”}, {“C”, “imp”}, {“D”, “req”}} (10.51)
```

We use that as the second argument to the **HighlightEdges** command. Given a graph and a set of edges, **HighlightEdges** changes the color of the specified edges. It can also take an optional third argument specifying the color to use.

```
> HighlightEdges(Figure10a, Figure10aResult[2])
```

```
> DrawGraph(Figure10a)
```



Subgraphs and Induced Subgraphs

Maple provides two primary methods for creating subgraphs. The **Subgraph** command takes two arguments: a graph and a list of edges. It returns the graph whose edge set is the given set of edges and whose vertex set is the vertices that are an endpoint of one of the given edges.

```
> hyper := SpecialGraphs[HypercubeGraph](3)
hyper := Graph 12: an undirected unweighted graph with 8 vertices and 12 edge(s) (10.52)
```

```
> subhyper := Subgraph(hyper, {{“100”, “101”}, {"100", "110"}, {"101", "111"}, {"110", "111"}})
subhyper := Graph 13: an undirected unweighted graph with 4 vertices and 4 edge(s) (10.53)
```

```
> Vertices(subhyper)
[“100”, “101”, “110”, “111”] (10.54)
```

```
> Edges(subhyper)
{{“100”, “101”}, {"100", "110"}, {"101", "111"}, {"110", "111"}} (10.55)
```

The second method for creating subgraphs is the **InducedSubgraph** command. This command expects a graph and a set (or list) of vertices of the graph. It returns the graph induced by the given vertices, that is, the graph consisting of the given vertices and all the edges from the original graph with endpoints in the set of vertices.

```
> prism := SpecialGraphs[PrismGraph](6)
prism := Graph 14: an undirected unweighted graph with 12 vertices and 18 edge(s) (10.56)
```

```
> subprism := InducedSubgraph(prism, {7, 8, 9, 10, 11, 12})
subprism := Graph 15: an undirected unweighted graph with 6 vertices and 6 edge(s) (10.57)
```

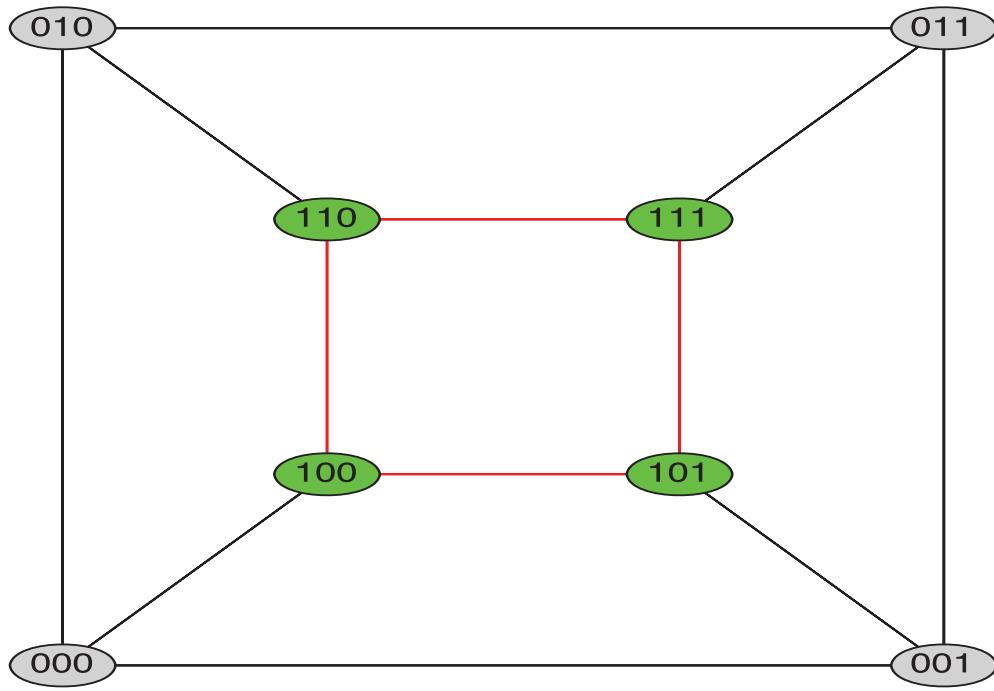
```
> Vertices(subprism)
[7, 8, 9, 10, 11, 12] (10.58)
```

```
> Edges(subprism)
{{7, 8}, {7, 12}, {8, 9}, {9, 10}, {10, 11}, {11, 12}} (10.59)
```

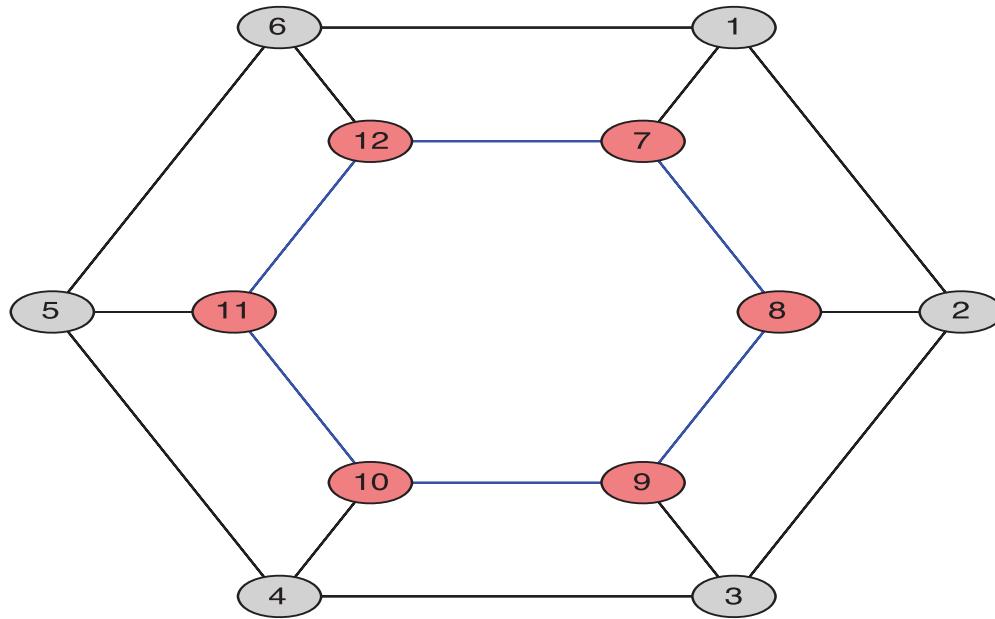
Maple also provides the command **HighlightSubgraph** to help visualize the structure of a subgraph as part of the original graph. By default, the vertices of the subgraph are set to green and the edges to red. Those color choices can be changed by passing colors to the command.

```
> HighlightSubgraph(hyper, subhyper)
```

```
> DrawGraph(hyper)
```



```
> HighlightSubgraph(prism, subprism, blue, "LightCoral")
> DrawGraph(prism)
```



Deleting Vertices and Edges

Subgraphs can also be produced by deleting vertices or edges. The **DeleteVertex** and **DeleteEdge** commands were described in the previous section, but are worth revisiting. The **DeleteVertex** command takes two arguments: a graph and a vertex or list of vertices. It returns a new graph with the vertex or vertices and all incident edges removed. Here, we highlight in green the subgraph of the complete graph K_4 that is obtained by deleting a vertex.

- > *ExDeleteVStart* := *CompleteGraph*(5)

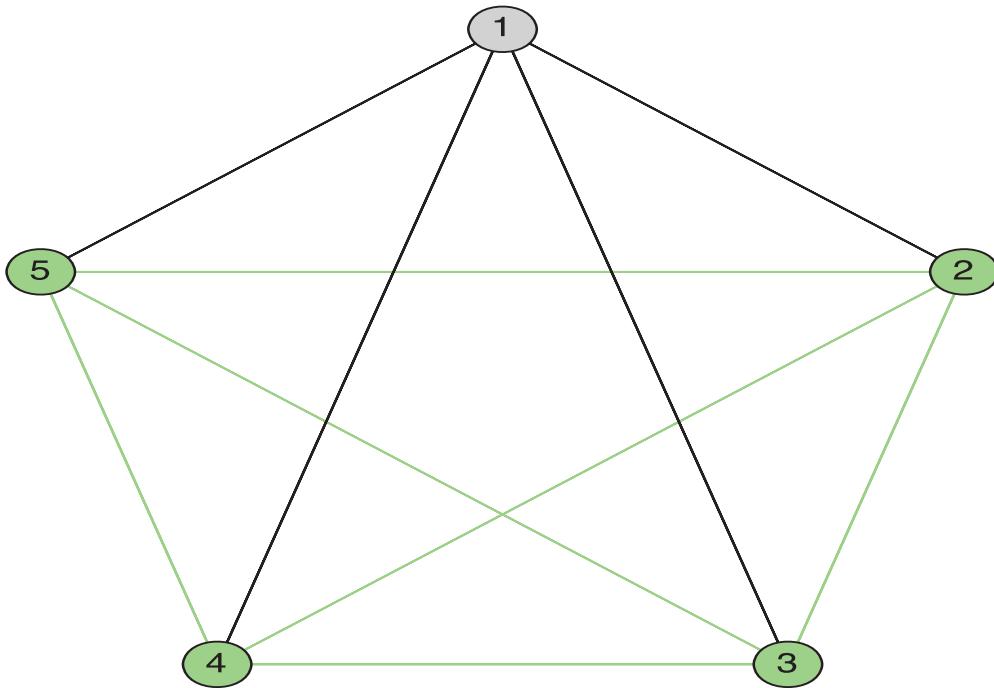
ExDeleteVStart := Graph 16: an undirected unweighted graph with 5 vertices
 and 10 edge(s) (10.60)

- > *ExDeletedV* := *DeleteVertex*(*ExDeleteVStart*, 1)

ExDeletedV := Graph 17: an undirected unweighted graph with 4 vertices and 6 edge(s) (10.61)

- > *HighlightSubgraph*(*ExDeleteVStart*, *ExDeletedV*, "LightGreen", "LightGreen")

- > *DrawGraph*(*ExDeleteVStart*)



DeleteEdge also takes two arguments, a name of an undirected graph and an edge or a set of edges. The set of edges can also be specified as a **Trail**. Recall from above that a **Trail** is a way to specify a set of edges by simply listing all of the vertices, in the order that the sequence of edges pass through them. For example, we can specify the outer ring of K_5 as follows:

- > *Trail*(1, 2, 3, 4, 5, 1)

Trail(1, 2, 3, 4, 5, 1) (10.62)

Note that the **Trail** command does not seem to evaluate. This is because the command is inert and only operates as a part of another command. We delete these edges from a complete graph.

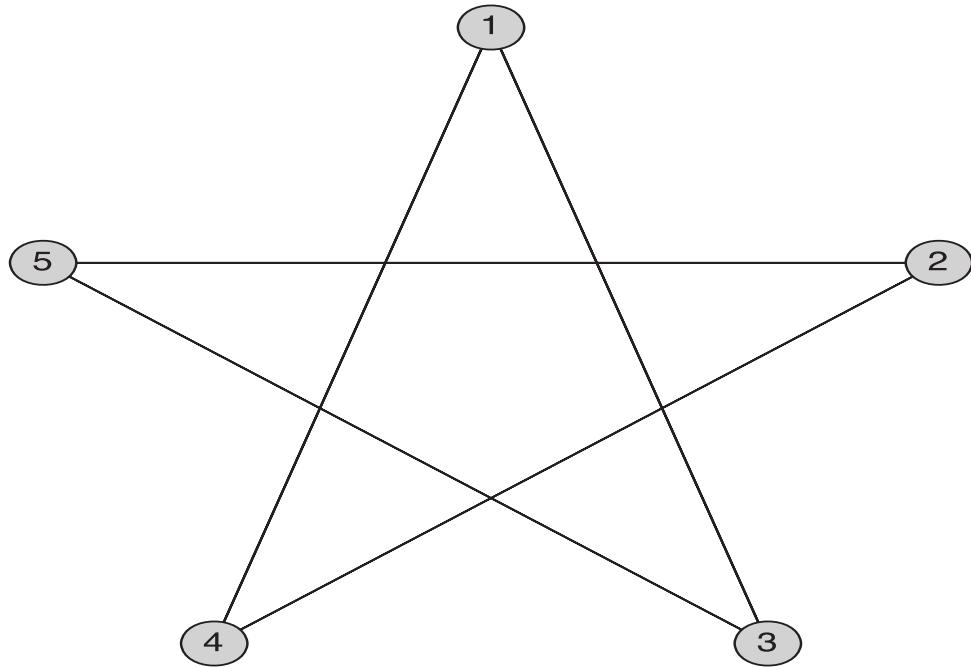
- > *ExDeleteE* := *CompleteGraph*(5)

ExDeleteE := Graph 18: an undirected unweighted graph with 5 vertices and 10 edge(s) (10.63)

- > *DeleteEdge*(*ExDeleteE*, *Trail*(1, 2, 3, 4, 5, 1))

Graph 18: an undirected unweighted graph with 5 vertices and 5 edge(s) (10.64)

> *DrawGraph(ExDeleteE)*



Observe that the **DeleteEdge** command modified its argument, as opposed to the **DeleteVertex** command which did not. To prevent modification of the graph, you can give the **inplace=false** option to **DeleteEdge**.

Adding Vertices and Edges

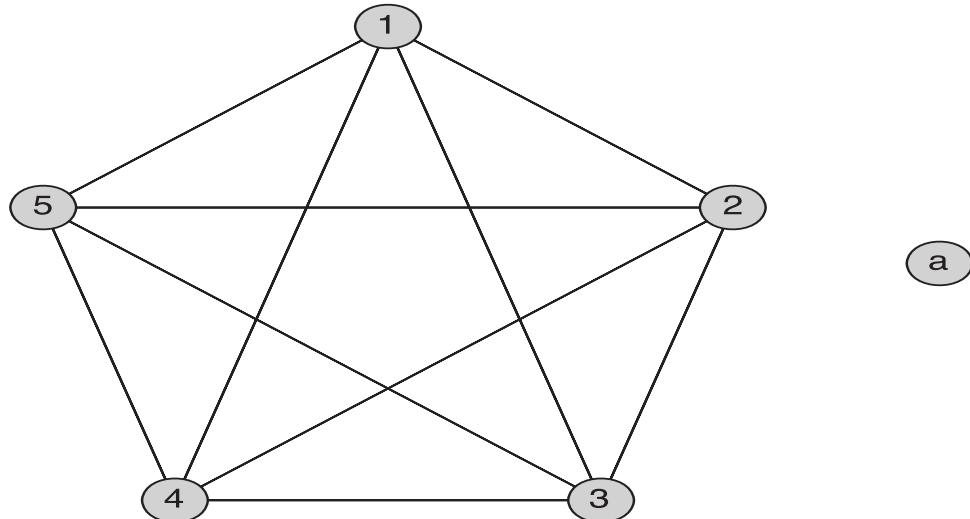
The commands for adding vertices and edges have very similar forms. **AddVertex** accepts a graph and either a vertex or a list of vertices to add to the graph. Again, the original is not modified.

> *ExAddV := AddVertex(CompleteGraph(5), "a")*

ExAddV := Graph 19: an undirected unweighted graph with 6 vertices and 10 edge(s)

(10.65)

> *DrawGraph(ExAddV)*

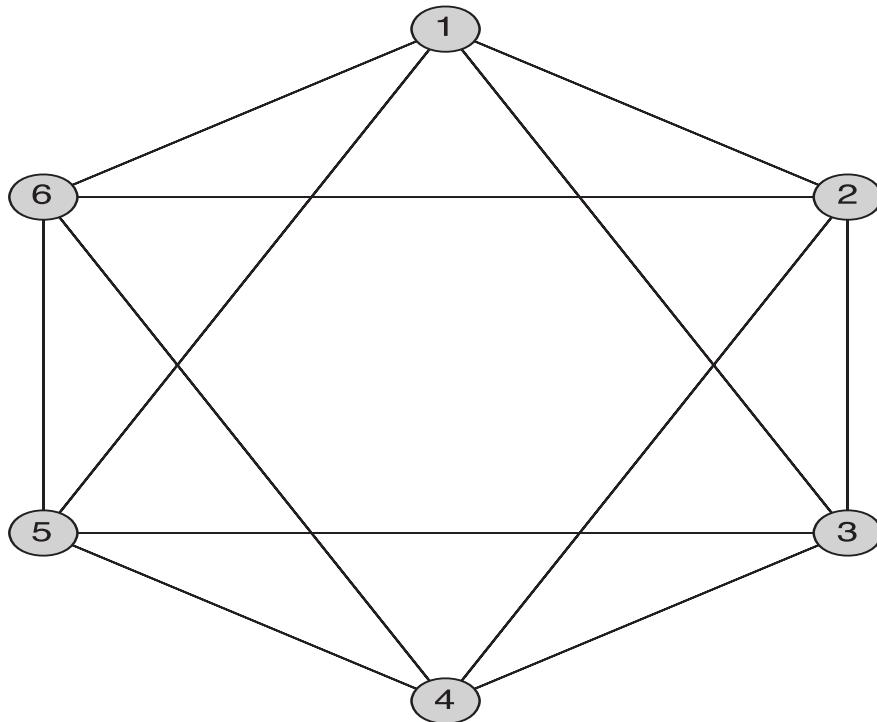


AddEdge acts on undirected graphs and accepts one edge, a set of edges, or a trail. Note that without the **inplace=false** option, this command will alter the original graph.

```
> ExAddE := CycleGraph(6)
ExAddE := Graph 20: an undirected unweighted graph with 6 vertices and 6 edge(s) (10.66)
```

```
> AddEdge(ExAddE, {{1,3},{1,5},{2,4},{2,6},{3,5},{4,6}})
Graph 20: an undirected unweighted graph with 6 vertices and 12 edge(s) (10.67)
```

```
> DrawGraph(ExAddE)
```



DeleteEdge and **AddEdge** apply only for undirected graphs. For directed graphs, use the commands **DeleteArc** and **AddArc**. The syntax and behavior of these commands are exactly the same as their undirected counterparts, although keep in mind that directed edges are represented as 2-element lists, rather than sets.

Edge Contraction

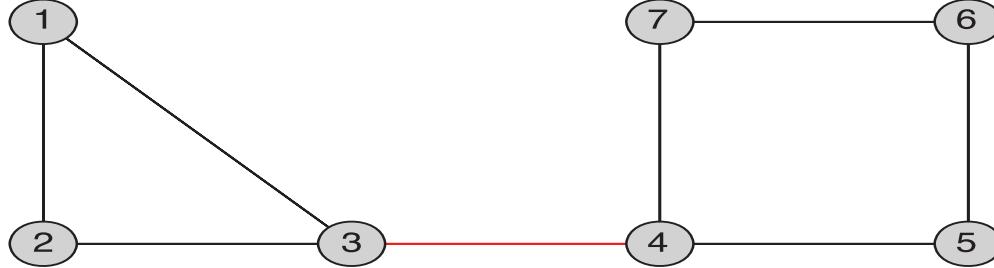
Recall that an edge contraction for an edge e with endpoints u and v consists of deleting the edge, merging u and v into a new vertex w , and preserving all edges (other than e) which had u or v as an endpoint by setting w as the new endpoint. As an illustration, consider the following graph:

```
> ExContraction := Graph({{1,2},{1,3},{2,3},{3,4},{4,5},{4,7},
{5,6},{6,7}})
ExContraction := Graph 21: an undirected unweighted graph with 7 vertices and 8 edge(s) (10.68)
```

```

> SetVertexPositions(ExContraction, [[0, 1], [0, 0], [1, 0], [2, 0], [3, 0],
[3, 1], [2, 1]])
> HighlightEdges(ExContraction, {3, 4})
> DrawGraph(ExContraction)

```



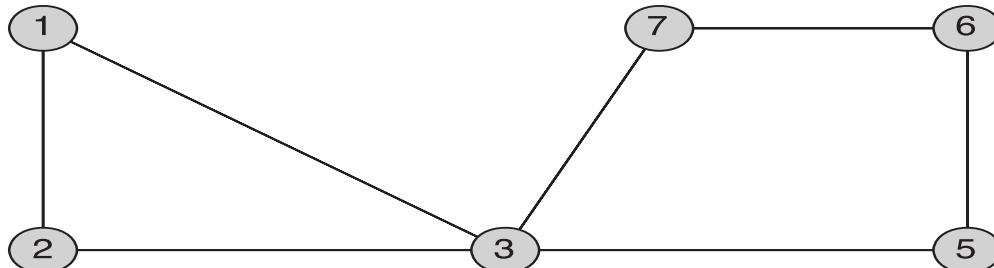
We will perform an edge contraction on the edge $\{3, 4\}$, which is highlighted in the image. The **Contract** command takes as arguments the name of the graph and an edge and performs the edge contraction. Note that the merged vertices will be represented by one of the vertices in the original pair, in this case vertex 3 will represent the pair. Also note that this command does not modify the original graph, which means that any style specifications or attributes will need to be reset.

```

> ExContracted := Contract(ExContraction, {3, 4})
ExContracted := Graph 22: an undirected unweighted graph with 6 vertices and 7 edge(s)
(10.69)

> SetVertexPositions(ExContracted, [[0, 1], [0, 0], [1.5, 0], [3, 0], [3, 1], [2, 1]])
> DrawGraph(ExContracted)

```



Unions and Complements of Graphs

Recall that the union of two graphs is the graph obtained by taking the union of the sets of vertices and the sets of edges from the two graphs.

As an example, we will “fill in” a prism graph by computing the union of the prism with the complete graph on the vertices in the inner ring.

```

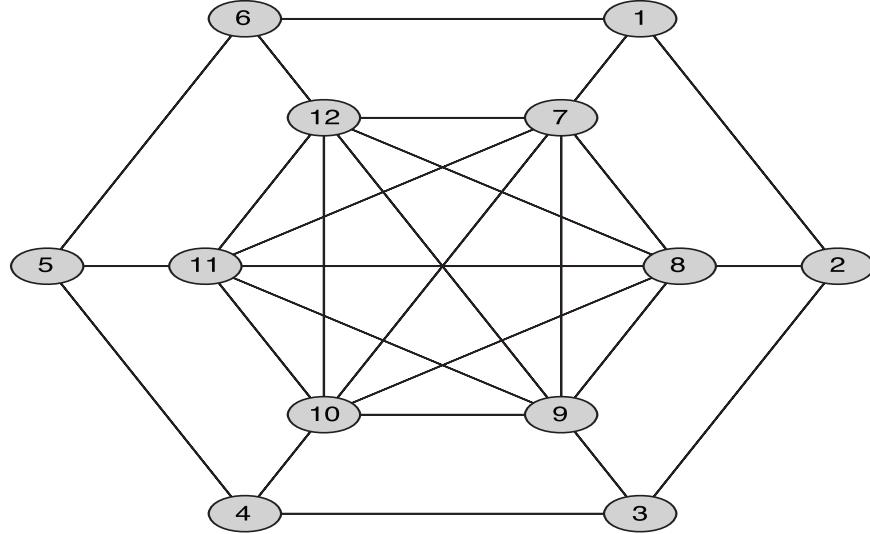
> unionA := SpecialGraphs[PrismGraph](6):
> unionB := CompleteGraph([7, 8, 9, 10, 11, 12]):
> unionAB := GraphUnion(unionA, unionB):

```

To get this graph to display in the way we described it, as the prism filled in with the complete graph on the inner set of vertices, we need to set the positions of the vertices. Otherwise, Maple will simply draw it in the circular style. We can access the vertex locations used in the display of the prism and impose those locations on this graph.

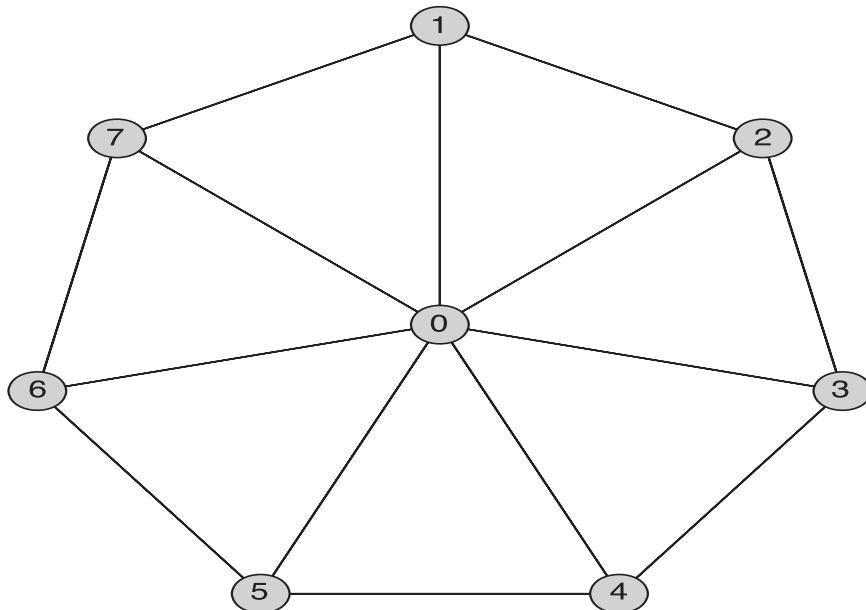
```
> SetVertexPositions(unionAB, GetVertexPositions(unionA))
```

```
> DrawGraph(unionAB)
```

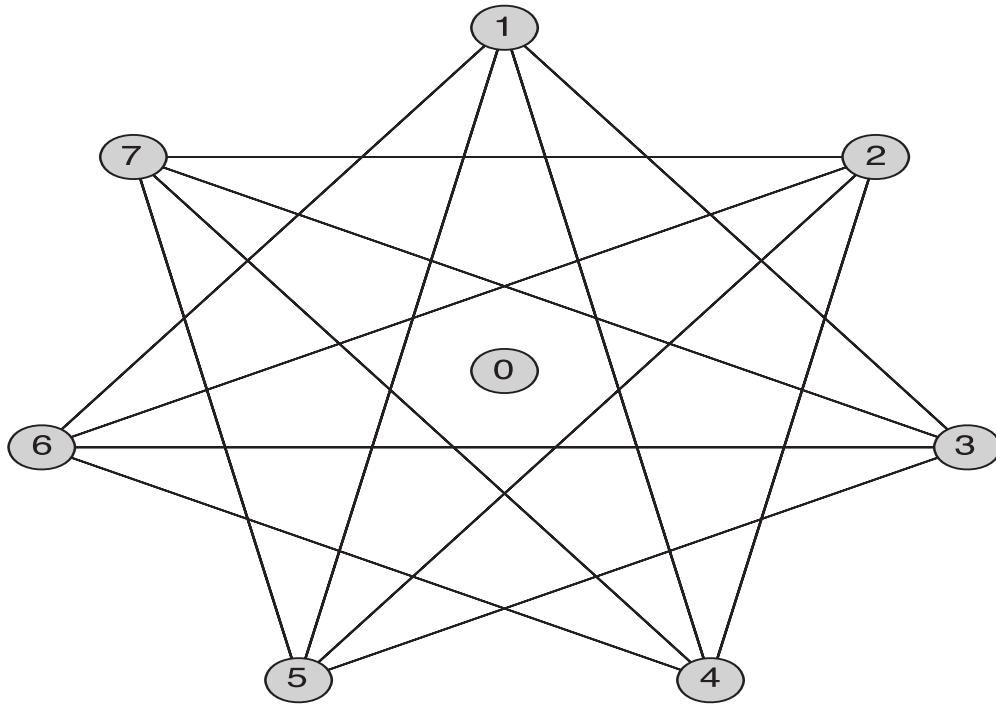


Finally, we consider graph complements, described in Exercise 61 in Section 10.2. The complement, \bar{G} , of a graph G is the graph whose vertex set is the same as that of G , but whose edge set is the set of all pairs of G that have no edge between them. In other words, if G has n vertices, then the edge set of \bar{G} is the complement of the edge set of G relative to K_n , the complete graph on n vertices. Maple has a command to compute the complement of a graph: **GraphComplement**.

```
> DrawGraph(SpecialGraphs[WheelGraph](7))
```



```
> DrawGraph(GraphComplement(SpecialGraphs[WheelGraph](7)))
```



10.3 Representing Graphs and Graph Isomorphism

In this section, we will see how to represent graphs in terms of adjacency lists, adjacency matrices, and incidence matrices. We will then use the adjacency matrix representation to help determine whether two graphs are isomorphic.

Adjacency Lists

Recall that a representation of a graph as an adjacency list consists of the lists of neighbors of each vertex.

In order to define a graph in Maple using an adjacency list, you apply the **Graph** command to a list of sets or a list of lists. For example,

```
> Graph([{2,3},{1,3,4},{1,2},{2,5},{4}])
```

Graph 23: an undirected unweighted graph with 5 vertices and 5 edge(s)

(10.70)

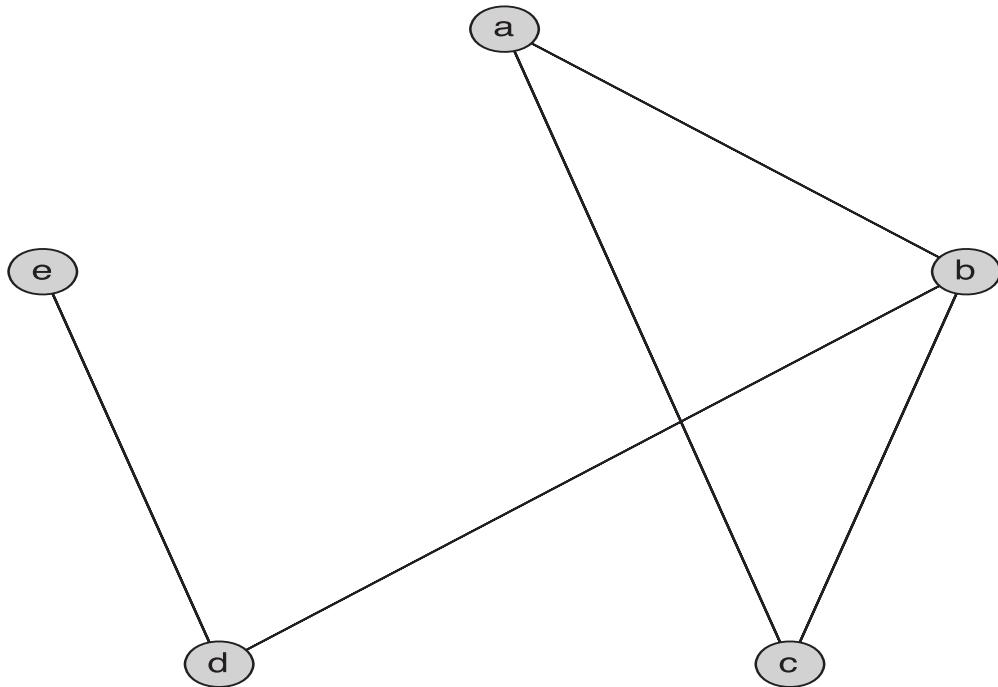
indicates that vertex 1 is incident to vertices 2 and 3; vertex 2 is incident to vertices 1, 3, and 4; vertex 3 is incident to vertices 1 and 2; and so on.

Note that graphs constructed this way are undirected or directed depending on the content of the adjacency lists. For example, in the previous example, removing 1 from the second set would make vertex 1 incident to vertex 2, but not *vice versa*. This would cause Maple to consider the graph directed.

Also note that the vertex labels can be specified by providing a list of the labels, but the adjacency list always needs to consist of integers corresponding to the index of the vertex.

```
> adjacencyEx := Graph(["a", "b", "c", "d", "e"],
  [{2, 3}, {1, 3, 4}, {1, 2}, {2, 5}, {4}])
adjacencyEx := Graph 24: an undirected unweighted graph with 5 vertices and 5 edge(s)
(10.71)

> DrawGraph(adjacencyEx)
```



In the other direction, we can use the Maple command **Departures**. For a directed graph, **Departures(G,v)** returns a list of all of the terminal vertices for edges whose initial vertex is v . If G is undirected, the same command returns all of v 's neighbors, or you can use the **Neighbors** command. Omitting the second argument returns a list of lists, where each sublist consists of the vertices adjacent to the vertex in the corresponding position of the graph's list of vertices.

```
> Vertices(adjacencyEx)
["a", "b", "c", "d", "e"]
(10.72)
```

```
> Departures(adjacencyEx)
[[["b", "c"], ["a", "c", "d"], ["a", "b"], ["b", "e"], ["d"]]]
(10.73)
```

Interpreting the output from **Departures** requires knowing the order of the vertices. We will write a procedure that, given a graph G , will produce more descriptive output. In order for the procedure to work with both undirected and directed graphs, we will use the Maple command **Departures**. Our procedure will also allow for loops by checking the “loop” attribute and listing the vertex as adjacent to itself when the “loop” attribute is true.

```

1 AdjacencyLists := proc(G :: Graph)
2   local v, AList;
3   uses GraphTheory;
4   for v in Vertices(G) do
5     AList := Departures(G, v);
6     if GetVertexAttribute(G, v, "loop") then
7       AList := [v, op(AList)];
8     end if;
9     printf("Vertex %a is adjacent to %a\n", v, AList);
10    end do;
11  end proc;

```

> *AdjacencyLists*(*adjacencyEx*)

```

Vertex "a" is adjacent to ["b", "c"]
Vertex "b" is adjacent to ["a", "c", "d"]
Vertex "c" is adjacent to ["a", "b"]
Vertex "d" is adjacent to ["b", "e"]
Vertex "e" is adjacent to ["d"]

```

> *AdjacencyLists*(*Exercise7*)

```

Vertex "a" is adjacent to ["b"]
Vertex "b" is adjacent to ["e"]
Vertex "c" is adjacent to ["c", "b", "d"]
Vertex "d" is adjacent to ["c"]
Vertex "e" is adjacent to ["e", "a", "d"]

```

Adjacency Matrices

The adjacency matrix of a graph G with n vertices is the $n \times n$ matrix whose (i, j) entry is 1 if there is an edge from vertex i to vertex j and 0 if not. As with adjacency lists, you can define a graph by passing an adjacency matrix to the **Graph** command.

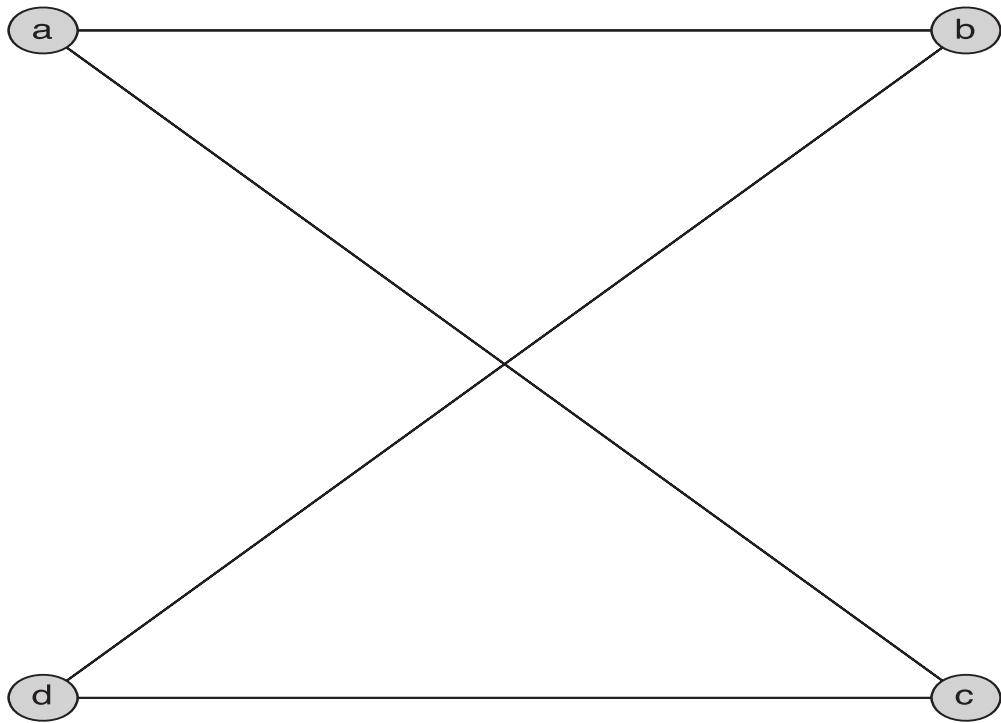
As an example, we reproduce Example 4 in Section 10.3.

> *exAdjM* := *Matrix*([[0, 1, 1, 0], [1, 0, 0, 1], [1, 0, 0, 1], [0, 1, 1, 0]])

$$exAdjM := \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \quad (10.74)$$

> *exAdjMGraph* := *Graph*(["a", "b", "c", "d"], *exAdjM*,
vertexpositions = [[0, 1], [1, 1], [1, 0], [0, 0]]):

> *DrawGraph*(*exAdjMGraph*)



Notice that this is the same graph as is produced in the textbook.

Maple also provides a command, **AdjacencyMatrix**, for computing the adjacency matrix of a simple graph.

> *AdjacencyMatrix(SpecialGraphs[WheelGraph](7))*

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad (10.75)$$

Incidence Matrices

The third representation of graphs we are considering is the incidence matrix. For a graph G with n vertices and m edges, the associated incidence matrix is the $n \times m$ matrix whose (i, j) entry is 1 if vertex i is an endpoint of edge j .

Unlike the other representations, Maple does not provide support for creating a graph from an incidence matrix. We will write a procedure to do so, at least for simple graphs.

To write this procedure, recall that the columns of the incidence matrix correspond to the edges of the graph. Therefore, we will use the columns to produce the list of edges. For each column, check

each entry and add the row index to a set representing the edge. Assuming the incidence matrix is properly formed, each column will have only two 1s so each column will produce a two-element set representing an edge. The set of all of these forms the set of edges, which we can pass to the **Graph** command.

```

1 GraphFromIncidence := proc (M::Matrix)
2   local r, c, e, E;
3   E := {};
4   for c from 1 to LinearAlgebra[ColumnDimension](M) do
5     e := {};
6     for r from 1 to LinearAlgebra[RowDimension](M) do
7       if M[r, c] = 1 then
8         e := e union {r};
9       end if;
10      end do;
11      E := E union {e};
12    end do;
13    return GraphTheory[Graph](E);
14  end proc;
```

As an example of our procedure, we reverse Example 6 in Section 10.3 and use the incidence matrix given in the solution in order to reproduce the graph.

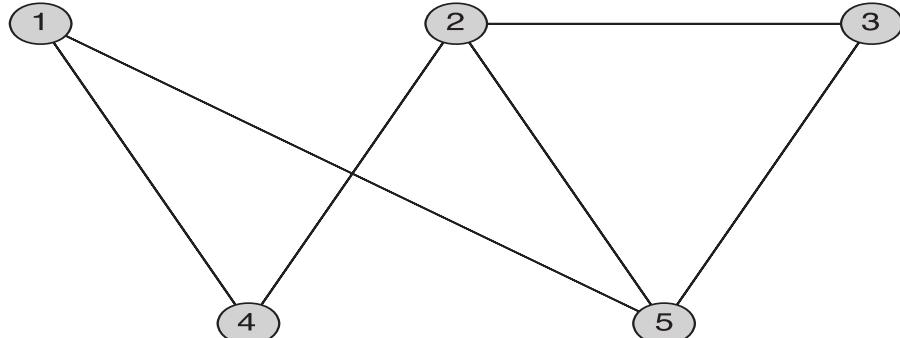
```

> exIncMatrix := Matrix([[1, 1, 0, 0, 0, 0], [0, 0, 1, 1, 0, 1], [0, 0, 0, 0, 1, 1],
[1, 0, 1, 0, 0, 0], [0, 1, 0, 1, 1, 0]])
exIncMatrix := 
$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$
 (10.76)
```

> exIncMGraph := GraphFromIncidence(exIncMatrix)
exIncMGraph := Graph 25: an undirected unweighted graph with 5 vertices and 6 edge(s) (10.77)

```

> SetVertexPositions(exIncMGraph, [[0, 1], [1, 1], [2, 1], [0.5, 0], [1.5, 0]])
> DrawGraph(exIncMGraph)
```



On the other hand, Maple does provide a command for computing the incidence matrix for a graph: **IncidenceMatrix**.

> *IncidenceMatrix(exIncMGraph)*

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix} \quad (10.78)$$

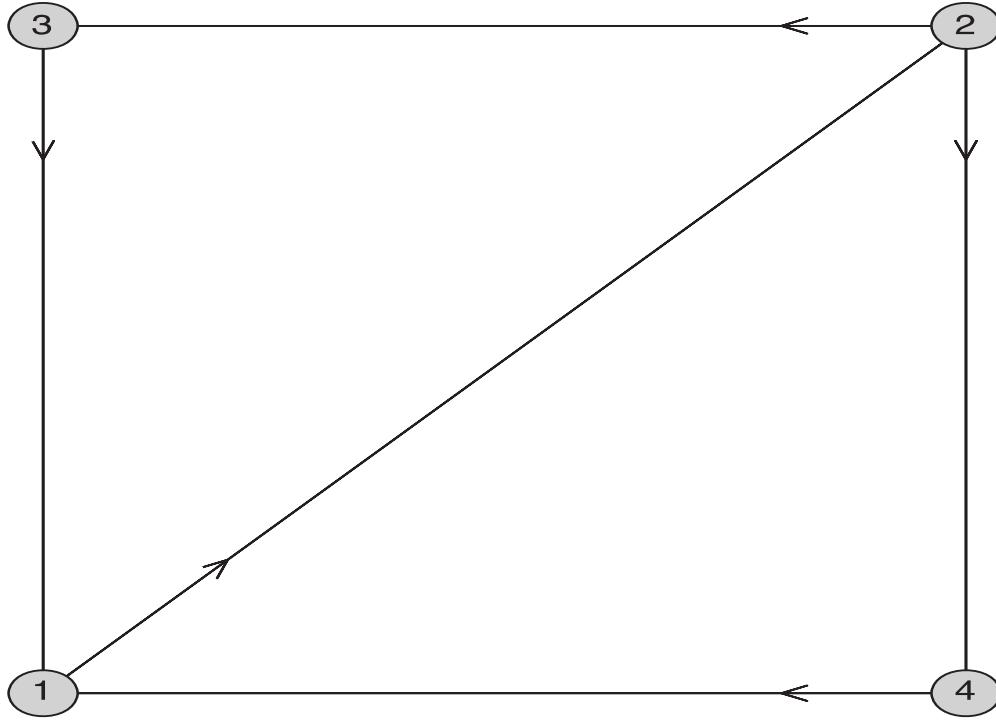
For a directed graph, the **IncidenceMatrix** command returns a matrix with a 1 in position (i, j) indicating that the vertex i is the head of edge j and an entry of -1 indicating that the vertex is the tail of the edge.

> *directedIncidence := Digraph(Trail(1, 2, 3, 1), Trail(2, 4, 1))*

directedIncidence := Graph 26: a directed unweighted graph with 4 vertices and 5 arc(s) (10.79)

> *SetVertexPositions(directedIncidence, [[0, 0], [1, 1], [0, 1], [1, 0]])*

> *DrawGraph(directedIncidence)*



> *IncidenceMatrix(directedIncidence)*

$$\begin{bmatrix} -1 & 0 & 0 & 1 & 1 \\ 1 & -1 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \end{bmatrix} \quad (10.80)$$

Isomorphism of Graphs

We conclude this chapter with a brief discussion of isomorphisms of graphs and graph invariants. Determining whether two graphs are isomorphic is a difficult problem. The naive approach (exhaustively checking each possible mapping) can require exponential time.

Graph invariants are useful tools for confirming that two graphs are not isomorphic. While there is no complete collection of graph invariants that will definitively conclude whether two graphs are or are not isomorphic, they can, for many pairs of graphs, quickly demonstrate the impossibility of an isomorphism. We will now create a procedure that will check some of the basic invariants that we have seen in this chapter: number of vertices, number of edges, whether the graph is directed, and whether it is bipartite. We also introduce another invariant: the degree sequence.

For a graph G , the degree sequence is the list of the degrees of the vertices of the graph sorted in ascending order. The Maple command **DegreeSequence** returns a list of the degrees of the vertices of a graph, listed in order of the vertices. Since this depends on the order in which Maple stores the vertices, it is not an invariant. However, applying the **sort** command to the result of the **DegreeSequence** command returns the degree sequence for the graph, as we defined it, which is an invariant.

The procedure defined below checks, one at a time, the invariants we have described. If any of the invariants indicate that the graphs are not isomorphic, the procedure prints a statement to that effect.

```
1 CheckInvariants := proc (G1 :: Graph, G2 :: Graph)
2   local notIso;
3   uses GraphTheory;
4   notIso := false;
5   if not (nops(Vertices(G1)) = nops(Vertices(G2))) then
6     notIso := true;
7     print("Different numbers of vertices");
8   end if;
9   if not (nops(Edges(G1)) = nops(Edges(G2))) then
10    notIso := true;
11    print("Different numbers of edges");
12  end if;
13  if IsDirected(G1) <> IsDirected(G2) then
14    notIso := true;
15    print("One is directed, one is undirected");
16  end if;
17  if IsBipartite(G1) <> IsBipartite(G2) then
18    notIso := true;
19    print("One is bipartite and the other is not");
20  end if;
21  if sort(DegreeSequence(G1)) <> sort(DegreeSequence(G2)) then
22    notIso := true;
23    print("Degree sequences do not match");
24  end if;
25  if notIso then
26    print("The graphs are not isomorphic");
27  else
```

```

28     print ("The graphs MAY be isomorphic") ;
29   end if ;
30 end proc;

```

> *CheckInvariants(directedIncidence, exIncMGraph)*
 “Different numbers of vertices”
 “Different numbers of edges”
 “One is directed, one is undirected”
 “Degree sequences do not match”
 “The graphs are not isomorphic” (10.81)

> *CheckInvariants(SpecialGraphs[HypercubeGraph](3),*
SpecialGraphs[PrismGraph](4))
 “The graphs MAY be isomorphic” (10.82)

Maple provides a command, **IsIsomorphic**, for definitively determining whether or not two graphs are isomorphic. For weighted graphs, the weights are ignored. The **IsIsomorphic** command accepts three arguments. The first two arguments are the two graphs to be compared. The third, optional, argument is a variable name in which the isomorphism, if it exists, is to be stored.

> *IsIsomorphic(SpecialGraphs[HypercubeGraph](3),*
SpecialGraphs[PrismGraph](4),'hyperprismiso')
 true (10.83)

> *hyperprismiso*
 [{"000" = 1, "001" = 2, "010" = 4, "011" = 3, "100" = 5, "101" = 6,
 "110" = 8, "111" = 7}] (10.84)

We can make an isomorphism visible by coloring corresponding vertices the same color. To choose the colors, we will make use of Maple’s built-in color schemes. The **ColorTools** package includes definitions of color palettes—predefined lists of colors. The **PaletteNames** command will produce a list of all the available palettes.

> *ColorTools[PaletteNames]()*
 [{"Niagara", "Nautical", "Spring", "OldPlots", "Mono", "Dalton",
 "Executive", "Bright", "Patchwork", "CSS", "HTML", "MapleV",
 "X11", "Resene", "Generic"}] (10.85)

We will make use of the Spring palette. You obtain a palette by applying the **GetPalette** command to the string naming the palette.

> *springcolors := ColorTools[GetPalette]("spring")*
springcolors := <Palette Spring: Blue Rose YellowGreen
BlueGreen Violet Cobalt Yellow PurpleRed GreenBlue
PaleGreen Orange Purple Green SeaBlue PaleYellow
PaleBlueGreen> (10.86)

With the palette assigned to a name, individual colors can be obtained through the usual selection operator, as if it were a list.

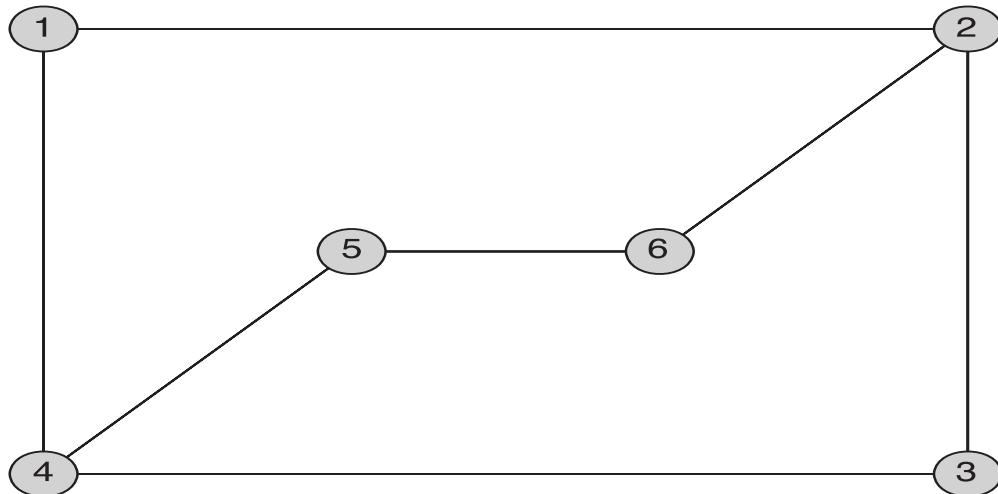
```
> springcolors[3]
⟨RGB : YellowGreen⟩ (10.87)
```

Previously, we saw how to change the color of vertices with attributes in order to explain more of the inner workings of graphs in Maple. Here, we will use the higher-level command **HighlightVertex**, which can be directly applied to these high-level color objects and can be used successively to assign different colors to different vertices.

We reproduce the graphs in Figure 12 of Section 10.3 of the textbook to illustrate how to draw the graphs with an isomorphism indicated by color-coding the vertices.

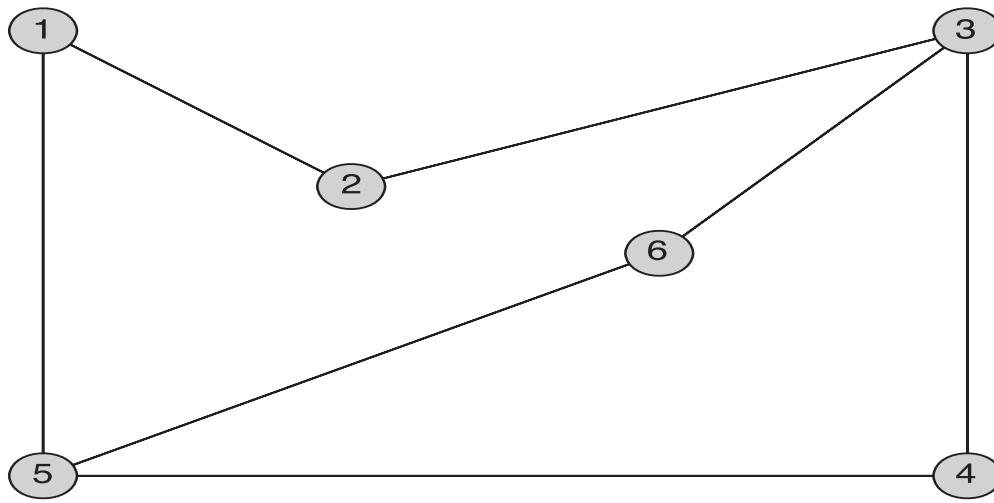
```
> figure12G := Graph({{1,2},{1,4},{2,3},{2,6},{3,4},{4,5},{5,6}},  
vertexpositions = [[0,2],[3,2],[3,0],[0,0],[1,1],[2,1]])  
figure12G := Graph 27: an undirected unweighted graph with 6 vertices and 7 edge(s) (10.88)
```

> DrawGraph(figure12G)



```
> figure12H := Graph({{1,2},{1,5},{2,3},{3,4},{3,6},{4,5},{5,6}},  
vertexpositions = [[0,2],[1,1.3],[3,2],[3,0],[0,0],[2,1]])  
figure12H := Graph 28: an undirected unweighted graph with 6 vertices and 7 edge(s) (10.89)
```

> DrawGraph(figure12H)



Applying **IsIsomorphic** confirms that the graphs are isomorphic and records the isomorphism in the name **figure12iso**.

```
> IsIsomorphic(figure12G,figure12H,'figure12iso')
true
```

(10.90)

```
> figure12iso
[1 = 4, 2 = 3, 3 = 6, 4 = 5, 5 = 1, 6 = 2]
```

(10.91)

We can use the **op** command to extract the information from this list of equations. Recall that, when applied to a list, set, or other structure, **op** produces the sequence of elements of the structure. But **op** can also accept an integer as its first argument, in which case it will return the element of the second argument in that position.

```
> op(3,figure12iso)
3 = 6
```

(10.92)

Note that this is similar to the usual selection operation.

```
> figure12iso[3]
3 = 6
```

(10.93)

Note that applying **op** to the equation yields the two vertex names in sequence.

```
> op(3 = 6)
3, 6
```

(10.94)

While we cannot directly apply the selection operation to the equation, we could use **op** to transform the equation into the sequence of its left- and right-hand sides and then apply selection. Using an integer as the first argument to **op** does this in one step.

```
> op(2,3 = 6)
6
```

(10.95)

With a list as the first argument to **op**, we can access the right-hand side of the 3rd equation in one command.

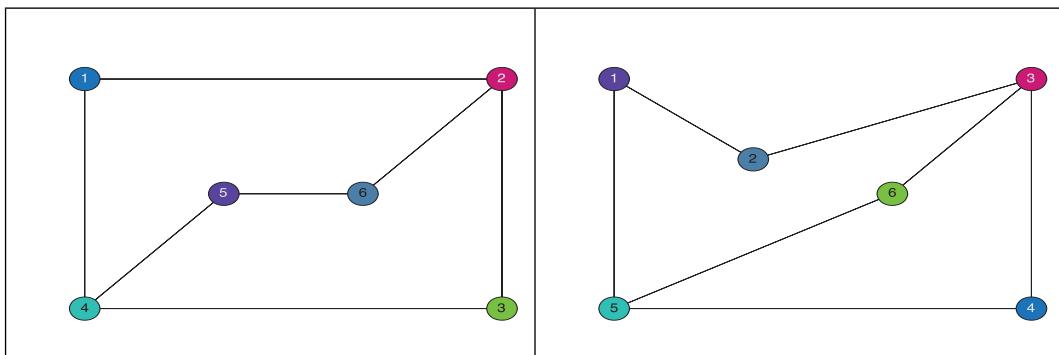
```
> op([3,2],figure12iso)
6
```

(10.96)

This is how we will assign colors to the vertices to illustrate the isomorphism. For each position in the list representing the isomorphism, we will assign the color in the corresponding position in the palette to the vertices on either side of the equation in their respective graph. For example, color 3 will be assigned to vertex 3 in the first graph and vertex 6 in the second. We execute the for loop below and then display the graphs. Note that **DrawGraph** can be applied to a list or set of graphs to display the graphs in a matrix.

```
> for i from 1 to nops(figure12iso) do
    HighlightVertex(figure12G, op([i,1],figure12iso), springcolors[i]);
    HighlightVertex(figure12H, op([i,2],figure12iso), springcolors[i]);
end do;

> DrawGraph ([figure12G,figure12H])
```



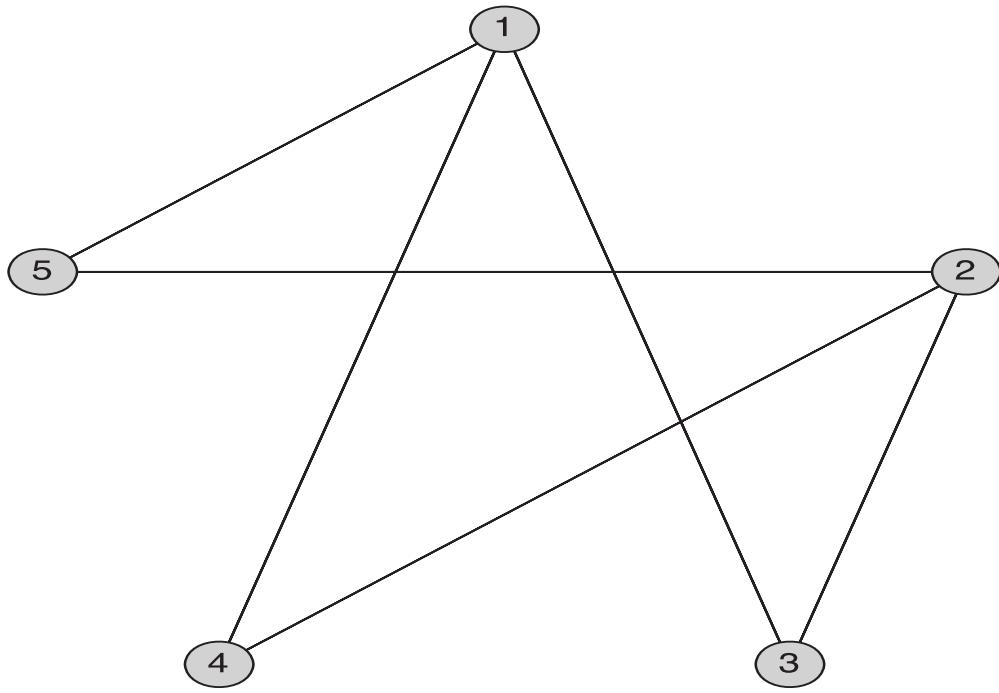
10.4 Connectivity

Maple provides a number of commands related to connectivity of graphs.

Connectedness in Undirected Graphs

The first such command that we consider is the **IsConnected** command. This command takes one argument, the name of the graph, and returns true or false. As an example, consider the complete bipartite graph $K_{2,3}$ and its complement.

```
> DrawGraph(CompleteGraph(2,3),style = circle)
```

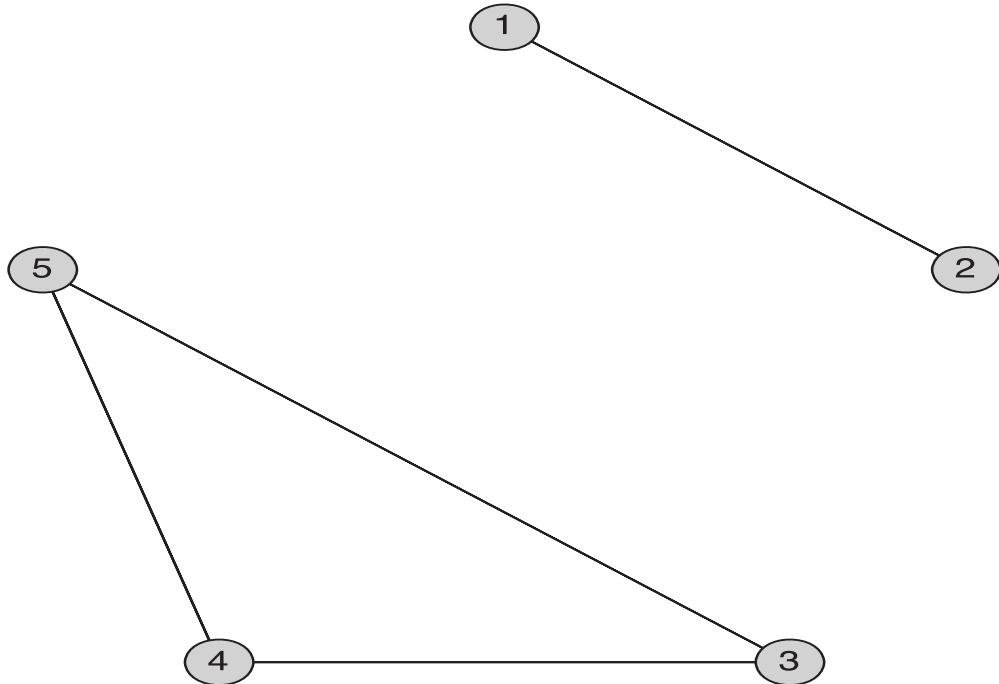


> *IsConnected(CompleteGraph(2, 3))*

true

(10.97)

> *DrawGraph(GraphComplement(CompleteGraph(2, 3)), style = circle)*



> *IsConnected(GraphComplement(CompleteGraph(2, 3)))*

false

(10.98)

Connectivity and Vertices

In addition to testing whether a graph is connected or not, Maple also has commands for determining which vertices are cut vertices (which Maple refers to as articulation points) and for calculating both the vertex and edge connectivity.

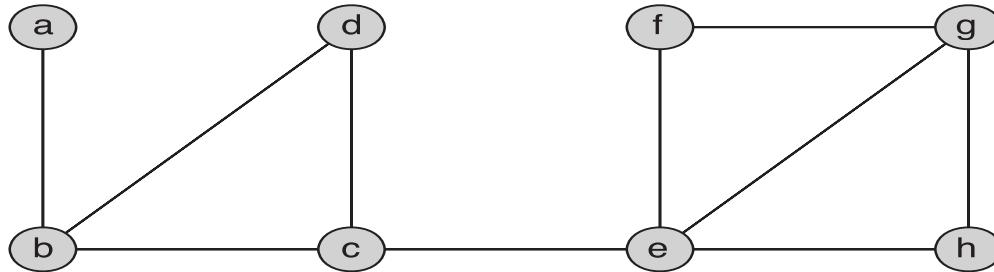
First, let us recreate two of the examples from Figure 4 in Section 10.4, namely G_1 and G_3 .

```
> Figure4G1 := Graph({{{"a", "b"}, {"b", "c"}, {"b", "d"}, {"c", "d"}, {"c", "e"}, {"e", "f"}, {"e", "g"}, {"e", "h"}, {"f", "g"}, {"g", "h}}})
```

Figure4G1 := Graph 29: an undirected unweighted graph with 8 vertices and 10 edge(s) (10.99)

```
> SetVertexPositions (Figure4G1, [[0, 1], [0, 0], [1, 0], [1, 1], [2, 0], [2, 1],  
[3, 1], [3, 0]])
```

> *DrawGraph*(Figure4G1)

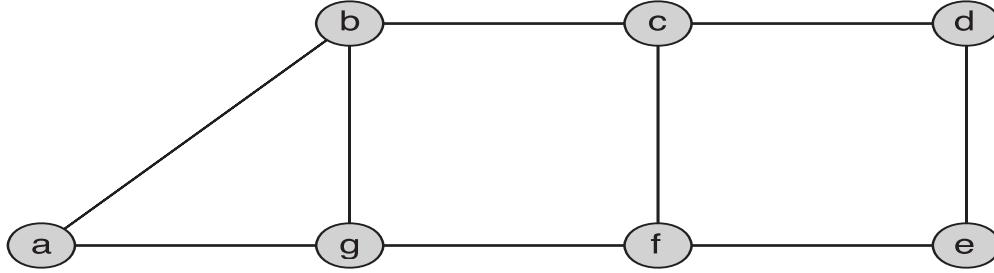


```
> Figure4G3 := Graph({{{"a","b"}, {"a","g"}, {"b","c"}, {"b","g"}, {"c","d"}, {"c","f"}, {"d","e"}, {"e","f"}, {"f","g}}})
```

Figure4G3 := Graph 30: an undirected unweighted graph with 7 vertices and 9 edge(s)
(10.100)

> *SetVertexPositions*(*Figure4G3*, [[0, 0], [1, 1], [2, 1], [3, 1], [3, 0], [2, 0], [1, 0]])

> *DrawGraph*(Figure4G3)



To find the cut vertices (or articulation points), we use the command **ArticulationPoints**. This command takes only one argument, the name of a graph, and returns a list of vertices that are articulation points, that is, vertices which, if removed, would disconnect the graph.

> *ArticulationPoints* (*Figure4G1*)

[“b”, “c”, “e”] **(10.101)**

> *ArticulationPoints*(Figure4G3)
 [] (10.102)

These results indicate that G_1 has three cut vertices while G_3 has none.

In other words, G_3 is nonseparable. Recall that a nonseparable graph will have vertex connectivity $\kappa(G) \geq 2$ and is thus referred to as 2-connected or biconnected, provided it has at least 3 vertices. The Maple command **IsBiconnected** also indicates that **Figure4G3** is nonseparable.

> *IsBiconnected*(Figure4G3)
 true (10.103)

Finally, the **VertexConnectivity** command computes $\kappa(G)$, the minimum number of vertices that must be deleted in order to disconnect a graph. The only argument is the name of the graph.

> *VertexConnectivity*(Figure4G1)
 1 (10.104)

> *VertexConnectivity*(Figure4G3)
 2 (10.105)

Finding a Vertex Cut

We conclude our discussion of vertex connectivity by developing a procedure for determining which sets of vertices in a graph G form a vertex cut of size $\kappa(G)$. That is, we want to find a minimal set of vertices which separate the graph. Maple tells us how many vertices are in such a set, but does not have a command for finding them.

First, we create a command that will test whether or not a given set of vertices is or is not a vertex cut. We do this by simply removing the vertices from the graph with the **DeleteVertex** command and then testing the resulting graph for connectedness with **IsConnected**.

```

1 IsVertexCut := proc (G::Graph, V::list)
2   local H, iscut;
3   uses GraphTheory;
4   H := DeleteVertex (G, V);
5   iscut := not IsConnected (H);
6   return iscut;
7 end proc;
```

We see that, in G_3 , $\{c, f\}$ separates the graph, while $\{a, d\}$ does not.

> *IsVertexCut*(Figure4G3, ["c", "f"])
 true (10.106)

> *IsVertexCut*(Figure4G3, ["a", "d"])
 false (10.107)

We now write the procedure to find all minimal vertex cuts. We will do this by brute force. First, the **Vertices** command produces the list of vertices in the graph. Then, the **choose** command from the

combinat package takes the list of vertices and the value of $\kappa(G)$ and produces a list of all sublists of vertices of that size. Finally, we check each of those sublists with **IsVertexCut** to see which are vertex cuts.

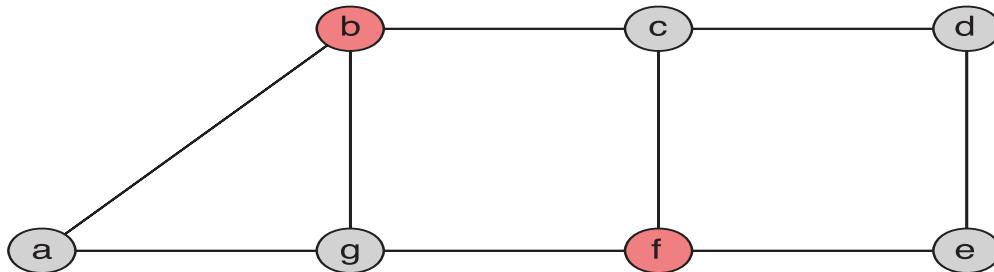
```

1 FindVertexCuts := proc (G::Graph)
2   local k, subLists, testVerts, result;
3   uses GraphTheory;
4   k := VertexConnectivity(G);
5   subLists := combinat[choose](Vertices(G), k);
6   result := [];
7   for testVerts in subLists do
8     if IsVertexCut(G, testVerts) then
9       result := [op(result), testVerts];
10    end if;
11  end do;
12  return result;
13 end proc;
```

We apply this to G_3 to find the possible minimum vertex cuts and then we redraw the graph with one of the choices highlighted.

```

> Figure4G3VCs := FindVertexCuts(Figure4G3)
Figure4G3VCs := [[“b”, “f”], [“b”, “g”], [“c”, “e”], [“c”, “f”],
[“c”, “g”], [“d”, “f”]](10.108)
> HighlightVertex(Figure4G3, Figure4G3VCs[1], “LightCoral”)
> DrawGraph(Figure4G3)
```



Connectivity and Edges

Maple offers similar functionality for edges. Unlike for vertices, however, Maple does not include a command like **ArticulationPoints** that will list all of a graph's bridges (recall that a bridge or a cut edge is an edge whose removal will disconnect the graph). Maple does, however, have a command to test whether an particular edge is a bridge.

The **IsCutSet** command takes two arguments. The first is the name of the graph. The second argument can be a single edge, in which case the function determines whether that edge is a bridge or not. Alternately, the second argument may be a set of edges, in which case the function determines whether or not that set is an edge cut. For example, we see that edge $\{c, e\}$ in G_1 is a bridge.

> *IsCutSet*(Figure4G1, {"c", "e"})
true (10.109)

We can also see that the pair of edges $\{b, c\}$ and $\{f, g\}$ form an edge cut of G_3 .

> *IsCutSet*(Figure4G3, {{"b", "c"}, {"f", "g"}})
true (10.110)

We now use this to create a command analogous to **ArticulationPoints**. This procedure works by checking each edge to see if it is a bridge. We accomplish this with the **select** command, which, when applied to a Boolean-valued function in one argument and a list, returns the list of those elements for which the function returns true. The second argument can also be a set, in which case a set is the result, or any other expression.

```

1 Bridges := proc (G::Graph)
2   uses GraphTheory;
3   select (e -> IsCutSet (G, e), Edges (G) )
4 end proc:
```

We can use this to see that G_1 has two bridges and that G_3 has none.

> *Bridges*(Figure4G1)
{{"a", "b"}, {"c", "e"}} (10.111)

> *Bridges*(Figure4G3)
 \emptyset (10.112)

Finally, Maple will compute $\lambda(G)$, the edge connectivity of the graph, with the command **EdgeConnectivity**. Just like **VertexConnectivity**, the only argument that is accepted is the name of the graph, and the command returns the minimum number of edges that must be deleted in order to disconnect the graph.

We have already seen that G_1 has bridges, and thus has edge connectivity 1.

> *EdgeConnectivity*(Figure4G1)
1 (10.113)

On the other hand, the **Bridges** command verified that G_3 does not have bridges, but it does have an edge cut of size 2.

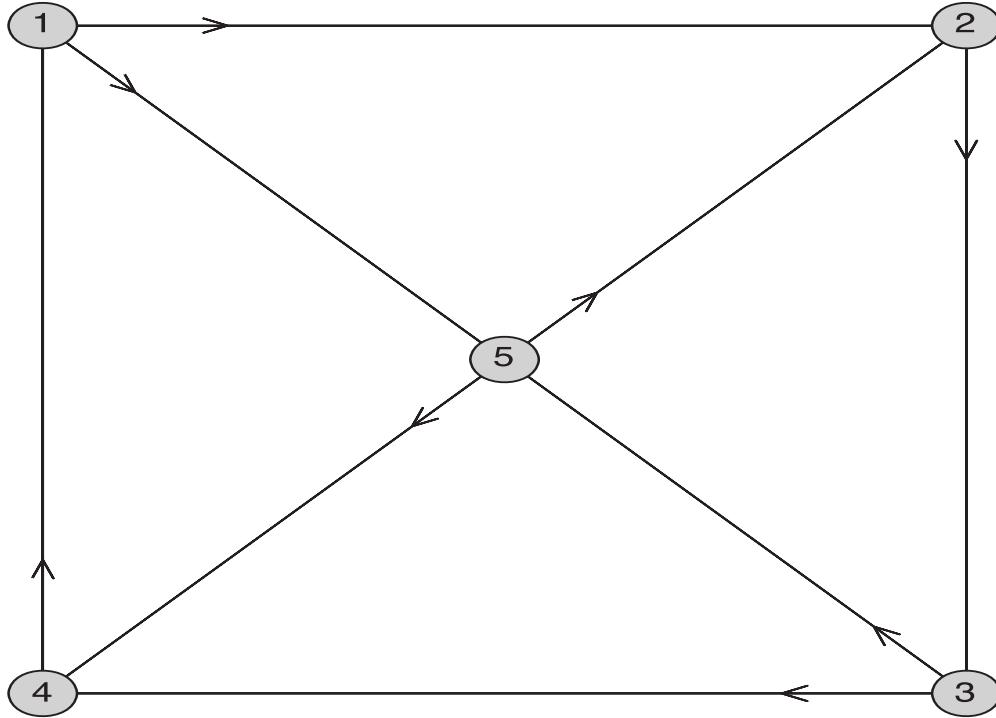
> *EdgeConnectivity*(Figure4G3)
2 (10.114)

Connectedness in Directed Graphs

When used with a directed graph, **IsConnected** returns true if the underlying undirected graph is connected. That is, for directed graphs, **IsConnected** determines whether or not the graph is weakly connected. To check if a directed graph is strongly connected, Maple has the command **IsStronglyConnected**.

We consider a pair of examples.

```
> strongEx := Digraph(Trail(1,2,3,4,1),[1,5],[3,5],[5,2],[5,4]):  
> SetVertexPositions(strongEx,[[0,1],[1,1],[1,0],[0,0],[0.5,0.5]])  
> DrawGraph(strongEx)
```



```
> IsConnected(strongEx)  
true  
(10.115)  
> IsStronglyConnected(strongEx)  
true  
(10.116)
```

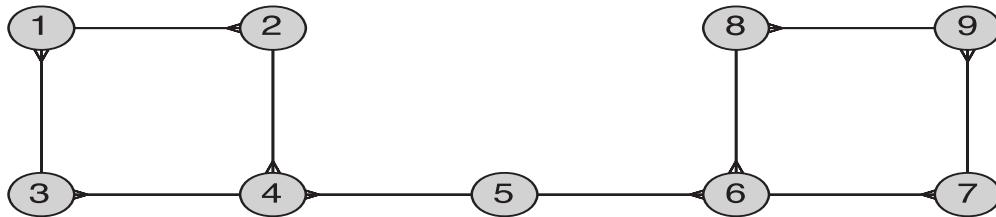
Applying **IsConnected** and **IsStronglyConnected** indicates that this graph is strongly connected.

```
> IsConnected(strongEx)  
true  
(10.117)  
> IsStronglyConnected(strongEx)  
true  
(10.118)
```

The second example we create will be seen to be weakly, but not strongly, connected.

```
> weakEx := Digraph(Trail(4,2,1,3,4,5), Trail(6,8,9,7,6,5))  
weakEx := Graph 31: a directed unweighted graph with 9 vertices and 10 arc(s)  
(10.119)  
> SetVertexPositions(weakEx,[[0,1],[1,1],[0,0],[1,0],[2,0],[3,0],[4,0],  
[3,1],[4,1]])
```

> *DrawGraph(weakEx)*



> *IsConnected(weakEx)*

true

(10.120)

> *IsStronglyConnected(weakEx)*

false

(10.121)

Maple also has commands to extract the connected components of a graph that is not connected. The **ConnectedComponents** command takes a graph as input and returns a list of lists of vertices. For directed graphs, **ConnectedComponents** is used to determine the weakly connected components of the graph. The strongly connected components are obtained with **StronglyConnectedComponents**.

> *ConnectedComponents(GraphComplement(CompleteGraph(2, 3)))*

[[1, 2], [3, 4, 5]]

(10.122)

The above indicates that the complement of $K_{2,3}$ has two connected components. The first component consists of the subgraph comprised of vertices 1 and 2, and the second connected component consists of the other three vertices.

Coloring the Components

Now, we present a procedure that will color code the strongly connected components in a directed graph. We will again select the colors using the Spring palette. Note that this palette has 16 colors, so we will design the procedure to recycle colors after using all that are available in the palette.

> *numcolors(springcolors)*

16

(10.123)

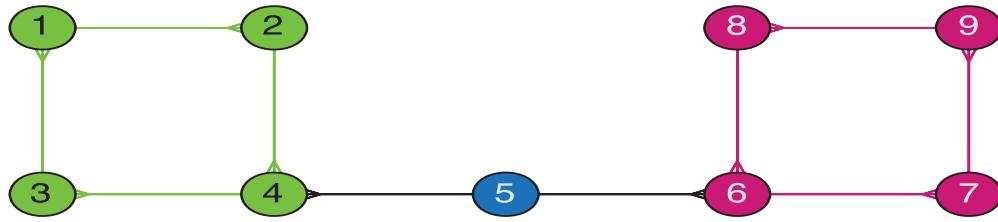
```
1 HighlightSCC := proc (G::Graph)
2   local colorList, components, c, i, H;
3   uses GraphTheory, ColorTools;
4   colorList := GetPalette("spring");
5   components := StronglyConnectedComponents(G);
6   c := 0;
7   for i from 1 to nops(components) do
8     c := c + 1;
9     if c > numcolors(colorList) then
10       c := 1;
11     end if;
12     if nops(components[i]) = 1 then
13       HighlightVertex(G, components[i], colorList[c]);
```

```

14
15     H := InducedSubgraph(G, components[i]);
16     HighlightSubgraph(G, H, colorList[c], colorList[c]);
17   end if;
18 end do;
19 DrawGraph(G);
20 end proc;

```

> *HighlightSCC(weakEx)*



Counting Paths Between Vertices

The last topic that we will consider in this section is determining the number of paths between two vertices of a given length. As described in the textbook, if A is the adjacency matrix for a graph (which may be undirected or directed and may include loops and multiple edges), then the (i, j) entry of the matrix A^r is the number of paths of length r from vertex i to vertex j .

As an example, consider the **strongEx** graph from above. We can obtain its adjacency matrix by applying the **AdjacencyMatrix** command to the name of the graph.

> *Amatrix := AdjacencyMatrix(strongEx)*

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \quad (10.124)$$

Next, compute some powers of the adjacency matrix.

> *Amatrix², Amatrix³, Amatrix⁴*

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 2 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 0 & 2 & 2 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 2 & 1 & 2 & 2 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 2 & 1 & 2 & 0 \end{bmatrix} \quad (10.125)$$

> *Amatrix*⁵, *Amatrix*⁶, *Amatrix*⁷

$$\begin{bmatrix} 2 & 3 & 2 & 2 & 1 \\ 0 & 2 & 1 & 2 & 2 \\ 2 & 2 & 2 & 3 & 1 \\ 1 & 2 & 0 & 2 & 2 \\ 2 & 1 & 2 & 1 & 2 \end{bmatrix}, \begin{bmatrix} 2 & 3 & 3 & 3 & 4 \\ 2 & 2 & 2 & 3 & 1 \\ 3 & 3 & 2 & 3 & 4 \\ 2 & 3 & 2 & 2 & 1 \\ 1 & 4 & 1 & 4 & 4 \end{bmatrix}, \begin{bmatrix} 3 & 6 & 3 & 7 & 5 \\ 3 & 3 & 2 & 3 & 4 \\ 3 & 7 & 3 & 6 & 5 \\ 2 & 3 & 3 & 3 & 4 \\ 4 & 5 & 4 & 5 & 2 \end{bmatrix} \quad (10.126)$$

We see that there are 4 paths of length 6 from vertex 3 to vertex 5, since the (3, 5) entry in the 6th power of the adjacency matrix is 4. We also see that there are cycles of length 3 for every vertex and there are no cycles of length less than 3. Finally, we know that the shortest path from vertex 2 to vertex 1 is of length 3, since the (2, 1) entry is 0 for the first and second powers of the matrix.

10.5 Euler and Hamilton Paths

In this section, we will show how to use Maple to solve two problems that seem closely related, but which are quite different in computational complexity. The two problems that will be analyzed are the problem of finding a simple circuit that contains every edge exactly once (an Euler circuit) and the problem of finding a simple circuit that contains every vertex exactly once (a Hamilton circuit). (Note that the textbook uses the term circuit while Maple uses the word cycle in its help pages. These two terms are synonymous.)

Euler Circuits in Simple Graphs

Maple comes equipped with a command to determine if a given simple graph has an Euler circuit or not. This command, **IsEulerian**, takes one or two arguments. The required argument is the graph. The second argument is an optional name in which Maple will store the Eulerian circuit it finds. As an example, we have Maple find an Euler circuit on K_5 .

> *IsEulerian*(*CompleteGraph*(5), 'K5EulerCircuit')

true (10.127)

> *K5EulerCircuit*

Trail(1, 2, 3, 1, 4, 2, 5, 3, 4, 5, 1) (10.128)

Now, we will have Maple help us visualize this path by creating an animation that successively highlights the edges in the path. To do this we will use the **animate** command. The **animate** command takes at least three arguments. The first is a Maple procedure that generates a plot. Typically, one uses one of the built-in commands, such as **plot** or **plot3d** as the first argument. In this case, however, we will create our own procedure, **plotPath**, for the first argument. We will return to **plotPath** in a moment. The second argument to the **animate** command is a list containing the arguments to the command given in the first argument. The third argument will be a parameter with a range specification of the form **t=a..b** which specifies the parameter used in the construction of the individual plots that make up the animation and their bounds. We will also be using two options. The **paraminfo=false** option turns off the display of the value of the parameter, while **frames=50** tells Maple to create 50 frames instead of the default 25, which in this case has the effect of slowing down the animation. (Note: There is no way to change the frame rate of the animation with the command

line, but you can increase and decrease the frames per second (FPS) in the context menu of an animation. You can also step through the animation one frame at a time to better see the progress of the path.)

We now return to the **plotPath** procedure, which will be the first argument to the **animate** command. The **plotPath** procedure will take as arguments a graph, a list of vertices representing a path, and a number representing the progress along the path (e.g., 1 indicates one edge traversed, 2 indicates two edges traversed, etc.).

It first makes a copy of the graph so that the modifications to the edge colors do not affect the original graph. The procedure uses the local variable **N** to ensure that it does not exceed the length of the list given in the second parameter and to ensure that the value representing the progress along the path is an integer. Assuming the requested path length is not 0, then the procedure takes a slice out of the list of vertices to represent a path of that length and highlights that “trail.” In the case that the third argument is 0, it skips the highlighting steps and just draws the graph.

```

1 plotPath := proc(G::Graph, P::list, n)
2   local Gcopy, path, N;
3   uses GraphTheory;
4   Gcopy := CopyGraph(G);
5   if n > nops(P) - 1 then
6     N := nops(P) - 1;
7   else
8     N := floor(n);
9   end if;
10  if N <> 0 then
11    path := P[1..(N+1)];
12    HighlightTrail(Gcopy, path);
13  end if;
14  DrawGraph(Gcopy);
15 end proc;
```

Now, we use the **plotPath** procedure as the basis for the following **animatePath** procedure. This procedure will take as input a graph and a path and create the animation using the **animate** command.

```

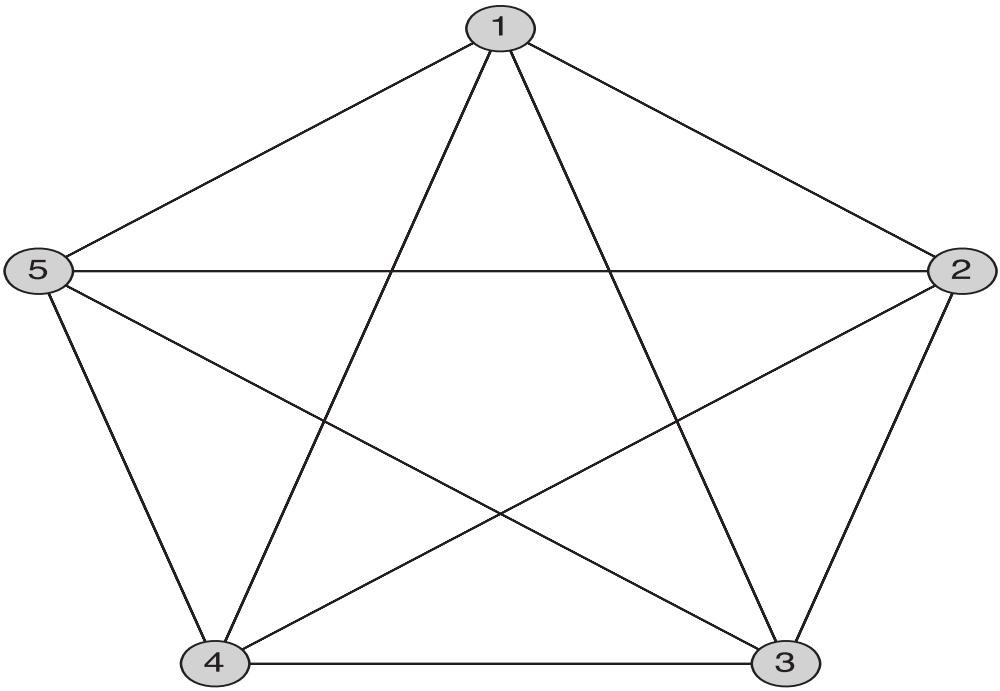
1 animatePath := proc(G::Graph, P::list)
2   local t;
3   plots[animate](plotPath, [G, P, t], t=0..(nops(P)-1),
4                 paraminfo=false, frames=50);
5 end proc;
```

To use this procedure, we just need to turn the “trail” that **IsEulerian** found above into a list of vertices.

```

> K5CircuitList := [op(K5EulerCircuit)]
K5CircuitList := [1, 2, 3, 1, 4, 2, 5, 3, 4, 5, 1] (10.129)

> animatePath(CompleteGraph(5), K5CircuitList)
```



You can see the Euler circuit traced out by clicking on the image of the graph and then clicking on the play button at the top of the window.

Euler Circuits in Multigraphs

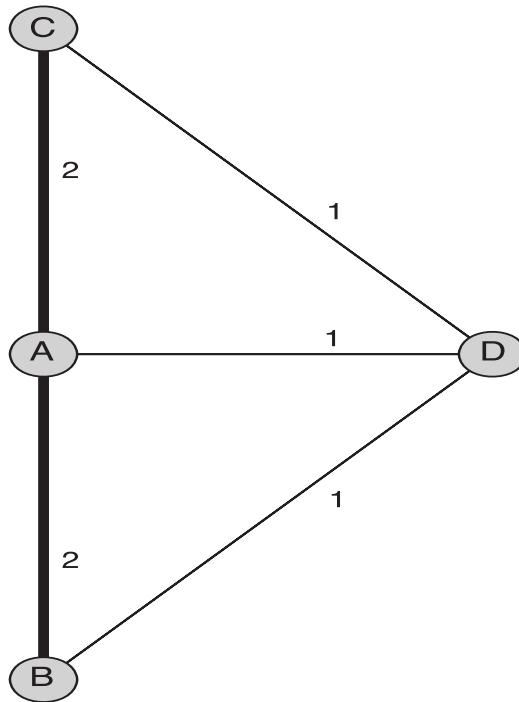
As usual, Maple's built-in function only applies to simple graphs, that is, graphs with no loops or multiple edges. We will examine the problem of finding Euler circuits in multigraphs. We know, from Theorem 1 in Section 10.5, that a connected multigraph with at least two vertices has an Euler circuit if and only if the degree of every vertex is even. It is easy to see that Theorem 1 extends to pseudographs as well. Using this fact, we can write a simple procedure for determining whether or not a pseudograph has an Euler circuit.

```

1  IsPseudoEulerian := proc(G::Graph)
2    local v;
3    uses GraphTheory;
4    if IsDirected(G) then
5      return FAIL;
6    end if;
7    if (not IsConnected(G)) or (nops(Vertices(G)) < 2) then
8      return false;
9    end if;
10   for v in Vertices(G) do
11     if type(PseudoDegree(G, v), odd) then
12       return false;
13     end if;
14   end do;
15   return true;
16 end proc;
```

We can use this procedure to solve the Bridges of Königsberg problem. First, we create a representation of the town and its bridges as a graph (this replicates Figure 2 in Section 10.5). Then we apply the test.

```
> Konigsberg := Graph({{{"A","B"},2},{{"A","C"},2}, {"A","D"},  
    {"B","D"}, {"C","D"}})  
Konigsberg := Graph 32: an undirected weighted graph with 4 vertices and 5 edge(s)  
(10.130)  
> SetVertexPositions(Konigsberg,[[0,1],[0,0],[0,2],[1,1]])  
> DrawPseudograph(Konigsberg)
```



```
> IsPseudoEulerian(Konigsberg)  
false  
(10.131)
```

Now that we have a test that tells us if a circuit exists, we will implement Algorithm 1 in Section 10.5 in order to find an Euler circuit, if it exists. The following procedure will find an Euler circuit in a multigraph. It could also be applied to a pseudograph without generating an error, but it will not include loops in the circuit.

```
1 FindMultiEuler := proc(G::Graph)  
2   local H, circuit, subC, i, v, insertPoint, e, w, buildingSub,  
3       oldC;  
4   uses GraphTheory;  
5   if not IsPseudoEulerian(G) then  
6     return false;  
7   end if;  
8   H := CopyGraph(G);  
9   circuit := [];  
10  while Edges(H) <> {} do  
11    # find a starting point
```

```

11  if circuit = [] then
12      subC := [Vertices(H)[1]];
13  else
14      for i from 1 to nops(circuit) do
15          if Neighbors(H, circuit[i]) <> [] then
16              subC := [circuit[i]];
17              insertPoint := i;
18              break;
19          end if;
20      end do;
21  end if;
22 # build a subcircuit
23 buildingSub := true;
24 while buildingSub do
25     v := subC[-1];
26     w := Neighbors(H, v)[1];
27     e := {v, w};
28     if IsWeighted(H) then
29         if GetEdgeWeight(H, e) > 1 then
30             SetEdgeWeight(H, e, GetEdgeWeight(H, e)-1);
31         else
32             DeleteEdge(H, e);
33         end if;
34     else
35         DeleteEdge(H, e);
36     end if;
37     subC := [op(subC), w];
38     if w = subC[1] then
39         buildingSub := false;
40     end if;
41 end do;
42 # splice the subcircuit into the main circuit
43 if circuit = [] then
44     circuit := subC;
45 else
46     oldC := circuit;
47     circuit := [];
48     if insertPoint >= 2 then
49         circuit := oldC[1..(insertPoint-1)];
50     end if;
51     circuit := [op(circuit), op(subC)];
52     if insertPoint < nops(oldC) then
53         circuit := [op(circuit), op(oldC[(insertPoint+1)..-1])];
54     end if;
55 end if;
56 end do;
57 return circuit;
58 end proc;

```

The program begins with a use of **IsPseudoEulerian** in order to avoid searching for a circuit that cannot exist. It then assigns to the variable **H** a copy of the graph. It is this copy that is used throughout the rest of the procedure, rather than the graph **G** that was passed to the algorithm. The benefit of using a copy is that the procedure will be able to manipulate it as the algorithm proceeds, for example, by deleting edges of **H** once they are included in the circuit so that those edges are not reused.

Recall the description of Algorithm 1 in Section 10.5. There are two key ideas at the heart of this algorithm. The first is that, for a graph whose vertices all have even degree, if you pick any vertex to start at and follow edges at random but without repetition, you will definitely return to the original vertex and create a circuit. The second key idea is that (for a connected graph), if your circuit does not include all of the edges of the graph, then some vertex used in the existing circuit can be made the starting point for a new subcircuit. This subcircuit can then be spliced into the main circuit. This will eventually use all the edges and the result will be a Euler circuit.

The variable **circuit** will hold the main circuit that, at the end of the procedure, is output to the user. The circuit will be stored as a list of the vertices through which the circuit passes and is initialized to the empty list. The main while loop consists of three parts: (1) determining the starting point for the subcircuit (named **subC**); (2) building the subcircuit; and (3) splicing the subcircuit into the main circuit.

The first step, finding the starting point for the subcircuit, depends on the state of the main circuit. If **circuit** is the empty list (i.e., it is the first pass through the main loop), then the starting point is the first vertex in the graph. If the main circuit is not empty, then the else clause looks at the vertices in the main circuit to find one that has neighbors (since edges are deleted from **H** as they are added to the circuit, only vertices that are an endpoint of an unused edge have neighbors). The first vertex that has a neighbor is used as the starting point for the subcircuit. The **insertPoint** variable is used to keep track of the index, relative to **circuit**, of the starting vertex for the subcircuit. This is used when the subcircuit is spliced into the main circuit.

The second step is to build **subC**. The **buildingSub** variable is used to control the while loop. It is initialized to true and is set to false once **subC** has returned to its starting vertex and is thus a circuit. The variable **v** is set to the last vertex currently included as part of the subcircuit and **w** represents a neighbor of **v**. The variable **e = {v,w}** is therefore an edge in the graph that has not already been traversed by the circuit. The nested if statements that follow the assignment of **e** effect the removal of the edge **e** from the graph **H**. In the case that **H** is weighted (i.e., is a multigraph), the weight is either decreased by 1 to represent the removal of one of several edges between the vertices or is deleted if there is only one such edge. In the unweighted case, the edge is always deleted from the graph. After the edge has been deleted, the vertex **w** is added to the subcircuit, representing the inclusion of the edge. Finally, the newest vertex is compared with the starting vertex to determine if the circuit has been closed. If the new vertex closes the circuit, then the **buildingSub** variable is set to false, which causes the while loop to terminate. Otherwise, the while loop continues building the subcircuit.

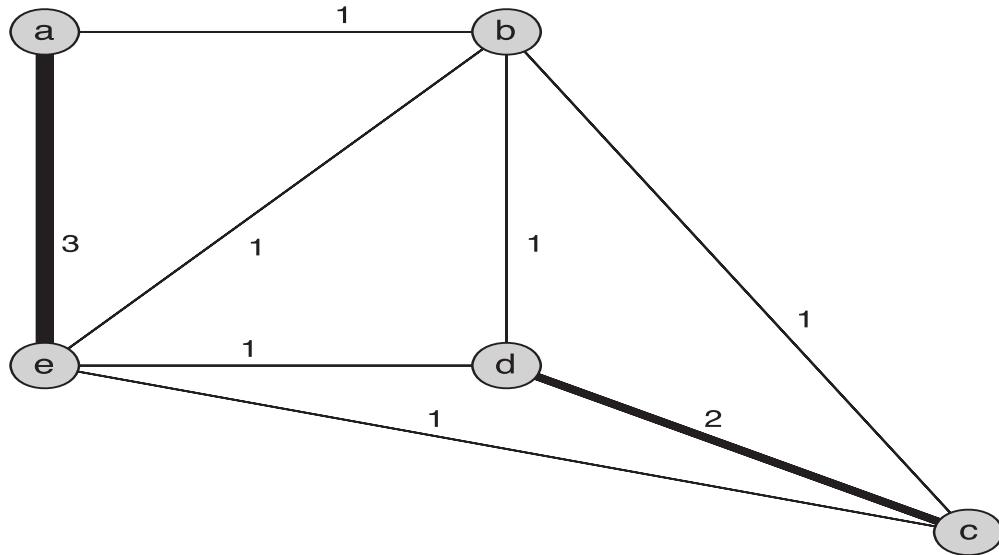
The third step, once the subcircuit has been built, is to splice it into the main circuit. In the first pass through the main loop, the main circuit is empty and so **subC** is just copied into **circuit**. In subsequent passes of the main loop, the variable **oldC** is used to store the “old” circuit. Recall that **insertPoint** stores the index of the starting vertex for **subC**. The goal is to put the subcircuit in that location. The new, more complete, circuit is built in three pieces. First, the part of the old circuit that

occurs before the insertion point (assuming the insertion point is not the initial vertex of the main circuit). Next, the subcircuit. Finally, the part of the old circuit that comes after the insertion point (assuming the insertion point is not the final vertex).

The main while loop continues until all the edges of the graph have been included in the circuit, making **circuit** an Euler circuit for the graph. As an example, consider Exercise 5 in Section 10.5.

```
> Ex5 := Graph({{{"a", "e"}, 3}, [{"c", "d"}, 2], {"a", "b"}, {"b", "c"}, {"b", "d"}, {"b", "e"}, {"c", "e"}, {"d", "e"}})
Ex5 := Graph 33: an undirected weighted graph with 5 vertices and 8 edge(s) (10.132)
```

```
> SetVertexPositions(Ex5, [[0, 2], [1, 2], [2, 0.5], [1, 1], [0, 1]])
> DrawPseudograph(Ex5)
```



```
> Ex5Path := FindMultiEuler(Ex5)
Ex5Path := ["a", "e", "c", "d", "e", "a", "b", "c", "d", "b", "e", "a"] (10.133)
```

Note that the edge between a and e is traversed three times: as the first edge in the circuit, shortly before the middle of the circuit, and again as the last edge in the circuit. This is consistent with there being three edges between a and e .

The following procedures can be used to create animations to visualize the circuit in a multigraph, as **animatePath** did above for simple graphs. **HighlightMultiTrail** replaces **HighlightTrail** and serves to transition multi-edges from blue to red in steps. Note the use of edge attributes to track the number of times a multi-edge has been traversed.

```
1 HighlightMultiTrail := proc(G::Graph, T)
2   local H, i, e, x, redshade;
3   uses GraphTheory;
4   if not IsWeighted(G) then
```

```

5      H := MakeWeighted(G);
6
7  else
8      H := CopyGraph(G);
9  end if;
10 for e in Edges(H) do
11     SetEdgeAttribute(H,e,"traversed"=0);
12 end do;
13 for i from 2 to nops(T) do
14     e := {T[i-1], T[i]};
15     x := GetEdgeAttribute(H,e,"traversed") + 1;
16     SetEdgeAttribute(H,e,"traversed"=x);
17 end do;
18 for e in Edges(G) do
19     redshade :=
20         GetEdgeAttribute(H,e,"traversed") / GetEdgeWeight(H,e);
21     HighlightEdges(G,{e}, COLOR(RGB,redshade,0,1-redshade));
22 end do;
23 return G;
24 end proc:
```

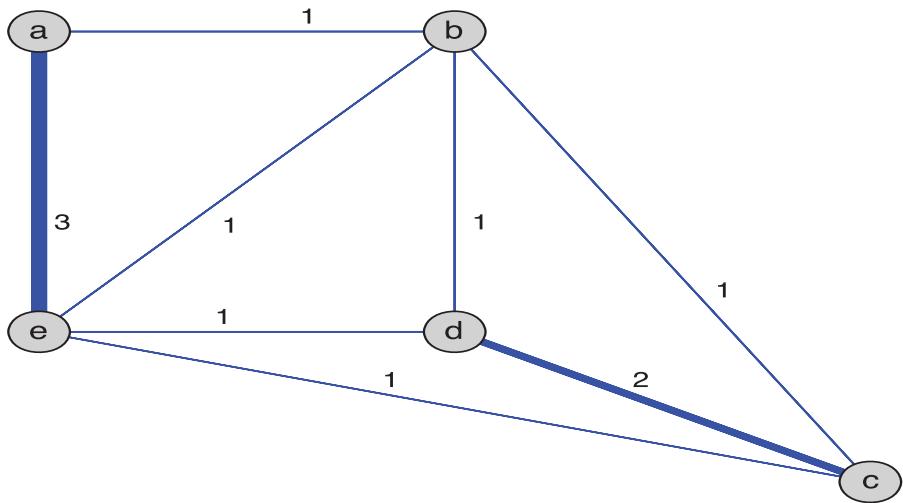
```

1 PlotMultiPath := proc(G::Graph, P::list, n)
2   local Gcopy, path, N;
3   uses GraphTheory;
4   Gcopy := CopyGraph(G);
5   if n > nops(P) - 1 then
6     N := nops(P) - 1;
7   else
8     N := floor(n);
9   end if;
10  if N <> 0 then
11    path := P[1..(N+1)];
12  else
13    path := [];
14  end if;
15  HighlightMultiTrail(Gcopy, path);
16  DrawGraph(Gcopy);
17 end proc:
```

```

1 AnimateMultiPath := proc(G::Graph, P::list)
2   local t;
3   plots[animate](PlotMultiPath, [G,P,t], t=0..(nops(P)-1),
4                 paraminfo=false, frames=50);
5 end proc:
```

> *AnimateMultiPath(Ex5, Ex5Path)*



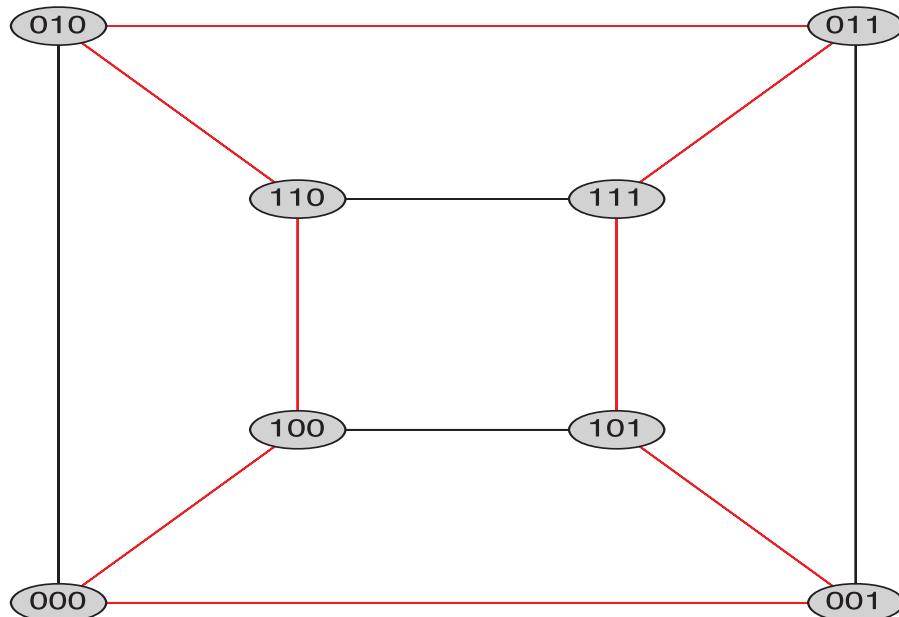
Hamilton Circuits

Turning our attention to Hamilton circuits, Maple provides the command **IsHamiltonian** for determining whether or not the graph contains a Hamilton circuit. This command, like **IsEulerian**, accepts one required and one optional argument. The required argument, of course, is a graph. If a variable name is provided as the second argument, then Maple will store the Hamilton circuit in the variable, which you can then use as the second argument to **HighlightTrail**.

```
> HCGraph := SpecialGraphs[HypercubeGraph](3)
HCGraph := Graph 34: an undirected unweighted graph with 8 vertices and 12 edge(s)
(10.134)
```

```
> IsHamiltonian(HCGraph, 'HCpath')
true
(10.135)

> HighlightTrail(HCGraph, HCpath)
> DrawGraph(HCGraph)
```



```
> IsHamiltonian(CompleteGraph(3,2))
false
```

(10.136)

Note that a pseudograph is Hamiltonian if and only if its underlying simple graph is Hamiltonian, so there is no need for us to extend the **IsHamiltonian** command to pseudographs.

10.6 Shortest-Path Problems

Among the most common problems in graph theory are the “shortest path problems.” Generally, in shortest path problems, we wish to determine a path between two vertices of a weighted graph that is minimal in terms of the total weight of the edges in the path.

In the previous sections of this chapter, we used edge weights as a way to get around Maple’s limitation of being able to represent simple graphs only. In this section, we will use weighted graphs in the way they are actually intended—to represent some sort of cost associated with traversing the edge. Note that pseudographs are rarely, if ever, of use in shortest path problems. There is no reason to consider multiple edges between two vertices since the edge of lowest weight is always preferred. Moreover, traversing a loop at a vertex would only increase the cost with no benefit.

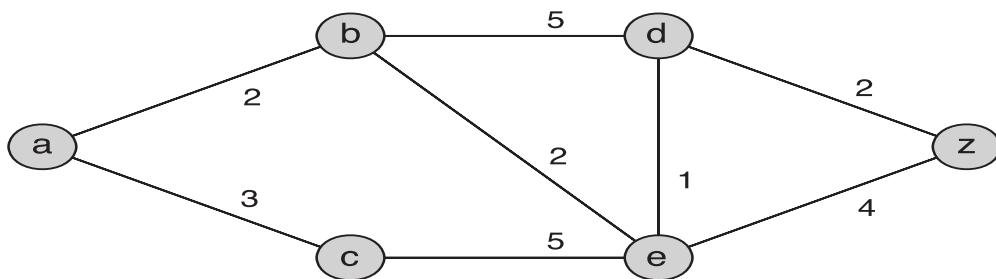
First, we reproduce Exercise 2 in Section 10.6 of the textbook to use as an example. Recall that when defining an undirected and weighted graph, we use the format **[{a,b},w]** to indicate that the graph has an edge between a and b with weight w .

```
> Ex2 := Graph({[{"a","b"},2],[{"a","c"},3],[{"b","d"},5],
  [{"b","e"},2],[{"c","e"},5],[ {"d","e"},1],[ {"d","z"},2],
  [{"e","z"},4]})

Ex2 := Graph 35: an undirected weighted graph with 6 vertices and 8 edge(s) (10.137)
```

```
> SetVertexPositions(Ex2, [[0, 0.5], [1, 1], [1, 0], [2, 1], [2, 0], [3, 0.5]])
```

```
> DrawGraph(Ex2)
```



Now, we will make use of Maple’s implementation of Dijkstra’s algorithm to compute the shortest path between a and z . To do this, we simply call the **DijkstrasAlgorithm** command with three arguments: the graph and the names of the starting and ending vertices.

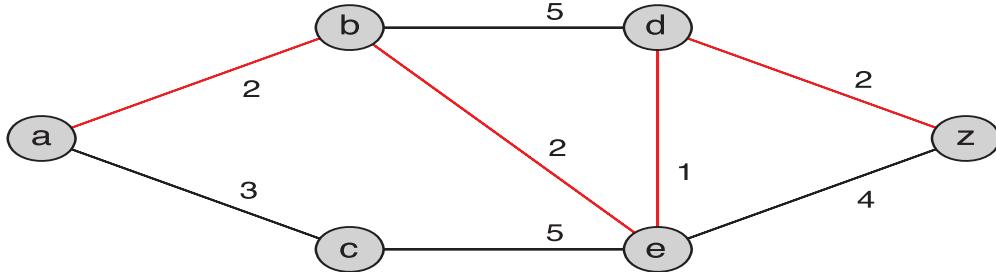
```
> Ex2shortest := DijkstrasAlgorithm(Ex2, "a", "z")
Ex2shortest := [{"a", "b", "e", "d", "z"}, 7]
```

(10.138)

The output informs us that the shortest path is a, b, e, d, z and that the length of this path is 7. We can display the shortest path by passing the list of vertices to the **HighlightTrail** command.

> *HighlightTrail(Ex2, Ex2shortest[1])*

> *DrawGraph(Ex2)*



There is an alternate form of **DijkstrasAlgorithm** for producing the shortest path from an initial vertex to several different vertices at once.

> *DijkstrasAlgorithm(Ex2, "a", ["d", "e", "z"])*
 $[[["a", "b", "e", "d"], 5], [[["a", "b", "e"], 4], [[["a", "b", "e", "d", "z"], 7]]]$

(10.139)

And for producing the shortest paths from the starting vertex to all other vertices.

> *DijkstrasAlgorithm(Ex2, "a")*
 $[[["a"], 0], [[["a", "b"], 2], [[["a", "c"], 3], [[["a", "b", "e", "d"], 5], [[["a", "b", "e"], 4], [[["a", "b", "e", "d", "z"], 7]]]]]]$

(10.140)

To determine the shortest path from every vertex to every other vertex, we use the **AllPairsDistance** command. This is an implementation of the Floyd–Warshall algorithm (also known as simply Floyd’s algorithm), which is described in Algorithm 2 in the Exercises of Section 10.6.

> *AllPairsDistance(Ex2)*

$$\begin{bmatrix} 0 & 2 & 3 & 5 & 4 & 7 \\ 2 & 0 & 5 & 3 & 2 & 5 \\ 3 & 5 & 0 & 6 & 5 & 8 \\ 5 & 3 & 6 & 0 & 1 & 2 \\ 4 & 2 & 5 & 1 & 0 & 3 \\ 7 & 5 & 8 & 2 & 3 & 0 \end{bmatrix}$$

(10.141)

Note that the command returned a matrix. The (i, j) entry in this matrix is the shortest distance from vertex i to vertex j .

Finally, Maple provides a **TravelingSalesman** command for solving the traveling salesperson problem on a given graph. Given a graph, the procedure returns two objects: a number representing

the minimum possible length of a Hamilton circuit and the list of vertices representing the minimal circuit.

```
> TravelingSalesman(Ex2)
21, ["a", "b", "d", "z", "e", "c", "a"]
```

(10.142)

10.7 Planar Graphs

This section explains how Maple can be used to explore the question of whether a graph is planar. We begin with a brief description of Maple's built-in functions. We then discuss how to use Maple to manipulate graphs in order to produce homeomorphic graphs and to apply Kuratowski's Theorem.

Maple includes the command **IsPlanar**, which returns true if and only if the given graph is a planar graph. For example, we can check that the graph $K_{3,2}$ is planar, but that $K_{3,3}$ is not.

```
> IsPlanar(CompleteGraph(3, 2))
true
```

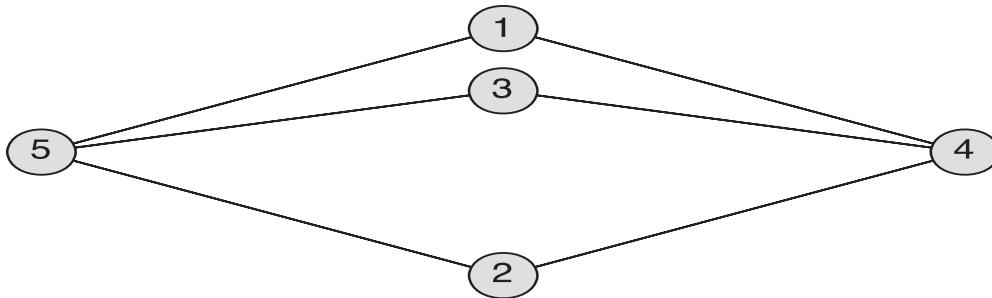
(10.143)

```
> IsPlanar(CompleteGraph(3, 3))
false
```

(10.144)

In addition, as mentioned above, for those graphs that are planar, the **DrawGraph** command includes the option to draw them as such, using the planar style.

```
> DrawGraph(CompleteGraph(3, 2), style = planar)
```



For graphs that are not planar, the planar style will cause an error to be raised.

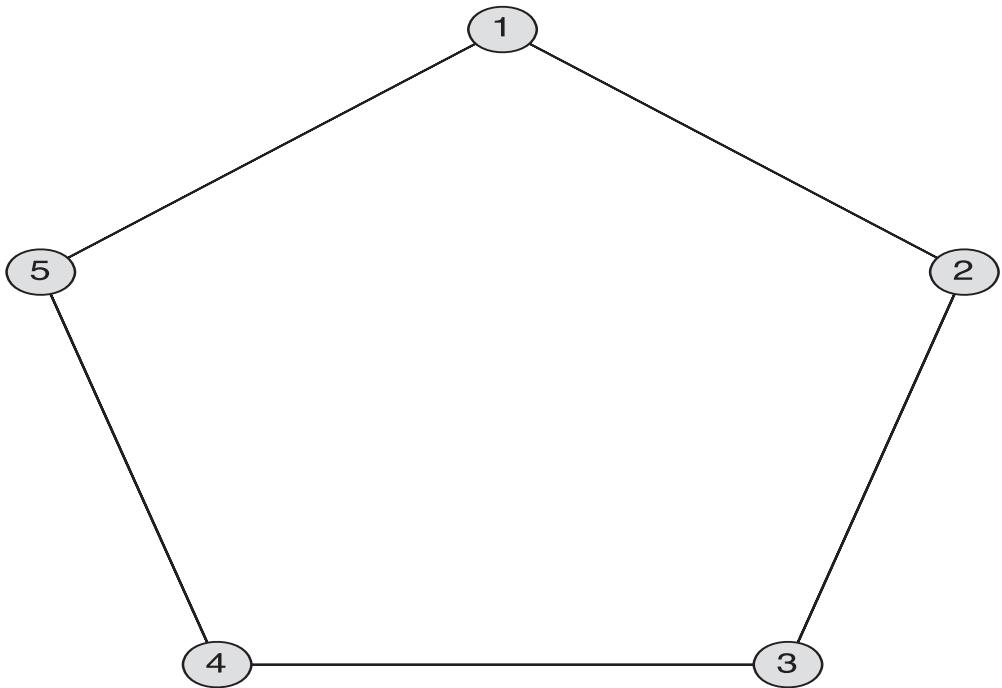
Elementary Subdivisions, Smoothing, and Homeomorphic Graphs

Recall that an elementary subdivision refers to the process of modifying a graph by removing an edge $\{u, v\}$ and replacing it with a vertex w and new edges $\{u, w\}$ and $\{v, w\}$. Effectively, this splits the original edge into two by inserting a vertex in the middle of it. Maple includes a command, **Subdivide**, for achieving this effect. If we apply this command to a graph and one of its edges, it returns a new graph obtained by performing an elementary subdivision on the given edge.

```
> SubdivideEx := CycleGraph(5)
SubdivideEx := Graph 36: an undirected unweighted graph with 5 vertices and 5 edge(s)
```

(10.145)

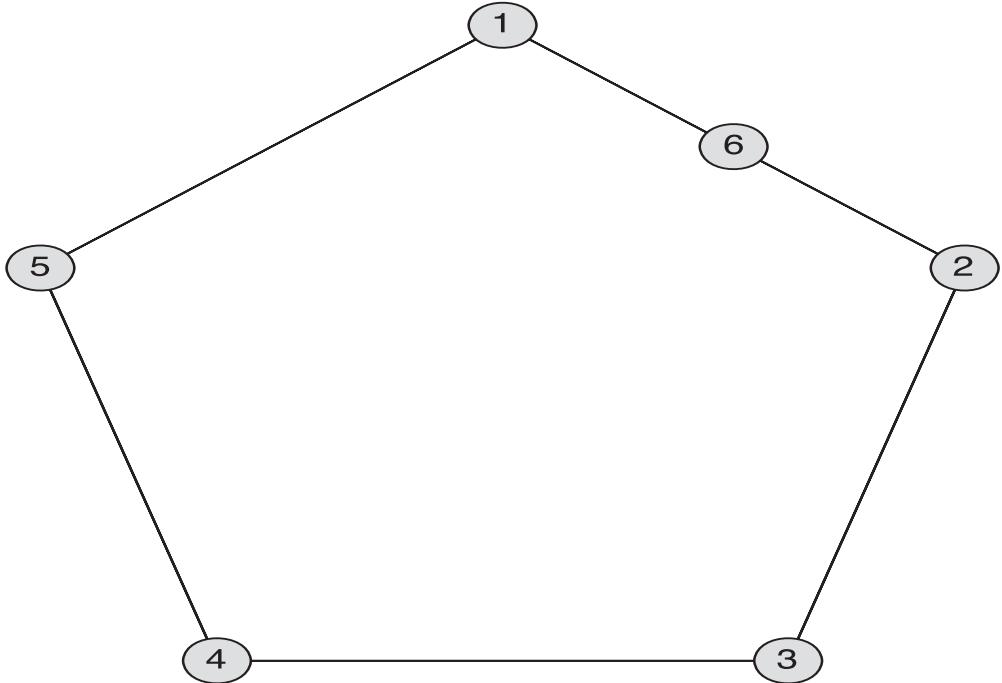
> *DrawGraph*(SubdivideEx)



> *SubdivideEx2* := *Subdivide*(SubdivideEx, {1, 2})

SubdivideEx2 := Graph 37: an undirected unweighted graph with 6 vertices and 6 edge(s)
(10.146)

> *DrawGraph*(SubdivideEx2)



The inverse operation of elementary subdivision is referred to as smoothing. To be precise, let v be a vertex of degree 2 with neighbors u and w and such that u and w are not adjacent. We smooth the

vertex v by deleting v and the edges incident to it and adding the edge $\{u, w\}$. Below we have created a procedure to implement smoothing. (Note that Maple's **Contract** command is more general than smoothing. The benefits of creating the **Smooth** procedure are that it is explicitly the inverse of elementary subdivision and that it is more natural, in this context, to think about smoothing the vertex rather than contracting one of the incident edges.)

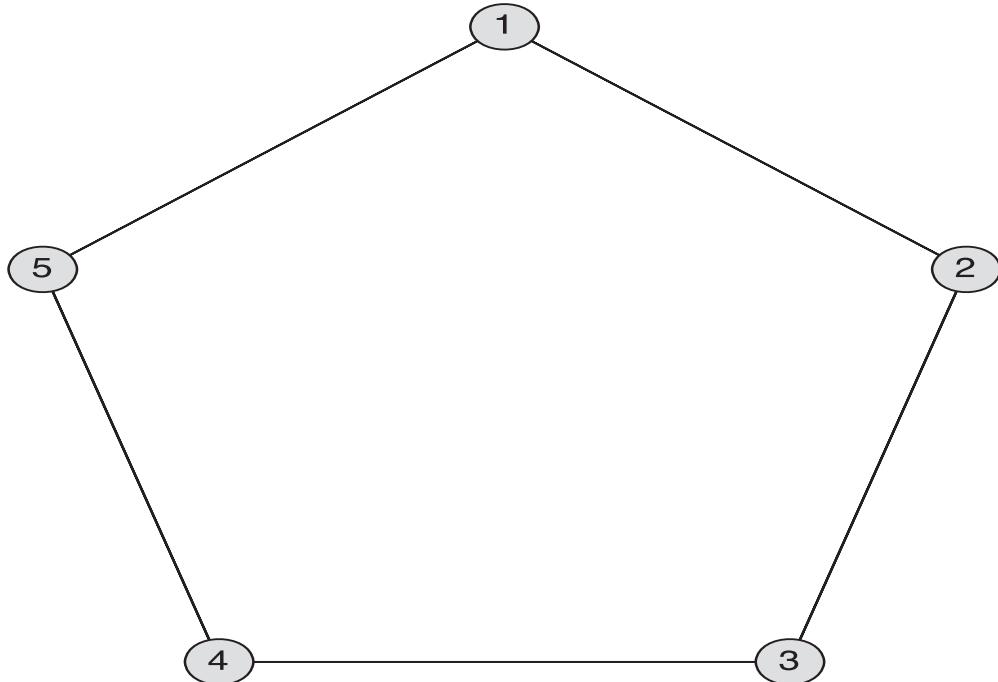
```

1  Smooth := proc (G::Graph, v)
2    local e, H;
3    e := {op(Neighbors(G, v))};
4    if (Degree(G, v) <> 2) or (e in Edges(G)) then
5      return FAIL;
6    else
7      H := DeleteVertex(G, v);
8      AddEdge(H, e);
9    end if;
10   return H;
11 end proc;
```

> *SubdivideEx3 := Smooth(SubdivideEx2, 6)*

SubdivideEx3 := Graph 38: an undirected unweighted graph with 5 vertices and 5 edge(s)
(10.147)

> *DrawGraph(SubdivideEx3)*



The textbook defines graphs to be homeomorphic if they can be obtained from the same graph from a sequence of elementary subdivisions. It is clear that if $G_1, G_2, G_3, \dots, G_n$ is a sequence of graphs, each of which can be obtained from the previous by an elementary subdivision, then

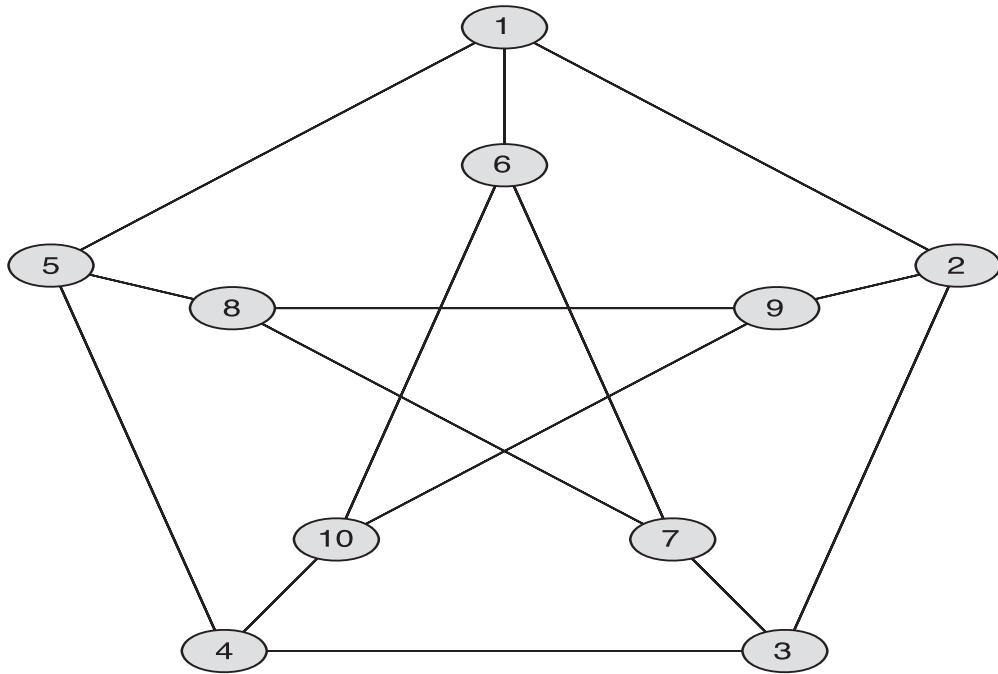
$G_n, \dots, G_3, G_2, G_1$ is a sequence of graphs, each of which can be obtained from the previous by a smoothing. Therefore, we can say that two graphs are homeomorphic if one can be transformed into the other by a sequence of elementary subdivisions and smoothings.

Applying Kuratowski's Theorem

Recall that Kuratowski's theorem asserts that a graph is nonplanar if and only if it contains a subgraph homeomorphic to either $K_{3,3}$ or K_5 . Using the commands above and those for creating subgraphs, we can use Maple to manipulate a graph and confirm that it is nonplanar using Kuratowski's theorem. We will illustrate this with the Petersen graph.

```
> petersen := SpecialGraphs[PetersenGraph]()
petersen := Graph 39: an undirected unweighted graph with 10 vertices and 15 edge(s)
(10.148)
```

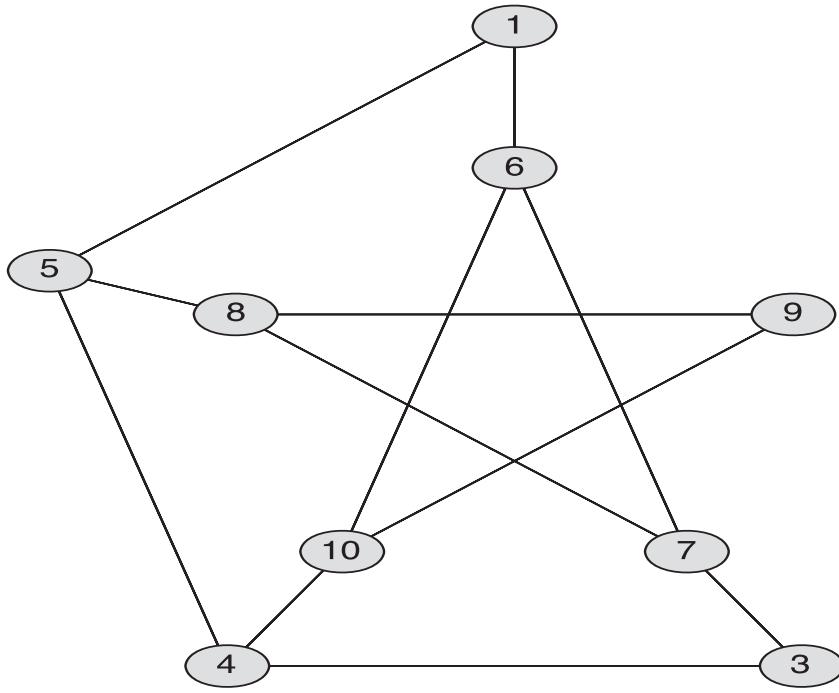
> DrawGraph(petersen)



First, we form the subgraph of the Petersen graph obtained by removing vertex 2 and the three edges incident to it.

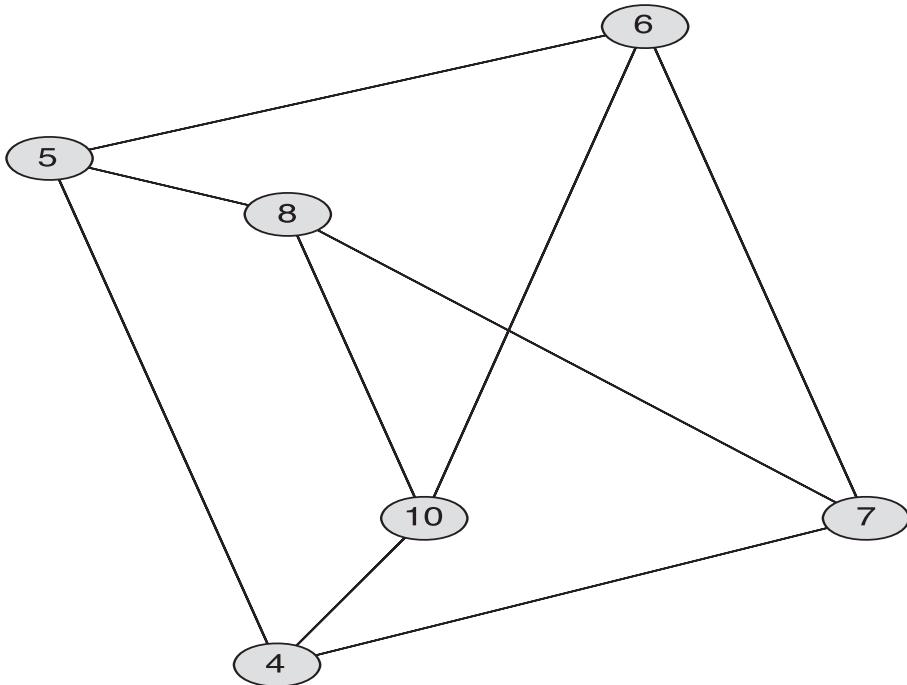
```
> petersen1 := DeleteVertex(petersen, 2)
petersen1 := Graph 40: an undirected unweighted graph with 9 vertices and 12 edge(s)
(10.149)
```

> DrawGraph(petersen1)



Now, we notice that there are three vertices that are smoothable: 1, 3, and 9. That is to say, those three vertices have degree 2 and their neighbors are not adjacent.

```
> petersen2 := Smooth(petersen1, 1):  
> petersen3 := Smooth(petersen2, 3):  
> petersen4 := Smooth(petersen3, 9):  
> DrawGraph(petersen4)
```

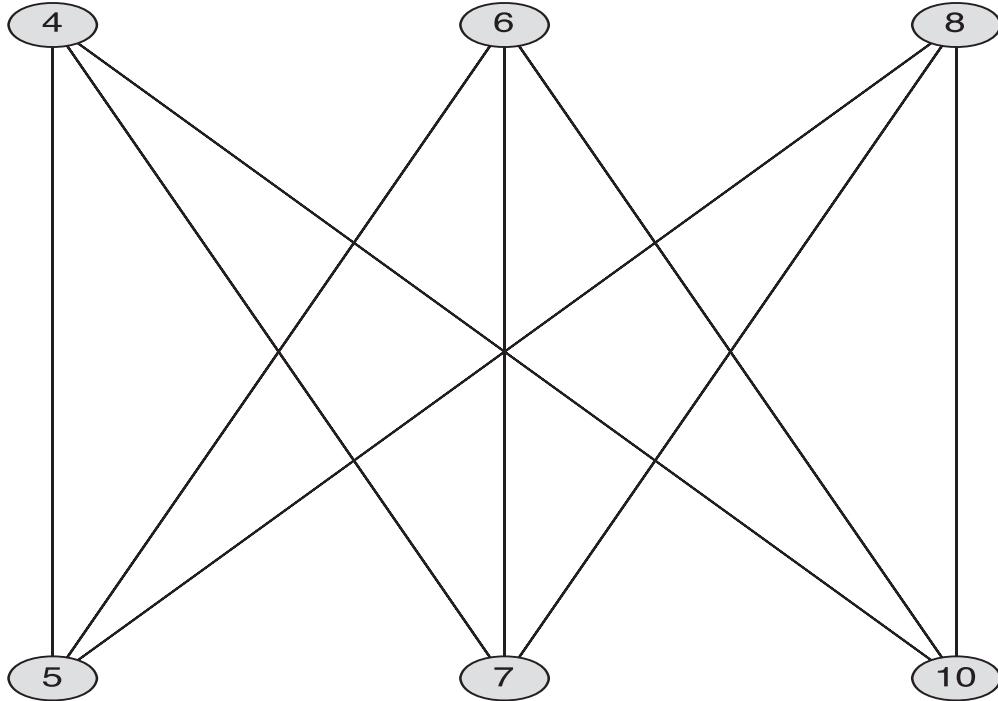


We now observe that this graph has 6 vertices, each of which has degree 3, just like $K_{3,3}$. Thus, there is a definite possibility that this graph is $K_{3,3}$. We check that it is bipartite and then have Maple draw it in that style.

```
> IsBipartite(petersen4)
true
```

(10.150)

```
> DrawGraph(petersen4, style = bipartite)
```



It is clear from inspection that this is $K_{3,3}$ and so we have demonstrated that the Petersen graph has a subgraph that is homeomorphic to $K_{3,3}$ and hence is nonplanar.

10.8 Graph Coloring

In this section, we consider the problem of how to properly color a graph; that is, how to assign to each vertex of a graph a color such that no vertex has the same color as any of its neighbors.

It is worth noting that, in terms of computational complexity, Hamilton circuits and graph coloring are equivalently difficult problems.

Maple's Command

Maple provides a **ChromaticNumber** command that uses a sophisticated backtracking technique for computing the chromatic number of a graph.

Given a graph, the **ChromaticNumber** command will report the minimal number of colors needed to color that graph.

> *CNExample* := *SpecialGraphs*[*WheelGraph*] (5)

CNExample := Graph 41: an undirected unweighted graph with 6 vertices and 10 edge(s)
(10.151)

> *ChromaticNumber*(*CNExample*)

4

(10.152)

If you provide a variable name as a second argument to the command, Maple will store a list of lists of vertices. These lists indicate which vertices should be assigned the same color.

> *ChromaticNumber*(*CNExample*, 'CNClasses')

4

(10.153)

> *CNClasses*

[[0], [2, 4], [3, 5], [1]]

(10.154)

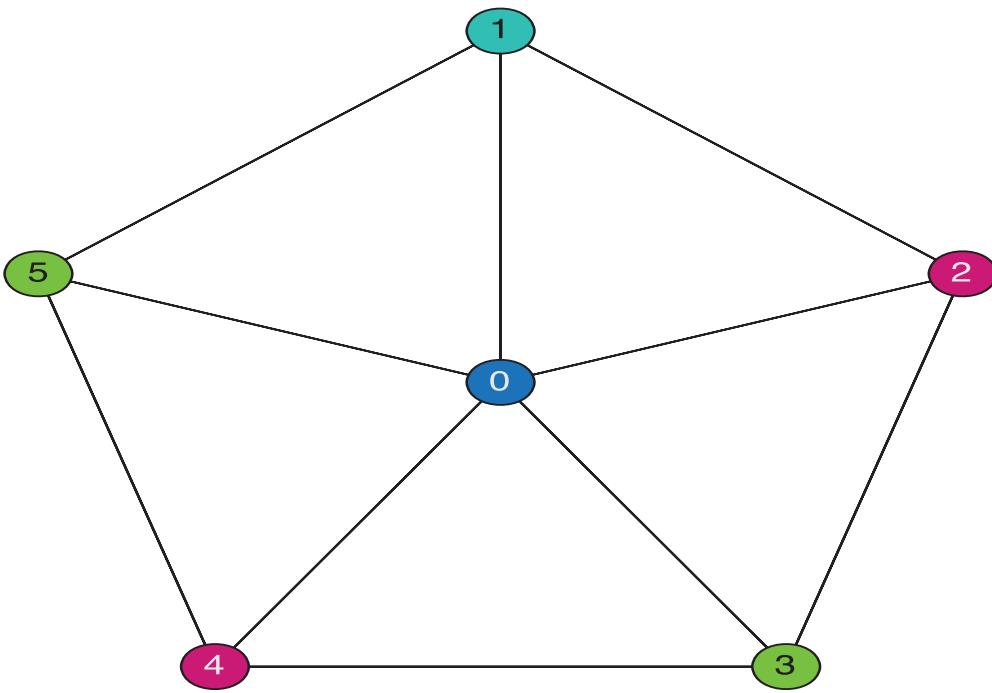
This output indicates that vertex 0 should be given one color, vertices 1 and 3 should be assigned a second color, vertices 2 and 4 a third color, and vertex 5 should be painted with the final color.

We can write a short procedure to display the graph with the vertices appropriately colored. Our procedure will call the **HighlightVertex** command with a list of vertices and a single color. This form of the command causes all of the vertices in the list to be shaded with the specified color.

```
1  CNColor := proc (G::Graph, colors)
2    local i, Vclasses;
3    uses GraphTheory;
4    ChromaticNumber (G, Vclasses);
5    if numelems (Vclasses) > numelems (colors) then
6      error "You must provide at least one color for each vertex class .";
7    end if ;
8    for i from 1 to numelems (Vclasses) do
9      HighlightVertex (G, Vclasses[i], colors[i]);
10    end do;
11    DrawGraph (G);
12  end proc;
```

This procedure requires two arguments: the graph and a list or other structure that contains colors that can be used with **HighlightVertex** and in conjunction with the selection operator.

> *CNColor*(*CNExample*, *ColorTools*[*GetPalette*] ("spring"))



A Greedy Coloring Algorithm

In this section, we will create a procedure based on the algorithm described in the preface to Exercise 29 in Section 10.8 of the text. It can be shown that this algorithm will color a graph using at most one more color than the maximal degree of the graph. It is considered a “greedy” algorithm because it makes optimal choices at each step but never reconsiders its choices. That is to say, it does the best it can at every step but never backtracks to make improvements. Greedy algorithms often lead to good, but nonoptimal, solutions.

The algorithm proceeds as follows. First, the vertices are sorted in order of descending degree. The first color is assigned to the first vertex in the list. In addition, assign color 1 to the first vertex in the list not adjacent to vertex 1, to the next vertex not adjacent to those already colored, etc. Then, move on to the second color. The first uncolored vertex in the list is assigned color 2, as are vertices further down the list not adjacent to ones previously assigned the second color. This continues until all of the vertices have been given a color.

Our first step in implementing this algorithm will be to sort the list of vertices in decreasing order of degree. For this, we will make use of Maple’s very flexible **sort** command. With no additional instructions, Maple will sort a list of numbers in increasing numerical order and a list of strings in lexicographical order. Moreover, the **sort** command accepts an optional argument that allows us to specify the way in which the list is sorted. Specifically, **sort** takes as an argument a procedure that is Boolean-valued on two arguments and returns true if the first argument precedes the second.

For our graph coloring procedure, this is further complicated by the fact that the procedure that we pass to the **sort** command must depend on the graph. We will create a functional operator that returns a boolean-valued procedure associated to the given graph.

```

1 MakeSorter := G -> proc (v, w)
2   uses GraphTheory;
3   evalb(Degree(G, v) > Degree(G, w));
4 end proc;

```

Applying **MakeSorter** to a graph returns a procedure that can be used as the optional argument to **sort**.

We now implement the greedy coloring algorithm.

```

1 GColor := proc (G::Graph, colors)
2   local Sorter, V, currentColor, excludeSet, i;
3   Sorter := MakeSorter(G);
4   V := sort(Vertices(G), Sorter);
5   for currentColor from 1 to numelems(colors) do
6     HighlightVertex(G, V[1], colors[currentColor]);
7     excludeSet := {op(Neighbors(G, V[1]))};
8     V := subsop(1=NULL, V);
9     i := 1;
10    while i <= nops(V) do
11      if not (V[i] in excludeSet) then
12        HighlightVertex(G, V[i], colors[currentColor]);
13        excludeSet := excludeSet union {op(Neighbors(G, V[i]))};
14        V := subsop(i=NULL, V);
15      else
16        i := i + 1;
17      end if;
18    end do;
19    if V = [] then
20      break;
21    end if;
22  end do;
23  if V <> [] then
24    error "Insufficiently many colors";
25  else
26    DrawGraph(G);
27  end if;
28 end proc;

```

Note that the set **V**, which is initialized to the list of vertices, sorted in decreasing order of degree, is used to track which vertices still need to be assigned a color. When a vertex has been assigned a color, it is deleted from the list **V** using **subsop(i=NULL,V)**. The **subsop** command is used to substitute a value in a list at a specified index. In this case, we are substituting the value **NULL** in the list at index **i**, which has the effect of removing it from the list.

The **excludeSet** variable is used to store all vertices which cannot be assigned the current color. Each time a vertex is assigned a color, all of its neighbors are added to the **excludeSet**. As the

procedure looks down the list of vertices that still need to have a color assigned, it checks to see if they are in this set.

The index **i**, which controls the while loop, is incremented in the else clause of the if statement that tests to see if a vertex can be assigned the color. If the vertex at index **i** is assigned the color, then it is removed from the list **V**, and thus the index **i** refers to a different vertex (the vertex previously in position **i+1**).

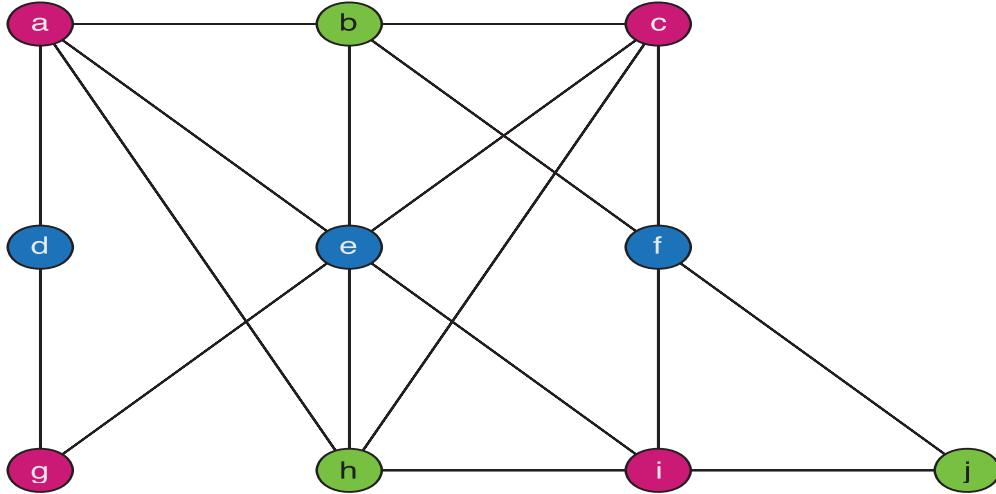
As an example, we solve Exercise 29 in Section 10.8.

```
> Exercise29 := Graph({{"a","b"}, {"a","d"}, {"a","e"}, {"a","h"}, {"b","c"}, {"b","e"}, {"b","f"}, {"c","e"}, {"c","f"}, {"c","h"}, {"d","g"}, {"e","g"}, {"e","h"}, {"e","i"}, {"f","i"}, {"f","j"}, {"h","i"}, {"i","j"}})
```

Exercise29 := Graph 42: an undirected unweighted graph with 10 vertices and 18 edge(s)
(10.155)

```
> SetVertexPositions(Exercise29,  
[[0,2], [1,2], [2,2], [0,1], [1,1], [2,1], [0,0], [1,0], [2,0], [3,0]])
```

```
> GColor(Exercise29, ColorTools[GetPalette]("spring"))
```



Solutions to Computer Projects and Computations and Explorations *Computations and Explorations 1*

Display all simple graphs with four vertices.

Solution: To solve this problem, we will generate all possible edge sets and then construct the graphs based on these edge sets. The possible edge sets are all of the subsets of the set of all possible edges, which we obtain from the complete graph on the vertices. We will generalize the question and have our procedure create all the simple graphs on n vertices.

```
1 AllGraphs := proc(n::posint)
2   local A, V, E, powerE, i, G;
```

```

3   uses GraphTheory;
4   A := {};
5   V := [seq(i, i=1..n)];
6   E := Edges(CompleteGraph(n));
7   powerE := combinat[powerset](E);
8   for i from 1 to nops(powerE) do
9     G[i] := Graph(V, powerE[i]);
10    A := A union {G[i]};
11  end do;
12  return A;
13 end proc;

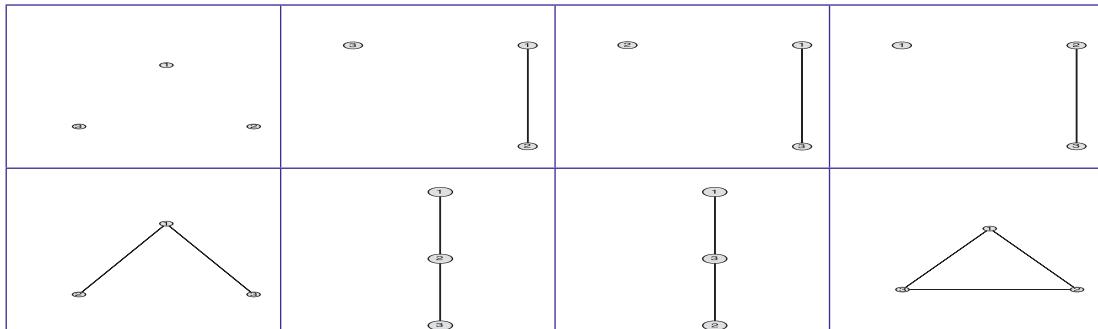
```

Recall that the complete graph on n vertices has $C(n, 2)$ edges, so there are $2^{C(n, 2)}$ graphs on n vertices. Thus, on 4 vertices, there are 64 graphs. For $n = 3$, there are only 8 graphs, which is more manageable.

```

> AllGraphs3 := AllGraphs(3):
> DrawGraph(AllGraphs3, width = 4)

```



Computations and Explorations 2

Display a full set of nonisomorphic simple graphs with six vertices.

Solution: The solution to this exercise is very similar to the previous question. The only difference is that, each time a graph is generated, we compare it to the graphs that have already been included using **IsIsomorphic**.

```

1 NonIsoGraphs := proc(n::posint)
2   local A, V, E, powerE, i, G, j, notisomorphic;
3   uses GraphTheory;
4   A := {};
5   V := [seq(i, i=1..n)];
6   E := Edges(CompleteGraph(n));
7   powerE := combinat[powerset](E);
8   for i from 1 to nops(powerE) do
9     G[i] := Graph(V, powerE[i]);
10    notisomorphic := true;
11    for j from 1 to nops(A) do

```

```

12      if IsIsomorphic(A[j], G[i]) then
13          notisomorphic := false;
14          break;
15      end if;
16  end do;
17  if notisomorphic then
18      A := A union {G[i]};
19  end if;
20 end do;
21 return A;
22 end proc:
```

Note that we have employed a rather brute-force approach by comparing each new graph to each of the previously identified graphs. More efficiency could be obtained by keeping track of a graph invariant and only fully checking pairs of graphs which agree on the invariant. The **IsIsomorphic** command already performs some invariant checking, including the number of edges and degree sequence, but for larger values of n , time spent computing an invariant in order to decrease the number of times **IsIsomorphic** is called can be beneficial.

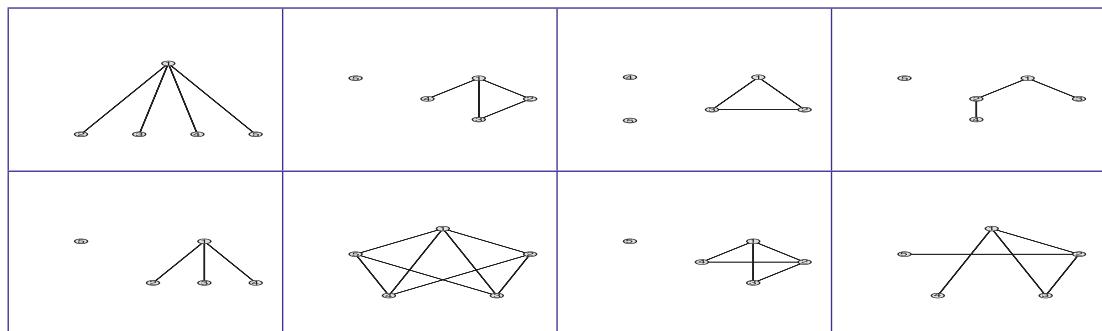
We apply this to five vertices, since six takes a bit more time to compute.

```

> NonIso5 := NonIsoGraphs(5):
> nops(NonIso5)
34
(10.156)
```

We see that there are 34 nonisomorphic simple graphs on 5 vertices. Here are the first eight.

```
> DrawGraph([seq(NonIso5[i], i = 1 .. 8)], width = 4)
```



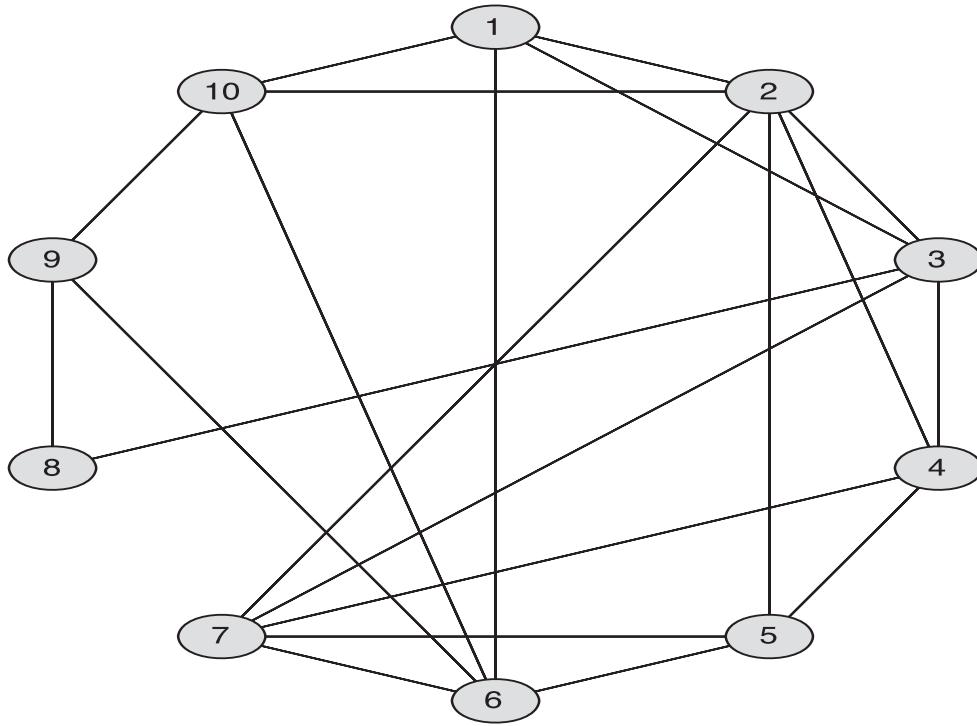
Computations and Explorations 9

Generate at random simple graphs with 10 vertices. Stop when you have constructed one with an Euler circuit. Display an Euler circuit in this graph.

Solution: To generate the random graphs, we will use the **RandomGraph** command in the **RandomGraphs** subpackage. By passing this command a number of vertices and a probability between 0 and 1, it produces a graph with the given number of vertices and with each possible edge

present with the given probability. To display a random graph on 10 vertices with each edge as likely to appear as not, we use the following command.

> *DrawGraph(RandomGraphs[RandomGraph](10, 0.5))*



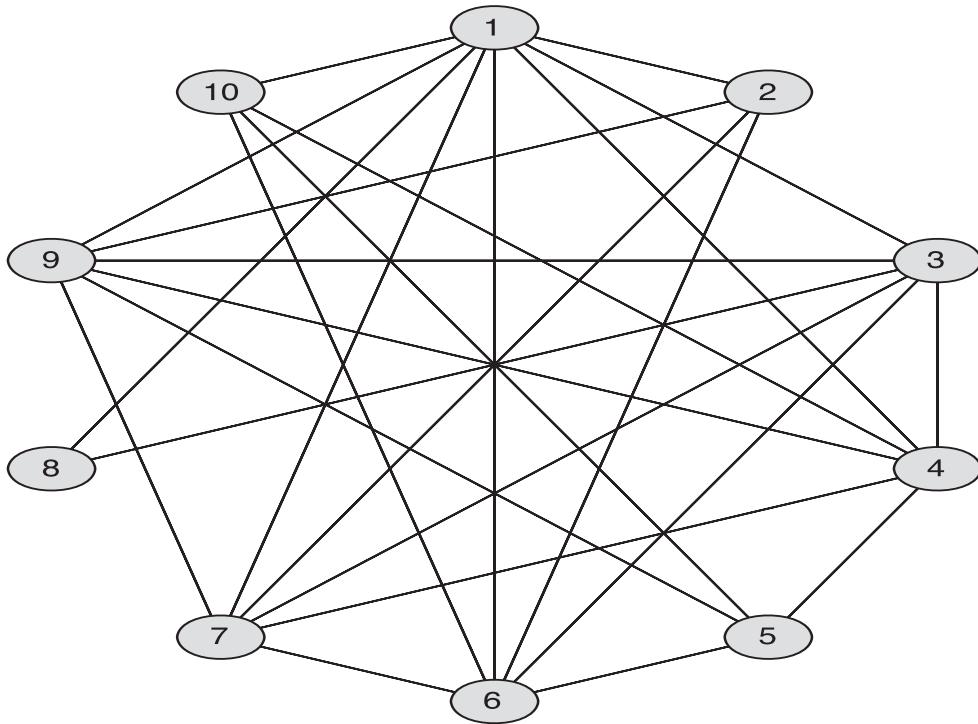
Recall the description of the **IsEulerian** command. When this command is given two arguments, specifically, a graph and an optional variable name, if there is an Euler circuit, then the command returns true and stores the circuit in the variable.

To satisfy the requirements of this problem, we use **RandomGraph** to generate a random graph **G**. Then, we test it for an Euler circuit using **IsEulerian**. As long as the randomly generated graph does not have an Euler circuit, we continue generating new random graphs. We display the path using the **animatePath** procedure we created in Section 10.5.

```

1 GenEuler := proc(n::posint)
2   local G, trail;
3   uses GraphTheory, RandomGraphs;
4   G := RandomGraph(n,.5);
5   while not IsEulerian(G,'trail') do
6     G := RandomGraph(n,.5);
7   end do;
8   animatePath(G,[op(trail)]);
9 end proc;
```

> *GenEuler(10)*



Computations and Explorations 13

Estimate the probability that a randomly generated simple graph with n vertices is connected for each possible integer n not exceeding ten by generating a set of random simple graphs and determining whether each is connected.

Solution: To solve this problem we create a procedure that generates a number of random graphs of the specified size and counts the number that are connected. We use the **RandomGraph** command to create the random graphs and the **IsConnected** command to test them for connectivity.

```

1 ConnectedProbability := proc (verts :: posint, total :: posint)
2   local G, count;
3   uses GraphTheory, RandomGraphs;
4   count := 0;
5   from 1 to total do
6     G := RandomGraph(verts, .5);
7     if IsConnected(G) then
8       count := count + 1;
9     end if;
10    end do;
11    return count/total;
12  end proc;
```

> [seq(ConnectedProbability(i, 100), i = 1 .. 10)]

$$\left[1, \frac{1}{2}, \frac{11}{25}, \frac{13}{25}, \frac{7}{10}, \frac{21}{25}, \frac{24}{25}, \frac{19}{20}, \frac{97}{100}, \frac{49}{50}\right]$$

(10.157)

Exercises

Exercise 1. Write a Maple procedure to find *all* maximal matchings for a bipartite graph.

Exercise 2. Write Maple procedures for calculating the adjacency and incidence matrices for a pseudograph.

Exercise 3. Write a Maple procedure for creating a pseudograph from an incidence matrix.

Exercise 4. Write a Maple procedure to automate the creation of vertex-colored graphs to illustrate graph isomorphisms, as was done with the graphs from Example 11 at the end of Section 10.3 of this manual.

Exercise 5. Write a Maple procedure to find all of the minimal edge cuts of a given graph.

Exercise 6. Write a Maple procedure to determine whether a mixed graph (with directed edges, multiple edges, and loops) has an Euler circuit and, if so, to find such a circuit.

Exercise 7. Use Maple to construct all regular graphs of degree n , given a positive integer n . (Regular is defined in the Exercises for Section 10.2.)

Exercise 8. For vertices u and v in a simple, undirected, and connected graph G , the local vertex connectivity $\kappa(u, v)$ is defined to be the minimum number of vertices that must be removed so that there is no path between vertex u and vertex v . Write a Maple procedure that calculates the local vertex connectivity of a graph and a pair of its vertices.

Exercise 9. For vertices u and v in a simple, undirected and connected graph G , the local edge connectivity $\lambda(u, v)$ is defined to be the minimum number of edges that must be removed so that there is no path between vertex u and vertex v . Write a Maple procedure that calculates the local edge connectivity of a graph and a pair of its vertices.

Exercise 10. Write a Maple procedure that computes the thickness of a nonplanar simple graph (see the Exercises in Section 10.7 for a definition of thickness).

Exercise 11. Write a Maple procedure for finding an orientation of a simple graph. (An orientation of a graph is defined in the Supplementary Exercises of Chapter 10.)

Exercise 12. Write a Maple procedure for finding the bandwidth of a simple graph. (The bandwidth of a graph is defined in the Supplementary Exercises of Chapter 10.)

Exercise 13. Write a Maple procedure for finding the radius and diameter of a simple graph. (The radius and diameter of a graph are defined in the Supplementary Exercises of Chapter 10.)

Exercise 14. Use Maple to find the minimum number of queens controlling an $n \times n$ chessboard for as many values of n as you can. Make use of the concept of a dominating set, described in the Supplementary Exercises of Chapter 10.

Exercise 15. Write a Maple procedure for finding all self-complementary graphs on n vertices. (A self-complementary graph is a graph which is isomorphic to its own complement.) Use your procedure to display the self-complementary graphs for as large a n as possible.

Exercise 16. Write a Maple procedure that finds a total coloring for a graph. A total coloring of a graph is an assignment of a color to each vertex and each edge such that: (a) no pair of adjacent vertices have the same color; (b) no two edges with a common endpoint have the same color; and (c) no edge has the same color as either of its endpoints.

Exercise 17. A sequence of positive integers is called graphic if there is a simple graph that has this sequence as its degree sequence. In this context, the degree sequence of a graph is the nondecreasing sequence made up of the degrees of the vertices of the graph. Develop a Maple procedure for determining whether a sequence of positive integers is graphic and, if it is, to construct a graph with this degree sequence.

11 Trees

Introduction

This chapter is devoted to exploring the computational aspects of the study of trees. Recall from the textbook that a tree is a connected simple graph with no simple circuits.

First, we will discuss how to represent, display, and work with trees using Maple. Specifically, we will see how to represent rooted trees and ordered rooted trees in addition to simple trees. We then use these representations to explore many of the topics discussed in the textbook. In particular, we will see how to use binary trees to store data in such a way as to make searching more efficient and we will see an implementation of Huffman codes. We will use Maple to carry out the different tree traversal methods described in the text. We will see how to construct spanning trees using both depth-first and breadth-first search and how to use backtracking to solve a variety of interesting problems. Finally, we will implement Prim's algorithm and Kruskal's algorithm for finding spanning trees of minimum weight for a weighted graph.

11.1 Introduction to Trees

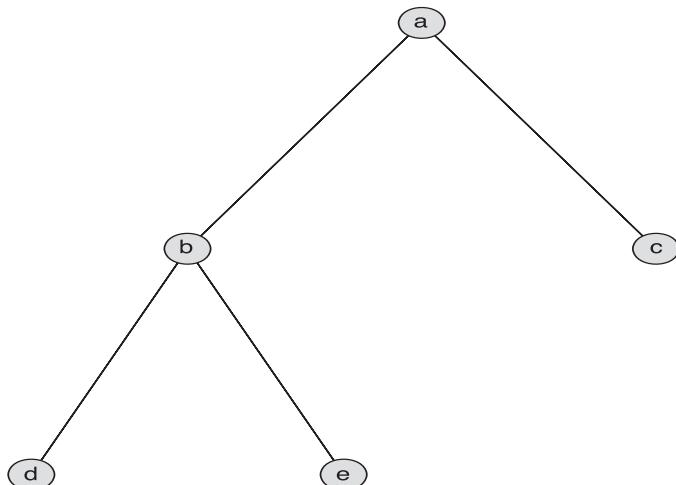
In this section, we will focus on how to construct trees in Maple and how to check basic properties, such as determining if a tree is balanced or not. To begin, we will consider the simplest case, unrooted trees, before moving on to rooted and ordered trees.

Unrooted Trees

Recall that a tree is defined to be a graph that is undirected, connected, and has no simple circuits (or cycles). To create a tree, we just create a graph as we did in the previous chapter. We begin by loading the graph theory package and then creating a simple tree with the **Graph** command.

```
> with(GraphTheory):  
> firstTree := Graph({{"a","b"}, {"a","c"}, {"b","d"}, {"b","e"}})  
firstTree := Graph 1: an undirected unweighted graph with 5 vertices and 4 edge(s) (11.1)
```

```
> DrawGraph(firstTree)
```



The first thing you may notice is that Maple has automatically drawn this in the traditional way, which tells us that Maple recognized the graph as a tree. The **IsTree** command can be used to check if an undirected graph is a tree.

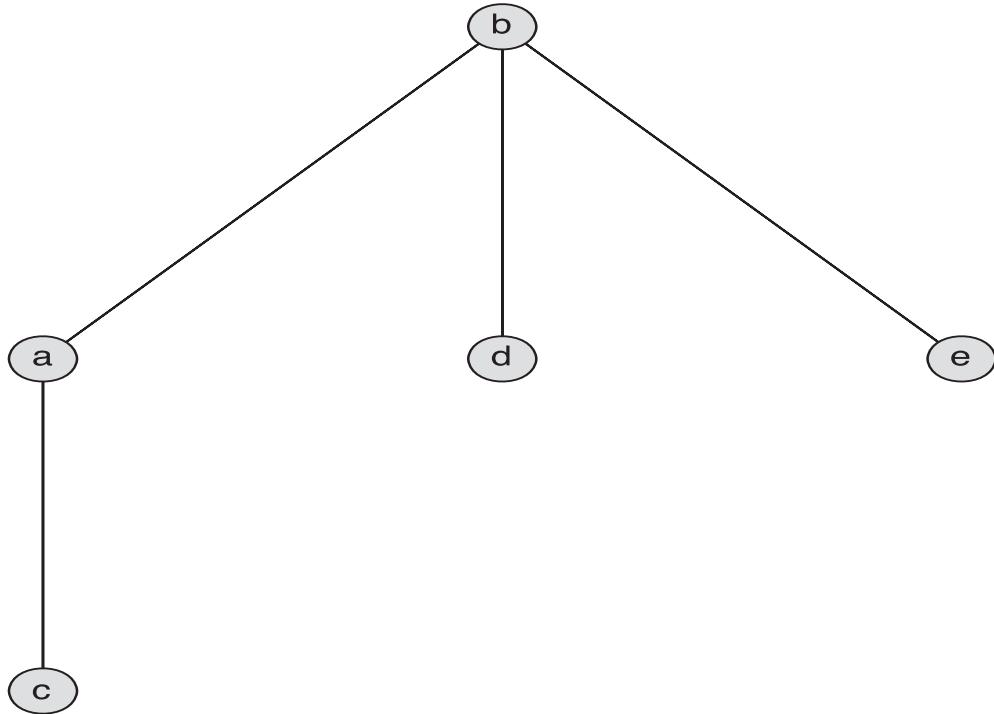
```
> IsTree(firstTree)
true
```

(11.2)

Maple also provides a command **IsForest** for checking whether a graph is a forest (i.e., a collection of trees). The only argument is the graph.

Recall from Section 10.1 of this manual that the **DrawGraph** command can take the optional **style=tree** argument in order to make sure Maple draws the graph as a tree. The **DrawGraph** command, when the tree style has been specified, can also take an optional argument of the form **root=r** to specify which vertex should be drawn as the root.

```
> DrawGraph(firstTree, style = tree, root = "b")
```

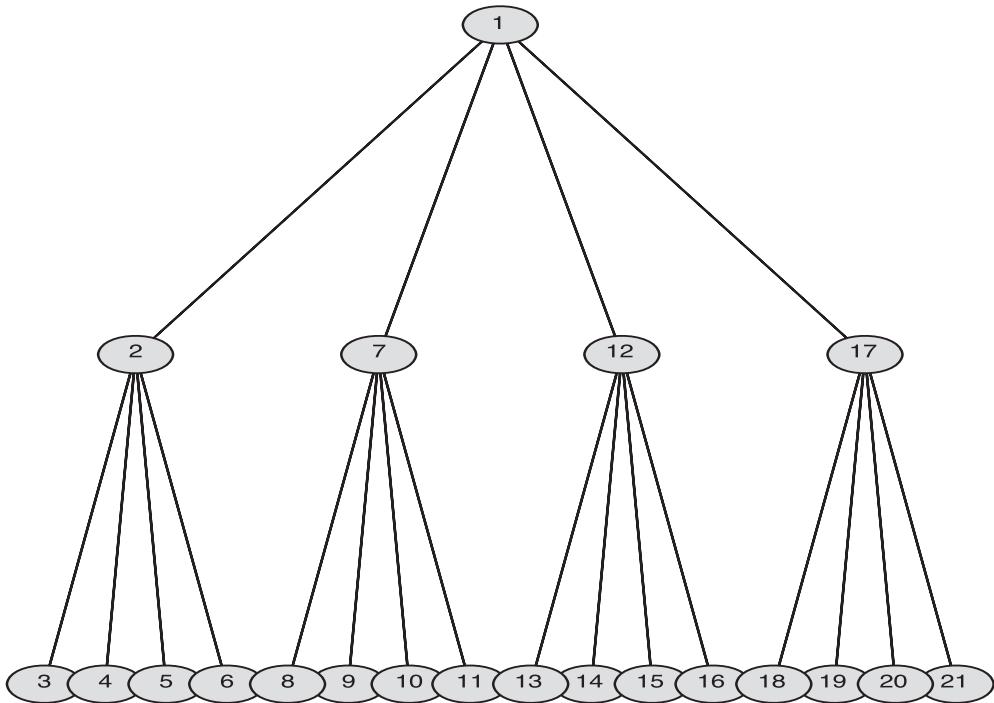


We make two comments about the **root=r** argument. First, it can only be used when the **style=tree** option is explicitly given. Second, while this option makes the tree appear to have the specified root, it does not make it a rooted tree, in the sense used by the textbook. That is to say, other than the positions of the vertices, there is no information stored in the data of **firstTree** to indicate that vertex *b* is the root or even that vertex *c* is a child rather than a parent of *a*.

Maple provides commands for creating certain kinds of trees. Recall that a rooted tree is called *k*-ary if every vertex has no more than *k* children. When *k* = 2, we say that it is a binary tree. Also recall that the height of a rooted tree is the maximum number of levels in the tree, that is, the

height is the length of the largest path from the root to any other vertex. (Maple uses the term depth synonymously with height.) The command **CompleteKaryTree** in the **SpecialGraphs** subpackage produces the unrooted version of the complete k -ary directed tree of height h , where k and h are given as the two arguments to the command.

```
> DrawGraph(SpecialGraphs[CompleteKaryTree](4, 2))
```



The **CompleteBinaryTree** command can be used in the case $k = 2$.

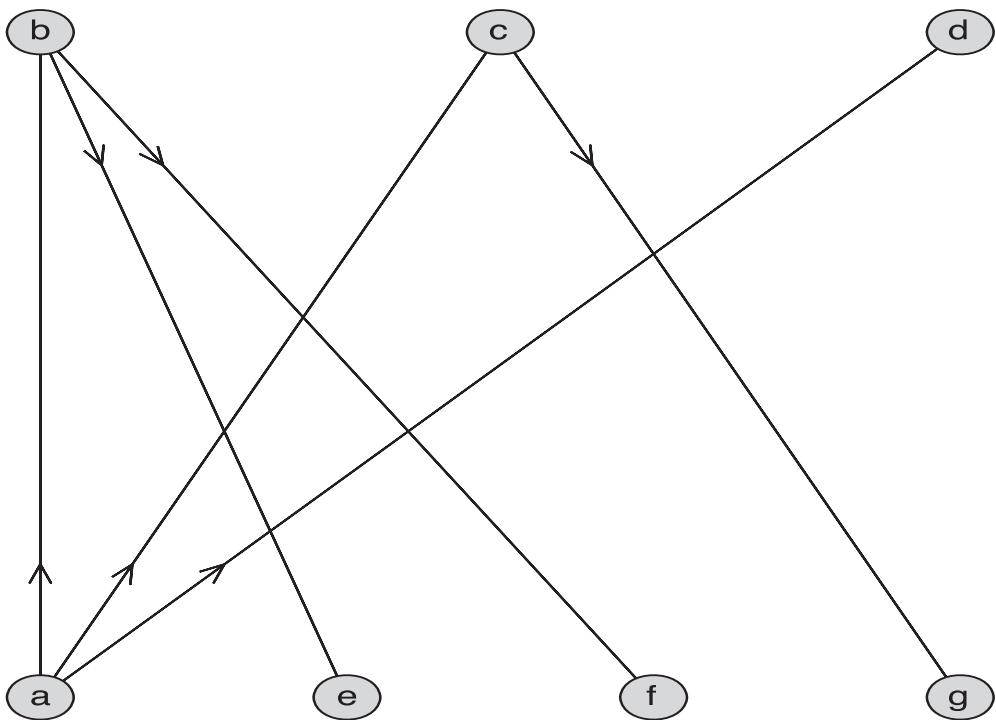
Rooted Trees

Next, we consider rooted trees. Recall that a rooted tree is a directed graph whose underlying undirected graph is a tree and in which one vertex is designated as the root with all edges directed away from the root. For example, the following graph is a rooted tree.

```
> firstRooted := Graph({["a", "b"], ["a", "c"], ["a", "d"], ["b", "e"],
  ["b", "f"], ["c", "g"]})
firstRooted := Graph 2: a directed unweighted graph with 7 vertices and 6 arc(s)
(11.3)
```

If we ask Maple to draw this graph with **DrawGraph**, however, it will not draw it in the form of a tree like it did with **firstTree**. This is because Maple does not recognize this graph as a tree.

```
> DrawGraph(firstRooted)
```



> *IsTree(firstRooted)*

Error, (in GraphTheory:-IsTree) graph must be undirected

While Maple provides some support for unrooted trees, its existing packages are not equipped to recognize and do computations with rooted trees. Much of the remainder of this section will be devoted to filling this gap in Maple’s functionality. This will provide tools you need to better explore trees in the remainder of the chapter.

A Tree Type

In Chapter 9, we discussed Maple types. It will once again be useful to declare types for our trees. This will enable automatic type checking for our procedures, which will help prevent attempts to use procedures on objects that are not trees. Moreover, the declaration of types for trees will formalize what it means to be a tree object in Maple.

A rooted tree, by definition, has a root. It will be convenient to store that information by setting a graph attribute that indicates which vertex is the root. This will be useful since it will free us from having to list the root as an argument to every procedure we create. It is also good programming practice—the root of a rooted tree is information that the tree should “know about itself,” that is to say, the tree should include the root as part of its data.

We begin by creating a type for unrooted trees before returning to the rooted situation. We define the type by creating a procedure which takes one argument, the potential tree, and returns true or false depending on whether it is or is not actually a tree. For unrooted trees, we will be calling the **IsTree** command to do the bulk of the work for us.

We will also use the **try** statement in this type definition. A **try** statement is the primary method for catching and handling errors. We saw above that applying the **IsTree** command to a directed

graph produces an error. In the type procedure below, we use the **try** statement as follows. First is the **try** keyword followed by the code that could potentially raise an error. In this case, it is the **IsTree** command that could raise an error. After the error-prone code, we use the **catch:** line. If anything in the code in the **try** block raises an error, the code following the **catch:** keyword is executed to “handle” the error. In this type definition, we handle any errors by setting the return value to false. (Note: the **try** statement structure is very flexible. Refer to the Maple help pages for more detail.)

```

1  'type/Tree' := proc(obj)
2    local result;
3    try
4      result := GraphTheory[IsTree](obj);
5    catch:
6      result := false;
7    end try;
8    return result;
9  end proc;
```

We can explicitly check to see if an object is of a specified type by using the **type** command with the object and the name of the type as arguments.

```
> type(firstTree, Tree)
true
```

(11.4)

```
> type(firstRooted, Tree)
false
```

(11.5)

Now, we create the type **RTree** for rooted trees. As discussed earlier, we insist that an object of this type stores its root as a graph attribute. To set an attribute, we use the **SetGraphAttribute** command with two arguments: the graph together with the attribute and its value in **tag=value** format. We use the tag “root” and the value will be the name of the root.

```
> SetGraphAttribute(firstRooted, "root" = "a")
```

The **GetGraphAttribute** command with the name of the graph and the tag, in this case “root,” will return the value of the tag.

```
> GetGraphAttribute(firstRooted, "root")
"a"
```

(11.6)

As with the **Tree** type, we will define the type by creating a procedure that returns true for rooted trees and false for all other objects. We will need to check three things: first, that the “root” graph attribute has been set; second, that the underlying undirected graph is a tree; and third, that all edges point away from the root. For the first part, we just check that **GetGraphAttribute** returns a name of a vertex for the tag “root.” Note that if a tag is not set, **GetGraphAttribute** will return **FAIL**.

For the second part, we can simply combine **IsTree** with the **UnderlyingGraph** command, which takes a directed graph and returns the underlying undirected graph. For example, we apply this to our example above.

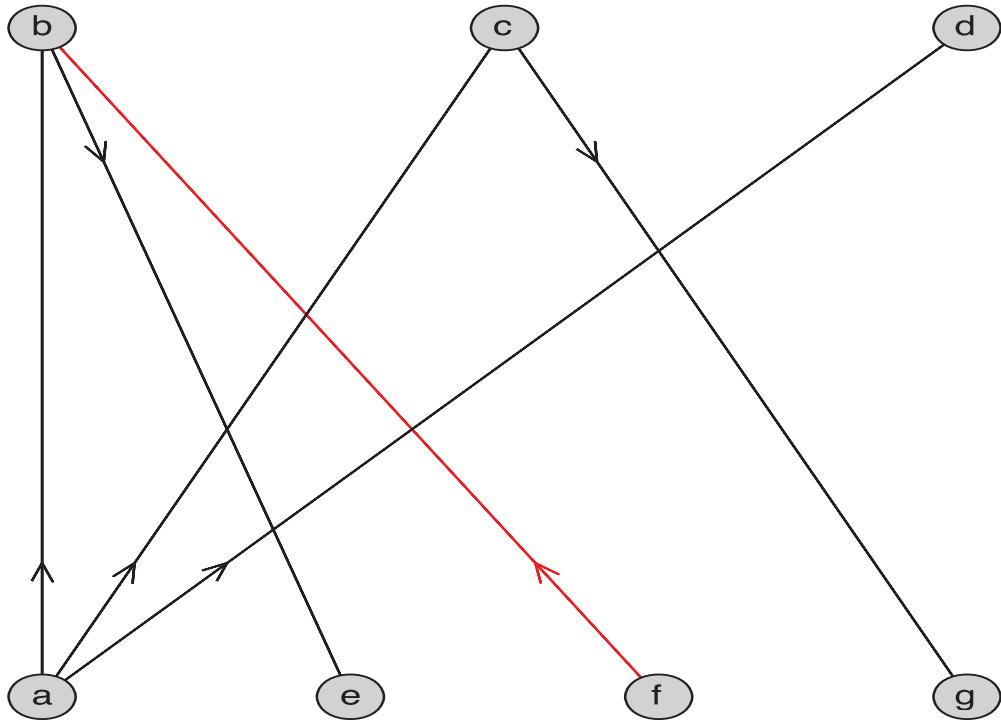
> *IsTree*(*UnderlyingGraph*(*firstRooted*))
true (11.7)

The third part of the test is that every edge is directed away from the root. Before we implement the test, consider the following example, which is identical to **firstRooted** except the edge $[a, e]$ has been reversed.

> *notRooted* := *Graph*([["a", "b"], ["a", "c"], ["a", "d"], ["b", "e"], ["c", "g"],
 ["f", "b"]])
notRooted := *Graph* 3: a directed unweighted graph with 7 vertices and 6 arc(s) (11.8)

```
> HighlightEdges(notRooted, {[“f”, “b”]} , “Red”)
```

> *DrawGraph*(*notRooted*)



> *IsTree*(UnderlyingGraph(notRooted))
true (11.9)

It is easy to see that the edge $[e, a]$ violates the requirement that all edges are directed away from the root. Checking this computationally, however, can be a bit tricky to do directly. Instead, we will take an indirect route. Instead of checking that all edges are directed away from the root, we will check that all the vertices are accessible from the root. Since the underlying graph is a tree, we know that there are no circuits. Thus, the only way for there to be a path from the root to a vertex is for all the edges to be directed away from root. Therefore, if all the vertices are reachable from the root, then all the edges are in the proper direction.

To implement this, we will build a list of vertices accessible from the root. This list will be initialized to contain the purported root. Then, we add to the list all of the vertices which are terminal

vertices of edges with the purported root as the initial vertex. (The **Departures** command, discussed in Section 10.3, is useful here.) Once that is done, the list of accessible vertices consists of the root and all of the children of the root. For the second element in the list, we add all of its children, that is, all the vertices accessible from it. Continuing in this fashion, for each element in the list of accessible vertices, we add its children to the list. When we reach the end of the list, it contains all of the vertices that can be reached from the root. If it has the same members as the list of all vertices in the graph, then the graph satisfies the second condition of having all edges directed away from the root.

We now put these three elements together to create the type.

```

1  'type' / RTree' := proc (obj)
2    local R, Alist, i;
3    uses GraphTheory;
4    if not type(obj, Graph) then
5      return false;
6    end if;
7    if not IsTree(UnderlyingGraph(obj)) then
8      return false;
9    end if;
10   R := GetGraphAttribute(obj, "root");
11   if not R in Vertices(obj) then
12     return false;
13   end if;
14   Alist := [R];
15   i := 1;
16   while i <= nops(Alist) do
17     Alist := [op(Alist), op(Departures(obj, Alist[i]))];
18     i := i + 1;
19   end do;
20   if {op(Alist)} = {op(Vertices(obj))} then
21     return true;
22   else
23     return false;
24   end if;
25 end proc:
```

> *type(firstRooted, RTree)*
true (11.10)

> *type(notRooted, RTree)*
false (11.11)

Drawing Rooted Trees

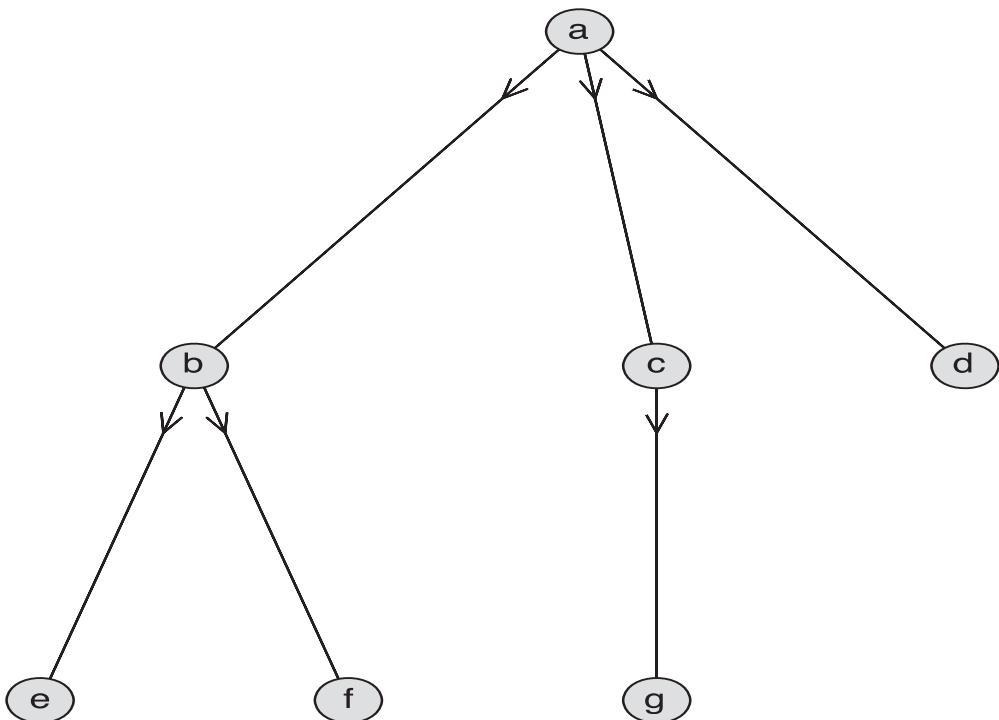
Now that we can test to see that a graph is in fact a rooted tree, we will have Maple draw rooted trees so that they look like trees. Since the underlying graph of a rooted tree is itself a tree, we can determine the best locations for drawing the vertices from the way Maple draws the underlying graph. To do this, we first apply **DrawGraph** to the underlying graph with the **style** and **root** options set but with the output suppressed. This is a necessary step since it is the **DrawGraph** command

that causes Maple to calculate vertex positions. We then use the command **GetVertexPositions** on the underlying graph. Finally, we use **SetVertexPositions** on the rooted tree and with the positions garnered from the underlying graph. (These commands were initially discussed in Section 10.2.)

```

1 DrawRTree := proc (G :: RTree)
2   local R, U, P;
3   uses GraphTheory;
4   R := GetGraphAttribute (G, "root");
5   U := UnderlyingGraph (G);
6   DrawGraph (U, style=tree, root=R):
7   P := GetVertexPositions (U, style=tree, root=R);
8   SetVertexPositions (G, P);
9   DrawGraph (G);
10 end proc:
```

> *DrawRTree(firstRooted)*



Parents, Children, Leaves, and Internal Vertices of Rooted Trees

We now consider commands related to identifying particular vertices in a rooted tree and relations between them.

We begin with the question of whether one vertex is the parent of another. Given the two vertices, checking this requires determining whether the directed edge from the parent to the child is actually in the tree.

```

1 IsParentOf := proc (T :: RTree, p, c)
2   return GraphTheory[HasArc] (T, [p, c]);
3 end proc:
```

```
> IsParentOf(firstRooted, "b", "f")
true
```

(11.12)

```
> IsParentOf(firstRooted, "b", "d")
false
```

(11.13)

Next, we consider the question of finding the parent of a given vertex. This can be done as an application of the **Arrivals** command, which returns the list of vertices that are initial vertices for the edges with the given vertex at the terminal end. Assuming the graph is in fact a rooted tree, there can be at most one such vertex. If the vertex is the root, there will be no parent and the procedure will return **NULL**, producing no output.

```
1 FindParent := proc (T :: RTree, v)
2   local A;
3   A := GraphTheory[Arrivals](T, v);
4   if nops(A) = 1 then
5     return op(A);
6   elif nops(A) = 0 then
7     return NULL;
8   else
9     error "The given graph is not a tree.";
10  end if;
11 end proc;
```

```
> FindParent(firstRooted, "d")
"a"
```

(11.14)

```
> FindParent(firstRooted, "root")
```

For the related question of determining all children of the given tree, we use the **Departures** command, which returns the list of vertices that are terminal vertices for the edges with the given vertex at the initial end.

```
1 FindChildren := proc (T :: RTree, v)
2   return GraphTheory[Departures](T, v);
3 end proc;
```

```
> FindChildren(firstRooted, "a")
["b", "c", "d"]
```

(11.15)

```
> FindChildren(firstRooted, "f")
[]
```

(11.16)

The **FindChildren** procedure also indicates how we can test a vertex to determine if it is an internal vertex or a leaf.

```
1 IsInternal := proc (T :: RTree, v)
2   if GraphTheory[Departures](T, v) <> [] then
3     return true;
```

```

4   else
5     return false;
6   end if ;
7 end proc:
8 IsLeaf := proc (T::RTree, v)
9   if GraphTheory[Departures](T, v) = [] then
10    return true;
11  else
12    return false;
13  end if ;
14 end proc:

```

> *IsInternal(firstRooted, "a")*
true (11.17)

> *IsLeaf(firstRooted, "a")*
false (11.18)

> *IsLeaf(firstRooted, "f")*
true (11.19)

We can determine all the leaves of a given tree by testing each vertex with **IsLeaf**. We use the **select** command to obtain the sublist of the list of vertices which are identified as leaves by **IsLeaf**.

```

1 FindLeaves := proc (T::RTree)
2   uses GraphTheory;
3   select(v -> IsLeaf(T, v), Vertices(T));
4 end proc:

```

> *FindLeaves(firstRooted)*
 $["d", "e", "f", "g"]$ (11.20)

Ordered Rooted Trees

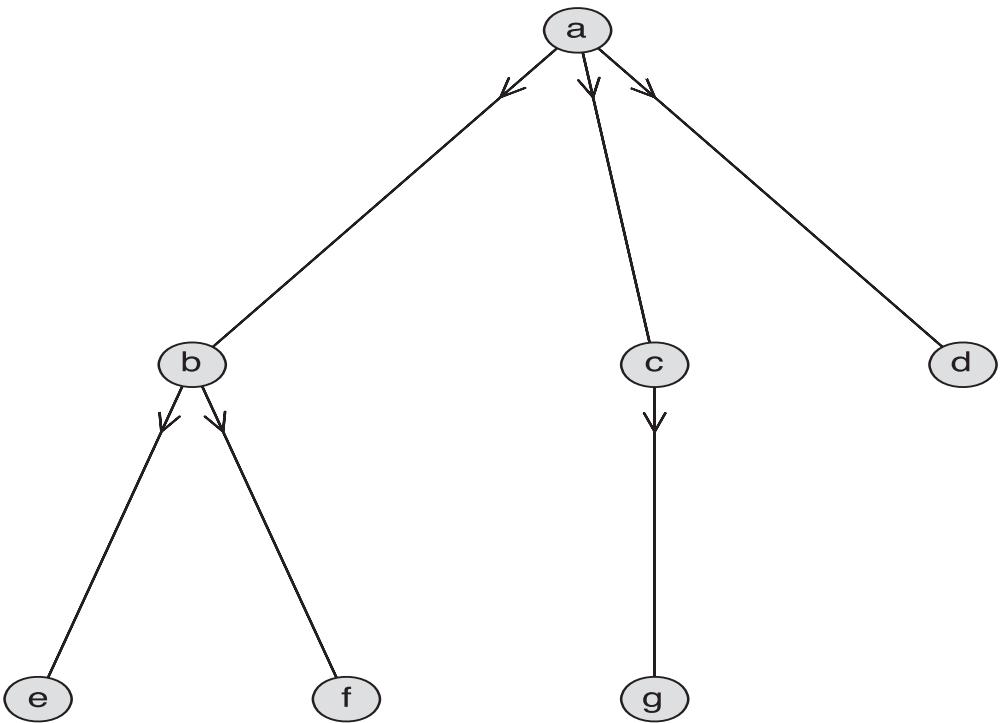
Recall that an ordered rooted tree is a rooted tree in which the children of each internal vertex are ordered.

Representing Ordered Rooted Trees

To represent an ordered rooted tree in Maple, we need to store the order of children. There are many ways to accomplish this, but perhaps the most straightforward is to mark each vertex with its order among its siblings. By way of illustration, we duplicate **firstRooted** from above and make the duplicate ordered.

> *firstOrdered := CopyGraph(firstRooted)*
firstOrdered := Graph 4: a directed unweighted graph with 7 vertices and 6 arc(s) (11.21)

> *DrawRTree(firstOrdered)*



Maple automatically draws the vertices in alphabetical order. But suppose instead we wanted the children of a to be in the order c, b, d , and the children of b to be in the order f then e . Internally, we will represent this by assigning, for each vertex, an “order” attribute. We set the “order” of the root to be 0, and for all other vertices, the “order” attribute will represent the position of that vertex among its siblings. In our example, c will have “order”=1, b will have “order”=2, and d will have “order”=3. Rather than manually using the **SetVertexAttribute** command for each vertex, we define a procedure.

The first argument to this procedure will be the name of a rooted tree. The second argument will be a list of values representing the “order” value to be assigned to each vertex. The procedure will loop through the vertices of the tree and set the “order” vertex attributes.

```

1 SetChildOrder := proc (T :: RTree, orderL :: list (nonnegint) )
2   local V, i;
3   uses GraphTheory;
4   V := Vertices (T);
5   if nops (V) <> nops (orderL) then
6     error "List must have one entry per vertex .";
7   end if ;
8   for i from 1 to nops (V) do
9     SetVertexAttribute (T, V[i], "order"=orderL[i]);
10  end do;
11 end proc;

```

Since the “order” values in the list given to the procedure must match the order of the vertices that is returned from **Vertices(T)**, it is a good idea to double-check the result of the **Vertices** command before using this procedure.

```

> Vertices(firstOrdered)
["a","b","c","d","e","f","g"] (11.22)

> SetChildOrder(firstOrdered,[0,2,1,3,2,1,1])

```

A Type for Ordered Rooted Trees

Now that we have ordered the children in Maple's representation of this tree, **firstOrdered** represents an ordered rooted tree. We will create an **ORTree** type. The requirements for being an ordered rooted tree are: the object must be a rooted tree, every vertex must have an "order" attribute set, and the root's "order" must be 0.

```

1 'type/ORTree' := proc (obj)
2   local v;
3   uses GraphTheory;
4   if not type(obj, RTree) then
5     return false;
6   end if;
7   for v in Vertices(obj) do
8     if GetVertexAttribute(obj, v, "order") = FAIL then
9       return false;
10    end if;
11  end do;
12  v := GetGraphAttribute(obj, "root");
13  if GetVertexAttribute(obj, v, "order") <> 0 then
14    return false;
15  end if;
16  return true;
17 end proc:

```

```

> type(firstOrdered, ORTree)
true (11.23)

```

Drawing Ordered Rooted Trees

While **firstOrdered** is now officially an ordered rooted tree and is storing the order of children, if we draw it, it will not appear with children in the proper order. To accomplish this, we need to create a **DrawORTree** procedure. Recall that when creating the **DrawRTree** procedure, our basic approach was to have Maple determine the correct positions of vertices for us by turning the rooted tree into a graph that it would draw in the manner we wished. We do the same thing in this case.

The key fact is that, in fact, Maple draws trees with the vertices in a fixed order. For example,

```

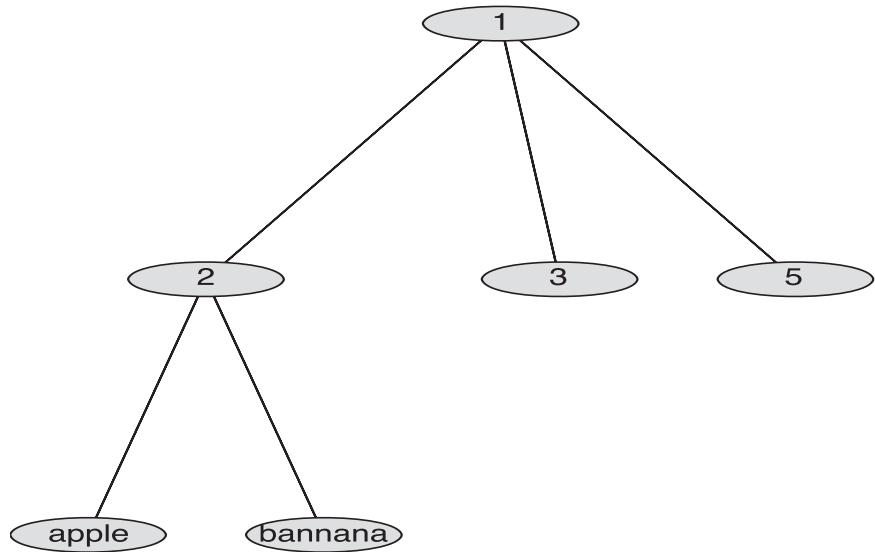
> orderEx1 := Graph({{1,2},{1,3},{1,5},{2,"apple"},{2,"banana"}})
orderEx1 := Graph 5: an undirected unweighted graph with 6 vertices and 5 edge(s) (11.24)

```

```

> DrawGraph(orderEx1)

```



Despite the order in which we listed the edges, Maple sorted the vertices with numeric names in numeric order and it sorted the vertices with string names in lexicographic order. This is because we provided only the edges and made Maple determine the graph's vertices for itself. Notice that when we use the **Vertices** command, the vertices are listed in the same order as they appear in the graph, reading left to right and top to bottom.

```

> Vertices(orderEx1)
[1, 2, 3, 5, "apple", "banana"]
(11.25)

```

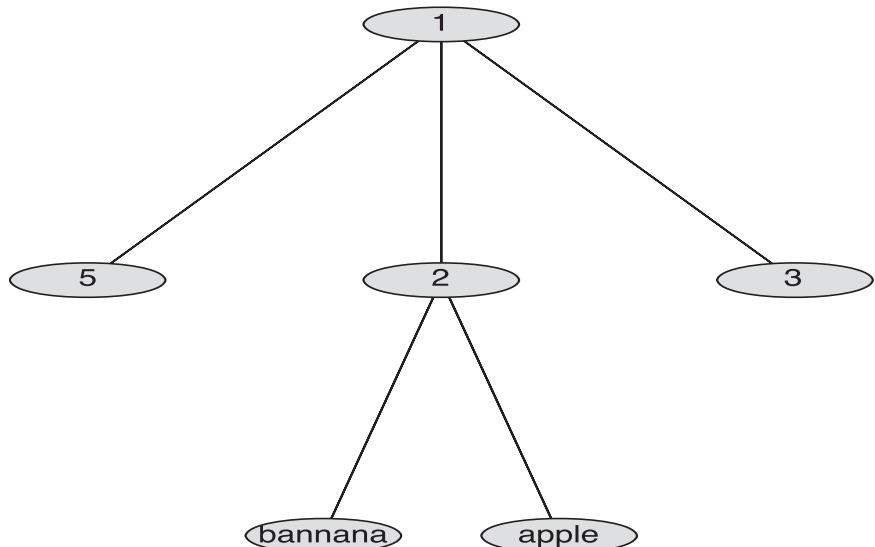
If we give the **Graph** command the list of vertices explicitly and in the order we want them, we can change the graph.

```

> orderEx2 := Graph([1, 5, 2, 3, "banana", "apple"],
    {{1,2}, {1,3}, {1,5}, {2, "apple"}, {2, "banana"}})
orderEx2 := Graph 6: an undirected unweighted graph with 6 vertices and 5 edge(s)
(11.26)

```

> DrawGraph(orderEx2)



We will use this feature in order to properly draw ordered rooted graphs. Given an **ORTree**, we use the **UnderlyingGraph** to access the undirected version of the graph, in particular, the undirected edges. We then determine an order for the vertices that is compatible with the ordering of children. And then we create a new graph using the order of the vertices and the edges from the underlying graph. Finally, as with **DrawRTree**, we use the vertex positions of that tree to draw our **ORTree**.

The key is to create the ordered list of vertices, which we call **Overts**. We will use an approach similar to how we determined that all edges were in the correct direction in the **RTree** type definition. We initialize **Overts** to the list containing only the root and we initialize a counter **i** to 1. We then use the **Departures** command to find all of the children of the root. (We could also use the **FindChildren** command we defined earlier, but using **Departures** directly is a bit more efficient.) We then sort the children in order of their “order” attribute. Once sorted, we add the children to the **Overts** list. Then, increment the counter **i** and repeat. At each step, the counter **i** is used as an index into **Overts** to determine which vertex is being processed. If that vertex has any children, they are sorted according to their “order” attribute and added to the list.

Before writing the main procedure, we create a helper procedure to aid in the sorting of children. Recall that passing the **sort** command a procedure as its optional second argument allows us to specify the order in which it arranges values. This procedure must be Boolean-valued and should return true if the first argument precedes the second argument. In our case, we need a procedure that compares two vertices by comparing their “order” attribute. However, we cannot access the vertex attribute without also having the name of the graph, since **GetVertexAttribute** requires the name of the graph. In other words, the procedure for comparing vertices depends on the graph, but we are only allowed to have two arguments in the procedure. Therefore, we create a function that takes a graph and returns a comparison procedure for that graph.

```

1 VOrderComp := G -> proc (u, v)
2   local uOrder, vOrder;
3   uses GraphTheory;
4   uOrder := GetVertexAttribute (G, u, "order");
5   vOrder := GetVertexAttribute (G, v, "order");
6   evalb (uOrder < vOrder);
7 end proc:
```

Observe that **VOrderComp** applied to our **firstOrdered** example returns a procedure that we can use as the optional argument to **sort**.

```
> sort ([“b”, “c”, “d”], VOrderComp (firstOrdered))
[“c”, “b”, “d”] (11.27)
```

Here, now, is the procedure **DrawORTree**.

```

1 DrawORTree := proc (T :: ORTree)
2   local E, sorter, R, Overts, i, children, G, VP, v, p;
3   uses GraphTheory;
4   E := Edges (UnderlyingGraph (T));
5   sorter := VOrderComp (T);
6   R := GetGraphAttribute (T, "root");
7   Overts := [R];
```

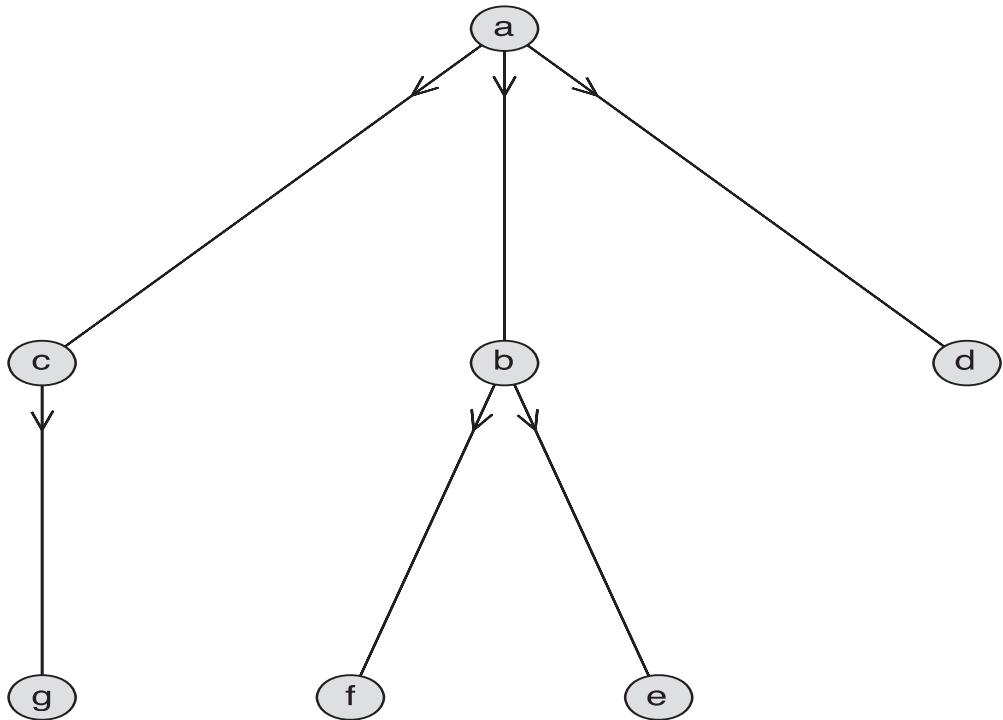
```

8   i := 1;
9   while i <= nops(Overts) do
10      children := Departures(T,Overts[i]);
11      children := sort(children,sorter);
12      Overts := [op(Overts),op(children)];
13      i := i + 1;
14  end do;
15  G := Graph(Overts,E);
16  DrawGraph(G,style=tree,root=R);
17  VP := GetVertexPositions(G);
18  for i from 1 to nops(Vertices(T)) do
19    v := Overts[i];
20    p := VP[i];
21    SetVertexAttribute(T,v,"draw-pos-user"=p);
22  end do;
23  DrawGraph(T);
24 end proc;

```

Finally, we can draw our ordered tree correctly.

> *DrawORTree(firstOrdered)*



Properties of Trees

We conclude this section with procedures for calculating the level of a vertex, the height of a tree, and for determining if a tree is balanced or not.

The level of a vertex in a rooted tree is the length of the path from the root to the vertex. We compute the level in reverse. We first initialize a counter to 0. If the given vertex is the root, then the level

is 0. Otherwise, increment the counter and look at the parent of the original vertex. If this vertex is a root, then the counter holds the level. Otherwise, increment the counter and back up to the parent of the current vertex. When we reach the root, then the value of the counter is the level of the vertex.

```

1 FindLevel := proc (T :: RTree, targetV)
2   local v, level, root;
3   uses GraphTheory;
4   level := 0;
5   v := targetV;
6   root := GetGraphAttribute (T, "root");
7   while v <> root do
8     level := level + 1;
9     v := FindParent (T, v);
10  end do;
11  return level;
12 end proc;
```

We can compute the levels of g and d in the **firstOrdered** tree.

> *FindLevel(firstOrdered, “g”)*
2 (11.28)

> *FindLevel(firstOrdered, “d”)*
1 (11.29)

The height of a tree is the maximum of the levels of the vertices. We can compute the height by checking each vertex's level. We use a variable to hold the largest level and each time we find a vertex with a level larger than the current maximum, we update the variable.

```

1 FindHeight := proc (T :: RTree)
2   local height, v, level;
3   uses GraphTheory;
4   height := 0;
5   for v in Vertices (T) do
6     level := FindLevel (T, v);
7     if level > height then
8       height := level;
9     end if;
10  end do;
11  return height;
12 end proc;
```

> *FindHeight(firstOrdered)*
2 (11.30)

Recall that a rooted tree of height h is balanced if all leaves are at level h or $h - 1$. To determine if a given tree is balanced, we need to: (1) calculate the height of the tree, (2) find all the leaves of the

tree with the **FindLeaves** procedure we wrote earlier, (3) test each leaf’s level and return false if it is higher than level $h - 1$.

```
1 IsBalanced := proc (T :: RTree)
2   local height, leaves, v;
3   height := FindHeight (T);
4   leaves := FindLeaves (T);
5   for v in leaves do
6     if FindLevel (T, v) < height - 1 then
7       return false;
8     end if;
9   end do;
10  return true;
11 end proc;
```

We see that our **firstOrdered** tree is balanced.

> *IsBalanced(firstOrdered)*
true (11.31)

11.2 Applications of Trees

This section is concerned with applications of trees, particularly binary trees. Specifically, we consider the use of trees in binary search algorithms as well as in Huffman codes. The reason we use binary trees is that we can use the binary structure of the tree to make binary decisions (e.g., less than/greater than) regarding search paths or insertion of elements. Additionally, the binary tree structure corresponds well with the way computers store information as binary data.

Recall that a tree is called a binary tree if all vertices in the tree have at most two children. In this section, we will be using ordered binary trees. The fact that the vertices are ordered means that the children of a vertex can be considered to be either a left child or a right child. By convention, we consider the left child to be the first child and the right child to be second.

Representation in Maple

Before we get to the applications, we will discuss how we can represent binary trees in Maple and develop some procedures to help us manipulate them. Since a binary tree is a particular kind of ordered rooted tree, our work here should be consistent with what we did above.

A Binary Tree Type

We will construct a type, **BTree**, for binary trees. We will impose three conditions for an object to be considered a **BTree**. First, it must be an ordered rooted tree, that is, an **ORTree**. Second, it must be binary, that is, each vertex must have at most two children. And third, each vertex other than the root must have “order” attribute 1 or 2, with 1 indicating that the vertex is a left child and 2 for right. The root will have its “order” attribute set to 0.

First, we construct an example of a binary tree. The tree we construct is the binary search tree for the letters D, B, F, A, C, E. Recall the **SetChildOrder** procedure we developed above. Its second argument is a list of the “order” attributes, listed in the same order as the vertices appear in the graph’s vertex list.

```

> firstBTree := Graph(["D", "B", "F", "A", "C", "E"], {[["B", "A"], 
    ["B", "C"], ["D", "B"], ["D", "F"], ["F", "E"]]})  

firstBTree := Graph 7: a directed unweighted graph with 6 vertices and 5 arc(s) (11.32)

```

```

> SetGraphAttribute(firstBTree, "root" = "D")  

> SetChildOrder(firstBTree, [0, 1, 2, 1, 2, 1])

```

Now that we have an example, we create the type. To check that the tree is in fact binary, we can use the **Departures** command and count the number of children of each vertex. If any vertex has more than two children, then it is not binary. Moreover, we make sure that each vertex is marked with an order of 1 or 2, or 0 in the case of the root.

```

1  'type/BTree' := proc(obj)
2    local R, v, vpos;
3    uses GraphTheory;
4    if not type(obj, ORTree) then
5      return false;
6    end if;
7    R := GetGraphAttribute(obj, "root");
8    for v in Vertices(obj) do
9      if nops(Departures(obj, v)) > 2 then
10        return false;
11      end if;
12      vpos := GetVertexAttribute(obj, v, "order");
13      if (vpos = 0 and v <> R) or
14        ((vpos <> 0) and (vpos <> 1) and (vpos <> 2)) then
15        return false;
16      end if;
17    end do;
18    return true;
19  end proc;

```

```

> type(firstBTree, BTree)  

true

```

(11.33)

Drawing Binary Trees

Next, we write a procedure for drawing binary trees. Note that, while **DrawORTree** will draw a binary tree, that procedure will display vertices that are “only children” directly below their parent rather than to the left or right. We will explicitly calculate the position of each vertex.

Think about the tree as being drawn in a 1 by 1 box with (0, 0) at the bottom left corner. The y-coordinate of each vertex will depend on the height of the tree and the level of the vertex. Specifically, the y-coordinate of any vertex is $1 - l/h$, where l is the level of the vertex and h is the height of the tree. This way, the root, which is at level 0, has y-coordinate 1 and the vertices in the last level have y-coordinate 0.

For the x -coordinates, the position of the vertex depends on the position of its parent and its level. We start by setting the x -position of the root to 1/2. The left child of the root will be drawn at

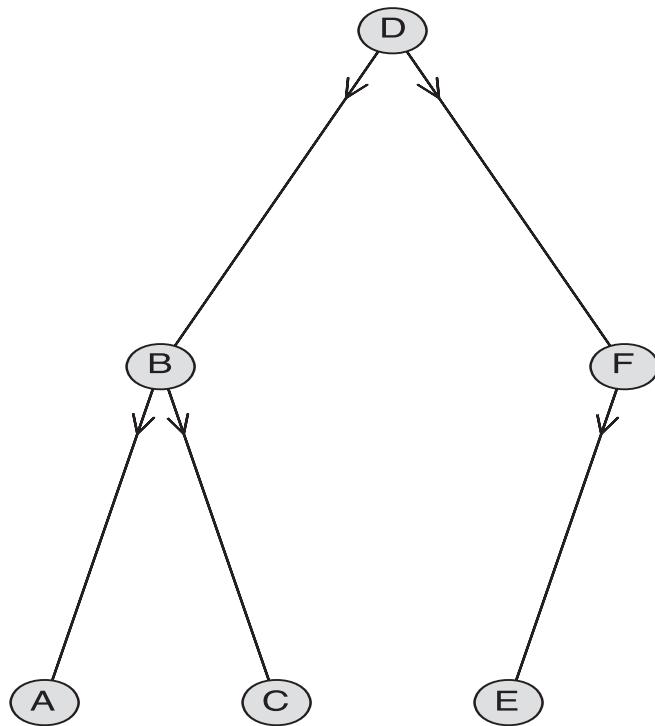
x -coordinate $1/4$ and the right child at $3/4$. We can think about the children of the root as being drawn $1/4$ to the left of the root and $1/4$ to the right of the root, respectively. That is, the x -coordinate of the root's left child is $\frac{1}{2} - \frac{1}{4}$, and the x -coordinate of the right child is $\frac{1}{2} + \frac{1}{4}$. Generally, for a vertex in level l , we can calculate its x -coordinate as the x -coordinate of its parent plus (for right children) or minus (for left) $\frac{1}{2^{l+1}}$.

The **DrawBTree** procedure is below. It begins by calculating the height of the tree with the **FindHeight** procedure we created earlier. It then processes the root of the tree by setting its position to $(1/2, 1)$. (Note that we set the position of the vertex by setting the “draw-pos-user” attribute of the vertex. This is in some ways more convenient than assembling all of the vertex positions and then using **SetVertexPositions**.) We also create a list, **Verts**, and populate it with the root's children. We then begin a loop with the same structure as the while loop in the **RTree** type definition. We have a counter **i** initialized to 1. This counter serves as an index into the **Verts** list. At each step in the while loop, we do several things. First, we use the **FindLevel** procedure we created earlier to determine the level of the vertex. Second, the y -coordinate is calculated by the formula $1 - l/h$. Third, the x -coordinate is calculated by accessing the x -coordinate of the parent and adding or subtracting $\frac{1}{2^{l+1}}$. Fourth, we set the “draw-pos-user” attribute for the vertex. Finally, the children of the current vertex are added to the **Verts** list and the counter is incremented.

```

1 DrawBTree := proc(T::BTree)
2   local height, v, i, level, Verts, x, y, parent, side;
3   uses GraphTheory;
4   height := FindHeight(T);
5   v := GetGraphAttribute(T, "root");
6   SetVertexAttribute(T, v, "draw-pos-user"=[1/2, 1]);
7   Verts := FindChildren(T, v);
8   i := 1;
9   while i <= nops(Verts) do
10    v := Verts[i];
11    level := FindLevel(T, v);
12    y := 1 - level/height;
13    parent := FindParent(T, v);
14    x := GetVertexAttribute(T, parent, "draw-pos-user") [1];
15    if GetVertexAttribute(T, v, "order") = 1 then
16      side := -1;
17    else
18      side := 1;
19    end if;
20    x := x + side * 1/(2^(level+1));
21    SetVertexAttribute(T, v, "draw-pos-user"=[x, y]);
22    Verts := [op(Verts), op(FindChildren(T, v))];
23    i := i + 1;
24  end do;
25  DrawGraph(T);
26 end proc;
```

> *DrawBTree(firstBTree)*



Parents and Children

In the previous section, we created the procedure **FindParent**, which returns the parent of a given vertex in the given tree. This procedure works on **BTrees** as well.

> *FindParent(firstBTree, "C")
"B"* (11.34)

We had also created the **FindChildren** procedure, which we could also use with binary trees. However, since we are dealing with binary trees, we want to be more specific and be able to determine the left and right children of a given vertex. Finding the left (respectively, right) child of a given vertex can be done by looking at each child of the vertex and checking the “order” attribute. The child with “order” 1 is the left child and is returned by the procedure (respectively, 2 and right child). If there is no left (respectively, right), child, the procedure will return **NULL**.

```

1 FindLeftChild := proc (T::BTREE, v)
2   local children, w, pos;
3   uses GraphTheory;
4   children := FindChildren(T, v);
5   for w in children do
6     pos := GetVertexAttribute(T, w, "order");
7     if pos = 1 then
8       return w;
9     end if;
10   end do;
11   return NULL;
12 end proc;
13 FindRightChild := proc (T::BTREE, v)
14   local children, w, pos;
  
```

```

15  uses GraphTheory;
16  children := FindChildren(T, v);
17  for w in children do
18    pos := GetVertexAttribute(T, w, "order");
19    if pos = 2 then
20      return w;
21    end if;
22  end do;
23  return NULL;
24 end proc;

```

> *FindLeftChild(firstBTree, “F”)*
“E”

> *FindRightChild(firstBTree, “F”)*

(11.35)

Building Binary Trees

We will also want procedures to create and build up a binary tree. Specifically, we create a procedure that, given the label for the root of a binary tree, creates the tree with that vertex as its root. We will then write procedures that, given a binary tree, a vertex in the tree, and a label for a new vertex, adds the new vertex as the left or right child of the given vertex.

The **NewBTree** procedure creates the binary tree consisting of a single vertex, the root of the tree.

```

1 NewBTree := proc (R)
2   local T;
3   uses GraphTheory;
4   T := Digraph([R]);
5   SetGraphAttribute(T, "root"=R);
6   SetVertexAttribute(T, R, "order"=0);
7   return T;
8 end proc;

```

Adding a child to a vertex in a binary tree requires three basic steps. We must add a vertex to the graph with the **AddVertex** command, add a directed edge from the parent to the new vertex with the **AddArc** command, and then use **SetVertexAttribute** to identify the new child as left or right by setting the “order” attribute to 1 or 2, respectively.

Recall, however, that the **AddVertex** command does not modify the original graph, but instead creates a new graph whose vertices are the original vertices together with the new vertex and whose edge set is identical to the original. The **AddVertex** command, however, does not preserve attributes. As a result, we will need to manually copy the “root” attribute for the graph and the “order” attributes on each vertex.

```

1 AddLeftChild := proc (T::BTree, v, newV)
2   local newT, temp, w;
3   uses GraphTheory;
4   if FindLeftChild(T, v) <> NULL then
5     error "Vertex already has a left child.";

```

```

6   end if ;
7   newT := AddVertex(T, newV) ;
8   AddArc(newT, [v, newV]) ;
9   temp := GetGraphAttribute(T, "root") ;
10  SetGraphAttribute(newT, "root"=temp) ;
11  for w in Vertices(T) do
12    temp := GetVertexAttribute(T, w, "order") ;
13    SetVertexAttribute(newT, w, "order"=temp) ;
14  end do ;
15  SetVertexAttribute(newT, newV, "order"=1) ;
16  return newT ;
17 end proc:
18 AddRightChild := proc (T::BTree, v, newV)
19   local newT, temp, w;
20   uses GraphTheory;
21   if FindRightChild(T, v) <> NULL then
22     error "Vertex already has a right child." ;
23   end if ;
24   newT := AddVertex(T, newV) ;
25   AddArc(newT, [v, newV]) ;
26   temp := GetGraphAttribute(T, "root") ;
27   SetGraphAttribute(newT, "root"=temp) ;
28   for w in Vertices(T) do
29     temp := GetVertexAttribute(T, w, "order") ;
30     SetVertexAttribute(newT, w, "order"=temp) ;
31   end do ;
32   SetVertexAttribute(newT, newV, "order"=2) ;
33   return newT ;
34 end proc:

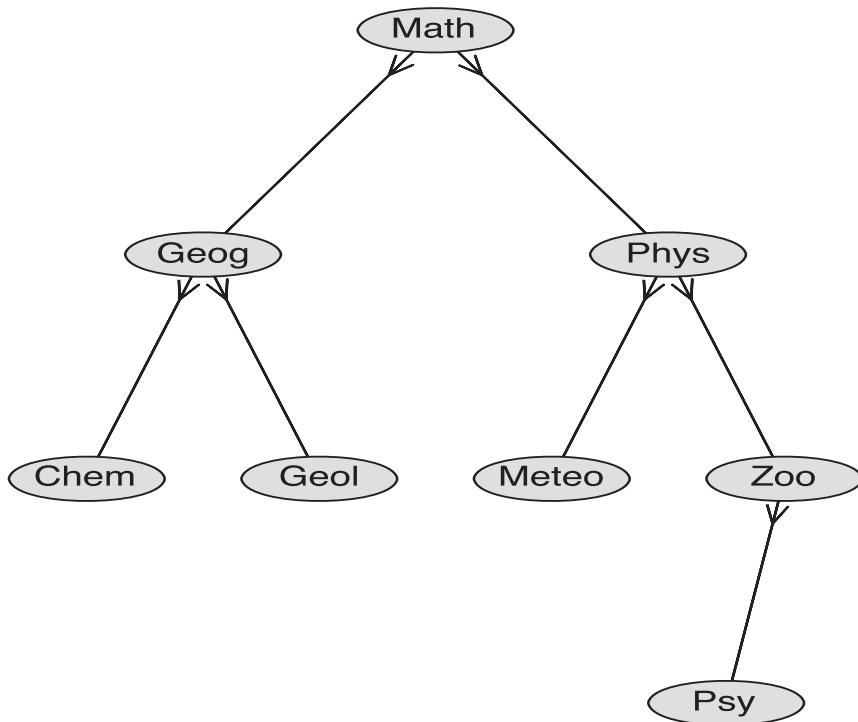
```

With **NewBTree**, **AddLeftChild**, and **AddRightChild**, we can now construct binary trees one vertex at a time. We will illustrate this by creating the binary search tree described in Example 1 of the text by following the steps illustrated in Figure 1. We abbreviate the words in order to make the image more readable.

```

> Fig1Tree := NewBTree("Math") :
> Fig1Tree := AddRightChild(Fig1Tree, "Math", "Phys") :
> Fig1Tree := AddLeftChild(Fig1Tree, "Math", "Geog") :
> Fig1Tree := AddRightChild(Fig1Tree, "Phys", "Zoo") :
> Fig1Tree := AddLeftChild(Fig1Tree, "Phys", "Meteo") :
> Fig1Tree := AddRightChild(Fig1Tree, "Geog", "Geol") :
> Fig1Tree := AddLeftChild(Fig1Tree, "Zoo", "Psy") :
> Fig1Tree := AddLeftChild(Fig1Tree, "Geog", "Chem") :
> DrawBTree(Fig1Tree)

```



Binary Insertion

A key benefit of binary search trees is that the search time required to find a specific element of the tree is logarithmic in the number of vertices of the tree. The drawback is that the initial insertion of a vertex is more expensive.

The procedure for constructing a binary search tree by insertion is described in Algorithm 1 in Section 11.2 of the textbook. We will implement this algorithm as the procedure **BInsertion**.

The **BInsertion** procedure will accept two input values: a binary search tree and a vertex to be found or added. The procedure returns true if the vertex is found to already be in the tree, and if not, it will add the vertex to the tree and return false.

We begin by locating the root of the tree by checking the tree’s “root” attribute and setting the local variable **v** to the root. Then, we start a while loop which continues provided two conditions are met. First, that **v <> NULL**. If we discover that the value we are searching for is not in the tree, then we will add it to the tree and set **v** to **NULL**, to indicate that we had to add a vertex, and this terminates the while loop. The second condition is that **v <> x**, where **x** is the value we are searching for. If **v = x**, then we have found the vertex and thus the loop should terminate. (Note that Maple identifies a vertex with its label, so, unlike the text, we do not distinguish between **v** and *label(v)*.)

Within the while loop, there are two possibilities. Either the sought-after value is less than **v** or it is greater than **v**. They cannot be equal because that is one of the terminating conditions for the while loop. If the target value is less than **v**, then we consider the left child of **v**. If there is no left child, then we know that the value is not in the tree and so we add the value as the left child of **v** and then set **v** to **NULL** to indicate that the desired value was not already in the tree. If there is a left child, then we set **v** equal to it and continue the loop. If the target value is greater than **v**, we proceed in exactly the same way, substituting right for left.

Once the loop terminates, we check the value of **v**. If **v** is **NULL**, then we know that the desired value was not found and the algorithm returns false. If **v** is not **NULL**, then we return true.

The **evaln** Parameter Modifier

Before providing the definition of the insertion procedure, we also mention the **evaln** parameter modifier. We want the **BInsertion** procedure to be able to modify the tree it was given. Normally, when you provide a parameter to a procedure, the procedure cannot assign to the parameter. Consider the following simple procedure.

```
> five := 5  
      five := 5
```

(11.36)

```
1 AddOne := proc (n)  
2     n := n + 1;  
3 end proc;
```

```
> AddOne (five)
```

Error, (in AddOne) illegal use of a formal parameter

This procedure produces an error when we attempt to assign a value to the parameter **n**. This is a feature of programming in Maple that is designed to encourage good programming practices. In particular, for a procedure to modify one of its arguments, we need to be very explicit that we really want to do so. This helps prevent unintended consequences—accidentally modifying a parameter can cause serious bugs in your programs.

You can think about what is going on in the **AddOne** procedure this way: when you call the procedure with the syntax **AddOne(five)**, all of the occurrences of the name **n** are resolved to the object **5**, which is the value stored in **five**. Thus, the command **n := n + 1;** resolves to **5 := 5 + 1;**. Clearly, that is not a legal command.

The **evaln** modifier provides a way around this. For example, if we declare the parameter to be **n::evaln** in the **AddOne** procedure, we are telling Maple to not evaluate the parameter **n** into the object that it refers to, but to evaluate the parameter into a name. Instead of evaluating the variable **five** to get the object **5** and replacing the parameter **n** with **5**, the variable **five** is evaluated into the name **five** and the parameter **n** is replaced with the name **five**. This means that the command becomes **five := five + 1;**. This still does not quite work, because the **five** on the right-hand side of the assignment is now a name, and we cannot add integers to names. We need to force the name **five** to be evaluated to the object **5** with the **eval** command.

```
1 AddOne2 := proc (n::evaln)  
2     n := eval(n) + 1;  
3 end proc;
```

Now, we can see that this procedure modifies the variable it is given.

```
> seven := 7  
      seven := 7
```

(11.37)

```
> AddOne2(seven)
8
```

(11.38)

```
> seven
8
```

(11.39)

To summarize, we can create procedures that modify a variable that is passed to them by marking the parameter with the modifier **evaln**. This allows the parameter to appear on the right side of an assignment, but the trade-off is that all other occurrences of the parameter must have an explicit **eval**.

Implementation of Binary Insertion

Here, now, is the binary insertion algorithm.

```
1 BInsertion := proc (BST::evaln, x)
2   local v;
3   uses GraphTheory;
4   if not type(eval(BST), BTree) then
5     BST := NewBTree(x);
6     return false;
7   end if;
8   v := GetGraphAttribute(eval(BST), "root");
9   while v <> NULL and v <> x do
10    if x < v then
11      if FindLeftChild(eval(BST), v) = NULL then
12        BST := AddLeftChild(eval(BST), v, x);
13        v := NULL;
14      else
15        v := FindLeftChild(eval(BST), v);
16      end if;
17    else
18      if FindRightChild(eval(BST), v) = NULL then
19        BST := AddRightChild(eval(BST), v, x);
20        v := NULL;
21      else
22        v := FindRightChild(eval(BST), v);
23      end if;
24    end if;
25  end do;
26  return evalb(v <> NULL);
27 end proc;
```

Note that we begin the procedure by testing to see if the object stored in the parameter **BST** is in fact a binary tree. If it is not, we use the **NewBTree** command to create a tree and store it in the name given by the parameter. This way we can use **BInsertion** to create a new tree in addition to inserting elements in an existing tree.

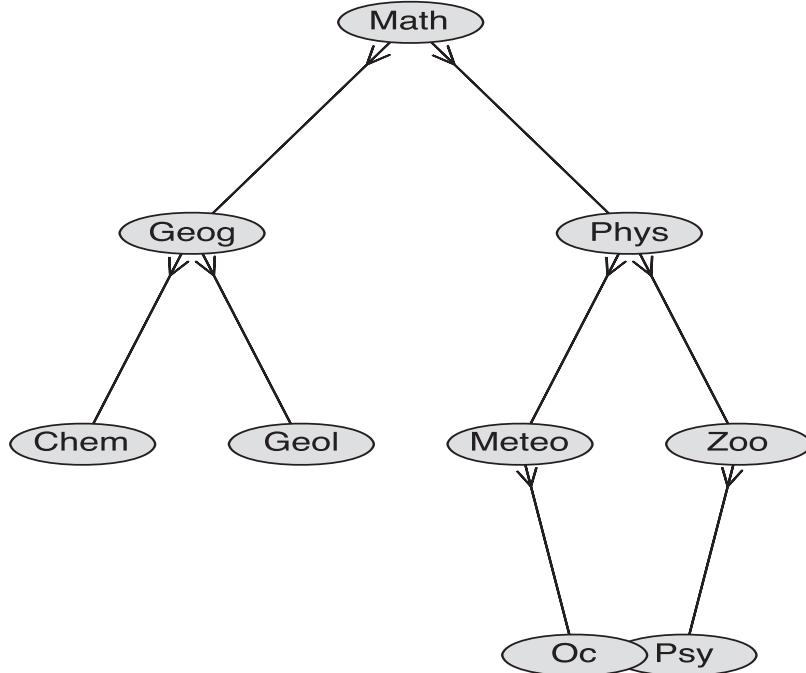
Now, we see if oceanography is in the **Fig1Tree** of academic subjects.

```
> BInsertion(Fig1Tree, "Oc")
false
```

(11.40)

The procedure returned false indicating that “Oc” was not found in the tree. Graphing **Fig1Tree**, we see that it was added as a child of meteorology.

> *DrawBTree (Fig1Tree)*



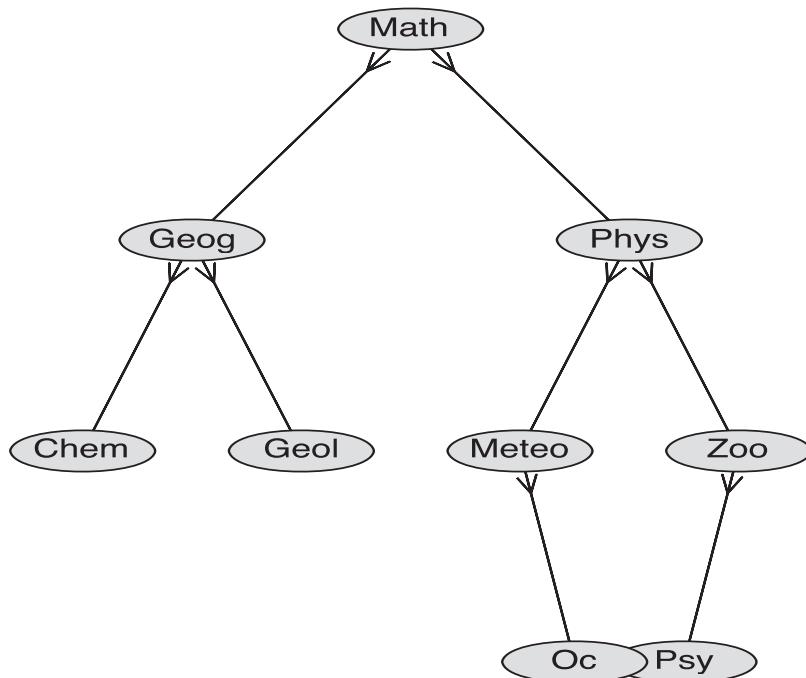
On the other hand, zoology is already in the tree and so the tree is not modified.

> *BInsertion (Fig1Tree, "Zoo")*

true

(11.41)

> *DrawBTree (Fig1Tree)*



Constructing a Binary Search Tree from a List

To conclude our discussion of binary search trees, we will create a procedure that takes a list of values and successively uses the **BInsertion** procedure to create a binary search tree for the given list.

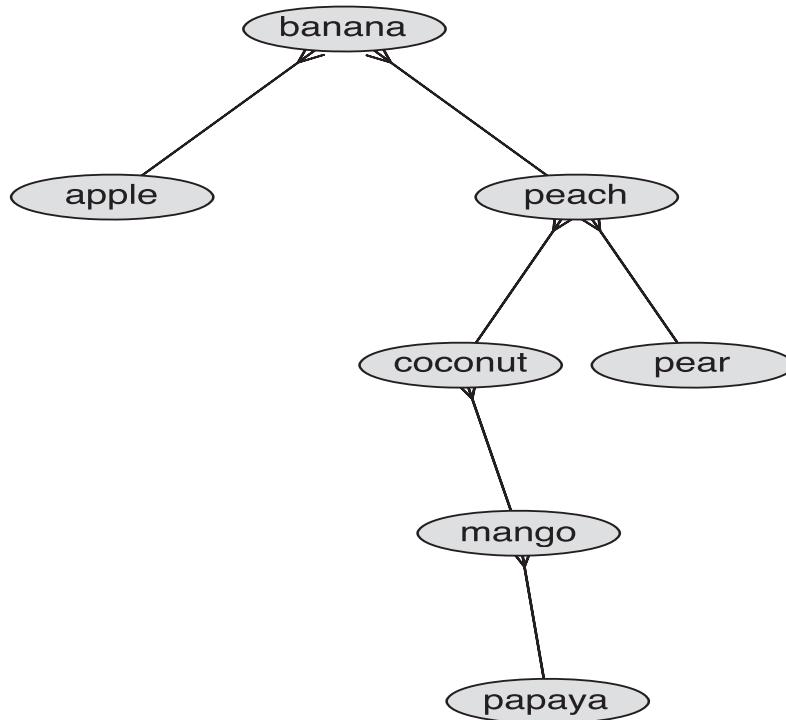
```
1 MakeBST := proc (L : :list)
2   local T, v;
3   for v in L do
4     BInsertion(T, v);
5   end do;
6   return T;
7 end proc:
```

We use this to complete Exercise 1 from Section 11.2.

> *Exercise1* := MakeBST(["banana", "peach", "apple", "pear",
"coconut", "mango", "papaya"])

Exercise1 := Graph 8: a directed unweighted graph with 7 vertices and 6 arc(s) (11.42)

> *DrawBTree(Exercise1)*



Huffman Coding

Huffman coding is a method for constructing an efficient prefix code for a set of characters. It is based on a greedy algorithm, where at each step the vertices with the least weight are combined. It can be shown that Huffman coding produces optimal prefix codes. The algorithm that we will implement is described in Algorithm 2 of Section 11.2.

Creating the Initial Forest

We begin with a list of symbols and their weights, or frequencies. The first step is to create the initial forest, which we will implement as a list of trees. For each symbol, we will create the binary tree consisting of a single vertex corresponding to the symbol.

We create a procedure, similar to **NewBTree**, which, in addition to creating the binary tree, also assigns a “weight” attribute to the graph to store the weight of the symbol.

```
1 NewHTree := proc (s, w)
2   local T;
3   uses GraphTheory;
4   T := Digraph ([s]);
5   SetGraphAttribute (T, "root"=s);
6   SetGraphAttribute (T, "weight"=w);
7   SetVertexAttribute (T, s, "order"=0);
8   return T;
9 end proc;
```

Now, we write an algorithm to create the initial forest. We assume that the data is provided as a list of pairs consisting of the symbol and the weight.

```
1 CreateForest := proc (L: :list (list) )
2   local forest, M, v, w;
3   forest := [ ];
4   for M in L do
5     v := M[1];
6     w := M[2];
7     forest := [op(forest), NewHTree (v, w)];
8   end do;
9   return forest;
10 end proc;
```

Using this procedure, we form the initial forest for Exercise 23 from Section 11.2. Note that the result is a list of graph objects.

```
> Ex23Forest := CreateForest ([[“a”, 0.20], [“b”, 0.10], [“c”, 0.15], [“d”, 0.25],
[“e”, 0.30]])
Ex23Forest := [Graph 9: a graph with 1 vertex and no edges,
Graph 10: a graph with 1 vertex and no edges,
Graph 11: a graph with 1 vertex and no edges,
Graph 12: a graph with 1 vertex and no edges,
Graph 13: a graph with 1 vertex and no edges] (11.43)
```

The main work of the Huffman coding algorithm is to determine the two members of the forest with the smallest weights. These two trees are then assembled into a single tree whose root is a new vertex with left and right children being the trees with the lowest and second lowest weights. The new tree’s weight is the sum of the weights of the two original trees.

Sorting the Forest

Recall that the **sort** command accepts an optional second argument, specifically, a procedure on two arguments that returns true if the first argument precedes the second in the desired order. The following procedure will be of this kind, accepting two binary trees as input and comparing their weights.

```
1 CompareTrees := proc (A :: BTree, B :: BTree)
2   local a, b;
3   uses GraphTheory;
4   a := GetGraphAttribute (A, "weight");
5   b := GetGraphAttribute (B, "weight");
6   return evalb(a < b);
7 end proc;
```

> *Ex23Forest := sort(Ex23Forest, CompareTrees)*

*Ex23Forest := [Graph 10: a graph with 1 vertex and no edges,
Graph 11: a graph with 1 vertex and no edges,
Graph 9: a graph with 1 vertex and no edges,
Graph 12: a graph with 1 vertex and no edges,
Graph 13: a graph with 1 vertex and no edges]*

(11.44)

Combining Two Trees

Next, we need to take two binary trees and create a new binary tree with one tree as the left subtree of the new root and the other as the right subtree. This procedure will require three arguments: the name of the new root, the left subtree, and the right subtree.

We create the new tree by: (1) combining the vertex lists of the original trees and adding the new vertex; (2) merging the two sets of edges of the original trees and adding two new edges linking the new root to the previous roots; (3) copying the “position” attribute from the original trees and then changing the “position” attributes for the two original roots to reflect their new status as left and right children in the new tree; (4) setting the graph attribute “root” and setting the “weight” attribute to be the sum of the weights of the two original trees. In addition, (5) we will use edge weights of 0 and 1 to label the edges, so the edge weights also need to be copied for the original edges and added to the new edges.

```
1 JoinHTrees := proc (newR, A :: BTree, B :: BTree)
2   local newT, newVerts, Aroot, Broot, newEdges, v, e, p, w;
3   uses GraphTheory;
4   newVerts := [newR, op(Vertices(A)), op(Vertices(B))];
5   Aroot := GetGraphAttribute (A, "root");
6   Broot := GetGraphAttribute (B, "root");
7   newEdges := Edges (A) union Edges (B) union
8     { [newR, Aroot], [newR, Broot] };
9   newT := Graph (newVerts, newEdges, weighted);
10  for v in Vertices (A) do
11    p := GetVertexAttribute (A, v, "order");
12    SetVertexAttribute (newT, v, "order"=p);
13  end do;
14  for v in Vertices (B) do
```

```

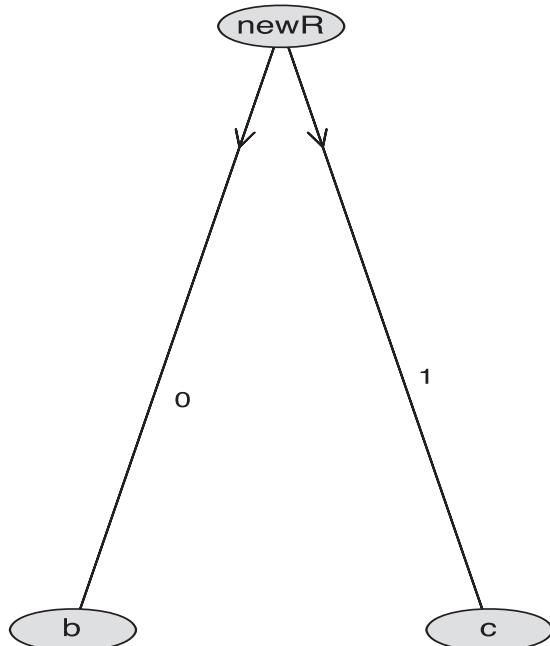
14      p := GetVertexAttribute(B, v, "order");
15      SetVertexAttribute(newT, v, "order"=p);
16  end do;
17  for e in Edges(A) do
18      p := GetEdgeWeight(A, e);
19      SetEdgeWeight(newT, e, p);
20  end do;
21  for e in Edges(B) do
22      p := GetEdgeWeight(B, e);
23      SetEdgeWeight(newT, e, p);
24  end do;
25  SetVertexAttribute(newT, Aroot, "order"=1);
26  SetVertexAttribute(newT, Broot, "order"=2);
27  SetVertexAttribute(newT, newR, "order"=0);
28  SetEdgeWeight(newT, [newR, Aroot], 0);
29  SetEdgeWeight(newT, [newR, Broot], 1);
30  SetGraphAttribute(newT, "root"=newR);
31  w := GetGraphAttribute(A, "weight") +
32      GetGraphAttribute(B, "weight");
33  SetGraphAttribute(newT, "weight"=w);
34  return newT;
end proc;

```

As an example, we join the first two graphs in our sorted **Ex23Forest**.

> *exampleJoin* := *JoinHTrees*("newR", *Ex23Forest*[1], *Ex23Forest*[2])
exampleJoin := Graph 14: a directed weighted graph with 3 vertices and 2 arc(s) (11.45)

> *DrawBTree*(*exampleJoin*)



Implementing the Main Procedure

We now have the major pieces of the Huffman algorithm assembled and we can write the **HuffmanCode** algorithm. This algorithm will accept as input the same list of symbol–weight pairs as the **CreateForest** procedure did. The procedure’s first step is to create the forest **F**. Then, we begin a while loop that continues as long as the list **F** contains more than one member. Inside the while loop, we first use the **CompareTrees** procedure to sort the forest in increasing order of weight. Then, we use the **JoinTrees** procedure to join the first two trees in the forest and we add that new tree to the list, replacing the original two.

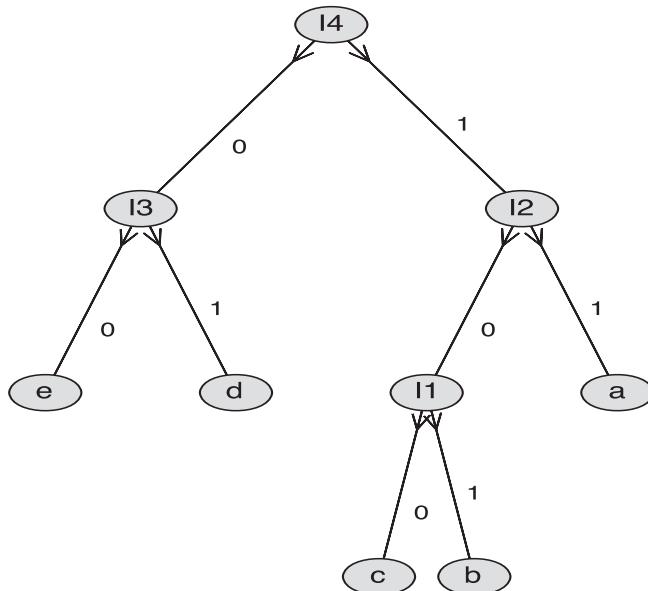
```

1  HuffmanCode := proc (L : : list (list) )
2      local F, i, tempT;
3      uses GraphTheory;
4      F := CreateForest (L) ;
5      i := 0;
6      while nops (F) > 1 do
7          F := sort (F, CompareTrees) ;
8          i := i + 1;
9          tempT := JoinHTrees (cat ("I", i) , F [2] , F [1] ) ;
10         F := [ op (F [3..-1]) , tempT ] ;
11     end do;
12     return F [1] ;
13 end proc;
```

Note that we need a name for the new root when we join two trees. Since these will be the internal vertices of the final tree, we call them I1, I2, I3, etc. We keep a counter **i** and use the string concatenation operator **cat** to create the names of the internal vertices.

> *Ex23HCode := HuffmanCode([[“a”, 0.20], [“b”, 0.10], [“c”, 0.15],
[“d”, 0.25], [“e”, 0.30]])*
Ex23HCode := Graph 15: a directed weighted graph with 9 vertices and 8 arc(s)
(11.46)

> *DrawBTree (Ex23HCode)*



Encoding Strings Using the Huffman Code Tree

We conclude this section by writing a procedure to encode a string of symbols using a given Huffman tree. Note that you can use the list selection notation on a string to access its individual characters. For example,

```
> exampleString := "Hello"  
exampleString := "Hello" (11.47)
```

```
> exampleString[2]  
"e" (11.48)
```

Since we encode a string by assembling the codes for individual characters, we start with a procedure for encoding a single **character**. We assemble the character's code right to left. Beginning with the vertex corresponding to the desired character, we find that vertex's parent. The last digit of the code is the weight of the corresponding edge. The next rightmost digit is the weight of the edge connecting the next parent. We continue until we reach the root.

```
1 EncodeCharacter := proc (H::BTree, c::character)  
2   local code, vertex, parent, digit;  
3   uses GraphTheory;  
4   vertex := c;  
5   code := "";  
6   while FindParent (H, vertex) <> NULL do  
7     parent := FindParent (H, vertex);  
8     digit := GetEdgeWeight (H, [parent, vertex]);  
9     code := cat ("", digit, code);  
10    vertex := parent;  
11  end do;  
12  return code;  
13 end proc:
```

```
> EncodeCharacter (Ex23HCode, "c")  
"100" (11.49)
```

(Note: in updating **code**, we begin the concatenation with the empty string, `""`. This is because **cat** returns an object of the same type as its first argument. If we did not begin with the empty string, the result would not be a string.)

To encode a string, we encode each character and assemble the results. We also make use of the **length** command in this procedure. Applied to a string, **length** returns the number of characters in the string.

```
1 EncodeString := proc (H::BTree, S::string)  
2   local code, charcode, i;  
3   code := "";  
4   for i from 1 to length (S) do  
5     charcode := EncodeCharacter (H, S[i]);
```

```

6   code := cat(code, charcode);
7   end do;
8   return code;
9 end proc:
```

We use this to encode the word “added”.

> *EncodeString(Ex23HCode, “added”)*
“1101010001” (11.50)

11.3 Tree Traversal

In this section, we show how to use Maple to carry out tree traversals. Recall that a tree traversal algorithm is a procedure for systematically visiting every vertex of an ordered rooted tree. We will provide procedures for three important tree traversal algorithms: preorder traversal, inorder traversal, and postorder traversal. We will then show how to use these traversal methods to produce the prefix, infix, and postfix notations for arithmetic expressions.

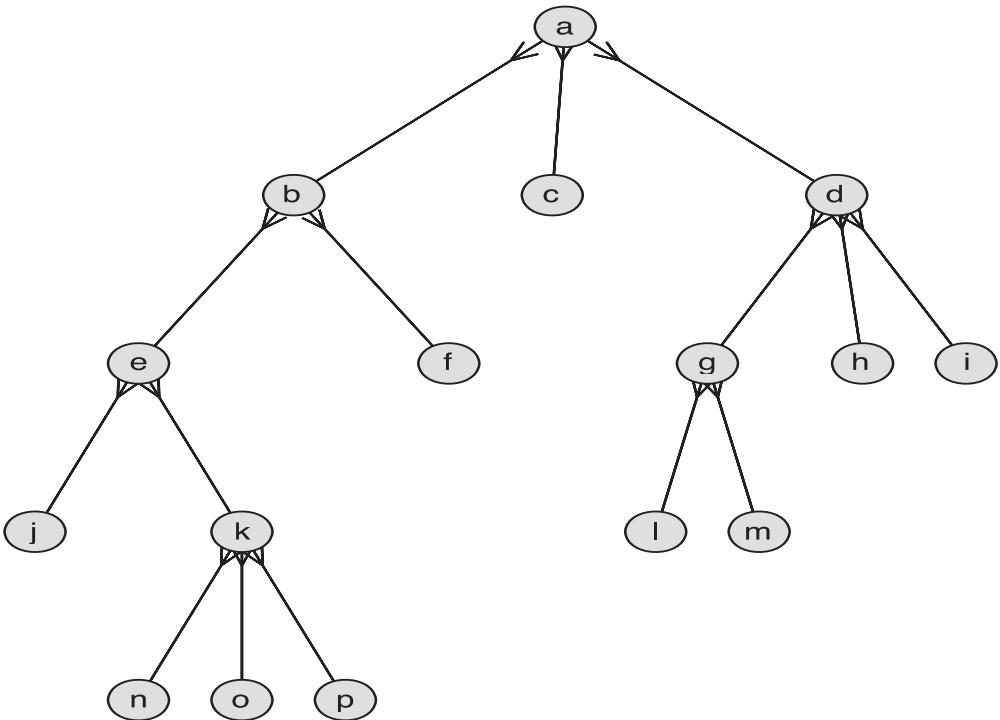
These tree traversal algorithms all require that the tree be rooted and ordered. Recall how we implemented ordered rooted trees in Section 1. An **ORTree** is an **RTree** with the additional restriction that each vertex has an “order” attribute.

Also recall that we created the function **VOrderComp**, which takes an ordered rooted tree and returns a procedure that compares vertices based on their “order” attribute. The procedure returned by **VOrderComp** can be used as an optional argument to **sort** to sort lists of vertices based on the “order” attribute.

To begin, we will create an ordered tree to use as an example as we explore the three traversal algorithms. This example is a reproduction of Figure 3 from Section 11.3.

> *Fig3ORTree := Graph({[“a”, “b”], [“a”, “c”], [“a”, “d”], [“b”, “e”],*
[“b”, “f”], [“d”, “g”], [“d”, “h”], [“d”, “i”], [“e”, “j”], [“e”, “k”], [“g”, “l”],
[“g”, “m”], [“k”, “n”], [“k”, “o”], [“k”, “p”]})
Fig3ORTree := Graph 16: a directed unweighted graph with 16 vertices and 15 arc(s) (11.51)

> *SetGraphAttribute(Fig3ORTree, “root” = “a”)*
> *Vertices(Fig3ORTree)*
[“a”, “b”, “c”, “d”, “e”, “f”, “g”, “h”, “i”, “j”, “k”, “l”, “m”, “n”,
“o”, “p”] (11.52)
> *SetChildOrder(Fig3ORTree, [0, 1, 2, 3, 1, 2, 1, 2, 3, 1, 2, 1, 2, 1, 2, 3])*
> *DrawORTree(Fig3ORTree)*



Subtrees

Before implementing the traversal algorithms, we need a procedure that determines a subtree of a tree. In particular, we want an algorithm that, given a tree and a vertex, will return the subtree with the given vertex as the root and that includes all of its descendants.

To produce the subtree, we will use the **InducedSubgraph** command on the list of vertices in the desired subtree. (Note that, unlike the **AddVertex** command, **InducedSubgraph** preserves the vertex attributes of the vertices in the subgraph.) The vertices that we want included in the subgraph are the given vertex together with all of its descendants. We begin by creating a procedure that finds all of the descendants of the given vertex. This procedure can apply to any rooted tree.

The approach is the same as we have used before. We begin with the given vertex and create a list consisting of its children, which we obtain with the **Departures** command. We then begin a loop over the list of descendants. At each step, we add all of the children of the current vertex to the list, and then move on to the next vertex in the list. This continues until we reach the end of the list and there are no more children to add. (Note: this is referred to as a level-order traversal.)

```

1 Descendants := proc(T::RTree, parent)
2     local Dlist, v, i;
3     uses GraphTheory;
4     Dlist := Departures(T, parent);
5     i := 1;
6     while i <= nops(Dlist) do
7         v := Dlist[i];
8         Dlist := [op(Dlist), op(Departures(T, v))];
9         i := i + 1;
10    end do;
11    return Dlist;
12 end proc;

```

Compute the descendants of e in the example tree above.

> Descendants (Fig3ORTree, "e")
["j", "k", "n", "o", "p"] (11.53)

To construct the subtree of an ordered rooted tree with a given vertex as its root, we need to: find the descendants of the given vertex; use the **InducedSubgraph** command on the given vertex and all its descendants; set the “root” attribute of the subgraph; and set the “order” attribute of the root to 0.

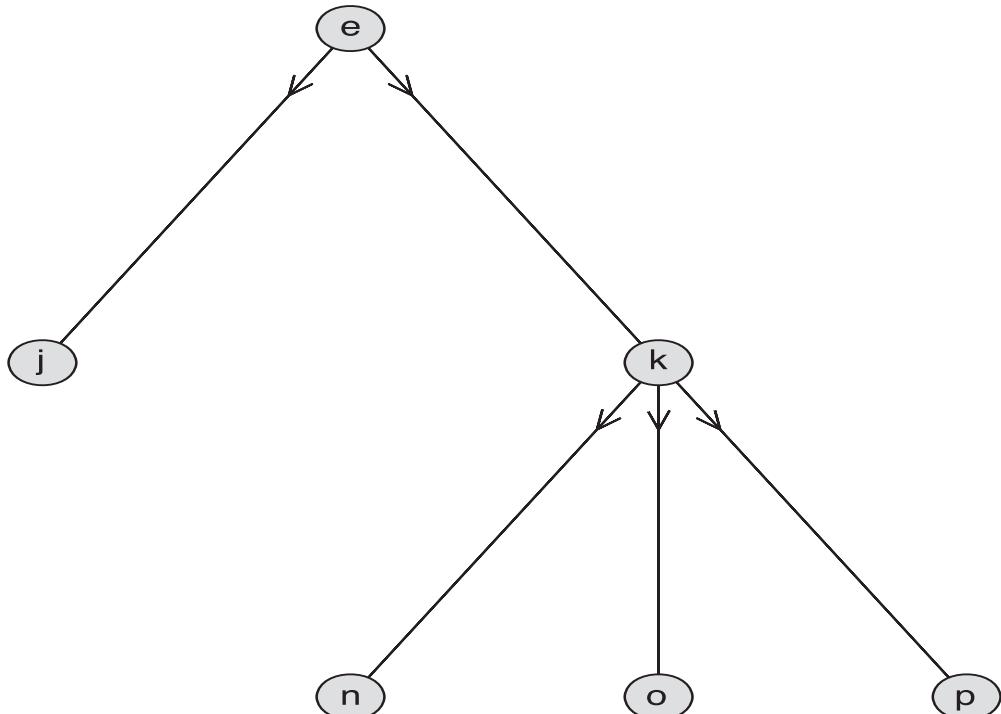
```
1 SubTree := proc (T::ORTree, newRoot)
2   local Vlist, subT;
3   uses GraphTheory;
4   Vlist := Descendants (T, newRoot);
5   Vlist := [newRoot, op(Vlist)];
6   subT := InducedSubgraph (T, Vlist);
7   SetGraphAttribute (subT, "root"=newRoot);
8   SetVertexAttribute (subT, newRoot, "order"=0);
9   return subT;
10 end proc;
```

Let us check this procedure by finding the subtree with root e and make sure it really is an **ORTree**.

> subEx := SubTree (Fig3ORTree, "e")
subEx := Graph 17: a directed unweighted graph with 6 vertices and 5 arc(s) (11.54)

> type (subEx, ORTree)
true (11.55)

> DrawORTree (subEx)



Traversal Algorithms

We now implement the three traversal algorithms described in Section 11.3 of the text. We begin with the preorder traversal algorithm, which is given as Algorithm 1 in the text.

Preorder

Given an ordered rooted tree, the preorder algorithm acts as follows. First, it adds the name of the root to the list of vertices that will be the procedure's output. Then, it calculates the children of the root and stores them in order. For each child, in order, the procedure recursively applies itself to the subtree with the given child as root and adds the subtree's output to the list of vertices.

```
1 Preorder := proc (T :: ORTree)
2   local root, Vlist, children, sorter, i, tempSubT;
3   uses GraphTheory;
4   root := GetGraphAttribute(T, "root");
5   Vlist := [root];
6   children := Departures(T, root);
7   sorter := VOrderComp(T);
8   children := sort(children, sorter);
9   for i from 1 to nops(children) do
10    tempSubT := SubTree(T, children[i]);
11    Vlist := [op(Vlist), Preorder(tempSubT)];
12  end do;
13  return Vlist;
14 end proc;
```

> *Preorder (Fig3ORTree)*

```
[“a”,[“b”,[“e”,[“j”],[“k”,[“n”],[“o”],[“p”]]],[“f”],[“c”],[“d”,
[“g”,[“l”],[“m”]],[“h”],[“i”]]] (11.56)
```

The nesting of the lists in this output is indicative of the subtree structure. You can confirm that this output is consistent with the preorder traversal demonstrated in Figure 4 of Section 11.3 of the textbook. Applying **Flatten** from the **ListTools** package will produce the list of vertices in the order they are visited.

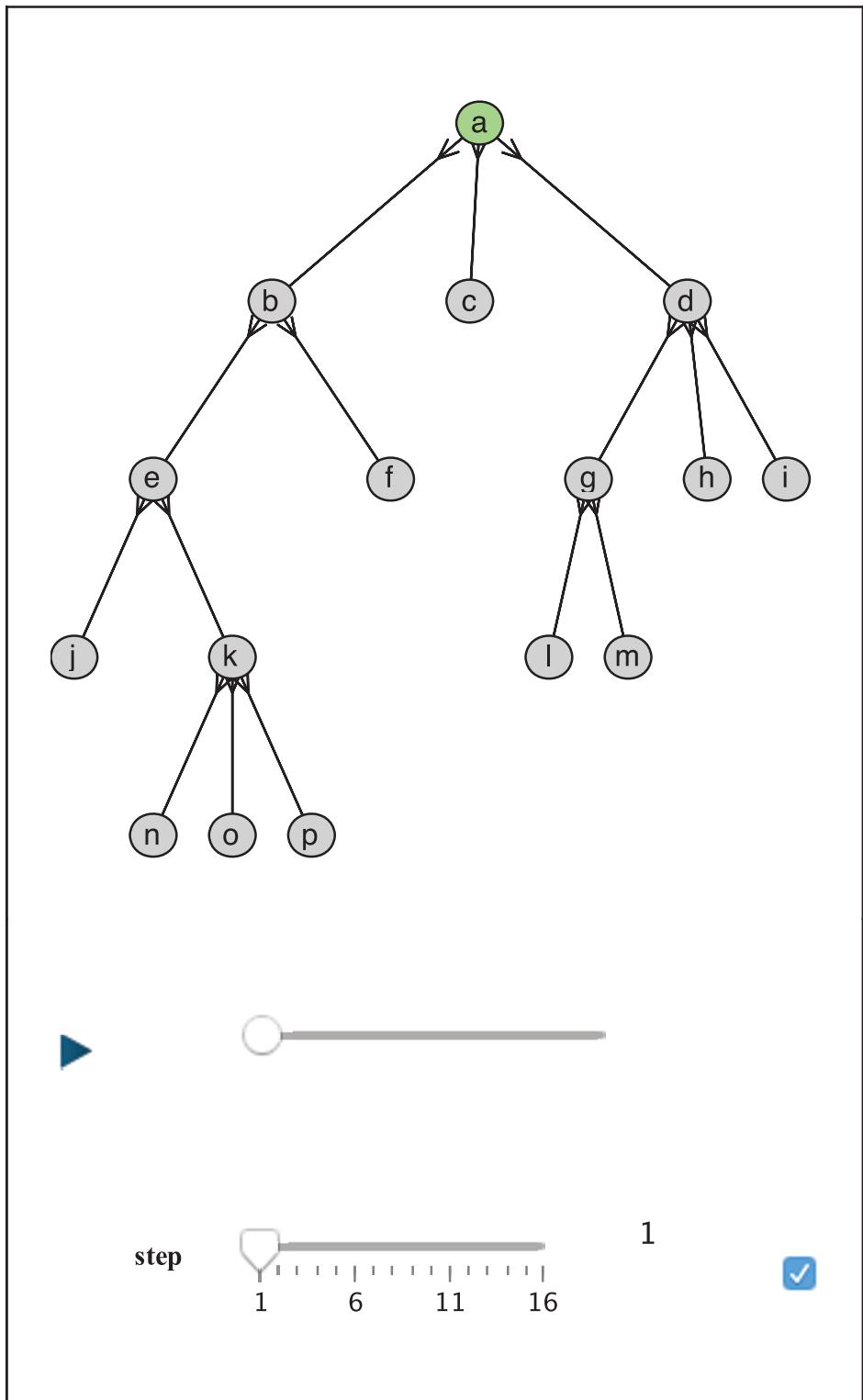
> *ListTools[Flatten] (11.56)*

```
[“a”,“b”,“e”,“j”,“k”,“n”,“o”,“p”,“f”,“c”,“d”,“g”,“l”,“m”,
“h”,“i”] (11.57)
```

Finally, with this output, we can create an animation that illustrates the traversal. To create the animation, we apply the **Explore** command to create a slider that controls how far along the traversal should be highlighted. We also use the **animate** option, which will allow you to click on the play button and watch the traversal progress automatically.

```
1 PreorderAnimation := proc (T :: ORTree)
2   local traversal, step;
3   uses GraphTheory, ListTools;
4   traversal := Flatten(Preorder(T));
5   Explore(DrawORTree(HighlightVertex(T, traversal[1..step],
inplace=false)), step=1..nops(traversal), animate=true);
6 end proc;
```

> PreorderAnimation (Fig3ORTree)



Postorder

Postorder traversal, described in Algorithm 3 of the text, is very similar to preorder traversal. The only change needed in the code is that, instead of adding the root to the output at the start of the algorithm, the root is processed after the loop is completed.

```

1 Postorder := proc(T::ORTree)
2   local root, Vlist, children, sorter, i, tempSubT;
3   uses GraphTheory;
4   root := GetGraphAttribute(T, "root");
5   Vlist := [];
6   children := Departures(T, root);
7   sorter := VOrderComp(T);
8   children := sort(children, sorter);
9   for i from 1 to nops(children) do
10    tempSubT := SubTree(T, children[i]);
11    Vlist := [op(Vlist), Postorder(tempSubT)];
12  end do;
13  Vlist := [op(Vlist), root];
14 end proc;

```

> *Postorder(Fig3ORTree)*

[[[["j"]], [{"n"}, {"o"}, {"p"}, {"k"}, {"e"}, {"f"}, {"b"}, {"c"}], [[[{"l"}], {"m"}], {"g"}],
[{"h"}, {"i"}, {"d"}, {"a"}]]

(11.58)

Inorder

In inorder traversal, the algorithm first applies itself recursively to the first child of the root, then it adds the root, and then it applies itself to the remainder of the children, in order.

```

1 Inorder := proc(T::ORTree)
2   local root, Vlist, children, sorter, i, tempSubT;
3   uses GraphTheory;
4   root := GetGraphAttribute(T, "root");
5   children := Departures(T, root);
6   sorter := VOrderComp(T);
7   children := sort(children, sorter);
8   if nops(children) <> 0 then
9     tempSubT := SubTree(T, children[1]);
10    Vlist := [Inorder(tempSubT), root];
11  else
12    Vlist := [root];
13  end if;
14  for i from 2 to nops(children) do
15    tempSubT := SubTree(T, children[i]);
16    Vlist := [op(Vlist), Inorder(tempSubT)];
17  end do;
18  return Vlist;
19 end proc;

```

> *Inorder(Fig3ORTree)*

[[[["j"]], {"e"}, [{"n"}, {"k"}, {"o"}, {"p"}]], {"b"}, [{"f"}], {"a"}, [{"c"}], [[[{"l"}], {"g"}],
[{"m"}], {"d"}, [{"h"}, {"i"}]]]

(11.59)

We leave it to the reader to create animations for inorder and postorder traversals.

Infix Notation

In the remainder of this section, we discuss how to use Maple to work with the infix, prefix, and postfix forms of arithmetic expressions, as described in Section 11.3 of the text. First, we will show how to create a tree representation of an infix expression. Then, we will explore how to evaluate expressions from their postfix and prefix forms.

Recall that infix notation is the usual notation for basic arithmetic and algebraic expressions. We will construct a Maple procedure that takes an infix expression and converts it into a tree representation. This tree representation can then be traversed using the traversals of the previous sections to form various arithmetic representation formats.

The algorithm we use to turn an arithmetic expression in infix notation into a tree is recursive. The basis case occurs when the expression consists of a single number or variable. In this case, the tree consists of a single vertex.

Otherwise, the expression consists of a left operand, an operator, and a right operand. In this case, we (1) apply the algorithm to the left and right operands, and (2) combine the resulting trees with the operator as the common root. Implementing this will require some preliminary work. In particular, we must address a few issues.

First, we need to represent arithmetic expressions in Maple in such a way that we can work with them and ensure that Maple will not evaluate them.

Second, we need to be able to distinguish the basis case from the recursive case.

Third, the leaves in the tree will be numbers and variables and the internal vertices will be operations. In an expression like $3 \cdot 4 + 7 \cdot (3 + x)$, we have repetition among the operators and the operands (two 3s, two additions, and two multiplications). We need a way to get Maple to consider each of these to be a distinct object, since Maple insists that the vertices in a graph be distinct.

Fourth, in the recursive step, we need to be able to identify the operator and separate the left and right operands. We must also deal with the special case of unary negation.

And fifth, we will need to implement a procedure to perform the combination of subtrees described in part (2) of the recursive step.

Representing Expressions

We cannot use the usual operations of arithmetic to represent an expression that will be turned into a tree, as Maple will automatically perform arithmetic operations. In order to prevent evaluation, we will use different operators, namely, **&+**, **&-**, **&***, **&/**, and **&^**.

Recall from Section 4.1 of this manual that the ampersand indicates to Maple that the symbol is a neutral operator. Before, we used neutral operators to define new infix operators for modular arithmetic. In this case, we only need to be able to write expressions in terms of infix operators and have the expression not evaluated. As long as we do not define **&+**, **&-**, **&***, **&/**, or **&^**, Maple will understand that they are operators but will not evaluate them.

Note that the typical order of operations, or precedence, is not respected by these neutral operators. Therefore, to enter expressions with these operators, we must fully parenthesize the expression. For example, to enter $2 \cdot 3 + 4$, type the following.

```
> (2 &* 3) &+ 4  
(2 &* 3) &+ 4
```

(11.60)

We will impose some additional restrictions on expressions, in order to avoid unnecessary complications. Specifically, we insist that the only symbols allowed are integers, variables, the binary neutral operators **&+**, **&-**, **&***, **&/**, **&^**, and parentheses **()**.

Distinguishing the Basis and Recursive Cases

Any arithmetic expression is either a single integer or variable, or it is two expressions joined by an arithmetic operator.

We can determine which kind it is by testing the object against the types **integer** and **symbol**. Remember that braces in a structured type mean that either type can be matched.

```
> type(5, {integer, symbol})  
true
```

(11.61)

```
> type(a, {integer, symbol})  
true
```

(11.62)

```
> type((2 &* 3) &+ 4, {integer, symbol})  
false
```

(11.63)

The above statements demonstrate that the **type** command can be used to distinguish between a single integer or variable and a complex expression.

Ensuring That Each Occurrence of an Object Is Considered Distinct

The tree associated to the expression $3 \cdot 4 + 7 \cdot (3 + x)$ will have 9 vertices. The internal vertices, the operations, consist of two additions and two multiplications. The leaves, the numbers and variables, consist of 4, 7, x , and two 3s.

In Maple graphs, each vertex must be unique and distinct from all other vertices. In order to make Maple consider two 3s or two **&+**'s to be different, we use the command **convert(R,'local')**, which has the effect of turning the expression **R** into a local name. This means that the object referred to by **R** is, as far as Maple is concerned, different from another copy of itself. For example, we can make 2 not equal to 2.

```
> evalb(convert(2, 'local') = 2)  
false
```

(11.64)

Note, however, that the object produced by **convert** appears to be an ordinary 2.

```
> convert(2, 'local')  
2
```

(11.65)

We incorporate this idea into a procedure analogous to **NewBTree** from Section 11.2.

```
1 NewExpressionTree := proc (R)
2   local T, r;
3   uses GraphTheory;
4   r := convert(R, 'local');
5   T := Digraph([r]);
6   SetGraphAttribute(T, "root"=r);
7   SetVertexAttribute(T, r, "order"=0);
8   return T;
9 end proc;
```

Identifying the Operator and the Operands

In the recursive case, we must separate a complex expression into its operator and the operands. Consider the following example.

```
> exampExpr := (3 &* 4) &+ (7 &* (3 &+ x))
exampExpr := (3 &* 4) &+ (7 &* 3 &+ x) (11.66)
```

This expression, $3 \cdot 4 + 7 \cdot (3 + x)$ consists of the sum of $3 \cdot 4$ and $7 \cdot (3 + x)$. In Maple, we use the **op** command to separate the expression into its parts.

Recall that **op** has several forms. Most frequently, we have used the form that accepts only one argument, typically a list or a set, and returns the sequence underlying the argument. Observe what happens if we apply **op** to our expression.

```
> op(exampExpr)
3 &* 4, 7 &* 3 &+ x (11.67)
```

It has removed the central addition and produced the sequence consisting of the two operands.

A different form of **op** will allow us to access the two operands directly. Given a positive integer as a first argument, **op** returns the specified operand.

```
> op(1, exampExpr)
3 &* 4 (11.68)
```

```
> op(2, exampExpr)
7 &* 4 &+ x (11.69)
```

Giving 0 as the first argument to **op** has slightly different meanings in different contexts. In this case, giving 0 as the first argument will return the operator.

```
> op(0, exampExpr)
&+ (11.70)
```

Combining Subtrees

Recall that, as part of Huffman coding in Section 2, we wrote a procedure, **joinHTrees**, for joining two existing trees at a new root. We create a similar procedure here, without weights and with flexibility in the number of trees being joined. Rather than taking a new root and two trees, this version will accept a new root and a list of trees. Each tree in the list will be a child of the new root, with the order among them determined by their order within the list.

```
1 JoinTrees := proc (newR,treeL::list(ORTree) )
2   local newT, newVerts, oldRoots, newEdges, t, v, p, i;
3   uses GraphTheory;
4   newVerts := [newR, seq(op(Vertices(t)), t in treeL)];
5   oldRoots := [seq(GetGraphAttribute(t, "root"), t in treeL)];
6   newEdges := {seq([newR, v], v in oldRoots), seq(op(Edges(t)), t in
7     treeL)};
8   newT := Graph(newVerts, newEdges);
9   for t in treeL do
10    for v in Vertices(t) do
11      p := GetVertexAttribute(t, v, "order");
12      SetVertexAttribute(newT, v, "order"=p);
13    end do;
14  end do;
15  for i from 1 to nops(oldRoots) do
16    SetVertexAttribute(newT, oldRoots[i], "order"=i)
17  end do;
18  SetVertexAttribute(newT, newR, "order"=0);
19  SetGraphAttribute(newT, "root"=newR);
20  return newT;
end proc:
```

The Procedure

With the **JoinTrees** procedure above and the **NewExpressionTree**, we are ready to write the procedure for turning infix expressions into binary trees.

The procedure accepts a single argument, **e**, the expression. We first test the type of **e**. If it is an integer or a symbol, then we use **NewExpressionTree** to create a new binary tree with **e** as the only vertex.

Otherwise, we are in the recursive case. We use **op** to determine the operands and the operator. Again, we apply **convert** to make the operator local, as it will be a vertex in the tree. After recursive calls to the procedure to create the trees for the operands, the subtrees are joined at the operator into the result tree.

Here is the procedure.

```
1 InfixToTree := proc (e)
2   local o, sube, operandTrees, result;
3   uses GraphTheory;
4   if type(e, {integer, symbol}) then
```

```

5      result := NewExpressionTree(e);
6      else
7          o := op(0, e);
8          operandTrees := [seq(InfixToTree(sube), sube in op(e))];
9          result := JoinTrees(convert(o, 'local'), operandTrees);
10     end if;
11     return result;
12 end proc;

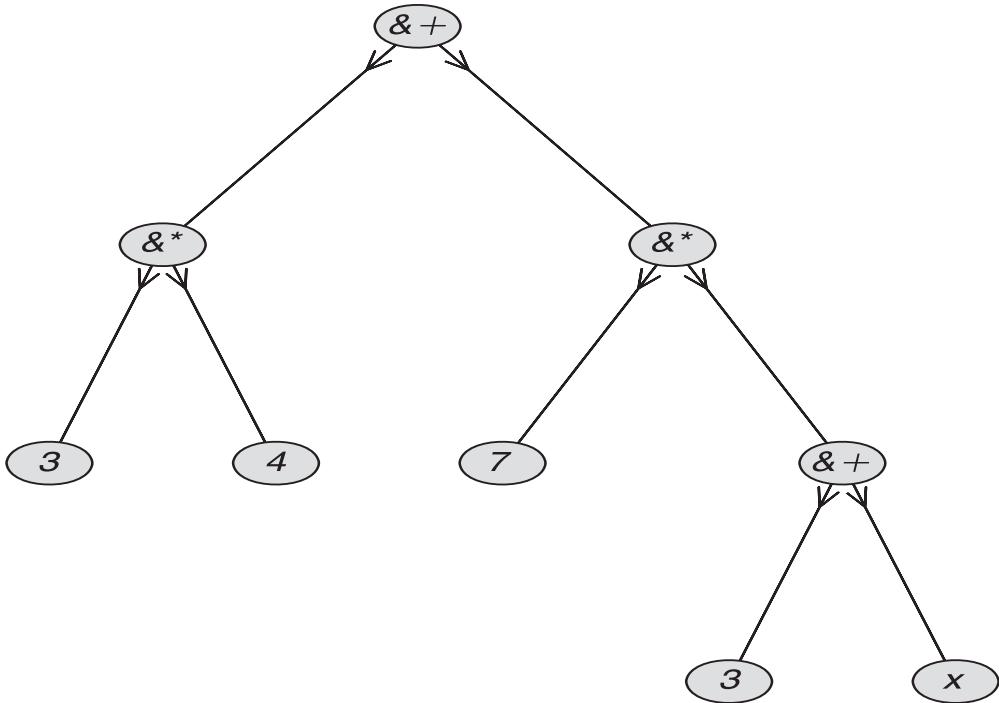
```

We test the procedure on the example expression.

> *exampExpr*
 $(3 \&* 4) \&+ (7 \&* 3 \&+ x)$ (11.71)

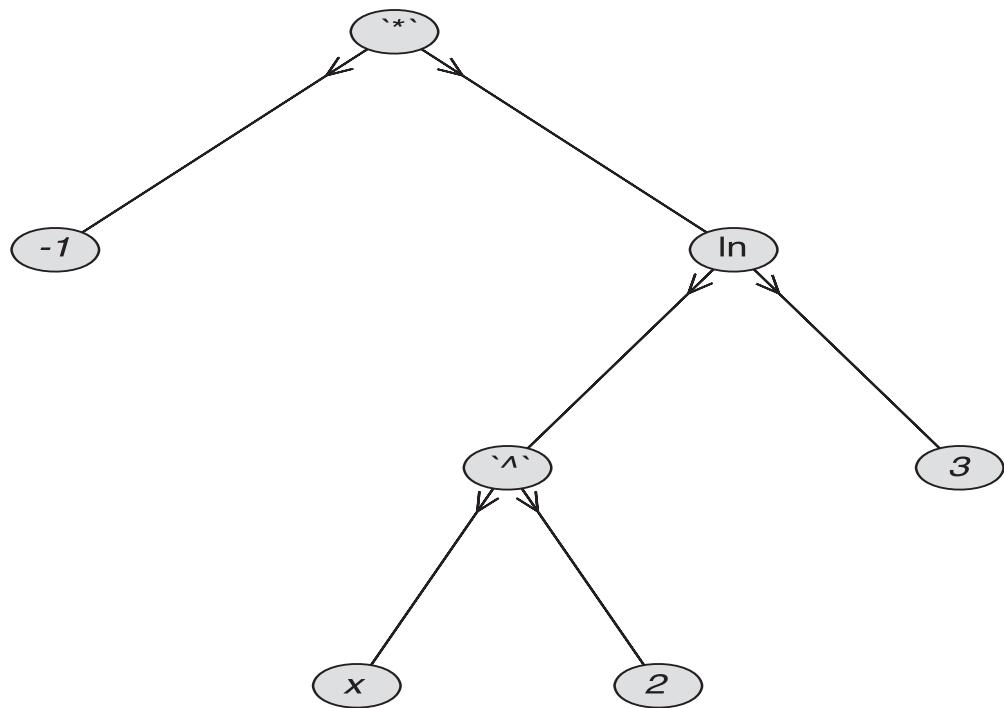
> *exampTree* := *InfixToTree* (*exampExpr*)
exampTree := Graph 18: a directed unweighted graph with 9 vertices and 8 arc(s) (11.72)

> *DrawORTree* (*exampTree*)

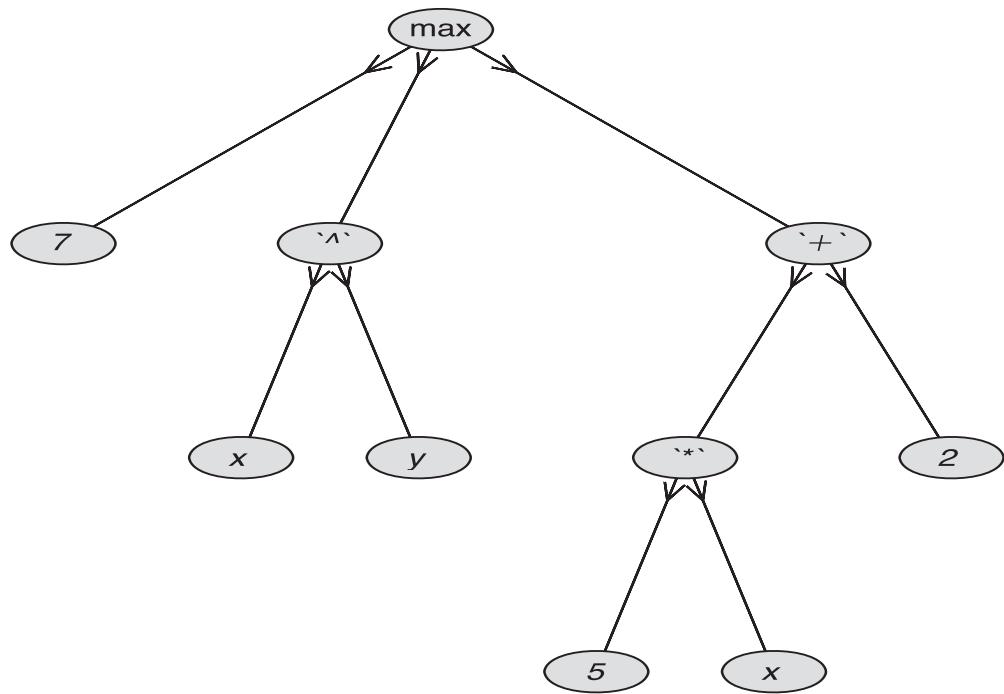


Note that this procedure is rather flexible. In fact, it can be applied to expressions involving the usual arithmetic operators, provided that you keep in mind that Maple will simplify them before constructing the tree. It also is not restricted to operators, but can be applied to expressions involving functions.

> *DrawORTree* (*InfixToTree* ($-\ln(x^2 + 3)$))



> *DrawORTree (InfixToTree (max (5 x + 2, x^y, 7)))*



Prefix and Postfix Notation

Suppose we are given a tree representation of an arithmetic expression. We can express these trees in postfix, infix, or prefix form by applying the respective traversal algorithm we designed above. For example, applying **Inorder** to the tree we created in the last example yields the correct sequence of symbols, with the list nesting representing the tree structure.

> *Inorder(exampTree)*

[[[3], &*, [4]], &+, [[7], &*, [[3], &+, [x]]]]

(11.73)

It is left to the reader to make the needed modifications to produce procedures that return accurate infix, prefix, and postfix expressions.

As a final example in this section, we demonstrate how to evaluate a given postfix expression. We will represent the postfix expression as a list of symbols, each of which is either a number or one of the arithmetic operations' symbols as a string.

Since we are considering postfix expressions, we read the list of symbols from left to right. Each time we encounter an operation, that operation is applied to the previous two numbers and we update the list by replacing the two numbers and the operation symbol by the result of the operation.

```
1 EvalPostfix := proc (Expr :: list)
2   local i, L;
3   L := Expr;
4   while nops (L) > 1 do
5     i := 1;
6     while not L[i] in {"+", "-", "*", "/", "^"} do
7       i := i + 1;
8     end do;
9     if L[i] = "+" then
10      L[i] := L[i-2] + L[i-1];
11    elif L[i] = "-" then
12      L[i] := L[i-2] - L[i-1];
13    elif L[i] = "*" then
14      L[i] := L[i-2] * L[i-1];
15    elif L[i] = "/" then
16      L[i] := L[i-2] / L[i-1];
17    elif L[i] = "^" then
18      L[i] := L[i-2] ^ L[i-1];
19    end if;
20    L := subsop (i-1=NULL, i-2=NULL, L);
21  end do;
22  return L[1];
23 end proc;
```

> *PostExample* := [7, 2, 3, “*”, “-”, 4, “{”, 9, 3, “/”, “+”]

PostExample := [7, 2, 3, “*”, “-”, 4, “{”, 9, 3, “/”, “+”]

(11.74)

> *EvalPostfix(PostExample)*

4

(11.75)

The reader is left to explore evaluation in the prefix case, which requires only a simple modification.

11.4 Spanning Trees

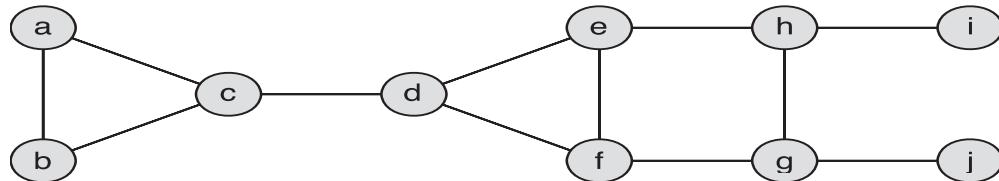
This section explains how to use Maple to construct spanning trees for graphs and how to use spanning trees to solve many different types of problems. Spanning trees have a myriad of applications, including coloring graphs, placing n queens on a $n \times n$ chessboard so that no two of the queens attack each other, and finding a subset of a set of numbers with a specified sum. All of these problems, which are described in detail in the text, will be explored computationally in this section. First, we will show how to use Maple to form spanning trees using two algorithms: depth-first search and breadth-first search. Then, we will show how to use Maple to solve the problems just mentioned.

Built-In Command

Maple includes a command, **SpanningTree**, for finding a spanning tree of an undirected graph. To illustrate, we reproduce the graph from Exercise 13 of Section 11.4.

```
> Exercise13 := Graph({{"a","b"}, {"a","c"}, {"b","c"}, {"c","d"}, {"d","e"}, {"d","f"}, {"e","f"}, {"e","h"}, {"f","g"}, {"g","h"}, {"g","j"}, {"h","i"}, {"i,"j"}})  
Exercise13 := Graph 19: an undirected unweighted graph with 10 vertices and 12 edge(s) (11.76)
```

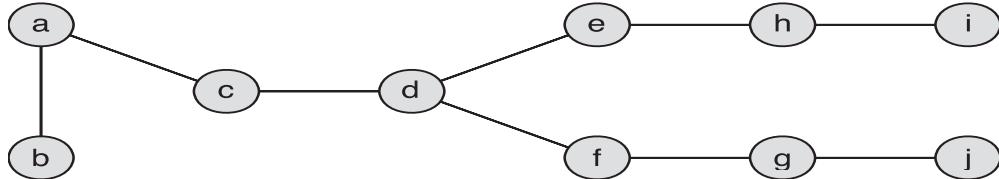
```
> SetVertexPositions(Exercise13, [[0, 1], [0, 0], [1, 0.5], [2, 0.5], [3, 1], [3, 0], [4, 0], [4, 1], [5, 1], [5, 0]])  
> DrawGraph(Exercise13)
```



Now, we use the **SpanningTree** command to find a spanning tree for this graph. The only required argument is the name of the graph whose spanning tree we wish to compute.

```
> Span1Exercise13 := SpanningTree(Exercise13)  
Span1Exercise13 := Graph 20: an undirected unweighted graph with 10 vertices and 9 edge(s) (11.77)
```

```
> DrawGraph(Span1Exercise13)
```



We can also specify one of the vertices of the graph as a second argument in order to specify the root of the spanning tree.

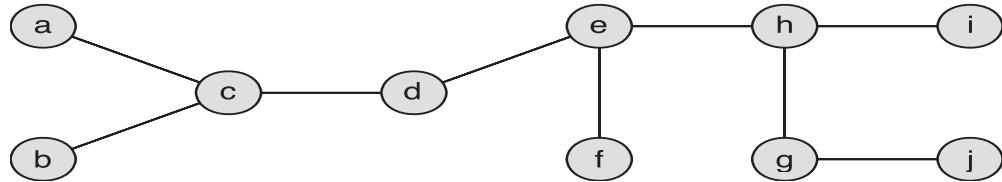
```
> Span2Exercise13 := SpanningTree(Exercise13, "i")
```

Span2Exercise13 := Graph 21: an undirected unweighted graph with 10 vertices

and 9 edge(s)

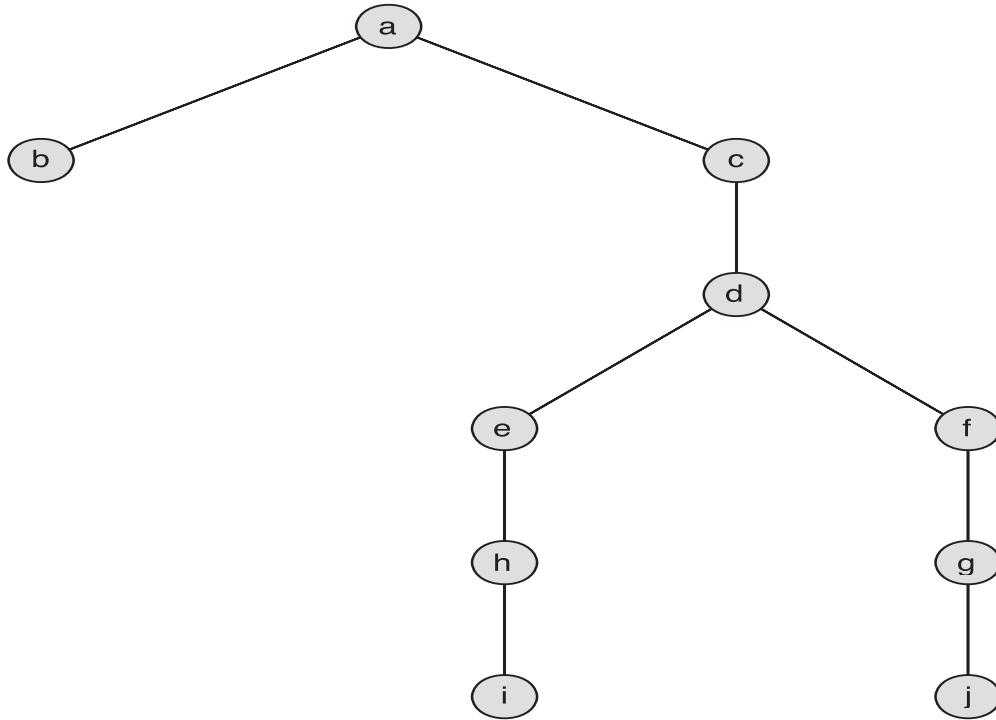
(11.78)

```
> DrawGraph(Span2Exercise13)
```



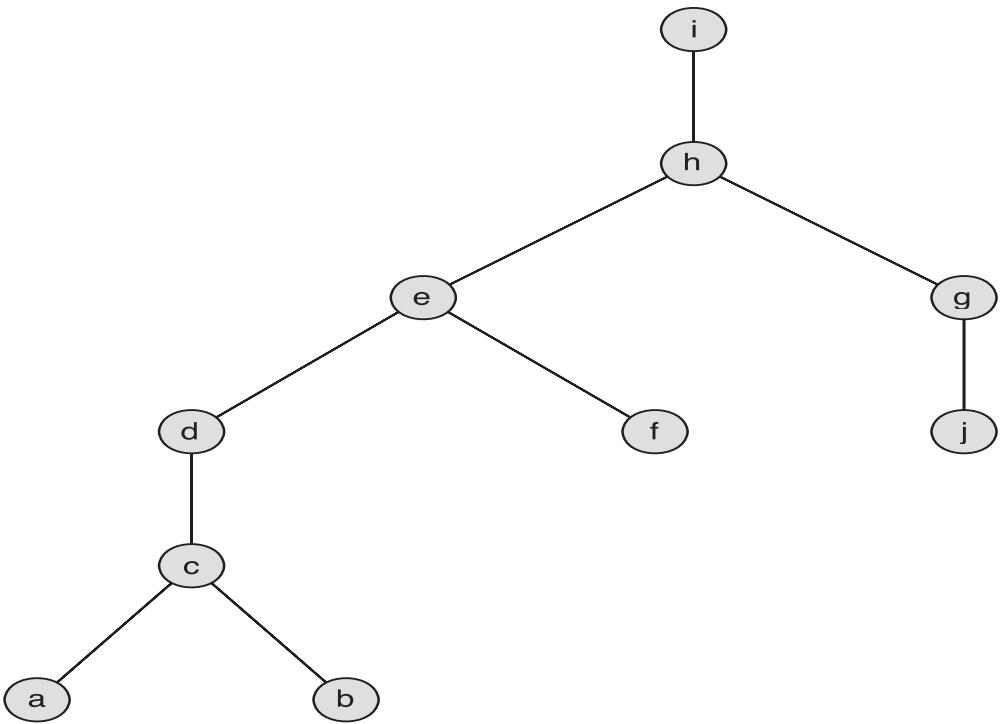
Observe that the specified vertex positions in the spanning tree are copied from the positions we set in the original graph. If you prefer a more tree-like image, you can apply the **tree style** option.

```
> DrawGraph(Span1Exercise13, style = tree)
```



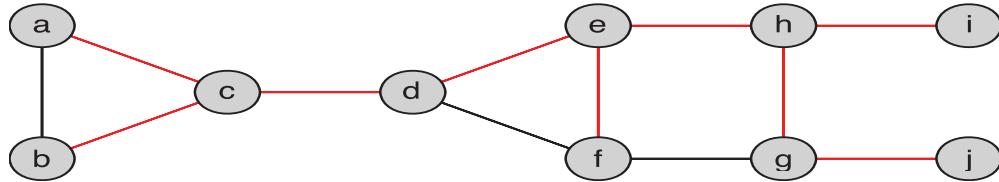
If you have a particular root in mind, you can specify it with the **root** option.

```
> DrawGraph(Span2Exercise13, style = tree, root = "i")
```



Alternatively, you might wish to view the spanning tree in the context of the original graph by highlighting the included edges. The easiest way to do this is using the **HighlightSubgraph** command, which takes a graph as its second argument. This command highlights both the edges and vertices of the subgraph and thus accepts two colors as optional arguments. The first color is applied to the edges and the second to the vertices.

```
> DrawGraph(HighlightSubgraph(Exercise13, Span2Exercise13, "Red",
                             "LightGray", inplace = false))
```



Despite Maple's existing **SpanningTree** command, we will develop two of our own using depth-first and breadth-first search algorithms as a way to illustrate these important algorithms.

Depth-First Search

We begin by implementing depth-first search. As the name of the algorithm suggests, vertices are visited in order of increasing depth of the spanning tree. Our implementation is based on Algorithm 1 of Section 11.4 of the textbook.

Recall the terminology defined in the textbook. We say that we are “exploring a vertex” v from the time the vertex is first added to the spanning tree until we have backtracked back to v for the last time. Note that at any step in the process, we are generally exploring multiple vertices. In particular,

the root of the spanning tree starts being explored at the very beginning of the process and continues being explored until the procedure terminates.

The procedure, which we call **DepthSearch**, will take two arguments: an undirected graph and a vertex in that graph. The procedure operates as follows:

1. First, we check that the graph is connected using Maple's **IsConnected** procedure. If not, there can be no spanning tree and the procedure returns **FAIL**.
2. Next, we initialize the following variables.
 - a) **ToVisit** will be the set of vertices of the graph that have not yet been visited. It is initialized to the set of vertices of the graph.
 - b) **Exploring** will be the list of vertices that are currently being explored. As vertices are visited, they are added to the end of the **Exploring** list. We remove a vertex from the **Exploring** list once it has been fully explored, that is, when it has no neighbors not already in the tree. **Exploring** is initialized to the vertex that is given as the second argument.
 - c) **T** will be the spanning tree that is constructed. It is initialized to the directed graph consisting of all the vertices of the graph, but with no edges. Provided that the graph is connected, we know that all the vertices will appear in **T** and this saves us from adding them one at a time. Note that **T** will be rooted and ordered. The ordering is accomplished by checking the number of children of a vertex when adding an edge from it and indicates the order in which edges are added to the tree.
3. Following initialization, we begin a while loop which terminates when the **Exploring** list is empty. The variable **v** is set to the last element of the **Exploring** list. We then compute the intersection, **N**, of the set of neighbors of **v** and the **ToVisit** set of vertices not already contained in the tree. Either,
 - a) **N** is nonempty, in which case, one of its elements is chosen as **w**, the next vertex to visit. The edge **{v,w}** is added to the tree **T**. In addition, **w** is removed from the **ToVisit** set and added to the end of the **Exploring** list. In the next iteration of the while loop, this new vertex will be set to be **v**.
 - b) **N** is empty, in which case the vertex **v** has been explored completely and so it can be removed from the **Exploring** list. The next iteration of the while loop will set **v** to be the vertex one step back in the **Exploring** list. This is the “backtracking” step.

Here, now, is the procedure.

```

1 DepthSearch := proc(G :: Graph, startV)
2   local ToVisit, Exploring, T, v, N, w;
3   uses GraphTheory;
4   if not IsConnected(G) then
5     return FAIL;
6   end if;
7   ToVisit := {op(Vertices(G))} minus {startV};
8   Exploring := [startV];
9   T := Digraph(Vertices(G));
10  SetGraphAttribute(T, "root"=startV);
11  SetVertexAttribute(T, startV, "order"=0);
12  while Exploring <> [] do
13    v := Exploring[-1];
14    N := {op(Neighbors(G, v))} intersect ToVisit;
15    if N <> {} then
16      w := N[1];
17      AddArc(T, [v, w]);

```

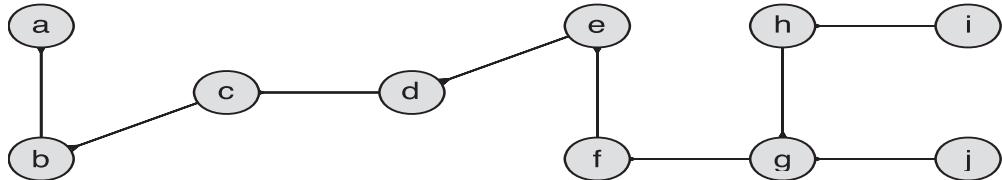
```

18     SetVertexAttribute(T,w,"order"=nops(Departures(T,v)));
19     ToVisit := ToVisit minus {w};
20     Exploring := [op(Exploring),w];
21 else
22     Exploring := subsop(-1=NULL,Exploring);
23 end if;
24 end do;
25 SetVertexPositions(T,GetVertexPositions(G));
26 return T;
27 end proc:
```

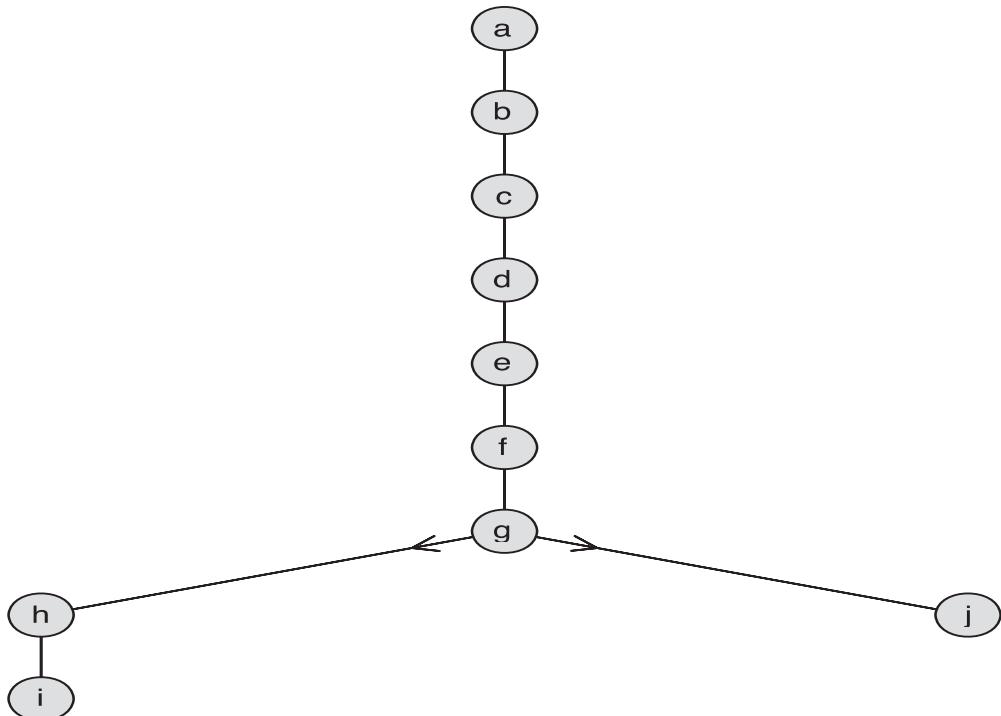
Let us test this with our Exercise 13 example from above.

> *DepthExercise13 := DepthSearch(Exercise13, "a")*
DepthExercise13 := Graph 22: a directed unweighted graph with 10 vertices and 9 arc(s)
(11.79)

> *DrawGraph(DepthExercise13)*



> *DrawORTree(DepthExercise13)*



So as to be able to animate the process of performing the depth-first search, we create a version of this procedure which, rather than adding the edges to a tree, returns a list of the edges in the order in which they are encountered. We edit the procedure replacing the tree **T** with a list of edges and removing the other commands having to do with building a graph object. In addition, since the original graph will be undirected, we add the edges as sets rather than lists so that they correspond to edges in the original graph.

```

1 DepthSearchList := proc (G::Graph, startV)
2   local ToVisit, Exploring, T, v, N, w;
3   uses GraphTheory;
4   if not IsConnected(G) then
5     return FAIL;
6   end if;
7   ToVisit := {op(Vertices(G))} minus {startV};
8   Exploring := [startV];
9   T := [];
10  while Exploring <> [] do
11    v := Exploring[-1];
12    N := {op(Neighbors(G, v))} intersect ToVisit;
13    if N <> {} then
14      w := N[1];
15      T := [op(T), {v, w}];
16      ToVisit := ToVisit minus {w};
17      Exploring := [op(Exploring), w];
18    else
19      Exploring := subsop(-1=NULL, Exploring);
20    end if;
21  end do;
22  return T;
23 end proc;
```

Applied to the exercise, this now produces a list of edges.

> *DepthSearchList(Exercise13, "a")*
 $\{ \{ "a", "b" \}, \{ "b", "c" \}, \{ "c", "d" \}, \{ "d", "e" \}, \{ "e", "f" \}, \{ "f", "g" \}, \{ "g", "h" \}, \{ "h", "i" \}, \{ "g", "j" \} \}$ (11.80)

We can now create a procedure, modeled on the **PreorderAnimation** from Section 11.3 of this manual, that produces an animation of the depth-first search progression. Note that we make a copy of the graph in order to permanently mark the starting vertex, and we use the **ifelse** functional form of an if statement to display the graph with no edges highlighted. The first argument of **ifelse** is a condition to be tested, the second argument is evaluated if the condition is true, and the third argument is evaluated if the condition is false. Also note that, when provided a list of edges to highlight, **HighlightEdges** requires a list of colors of the same length.

```

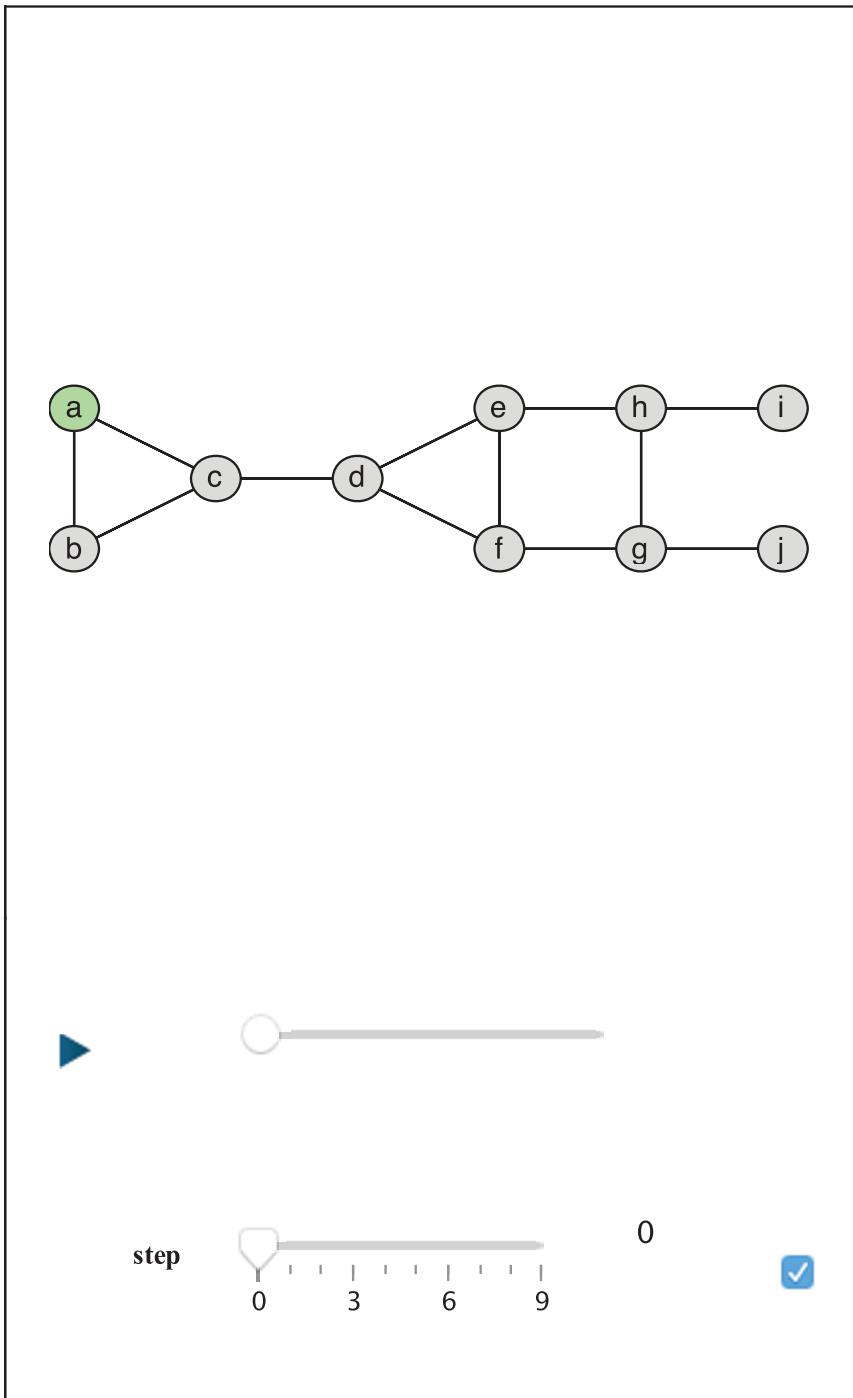
1 DepthSearchAnimation := proc (G::Graph, startV)
2   local H, search, step;
3   uses GraphTheory, ListTools;
4   H := CopyGraph(G);
5   HighlightVertex(H, startV);
```

```

6   search := DepthSearchList (H, startV) ;
7   Explore (
8     ifelse (step=0,
9        DrawGraph (H),
10       DrawGraph (HighlightEdges (H, search [1..step],
11         ["Red" $ step], inplace=false)) ),
12       step=0..nops (search), animate=true);
end proc:

```

> *DepthSearchAnimation(Exercise13, "a")*



Breadth-First Search

We now turn to an implementation of breadth-first search. Recall that the breadth-first algorithm works by examining all vertices at the current depth of the spanning tree before moving on to the next level of the graph. Our implementation will follow Algorithm 2 of Section 11.4 of the text.

The procedure, to be called **BreadthSearch**, again takes two arguments: an undirected graph and a vertex to act as the starting point. It proceeds as follows:

1. First, we check that the graph is connected using Maple's **IsConnected** procedure.
2. Next, we initialize the following variables.

- a) **ToVisit**, as before, will be the set of vertices of the graph not yet visited. It is initialized to the set of vertices of the graph with the initial vertex excluded.
- b) **ToProcess** will be the list of vertices that have been determined to be incident to a vertex in the tree but which have not yet been processed. **ToProcess** is initialized to the vertex that is given as the second argument to the procedure.
- c) **T** will be the spanning tree that is constructed. Once again, it is initialized to the tree consisting of all the vertices of the given graph, but with no edges.
- d) Following initialization, we begin a while loop that terminates when the **ToProcess** list is empty. The variable **v** is set to the first element of the **ToProcess** list. We then compute the intersection, **N**, of the set of neighbors of **v** and the **ToVisit** set. For each element **w** of **N**, an edge $\{v,w\}$ is added to **T** and **w** is added to the end of the **ToProcess** list and removed from the **ToVisit** set. Then, **v** is removed from **ToProcess**.

Observe that, since neighbors are added to the end of the **ToProcess** list and are processed from the beginning of the list, we are assured that all vertices on a given level will be processed before any vertex at a lower level.

Here is the implementation.

```
1 BreadthSearch := proc (G::Graph, startV)
2   local ToVisit, ToProcess, T, v, N, w;
3   uses GraphTheory;
4   if not IsConnected(G) then
5     return FAIL;
6   end if ;
7   ToVisit := {op(Vertices(G))} minus {startV};
8   ToProcess := [startV];
9   T := Digraph(Vertices(G));
10  SetGraphAttribute(T, "root"=startV);
11  SetVertexAttribute(T, startV, "order"=0);
12  while ToProcess <> [] do
13    v := ToProcess[1];
14    N := {op(Neighbors(G, v))} intersect ToVisit;
15    for w in N do
16      AddArc(T, [v, w]);
17      SetVertexAttribute(T, w, "order"=nops(Departures(T, v)));
18      ToProcess := [op(ToProcess), w];
19      ToVisit := ToVisit minus {w};
20    end do;
21    ToProcess := subsop(1=NULL, ToProcess);
22  end do;
```

```

23     SetVertexPositions(T, GetVertexPositions(G));
24     return T;
25 end proc;

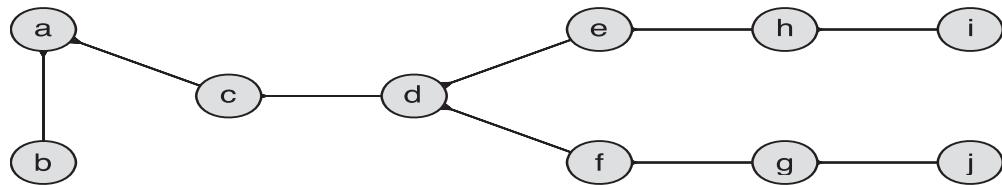
```

Once again, we illustrate using Exercise 13.

> *BreadthExercise13 := BreadthSearch(Exercise13, "a")*

BreadthExercise13 := Graph 23: a directed unweighted graph with 10 vertices and 9 arc(s)
(11.81)

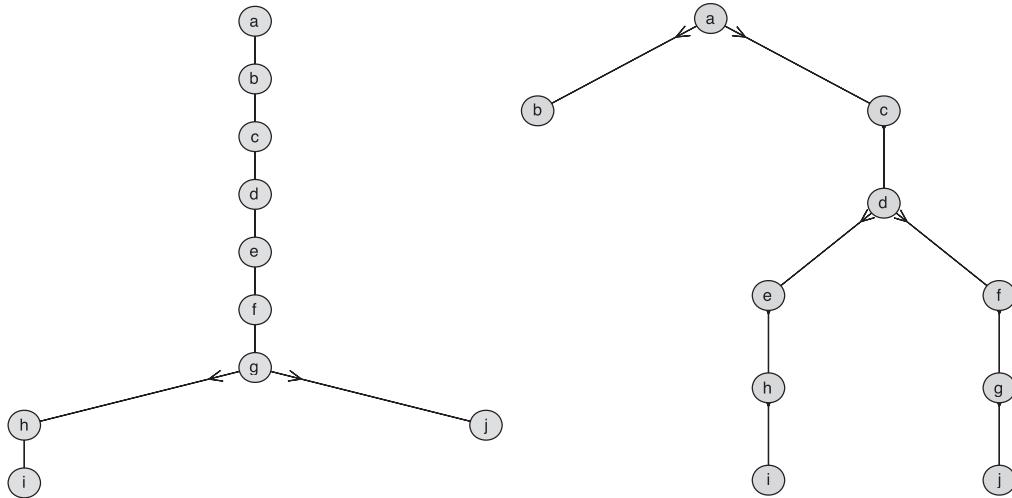
> *DrawGraph(BreadthExercise13)*



Observe that this is the same graph as was produced by Maple's **SpanningTree** command, which suggests that Maple's command uses a breadth-first algorithm.

Creating the procedures for animating depth-first search is left to the reader. However, before moving on to backtracking, take a moment to compare the trees produced by the two algorithms.

> *DrawORTree(DepthExercise13), DrawORTree(BreadthExercise13)*



Notice that the two spanning trees are quite different, even though they are both drawn rooted at vertex *a*. In particular, the depth-first search has a deep and thin structure, whereas the breadth-first search is shorter and wider appearing.

Graph Coloring via Backtracking

Backtracking is a method that can be used to find solutions to problems that might be impractical to solve using exhaustive search techniques. Backtracking is based on the systematic search for a solution to a problem using a decision tree. (See the text for a complete discussion.) Here we

show how to use backtracking to solve several different problems, including coloring a graph, the n -queens problem, and the subset sum problem.

The first problem we will attack via a backtracking procedure is the problem of coloring a graph using n colors, where n is a positive integer. Given a graph, we will attempt to color it using n colors using the method described in Example 6 of Section 11.4.

1. Fix an order on the vertices of the graph, say v_1, v_2, \dots, v_m and fix an ordering of the colors as color 1, color 2, ..., color n . We will use the ordering of the vertices that Maple automatically imposes. For the colors, we will require an ordered list of colors as one of the arguments to the procedure.
2. We store the current state of the coloring in a list we will call **coloring**. The i th entry in this list will correspond to the color of i th vertex. For example, **coloring = [1,2,1]** corresponds to vertex v_1 assigned color 1, vertex v_2 assigned color 2, and vertex v_3 assigned color 1. This **coloring** list is similar to the **Exploring** list from **DepthSearch**. In both cases, you can think of the list as storing the path from the root of the tree to the current vertex. In this case, the level, which corresponds to the position in the list, carries additional information. Specifically, level k in the decision tree (that is, position k in the **coloring** list) corresponds to deciding the color of vertex v_k .
3. We initialize **coloring** to **[1]** and set a counter variable **i** to 2. The variable **i** will indicate the vertex that requires a decision.
4. Set **N** equal to the neighbors of the **i**th vertex of the graph, and then construct a set, **used**, consisting of the indices of the colors assigned to the neighbors. The **i**th vertex will be assigned the color with the smallest index not in **used**, assuming there are any remaining colors.
5. If there are no possible colors for the **i**th vertex, then we must backtrack. We decrease **i** by one. To ensure that we do not repeat a choice already made when we revisit a vertex, we make the following modification to how colors are chosen. If the **i**th position of **coloring** has already been set, then we know we are in the process of backtracking. We insist that the new choice for the color of vertex **i** is the smallest possible color greater than the current color.
6. The procedure terminates in one of two cases. If **i** is set to a value greater than the number of vertices, then we know that **coloring** contains a valid assignment for all vertices. On the other hand, if **i** is ever set to 1, then we know that we have backtracked all the way to the root. Since the color of the first vertex does not affect the validity of the coloring, this indicates that we have exhausted all possible colorings and that the graph cannot be colored with n colors.

Our procedure will be called **BackColor**. It will accept two arguments: the graph to be colored and a list of colors. If it is successful, it will display the graph with the vertices colored using **HighlightVertex**. If it determines that there is no n -coloring of the graph, it will return **FAIL**.

```

1 BackColor := proc (G::Graph, C::list)
2   local Verts, numverts, allcolorsL, k, coloring, i, N, j, used,
3       available, colorL;
4   uses GraphTheory;
5   Verts := Vertices(G);
6   numverts := nops(Verts);
7   allcolorsL := {seq(k, k=1..nops(C))};
8   coloring := [1];
9   i := 2;
10  while i > 1 and i <= numverts do
11    N := Neighbors(G, Verts[i]);
12    used := {};
13    for j from 1 to i-1 do
14      if Verts[j] in N then

```

```

14      used := used union {coloring[j]};
15      end if;
16  end do;
17  if nops(coloring) >= i then
18      used := used union {seq(k, k=1..coloring[i])};
19  end if;
20  available := allcolorsL minus used;
21  if available <> {} then
22      coloring := [op(coloring[1..i-1]), available[1]];
23      i := i + 1;
24  else
25      if nops(coloring) >= i then
26          coloring := coloring[1..(i-1)];
27      end if;
28      i := i - 1;
29  end if;
30 end do;
31 if i > numverts then
32     print(coloring);
33     colorL := [];
34     for k from 1 to numverts do
35         colorL := [op(colorL), C[coloring[k]]];
36     end do;
37     HighlightVertex(G, Verts, colorL);
38     DrawGraph(G);
39 else
40     return FAIL;
41 end if;
42 end proc;

```

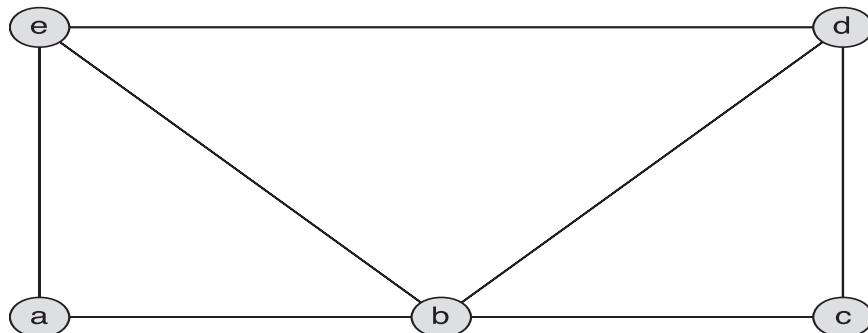
We test our procedure on the example given in Figure 11 of Section 11.4 of the text.

```

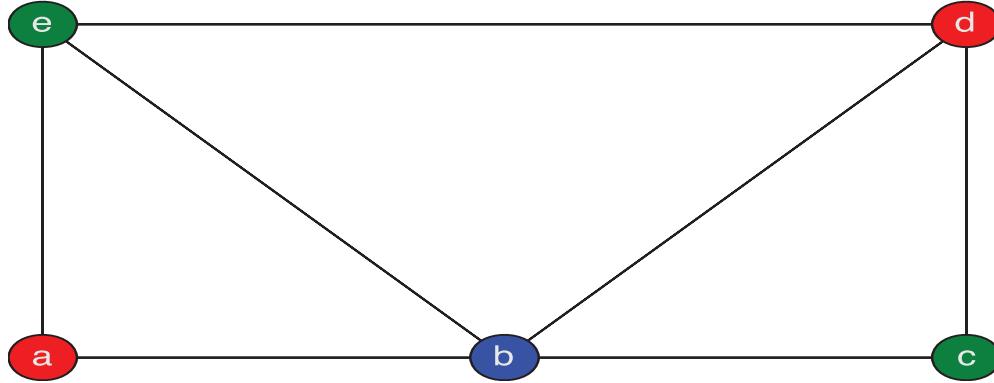
> Fig11Graph := Graph({{"a","b"}, {"a","e"}, {"b","c"}, {"b","d"}, {"b","e"}, {"c","d"}, {"d","e"}})
Fig11Graph := Graph 24: an undirected unweighted graph with 5 vertices and 7 edge(s)
(11.82)

> SetVertexPositions(Fig11Graph, [[0,0], [1,0], [2,0], [2,1], [0,1]])
> DrawGraph(Fig11Graph)

```



```
> BackColor(Fig11Graph, ["Red", "Blue", "Green"])
[1, 2, 3, 1, 3]
```



On the other hand, the complete graph on 5 vertices cannot be 3-colored or 4-colored. This time, rather than specifying colors by name, we'll use a sublist of a palette.

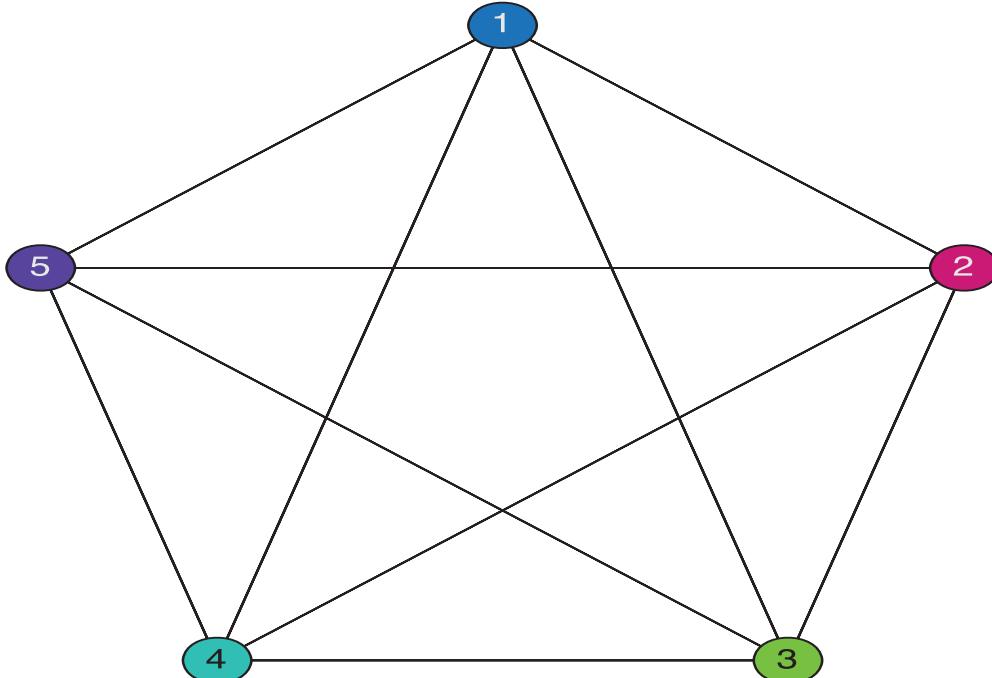
```
> K5 := CompleteGraph(5)
K5 := Graph 25: an undirected unweighted graph with 5 vertices and 10 edge(s) (11.83)
```

```
> springcolors := ColorTools[GetPalette]("Spring"):
```

```
> BackColor(K5, springcolors[1..3])
FAIL (11.84)
```

```
> BackColor(K5, springcolors[1..4])
FAIL (11.85)
```

```
> BackColor(K5, springcolors[1..5])
[1, 2, 3, 4, 5]
```



Before moving on to the n -queens problem, we illustrate how we can modify our backtracking procedure to record and display the decision tree. Instead of displaying the graph, our modified algorithm will produce an animation showing how the decision tree is built up. We do this by keeping a list of trees. Each time a color is assigned, we create a new tree in the list by adding the current state of the **coloring** list (converted to a string) as a vertex.

```

1 BackColorDT := proc(G :: Graph, C :: list)
2   local Verts, numverts, allcolorsL, k, coloring, i, N, j, used,
3       available, DTList, parentV, newV, newT, Vpos, plotList;
4   uses GraphTheory;
5   Verts := Vertices(G);
6   numverts := nops(Verts);
7   allcolorsL := {$1..nops(C)};
8   coloring := [1];
9   newV := convert(coloring, 'string');
10  DTList := [Graph([newV])];
11  i := 2;
12  while i > 1 and i <= numverts do
13    N := Neighbors(G, Verts[i]);
14    used := {};
15    for j from 1 to i-1 do
16      if Verts[j] in N then
17        used := used union {coloring[j]};
18    end if;
19    end do;
20    if nops(coloring) >= i then
21      used := used union {seq(k, k=1..coloring[i])};
22    end if;
23    available := allcolorsL minus used;
24    if available <> {} then
25      parentV := convert(coloring[1..i-1], 'string');
26      coloring := [op(coloring[1..i-1]), available[1]];
27      newV := convert(coloring, 'string');
28      newT := AddVertex(DTList[-1], newV);
29      AddEdge(newT, {parentV, newV});
30      DTList := [op(DTList), newT];
31      i := i + 1;
32    else
33      if nops(coloring) >= i then
34        coloring := coloring[1..(i-1)];
35      end if;
36      i := i - 1;
37    end if;
38  end do;
39  DrawGraph(DTList);
40  Vpos := GetVertexPositions(DTList[-1]);
41  for k from 1 to nops(DTList) do
42    SetVertexPositions(DTList[k], Vpos[1..k]);
43  end do;

```

```

43 plotList := [seq(DrawGraph(DTList[k]), k=1..nops(DTList))];
44 plots[display](plotList, insequence=true);
45 end proc:

```

> *BackColorDT(Fig11Graph, [red, blue, green])*

(1)

> *BackColorDT(K5, [red, blue, green, yellow])*

(2)

n-Queens Problem via Backtracking

Another problem with an elegant backtracking solution is the problem of placing n queens on an $n \times n$ chessboard so that no queen can attack another. This means that no two queens can be placed on the same horizontal, vertical, or diagonal line. We will solve this problem with a backtracking algorithm. The solution we present here is based on the solution given in Example 7 in Section 11.4. We place queens in a greedy fashion on the chessboard until either all the queens are placed or there is no available position for a queen to be placed without coming under attack from a queen already on the board.

Following the textbook, the i th step in the backtracking algorithm will be to place a queen in the i th column (or file, in chess terms). Like the **coloring** list and **Exploring** list, the algorithm will build a **queens** list. In this case, **queens[i] = j** will indicate that a queen is placed in the j th row (rank) in the i th column (file). We will build a helper procedure, **ValidQueens**, that, given the dimension of the board and the current **queens** list, will determine the possible locations for a queen in the next column.

To implement **ValidQueens**, we will need a representation of the status of the board; specifically, for each square on the board, whether it is safe or under attack. It is natural to represent the board as a matrix with an entry 1 indicating that the corresponding square is safe and 0 that it is under attack. Note that we can create a square matrix with all entries initialized to 1 by issuing the **Matrix** command with two arguments: the dimension of the matrix and the formula **fill=1**. For example,

> *Matrix(5, fill = 1)*

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

(11.86)

initializes the matrix representing the board on which no queens have been placed.

We now build a **BoardStatus** procedure that, given the current list of queen locations and the dimension of the board will return a matrix representing the board with that configuration.

The matrix will contain 1 in positions not under attack, 0 in positions under attack, and will represent the location of queens by the symbol Q.

```

1 BoardStatus := proc(curQueens, dim)
2   local board, i, dif, Qrank, Qfile;
3   board := Matrix(dim, fill=1);
4   for Qfile from 1 to nops(curQueens) do
5     Qrank := curQueens[Qfile];
6     for i from 1 to dim do
7       board[Qrank, i] := 0;
8       board[i, Qfile] := 0;
9       dif := i - Qfile;
10      if Qrank + dif <= dim and Qrank + dif >= 1 then
11        board[Qrank+dif, i] := 0;
12      end if;
13      if Qrank - dif <= dim and Qrank - dif >= 1 then
14        board[Qrank-dif, i] := 0;
15      end if;
16    end do;
17  end do;
18  for Qfile from 1 to nops(curQueens) do
19    Qrank := curQueens[Qfile];
20    board[Qrank, Qfile] := 'Q';
21  end do;
22  return board;
23 end proc:
```

For example, on a 10×10 board, with the first queen in the second row and the second queen in the seventh row, the board looks as follows.

> *BoardStatus* ([2, 7], 10)

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ Q & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & Q & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \quad (11.87)$$

The **ValidQueens** procedure will take the same arguments, the list of queen locations and dimension of the board, pass them to **BoardStatus**, and use the resulting matrix to determine available positions in the next column. We could omit the **BoardStatus** procedure and instead create the

ValidQueens procedure independently, but, as you see above, the **BoardStatus** procedure provides a useful visualization.

```

1 ValidQueens := proc(curQueens, dim)
2   local B, file, i, freeSet;
3   B := BoardStatus(curQueens, dim);
4   file := nops(curQueens) + 1;
5   freeSet := {};
6   for i from 1 to dim do
7     if B[i, file] = 1 then
8       freeSet := freeSet union {i};
9     end if;
10    end do;
11    return freeSet;
12  end proc;
```

With this preliminary work out of the way, we are ready to write the main program, **nQueens**. It will work in much the same way as our previous examples.

1. We keep a **queens** list, initialized to the empty list, that records the locations of queens.
2. We initialize a counter **file** to 1. This indicates the column in which we need to place a queen. Notice that, in the **BackColor** algorithm, we initialized the counter to 2. The reason for the difference is that, in the coloring algorithm, the color of the first vertex was arbitrary and changing it from color 1 to a different color could not possibly affect the outcome. In this case, it may be the case that there is no solution with the first queen in file 1, rank 1, but there is a solution if the file 1 queen is in a different rank.
3. Apply the **ValidQueens** procedure with the **queens** list and the board dimension. Store the resulting set as **open**.
4. As with the **BackColor** algorithm, we determine if the current assignment is a new assignment or a result of backtracking. If it is a backtracking step, we remove from **open** the positions equal to or smaller than the previous attempt.
5. We terminate when **file** either exceeds the board dimension, in which case we have found a solution, or when it is backtracked to 0, in which case we have exhausted all possibilities.

```

1 nQueens := proc(boardDim::posint)
2   local queens, file, open, i;
3   queens := [];
4   file := 1;
5   while file > 0 and file <= boardDim do
6     open := ValidQueens(queens[1..(file-1)], boardDim);
7     if nops(queens) >= file then
8       open := open minus {seq(i, i=1..queens[file])};
9     end if;
10    if open <> {} then
11      queens := [op(queens[1..(file-1)]), open[1]];
12      file := file + 1;
13    else
14      if nops(queens) >= file then
15        queens := queens[1..(file-1)];
16      end if;
17      file := file - 1;
```

```

18      end if ;
19  end do ;
20  if file > boardDim then
21    return BoardStatus(queens, boardDim) ;
22  else
23    return FAIL ;
24  end if ;
25 end proc;

```

We can use this to find one solution to the 8-queens problem (8×8 is the size of the standard board).

> *nQueens*(8)

$$\begin{bmatrix} Q & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & Q & 0 \\ 0 & 0 & 0 & 0 & Q & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & Q \\ 0 & Q & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & Q & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & Q & 0 & 0 \\ 0 & 0 & Q & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (11.88)$$

Subset Sum Problem via Backtracking

Finally, we consider the subset sum problem. Given a set of integers S and a value M , we want to find a subset B of S whose sum is M . To use backtracking on this problem, we first impose an ordering on the set S . We successively select integers from S to include in B until the sum of the elements of B equals or exceeds M , and backtrack when the sum exceeds M .

Before we get to the main algorithm, it is worth reviewing two items of syntax. First, given a list of values, we can compute their sum with the **add** command as follows.

> *listofvalues* := [3, 7, 11, 15, -4]
listofvalues := [3, 7, 11, 15, -4] (11.89)

> *add*(*listofvalues*)
32 (11.90)

Second, given a list of values and a second list consisting of indices into the first list, we can obtain the sublist of values corresponding to the positions described by the second list as follows.

> *listofstuff* := ["a", "b", "c", "d", "e", "f", "g"]
listofstuff := ["a", "b", "c", "d", "e", "f", "g"] (11.91)

> *listofindices* := [1, 3, 4, 7]
listofindices := [1, 3, 4, 7] (11.92)

```
> listofstuff[listofindices]
["a", "c", "d", "g"]
```

(11.93)

```
> listofstuff[[3, 4, 6]]
["c", "d", "f"]
```

(11.94)

As the general pattern of backtracking algorithms should be clear by this point, we omit a detailed description of the procedure.

```

1 SubSum := proc (S :: set(integer), M :: integer)
2   local SList, Bindices, allIndices, i, availIndices, k, currSum;
3   SList := [op(S)];
4   Bindices := [];
5   allIndices := {seq(k, k=1..nops(SList))};
6   i := 1;
7   currSum := 0;
8   while i > 0 and currSum <> M do
9     availIndices := allIndices minus {op(Bindices)};
10    if nops(Bindices) >= i then
11      availIndices := availIndices minus
12        {seq(k, k=1..Bindices[i])};
13    end if;
14    for k in availIndices do
15      if currSum + SList[k] > M then
16        availIndices := availIndices minus {k};
17      end if;
18    end do;
19    if availIndices <> {} then
20      Bindices := [op(Bindices[1..(i-1)]), availIndices[1]];
21      i := i + 1;
22    else
23      if nops(Bindices) >= i then
24        Bindices := Bindices[1..(i-1)];
25      end if;
26      i := i - 1;
27    end if;
28    currSum := add(SList[Bindices[1..(i-1)]]);
29  end do;
30  if i = 0 then
31    return FAIL;
32  else
33    return SList[Bindices];
34  end if;
end proc;
```

```
> SubSum({5, 7, 11, 15, 27, 31}, 39)
[5, 7, 27]
```

(11.95)

```
> SubSum({5, 7, 11, 15, 27, 31}, 40)
FAIL
```

(11.96)

11.5 Minimum Spanning Trees

This section explains how to use Maple to find the minimum spanning tree of a weighted graph. Recall that a minimum spanning tree T of a weighted graph G is a spanning tree of G with the minimum weight of all spanning trees of G . The two best-known algorithms for constructing minimum spanning trees are called Prim's algorithm and Kruskal's algorithm. While Maple includes commands, **PrimsAlgorithm** and **KruskalsAlgorithm**, that implement these algorithms, this is another case in which understanding the implementation can help you better understand the algorithms. We will develop our own procedures that implement these two algorithms rather than using Maple's.

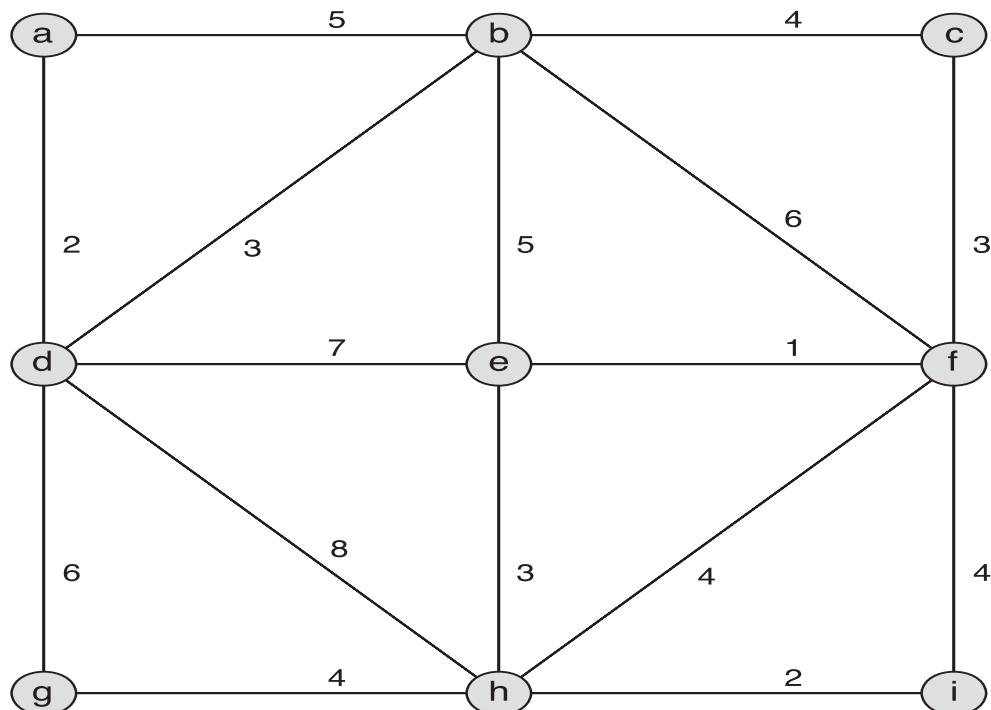
First, we construct a graph to use as an example. We will recreate Exercise 3 from Section 11.5. Recall the syntax for defining undirected, weighted edges: the members of the edge set passed to the **Graph** command are two-element lists whose first element is the undirected edge and whose second element is the weight. For example, `["a","b"],5` represents an edge between a and b with weight 5.

```
> Exercise3 := Graph({[{"a", "b"}, 5], [{"a", "d"}, 2], [{"b", "c"}, 4],  
  [{"b", "d"}, 3], [{"b", "e"}, 5], [{"b", "f"}, 6], [{"c", "f"}, 3],  
  [{"d", "e"}, 7], [{"d", "g"}, 6], [{"d", "h"}, 8], [{"e", "f"}, 1],  
  [{"e", "h"}, 3], [{"f", "h"}, 4], [{"f", "i"}, 4], [{"g", "h"}, 4],  
  [{"h", "i"}, 2]})
```

Exercise3 := Graph 26: an undirected weighted graph with 9 vertices and 16 edge(s)
(11.97)

```
> SetVertexPositions(Exercise3, [[0, 2], [1, 2], [2, 2], [0, 1], [1, 1], [2, 1], [0, 0],  
  [1, 0], [2, 0]])
```

```
> DrawGraph(Exercise3)
```



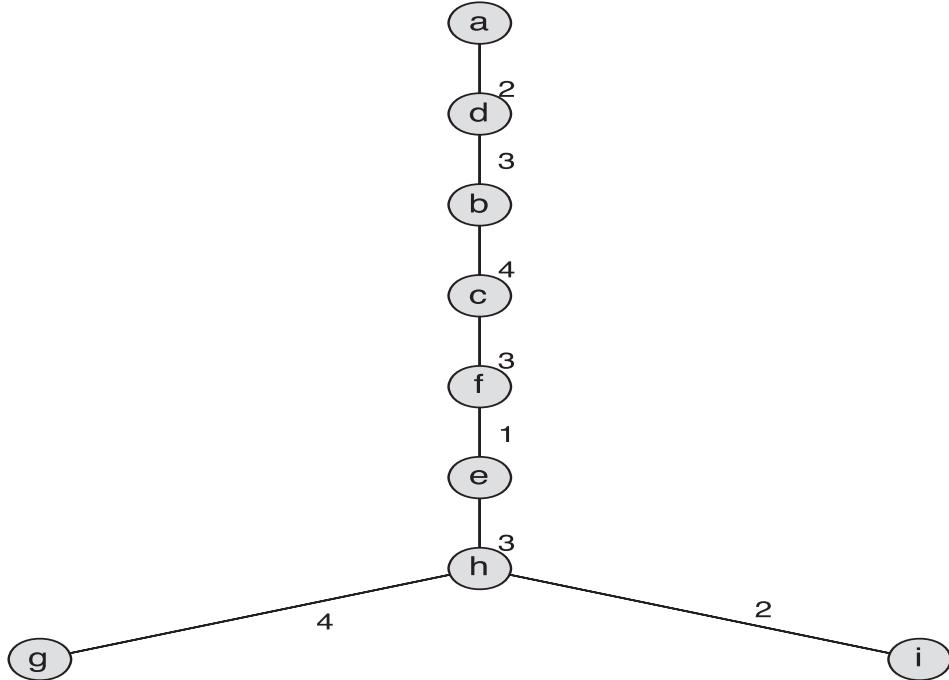
Built-In Commands

Before implementing our own versions of Prim's algorithm and Kruskal's algorithm, we mention how to use Maple's built-in implementations of these algorithms, **PrimsAlgorithm** and **KruskalsAlgorithm**. There is also a command called **MinimalSpanningTree**, which uses Kruskal's algorithm. The syntax for both commands is the same. There is one required argument, the name of the graph. The commands both return a graph object that is a minimum spanning tree for the input. We will use **PrimsAlgorithm** to illustrate, but recall that **KruskalsAlgorithm** uses the same syntax.

```
> Exercise3Prim := PrimsAlgorithm(Exercise3)
```

```
Exercise3Prim := Graph 27: an undirected weighted graph with 9 vertices and 8 edge(s) (11.98)
```

```
> DrawGraph(Exercise3Prim)
```



Both commands have two optional arguments. The first is a variable name, which will be assigned the weight of the minimal spanning tree.

```
> PrimsAlgorithm(Exercise3, 'f')
```

```
Graph 28: an undirected weighted graph with 9 vertices and 8 edge(s) (11.99)
```

```
> f
```

```
22 (11.100)
```

The second optional argument is the keyword **animate**. If this argument is given, the procedures will return an animation showing how the minimal spanning tree is built. Note, however, that this command will not work if strings are used to name vertices. Instead, we must use either numbers or variable names that have not been assigned values.

Below, we define **Exercise3names**, which is identical to **Exercise3** but using names instead of strings. Note that the name **f** already stores a value. Even putting it in left single quotes to delay evaluation does not prevent it from appearing as its numerical value in the drawing of the graph.

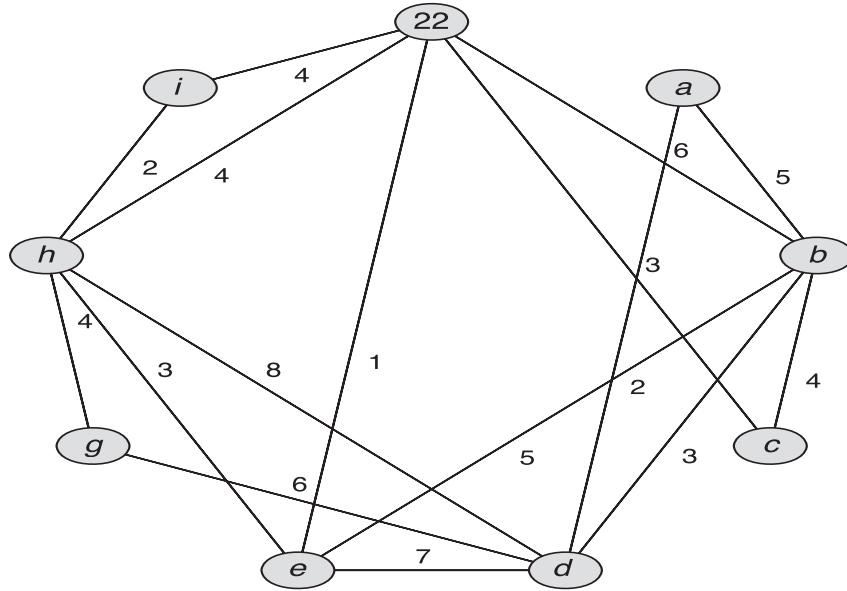
```

> Exercise3names := Graph({[{a,b},5], [{a,d},2], [{b,c},4], [{b,d},3],
  [{b,e},5], [{b,f},6], [{c,f},3], [{d,e},7], [{d,g},6], [{d,h},8],
  [{e,f},1], [{e,h},3], [{f,h},4], [{f,i},4], [{g,h},4], [{h,i},2]})

Exercise3names := Graph 29: an undirected weighted graph with 9 vertices
and 16 edge(s) (11.101)

```

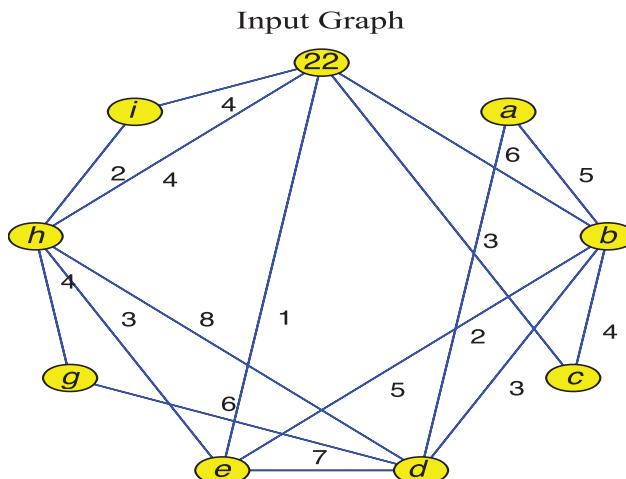
> DrawGraph(Exercise3names)



This is why, elsewhere in this chapter, we always use numbers or strings to name vertices in graphs. Note that it is also not possible to use the **animate** option after setting vertex positions with **SetVertexPositions**.

We are now able to apply **PrimsAlgorithm** with the **animate** option.

> PrimsAlgorithm(Exercise3names, animate)



Prims algorithm builds a Minimum Cost Spanning Tree from a given starting vertex by adding to itself a 'fringe' edge of minimal weight at each iteration. If the edge would create a cycle in the tree it is discarded.

The first frame of the animation shows the graph and some explanation. Subsequent frames show the process of building the minimal spanning tree one edge at a time.

Prim's Algorithm

We will now build our own versions of both algorithms. We will also see how to create animations that illustrate the process of building the spanning trees, but without the restrictions imposed by the built-in commands.

Since both algorithms depend on choosing an edge of smallest weight, it will be useful to be able to apply **sort** to a list of edges. Recall that **sort** accepts an optional second argument: a procedure that takes two arguments and returns true if the first argument is “less than” the second. As we have seen before, this procedure needs to also depend on the graph, so we will create a functional operator that takes a graph and produces the right kind of procedure that can be used by **sort**.

```
1  edgeCompare := G -> proc (a, b)
2      local Wa, Wb;
3      Wa := GetEdgeWeight (G, a);
4      Wb := GetEdgeWeight (G, b);
5      return evalb (Wa < Wb);
6  end proc;
```

We will now consider Prim's algorithm, which is given as Algorithm 1 in Section 11.5 of the textbook. Prim's algorithm constructs a minimum spanning tree by successively selecting an edge of smallest weight that extends the tree without creating any loops.

To simplify our implementation of Prim's algorithm, we will create a procedure **MinEdge**. Given the original graph and the list of vertices already included in the spanning tree, **MinEdge** determines which edge of the graph should be added next.

MinEdge determines the set of edges that are incident with a vertex currently in the tree using the **IncidentEdges** command. **IncidentEdges** takes two arguments. The first argument is a graph, and the second is either a vertex or a list of vertices. It returns the set of edges incident to the given vertex or vertices. The **MinEdge** procedure then eliminates any edge with both ends already in the spanning tree. (This is equivalent to the condition that the edge not introduce a simple circuit.) Once it has determined the valid candidates, the **MinEdge** procedure returns the edge with smallest weight. We also include the special case that the spanning tree has not yet been started, in which case we call the procedure with the empty list as the second argument.

```
1  MinEdge := proc (G::Graph, V::list)
2      local possibleEdges, e, edgeList;
3      uses GraphTheory;
4      if V = [] then
5          possibleEdges := Edges (G);
6      else
7          possibleEdges := IncidentEdges (G, V);
8      end if;
9      for e in possibleEdges do
10         if e[1] in V and e[2] in V then
11             possibleEdges := possibleEdges minus {e};
12         end if;

```

```

13  end do;
14  if possibleEdges = {} then
15      return NULL;
16  end if;
17  edgeList := [op(possibleEdges)];
18  edgeList := sort(edgeList, edgeCompare(G));
19  return edgeList[1];
20 end proc:

```

With this procedure in place, Prim's algorithm is fairly straightforward to implement.

1. Begin building the spanning tree by finding the edge of minimum weight with the command **MinEdge(G,[])**.

2. Continue building the spanning tree one edge at a time by adding the edge returned by **MinEdge**. (Note that we must add the new vertex before the edge, since **AddEdge** expects both endpoints of the edge to be added to already be in the graph.)
3. After $n - 2$ repetitions of step 2, where n is the number of vertices in the graph, the spanning tree is complete.
4. For the sake of displaying the resulting tree, we conclude the procedure by checking to see if the original graph has had its vertex positions explicitly set, and, if so, those positions are copied to the tree. (Note that we cannot conveniently use **SetVertexPositions** in this case, because the vertices in the spanning tree are likely in a different order than they are in the graph. Specifically, the order of the vertices in the spanning tree is the order in which they are added to the tree.)

```

1 Prim := proc (G :: Graph)
2     local newEdge, T, n, v, pos;
3     uses GraphTheory;
4     newEdge := MinEdge (G, []);
5     T := Graph ({newEdge}, weighted);
6     SetEdgeWeight (T, newEdge, GetEdgeWeight (G, newEdge));
7     n := nops (Vertices (G));
8     from 1 to n-2 do
9         newEdge := MinEdge (G, Vertices (T));
10        if newEdge[1] in Vertices (T) then
11            T := AddVertex (T, newEdge[2]);
12        else
13            T := AddVertex (T, newEdge[1]);
14        end if;
15        AddEdge (T, newEdge);
16        SetEdgeWeight (T, newEdge, GetEdgeWeight (G, newEdge));
17    end do;
18    for v in Vertices (T) do
19        pos := GetVertexAttribute (G, v, "draw-pos-user");
20        if pos <> FAIL then
21            SetVertexAttribute (T, v, "draw-pos-user"=pos);
22        end if;
23    end do;
24    return T;
25 end proc:

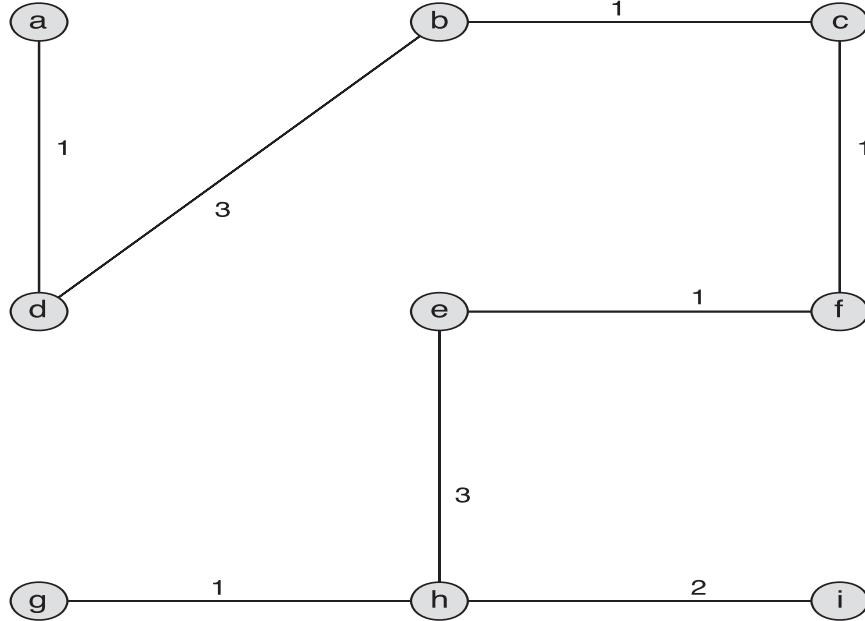
```

We can now use this algorithm to find the minimum spanning tree for Exercise 3.

> *PrimExercise3 := Prim(Exercise3)*

PrimExercise3 := Graph 30: an undirected weighted graph with 9 vertices and 8 edge(s)
(11.102)

> *DrawGraph(PrimExercise3)*



Before moving on to Kruskal's algorithm, we will create a procedure to produce an animation demonstrating Prim's algorithm in action. First, we modify the Prim procedure to record the list of edges that form the tree and return this list of edges rather than the tree.

```

1 PrimEdges := proc (G::Graph)
2     local newEdge, T, edgeList, n;
3     uses GraphTheory;
4     newEdge := MinEdge (G, []);
5     T := Graph ({newEdge}, weighted);
6     SetEdgeWeight (T, newEdge, GetEdgeWeight (G, newEdge));
7     edgeList := [newEdge];
8     n := nops (Vertices (G));
9     from 1 to n-2 do
10        newEdge := MinEdge (G, Vertices (T));
11        if newEdge [1] in Vertices (T) then
12            T := AddVertex (T, newEdge [2]);
13        else
14            T := AddVertex (T, newEdge [1]);
15        end if;
16        AddEdge (T, newEdge);
17        SetEdgeWeight (T, newEdge, GetEdgeWeight (G, newEdge));
18        edgeList := [op (edgeList), newEdge];
19    end do;
20    return edgeList;
21 end proc;

```

> *PrimEdges (Exercise3)*

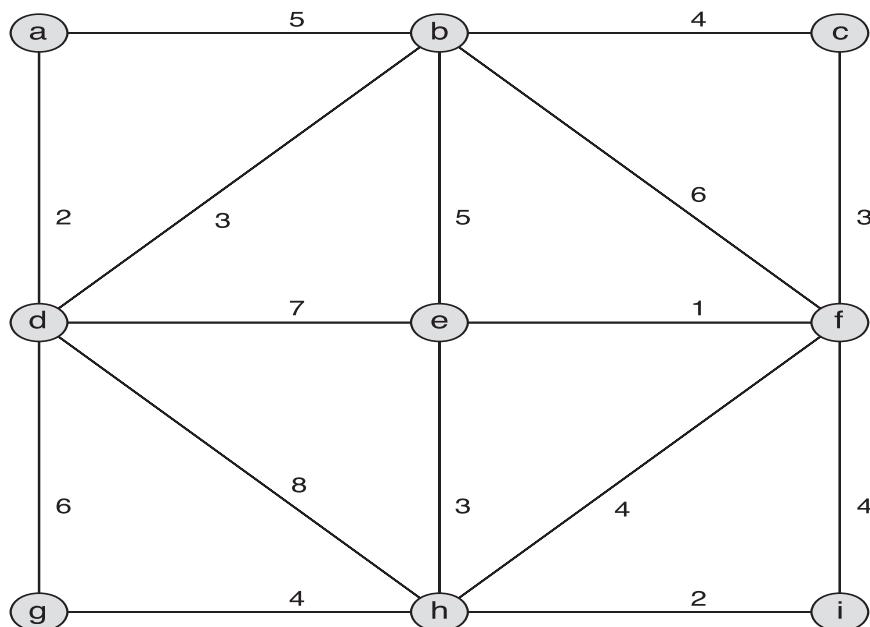
$\{ \{ "e", "f" \}, \{ "e", "h" \}, \{ "h", "i" \}, \{ "c", "f" \}, \{ "g", "h" \}, \{ "b", "c" \}, \{ "b", "d" \}, \{ "a", "d" \} \}$

(11.103)

We now write the procedures that will produce the animation. Since these are nearly identical to the **plotPath** and **animatePath** procedures from Chapter 10, we refer the reader to Section 10.5 of this manual for a detailed explanation.

```
1 plotTree := proc(G::Graph, T::list, n)
2   local Gcopy, subtree, N;
3   uses GraphTheory;
4   Gcopy := CopyGraph(G);
5   if n > nops(T) then
6     N := nops(T);
7   else
8     N := floor(n);
9   end if;
10  if N <> 0 then
11    subtree := T[1..N];
12    HighlightEdges(Gcopy, subtree, ["Red" $ N]);
13  end if;
14  DrawGraph(Gcopy);
15 end proc;
16 animateTree := proc(G::Graph, T::list)
17   local t;
18   plots[animate](plotTree, [G, T, t],
19   t=0..(nops(T)), paraminfo=false, frames=50);
20 end proc;
```

> *animateTree (Exercise3, PrimEdges (Exercise3))*



Kruskal's Algorithm

Recall that Kruskal's algorithm, Algorithm 2 in Section 11.5, begins in the same way as Prim's algorithm, with the edge of smallest weight. The difference is that at each step, Kruskal's algorithm adds whatever edge is of least weight which does not create a simple circuit, regardless of whether it is incident to an edge already in the graph.

We begin with a procedure to test whether or not a given edge will create a simple circuit. Note that, during the steps of Kruskal's algorithm, we have a forest of trees. An edge will create a simple circuit if and only if both of its endpoints are in the same tree within the forest. We use the **ConnectedComponents** command to find the trees. The **ConnectedComponents** command returns a list of lists, where each inner list is the vertices within one of the connected components of the graph. We test an edge by looping through each of the connected components and making sure that both ends do not appear in the same component.

```
1 NotFormsCircuit := proc (G::Graph, edge)
2   local components, C;
3   uses GraphTheory;
4   components := ConnectedComponents (G);
5   for C in components do
6     if edge [1] in C and edge [2] in C then
7       return false;
8     end if;
9   end do;
10  return true;
11 end proc;
```

Now, we implement Kruskal's algorithm. The procedure is as follows:

1. Initialize **edges** to the list of edges of the given graph and sort this list using the **edgeCompare** procedure created above.
2. Initialize **T** to the empty graph.
3. Consider the first edge in the **edges** list. Use **NotFormsCircuit** to determine if it is safe to add to the tree. If we can add it to the tree, add one or both of its ends as new vertices to **T** as needed and add the edge.
4. Regardless of whether **NotFormsCircuit** approves the addition of the first edge in **edges** to the tree, remove the edge from **edges**—either the edge is now used in the tree and so will not be used again, or its addition would create a circuit (a fact which will not change later).
5. Repeat steps 3 and 4 until $n - 1$ edges have been added, where n is the number of vertices.

```
1 Kruskal := proc (G::Graph)
2   local edges, T, n, i, newEdge, v, pos;
3   uses GraphTheory;
4   edges := [op(Edges (G))];
5   edges := sort (edges, edgeCompare (G));
6   T := Graph (weighted);
7   n := nops (Vertices (G));
8   i := 1;
9   while i <= n-1 do
10    newEdge := edges [1];
11    if NotFormsCircuit (T, newEdge) then
12      if not newEdge [1] in Vertices (T) then
```

```

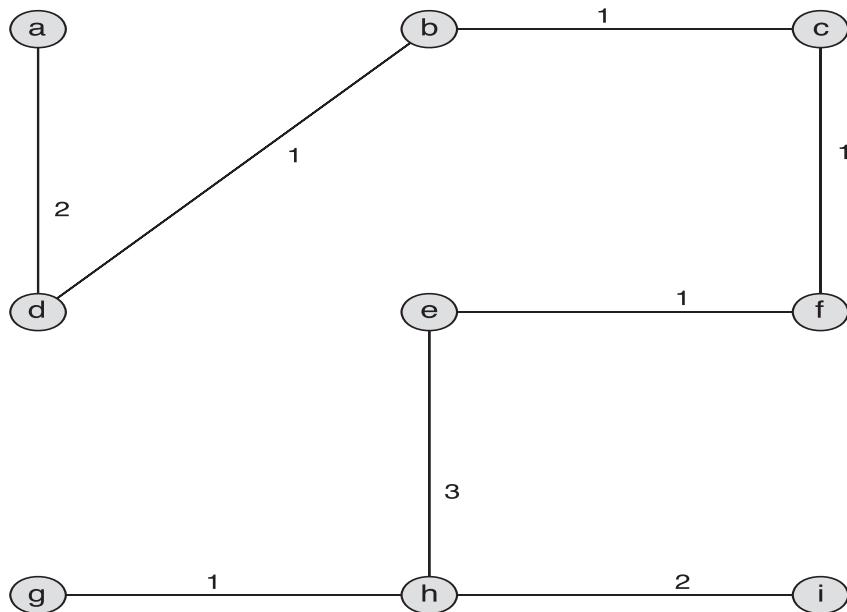
13      T := AddVertex(T, newEdge[1]);
14  end if;
15  if not newEdge[2] in Vertices(T) then
16      T := AddVertex(T, newEdge[2]);
17  end if;
18      AddEdge(T, newEdge);
19      SetEdgeWeight(T, newEdge, GetEdgeWeight(G, newEdge));
20      i := i + 1;
21  end if;
22  edges := subsop(1=NULL, edges);
23 end do;
24 for v in Vertices(T) do
25     pos := GetVertexAttribute(G, v, "draw-pos-user");
26     if pos <> FAIL then
27         SetVertexAttribute(T, v, "draw-pos-user"=pos);
28     end if;
29 end do;
30 return T;
31 end proc;

```

Note that Kruskal's algorithm produces the same minimum spanning tree for Exercise 3 as did our implementation of Prim's algorithm (in fact, this graph has a unique minimum spanning tree).

> *KruskalExercise3 := Kruskal(Exercise3)*
KruskalExercise3 := Graph 31: an undirected weighted graph with 9 vertices and 8 edge(s) (11.104)

> *DrawGraph(KruskalExercise3)*



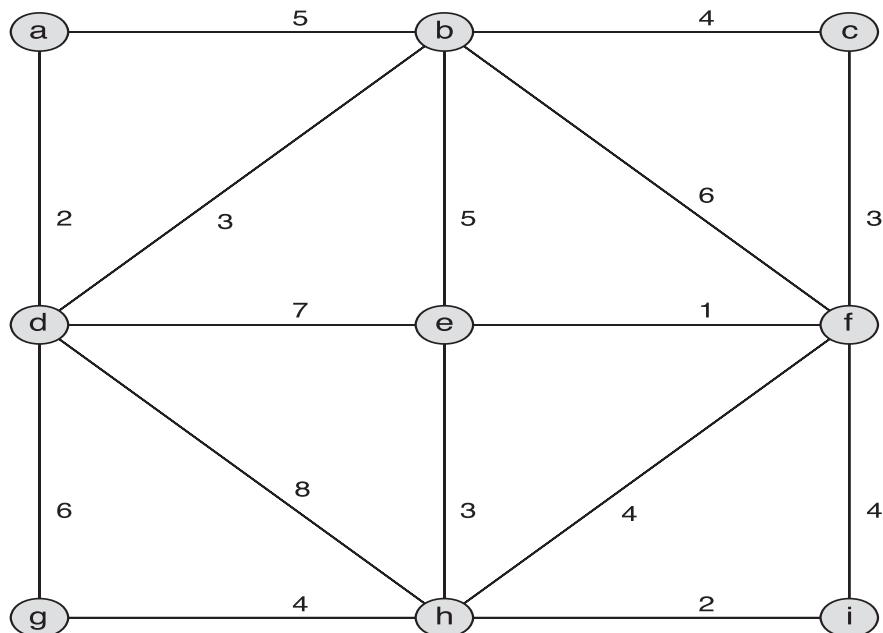
We can produce an animation, like we did for Prim's algorithm, by modifying the procedure to produce the list of edges in the order they are added and then using the **animateTree** command once again.

```

1 KruskalEdges := proc (G::Graph)
2   local edges, edgeList, T, n, i, newEdge;
3   uses GraphTheory;
4   edges := [op(Edges(G))];
5   edges := sort(edges, edgeCompare(G));
6   edgeList := [];
7   T := Graph(weighted);
8   n := nops(Vertices(G));
9   i := 1;
10  while i <= n-1 do
11    newEdge := edges[1];
12    if NotFormsCircuit(T, newEdge) then
13      if not newEdge[1] in Vertices(T) then
14        T := AddVertex(T, newEdge[1]);
15      end if;
16      if not newEdge[2] in Vertices(T) then
17        T := AddVertex(T, newEdge[2]);
18      end if;
19      AddEdge(T, newEdge);
20      SetEdgeWeight(T, newEdge, GetEdgeWeight(G, newEdge));
21      edgeList := [op(edgeList), newEdge];
22      i := i + 1;
23    end if;
24    edges := subsop(1=NULL, edges);
25  end do;
26  return edgeList;
27 end proc;

```

> *animateTree(Exercise3, KruskalEdges(Exercise3))*



By comparing the two animations, you can see how the two algorithms provide different routes to the same end result.

Solutions to Computer Projects and Computations and Explorations

Computer Projects 6

Given the ordered list of edges of an ordered rooted tree, find the universal addresses of its vertices.

Solution: Recall that the universal address of a vertex in an ordered rooted tree is defined as follows. The root has address 0 and its children have addresses 1, 2, 3, etc., in order. The address of every other vertex is defined recursively as $p.n$ where p is the address of the vertex's parent and n is 1 if the vertex is the first child of its parent, 2 if it is the second child, etc.

Before solving this problem, we will make the following assumption on the input: the edges are sorted according to the lexicographical order of the universal address of their terminal vertex. That is to say, the edges are listed in the order of their appearance from left to right and top to bottom when the tree is drawn in the usual way. This makes some of what follows slightly easier. The reader is encouraged to generalize the procedure we create here so as to not depend on this restriction.

We build the **ORTree** determined by the list of edges and add a “univ-address” attribute to each vertex containing the vertex's universal address. First, here is an ordered list of edges for an ordered rooted tree.

```
> CP6Example := [[“D”, “E”], [“D”, “C”], [“D”, “I”], [“E”, “L”], [“E”, “F”],  
    [“T”, “B”], [“T”, “K”], [“T”, “H”], [“F”, “J”], [“F”, “G”], [“F”, “A”]]  
CP6Example := [[“D”, “E”], [“D”, “C”], [“D”, “I”], [“E”, “L”],  
    [“E”, “F”], [“I”, “B”], [“T”, “K”], [“T”, “H”], [“F”, “J”],  
    [“F”, “G”], [“F”, “A”]]  
(11.105)
```

We have purposefully chosen vertex names that are out of order with respect to the ordering of the edges so that our construction of the **ORTree** is sure to rely only on the ordering of the edges and not the vertex labels.

Note that creating a graph with these edges only requires turning the list of edges into a set and passing it to the **Graph** command.

```
> CP6ExGraph := Graph({op(CP6Example)})  
CP6ExGraph := Graph 32: a directed unweighted graph with 12 vertices and 11 arc(s)  
(11.106)
```

Our first task is to turn this into an ordered rooted tree. Since we provided directed edges, all that is required to make the graph rooted is to set the graph attribute “root.” Since the edges were ordered,

the first edge is the edge from the root to the root’s first child. This means that we can get the root by accessing the first element of the first edge.

```
> CP6ExRoot := (CP6Example[1])[1]  
CP6ExRoot := "D" (11.107)
```

```
> SetGraphAttribute(CP6ExGraph, "root" = CP6ExRoot)  
> type(CP6ExGraph, RTree)  
true (11.108)
```

To make this graph an ordered rooted tree, we need to set the “order” attribute for each vertex. For the root, we just set the root’s order to 0.

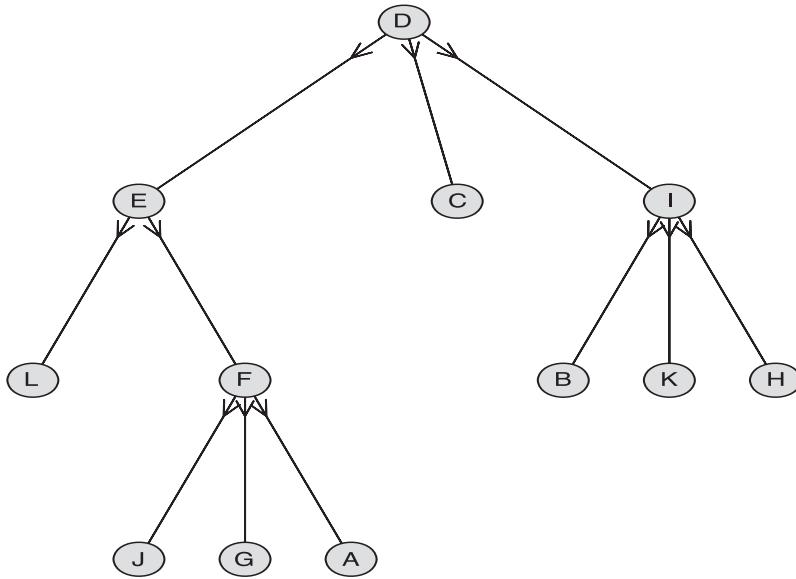
```
> SetVertexAttribute(CP6ExGraph, CP6ExRoot, "order" = 0)
```

For the rest of the vertices, we need to do some more work. Our approach will be as follows. Loop through all of the edges in the original edge list, in order, keeping track of two variables, **curParent**, the “current parent,” and the **childOrder**. The **curParent** will initially be set to the root of the tree and **childOrder** will be initialized to 1. For the first edge in the list, we assign the child vertex (the terminal vertex of the ordered edge) “order” equal to **childOrder**. We then go to the next edge. If the **curParent** is the same as the parent vertex in this edge, then **childOrder** is incremented and we assign the child vertex of this edge an “order” equal to the new **childOrder** value. Otherwise, the parent vertex of this new edge is different from **curParent**. This indicates that we have moved on to a new parent with a new set of children, so we set **curParent** to this new parent and reset **childOrder** to 1. Here is the code for this step in the process.

```
> curParent := CP6ExRoot:  
childOrder := 1:  
for thisEdge in CP6Example do  
  if thisEdge[1] ≠ curParent then  
    curParent := thisEdge[1];  
    childOrder := 1;  
  end if;  
  SetVertexAttribute(CP6ExGraph, thisEdge[2], "order" = childOrder);  
  childOrder := childOrder + 1;  
end do :
```

Now that the order attributes are set, our graph is an ordered rooted tree.

```
> type(CP6ExGraph, ORTree)  
true (11.109)  
> DrawORTree(CP6ExGraph)
```



The preliminaries are now done and we can determine the universal addresses of the vertices. Again, the root will have universal address 0.

> *SetVertexAttribute (CP6ExGraph, CP6ExRoot, "univ-address" = "0")*

The universal address of the root's children will be the same as their "order".

```

> for CPRootChild in FindChildren(CP6ExGraph, CP6ExRoot) do
    tempAddress := GetVertexAttribute(CP6ExGraph, CPRootChild, "order");
    tempaddress := convert(tempAddress, {string});
    SetVertexAttribute(CP6ExGraph, CPRootChild, "univ-address" = tempaddress);
end do :

```

After that, we proceed as in a breadth-first search through the tree. We initialize a list with the children of the root.

> *ToProcessAddress := FindChildren (CP6ExGraph, CP6ExRoot)*
ToProcessAddress := ["C", "E", "I"] (11.110)

We process a vertex by (1) adding its children to the list; (2) setting the children's universal addresses to the result of concatenating the parent's address and the child's order; and (3) removing it from the list.

```

> while ToProcessAddress ≠ [] do
    processV := ToProcessAddress[1];
    Vaddress := GetVertexAttribute(CP6ExGraph, processV, "univ-address");
    for CToAddress in FindChildren(CP6ExGraph, processV) do
        ToProcessAddress := [op(ToProcessAddress), CToAddress];
        Caddress := GetVertexAttribute(CP6ExGraph, CToAddress, "order");
        Caddress := cat(Vaddress, ".", Caddress);
        SetVertexAttribute(CP6ExGraph, CToAddress, "univ-address" = Caddress);
    end do :
    ToProcessAddress := subsop(1 = NULL, ToProcessAddress);
end do :

```

Now, we have set the universal address attribute for all the vertices. For example,

> *GetVertexAttribute (CP6ExGraph, "A", "univ-address")*
“1.2.3” (11.111)

indicates that A is the root's first child's second child's third child. Or, A is the third child of the second child of the first child of the root.

We now put this together in a procedure.

```
1 UniversalAddress := proc(edgeList : :list)
2   local T, root, curParent, childOrder, thisEdge, V, tempAddress,
3       ToProcessAddress, Vaddress, C, Caddress;
4   uses GraphTheory;
5   T := Graph ({op(edgeList)} );
6   root := edgeList [1] [1];
7   SetGraphAttribute(T, "root"=root);
8   SetVertexAttribute(T, root, "order"=0);
9   curParent := root;
10  childOrder := 1;
11  for thisEdge in edgeList do
12    if thisEdge [1] <> curParent then
13      curParent := thisEdge [1];
14      childOrder := 1;
15    end if;
16    SetVertexAttribute(T, thisEdge [2], "order"=childOrder);
17    childOrder := childOrder + 1;
18  end do;
19  SetVertexAttribute(T, root, "univ-address"="0");
20  for V in FindChildren(T, root) do
21    tempAddress := GetVertexAttribute(T, V, "order");
22    tempAddress := convert(tempAddress, 'string');
23    SetVertexAttribute(T, V, "univ-address"=tempAddress);
24  end do;
25  ToProcessAddress := FindChildren(T, root);
26  while ToProcessAddress <> [] do
27    V := ToProcessAddress [1];
28    Vaddress := GetVertexAttribute(T, V, "univ-address");
29    for C in FindChildren(T, V) do
30      ToProcessAddress := [op(ToProcessAddress), C];
31      Caddress := GetVertexAttribute(T, C, "order");
32      Caddress := cat(Vaddress, ".", Caddress);
33      SetVertexAttribute(T, C, "univ-address"=Caddress);
34    end do;
35    ToProcessAddress := subsop (1=NULL, ToProcessAddress);
36  end do;
37  return T;
end proc;
```

Computations and Explorations 1

Display all trees with six vertices.

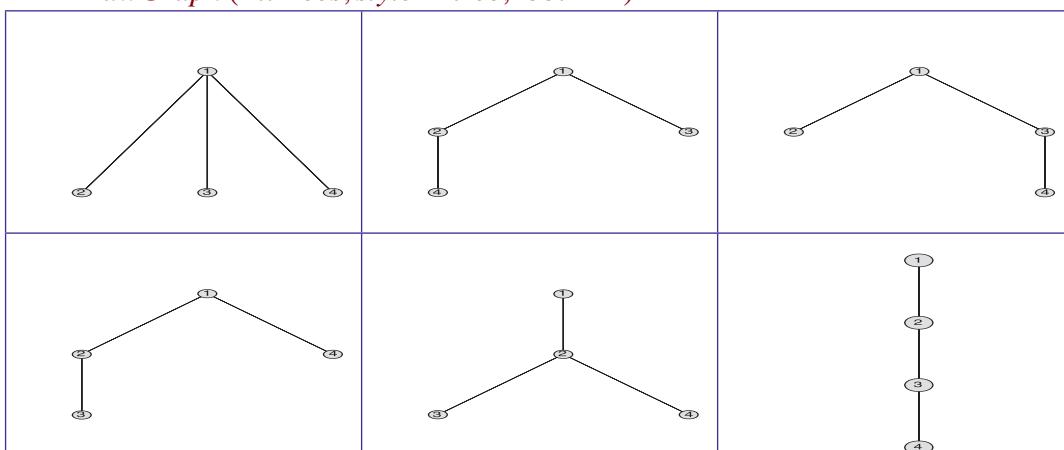
Solution: To solve this problem, we make use of a recursive definition of trees. The empty graph is a tree and the graph with a single vertex is also a tree. Given any tree, we can form a new tree with one additional vertex by adding the new vertex as a leaf connected to any one of the original vertices. (The reader can verify that this indeed creates all trees with one more vertex.)

We shall create a procedure, called **ExtendTrees**, that accepts as input a set of trees on n vertices and returns the resulting set of trees on $n + 1$ vertices. For each of the original trees, we consider each vertex of the tree and create a new tree by adding a leaf to it.

```
1 ExtendTrees := proc(Trees::set)
2   local newTrees, newV, T, v, newT;
3   uses GraphTheory;
4   newTrees := {};
5   newV := nops(Vertices(Trees[1])) + 1;
6   for T in Trees do
7     for v in Vertices(T) do
8       newT := AddVertex(T, newV);
9       AddEdge(newT, {v, newV});
10      newTrees := newTrees union {newT};
11    end do;
12  end do;
13  return newTrees;
14 end proc;
```

We can now use this procedure to determine all trees on four vertices. Finding all the trees of larger sizes is left to the reader.

```
> AllTrees := {Graph([1])}
AllTrees := {Graph 33: a graph with 1 vertex and no edges} (11.112)
> for i from 2 to 4 do
  AllTrees := ExtendTrees(AllTrees)
end do:
> DrawGraph(AllTrees, style = tree, root = 1)
```



Computations and Explorations 3

Construct a Huffman code for the symbols with ASCII codes given the frequency of their occurrence in representative input.

Solution: ASCII, which stands for American Standard Code for Information Interchange, includes 128 characters, including 33 nonprinting characters. Most of the nonprinting characters, with the exception of the space and the carriage return or newline character, however, are rarely used. We will focus on the standard characters of English and the newline character.

Since we have already created the procedure **HuffmanCode**, which creates a Huffman code based on a list of character/weight pairs, the main work we need to do is determine the frequencies of characters in a sample input. We use the following input, which contains letters, punctuation, and newline characters. (Note: You enter newline characters in a string in the same way as you create new lines when entering a procedure, by pressing the shift and enter/return keys simultaneously. You enter quotation marks inside a string by preceding the quotation mark with a backslash.)

```
> inputText := "The quick brown fox said,  
  \"How do you do, my friend?\"  
  Then he ran very quickly off into the sunset."  
inputText := "The quick brown fox said,  
  \"How do you do, my friend?\"  
  Then he ran very quickly off into the sunset." (11.113)
```

Note that characters in a string can be accessed as if the string were a list. For instance, the fifth character in the text above is

```
> inputText[5]  
"q" (11.114)
```

Using **printf**, Maple will display the newline characters for us explicitly. Note that Maple differentiates between a newline character and a space character.

```
> printf(%a, inputText[26])  
"\r"  
  
> printf(%a, inputText[4])  
"  
  
> evalb (inputText[4] = inputText[26])  
false (11.115)
```

We can obtain the frequencies of characters using the **StringTools** command **CharacterFrequencies**. This command can be applied to a string and produces a sequence of equations identifying characters with the number of times they occur in the string. We can see that the input text includes seven e's, two newline characters, and 17 spaces.

```

> StringTools[CharacterFrequencies](inputText)
“
“ = 2, “” = 17, “”” = 2, “,” = 2, “.” = 1, “?” = 1, “H” = 1, “T” = 2,
“a” = 2, “b” = 1, “c” = 2, “d” = 4, “e” = 7, “f” = 4, “h” = 4,
“i” = 5, “k” = 2, “l” = 1, “m” = 1, “n” = 6, “o” = 8, “q” = 2,
“r” = 4, “s” = 3, “t” = 3, “u” = 4, “v” = 1, “w” = 2, “x” = 1,
“y” = 4

```

(11.116)

All that remains is to transform this data into the format expected by the Huffman algorithm: a list of pairs of the characters and the relative frequency of their occurrence. We use **op** on an equation with first argument 1 to obtain the left-hand side and with first argument 2 to obtain the right-hand side. The frequency is divided by the number of characters in the string, which we obtain with the **length** command.

```

> op(1, “d” = 4)
“d”

```

(11.117)

```

>  $\frac{\text{op}(2, “d” = 4)}{\text{length}(\text{inputText})}$ 
 $\frac{4}{99}$ 

```

(11.118)

```

1 FrequencyList := proc(s :: string)
2   local len, Flist, c, x;
3   len := length(s);
4   Flist := [];
5   for c in StringTools[CharacterFrequencies](s) do
6     x := [op(1, c), op(2, c)/len];
7     Flist := [op(Flist), x];
8   end do;
9   return Flist;
10 end proc;

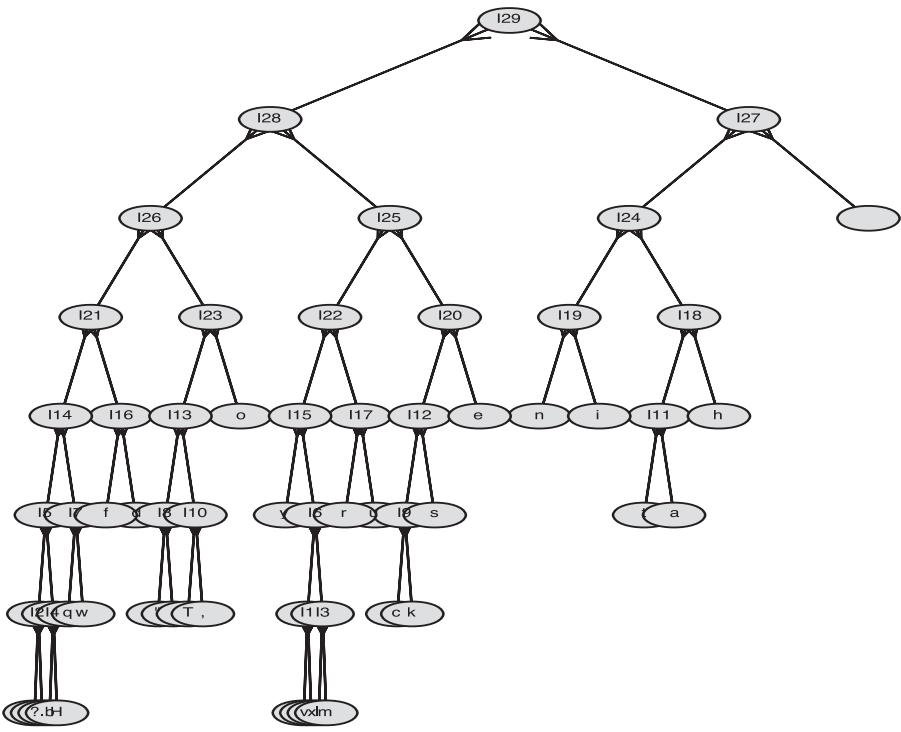
```

Finally, we are able to apply the **HuffmanCode** procedure.

```

> foxList := FrequencyList(inputText):
> foxCode := HuffmanCode(foxList):
> DrawBTree(foxCode)

```



You will need to enlarge the image substantially in order to see the result in a readable format. In addition, note that Maple automatically suppresses the display of the edge weights for a tree this large.

Computations and Explorations 4

Compute the number of different spanning trees of K_n for $n = 1, 2, 3, 4, 5, 6$. Conjecture a formula for the number of such spanning trees whenever n is a positive integer.

Solution: Maple provides a command, **NumberOfSpanningTrees**, which returns the number of unique spanning trees of an undirected graph. Thus, the number of spanning trees on K_n for n from 1 to 6 can be computed as follows.

```
> seq(NumberOfSpanningTrees(CompleteGraph(n)), n = 1 .. 6)
1, 1, 3, 16, 125, 1296
(11.119)
```

We leave it to the reader to make a conjecture for the formula.

Computations and Explorations 8

Draw the complete game tree for a game of checkers on a 4×4 board.

Solution: We will provide a partial solution to this problem; the reader is left to complete the full solution. Specifically, we will create a Maple procedure called **MovePiece** that will determine all possible new checker arrangements given the current state of the board and the player whose turn it is. Once this procedure is created, the reader must determine how to represent these board positions as vertices and edges, how to determine the next level of the game tree, as well as the halting conditions.

Before writing this procedure, however, we must establish a representation of the board. Naturally, we will use a matrix whose size is the size of the board. Empty board spaces will contain 0. Board spaces in which a regular white or black piece is sitting will be represented by 1 or 2, respectively. Kings will be represented by negative values, -1 for a white king and -2 for a black king. The following represents an initial board before any moves have been made.

$$> \text{CheckersStart} := \text{Matrix}([[0, 2, 0, 2], [0, 0, 0, 0], [0, 0, 0, 0], [1, 0, 1, 0]])$$

$$\text{CheckersStart} := \begin{bmatrix} 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \quad (11.120)$$

Given a matrix representing a board state and an integer representing which side's turn it is, the procedure **MovePiece** will list all of the possible results of the player's move. It operates as follows:

1. Initialize **newBoards**, which will be the list of all possible boards that result from the current move, to the empty list.
2. If **side** is 1, then normal pieces move up the board from bottom to top and we set **direction** to -1, since the index of rows in a matrix decrease as we move up the board. If the **side** is 2, then **direction** is set to 1.
3. Begin a pair of for loops, with indices **r** and **c**. These for loops allow us to consider each possible board location. In each position, we want to know if that location holds a piece belonging to the current player. If it is 1's turn, then that player's normal pieces are represented by a 1 in the position and the player's kings are represented by a -1 in the position. Likewise, 2's pieces are represent by 2 or by -2. Thus, we can determine if a position holds a player's piece by comparing the absolute value of the matrix entry with the **side**. If the square does not hold a piece belonging to the current player, we simply move on to the next location.
4. Check to see if the piece is a king and set the variable **isKing** to 1 if it is a king or 0 if not. We then begin a for loop from 0 to the value of **isKing**. If **isKing** is 0, the loop executes only once. If **isKing** is 1, then the loop will execute twice. The index of this loop, **King**, is used to control **rowDir**, the current direction being considered. **rowDir** is either the same as **direction** or, in the case of the second iteration for a king, the reverse direction.
5. We now check to see if the possible moves keep the piece on the board. First, we make sure that **r+rowDir**, that is, the row in which the piece would move to, is still between 1 and 4 (the possible rows).
6. Assuming moving the piece will not take it off the top or bottom of the board, consider the left and right moves. We do this with a for loop which sets the variable **colDir** to -1 and then to 1. Again, we check to see that **c+colDir**, the current column plus the proposed change to the column position, is still on the board.
7. At this point, we know that the board position (**r+rowDir,c+colDir**) is actually a board position. There are now three possibilities: the position is empty, there is an enemy piece in the square, or there is a friendly piece in the square.

8. In the first case, the position is empty, we want to move the piece to that location. We make a copy of the **Board** matrix with **LinearAlgebra[Copy]** (so we do not modify the original board). Then, we make the move and add the new board to the list. Note that we also check to see if the piece becomes a king by moving into this position.
9. In the second case, there is an enemy in the square, then we test to see if it is possible to jump. That is, we must make sure that the landing location after the jump is both on the board and empty. If so, we make the jump, that is, we copy the **Board** and make the necessary modifications. If not, then the move is not possible.
10. In the third case, the move is not possible and we do nothing.

Here is the procedure:

```

1 MovePiece := proc (Board :: Matrix, side :: integer)
2   local newBoards, direction, r, c, isKing, King, rowDir, colDir,
3       newB;
4   newBoards := [];
5   if side = 1 then
6     direction := -1;
7   else
8     direction := 1;
9   end if;
10  for r from 1 to 4 do
11    for c from 1 to 4 do
12      if abs (Board[r, c]) = side then
13        if Board[r, c] < 0 then
14          isKing := 1;
15        else
16          isKing := 0;
17        end if;
18        for King from 0 to isKing do
19          if King = 0 then
20            rowDir := direction;
21          else
22            rowDir := -1 * direction;
23          end if;
24          if r+rowDir >= 1 and r+rowDir <= 4 then
25            for colDir from -1 to 1 by 2 do
26              if c+colDir >= 1 and c+colDir <= 4 then
27                if Board[r+rowDir, c+colDir] = 0 then
28                  newB := LinearAlgebra [Copy] (Board);
29                  if ((r+rowDir=1 and side=1) or (r+rowDir=4 and side=2)) and
30                    Board[r, c] > 0 then
31                    newB[r+rowDir, c+colDir] := -1*Board[r, c];
32                  else
33                    newB[r+rowDir, c+colDir] := Board[r, c];
34                  end if;
35                  newB[r, c] := 0;
36                  newBoards := [op(newBoards), newB];
37                elif abs (Board[r+rowDir, c+colDir]) <> side then
38                  if r+2*rowDir >= 1 and r+2*rowDir <= 4 and c+2*colDir >= 1 and
39                    c+2*colDir <= 4 then

```

```

37   if Board[r+2*rowDir, c+2*colDir] = 0 then
38     newB := LinearAlgebra[Copy](Board);
39     if ((r+2*rowDir=1 and side=1) or (r+2*rowDir=4 and side=2))
40       and Board[r, c] > 0 then
41       newB[r+2*rowDir, c+2*colDir] := -1*Board[r, c];
42     else
43       newB[r+2*rowDir, c+2*colDir] := Board[r, c];
44     end if;
45     newB[r, c] := 0;
46     newB[r+rowDir, c+colDir] := 0;
47     newBoards := [op(newBoards), newB];
48   end if;
49   end if;
50   end if;
51   end do;
52   end if;
53   end do;
54   end if;
55   end do;
56   end do;
57   return newBoards;
58 end proc:

```

We now demonstrate a few steps using the procedure.

> *Move1* := *MovePiece*(*CheckersStart*, 1)

$$\left[\begin{bmatrix} 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \right] \quad (11.121)$$

> *Move2* := *MovePiece*(*Move1*[1], 2)

$$\left[\begin{bmatrix} 0 & 0 & 0 & 2 \\ 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \right] \quad (11.122)$$

> *Move3* := *MovePiece*(*Move2*[3], 1)

$$\left[\begin{bmatrix} 0 & 2 & 0 & 0 \\ 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 2 & 0 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \right] \quad (11.123)$$

> *Move4* := *MovePiece*(*Move3*[2], 2)

$$\begin{bmatrix} 0 & 0 & 0 & -1 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

(11.124)

Exercises

Exercise 1. Write Maple procedures for finding the eccentricity of a vertex in an unrooted tree and for finding the center of an unrooted tree. (Eccentricity and center are defined in prelude to Exercise 39 of Section 11.1 of the text.)

Exercise 2. Develop a Maple procedure for constructing rooted Fibonacci trees. (See the prelude to Exercise 45 of Section 11.1 for a definition of a Fibonacci tree.)

Exercise 3. Develop a Maple procedure for listing the vertices of an ordered rooted tree in level order.

Exercise 4. Compare the performance of binary search trees to linear search as follows:

- a) Write a procedure, **LinearSearch**, that takes two inputs, a list of integers and an integer to find, and checks each element of the list in order until the input is found, at which time it returns true. If the desired integer is not found, it is added at the end of the list.
- b) Use the command **combinat[randperm](n)** for a positive integer n to create a list of the first n integers in random order, with an appropriately large n . Apply the **MakeBST** command to the list to create a binary search tree for the data.
- c) Randomly select some positive integers to search for. The **randcomb** command could be useful here.
- d) Use both **LinearSearch** and **BInsertion** to find the integers from part c in the list and tree, respectively. Time them using the typical **st := time(): procedure: time() - st;** structure. Repeat this for 100 different permutations and compare the resulting times. Compare these data (representing average-case complexity) with the theoretical worst-case results of n comparisons for **LinearSearch** and $\lceil \log(n + 1) \rceil$ for **BInsertion**.

Exercise 5. Construct a Maple procedure for decoding a message which was encoded with a Huffman code. That is, given a Huffman coding tree produced by the **HuffmanCode** procedure and a message encoded by the **EncodeString** procedure, the algorithm should return the original string.

Exercise 6. Use the Shakespearean sonnets to estimate the frequency of characters used by Shakespeare. (See Section 7.3 of this manual to see how to read the data into Maple and make use of the procedures given in the solution to Computations and Explorations 3 above to compute the frequencies of characters used in the poems.) Then, create a Huffman code based on the sonnets and encode the *ShakespeareData.txt* with the Huffman code. Compare the storage space required by the Huffman encoded version of the file as opposed to the space that would be used to encode the file in ASCII format, assuming each ASCII character requires 7 bits.

Exercise 7. Compare the performance of the breadth-first and depth-first functions for finding spanning trees. Use the **RandomGraph** command (available in the **RandomGraphs** subpackage of **GraphTheory**) to create random spanning trees. The expression **RandomGraphs[RandomGraph](v,e)** will create a random graph with **v** vertices and **e** edges, but

you will need to ensure the graph is connected before applying the functions for creating the spanning tree.

Exercise 8. Construct an undirected weighted graph which has at least two different minimum spanning trees and for which the **Prim** and **Kruskal** algorithms will return different results.

Exercise 9. Write a Maple procedure implementing the reverse-delete algorithm for constructing minimal spanning trees. (The reverse-delete algorithm is described in the prelude to Exercise 34 in Section 11.5.)

Exercise 10. Explore the relative complexity of **Prim**, **Kruskal**, and the reverse-delete procedure you created in the previous exercise. Use the **RandomGraph** command (available in the **RandomGraphs** subpackage of **GraphTheory**) to experiment with their performance. The command **RandomGraph(v,e,connected,weights=x..y)** will produce a random weighted connected graph with **v** vertices, **e** edges, and edge weights chosen randomly between **x** and **y** (with **x < y**). For each algorithm, can you find properties that you can impose on the graphs that will ensure that the algorithm will outperform the others?

Exercise 11. Develop a Maple procedure for producing degree-constrained spanning trees, which are defined in the Supplementary Exercises for Chapter 11. Use this procedure on a set of randomly generated graphs to attempt to construct degree-constrained spanning trees in which each vertex has degree no larger than 3.

Exercise 12. Use Maple to analyze the game of Nim with different starting conditions via the technique of game trees. (See Example 6 in Section 11.2 for a description of the game of Nim.)

Exercise 13. Use Maple to analyze the game of checkers on square boards of different sizes via the technique of game trees. (See the solution to Computations and Explorations 8 for the beginnings of a solution.)

Exercise 14. Develop Maple procedures for finding a path through a maze using the technique of backtracking.

Exercise 15. Develop Maple procedures for solving Sudoku puzzles using the technique of backtracking.

Exercise 16. Use Maple to generate as many graceful trees as possible. (See the Supplementary Exercises of Chapter 11 for a definition of graceful.) Based on the examples you find, make conjectures about graceful trees.

Exercise 17. Alter the postfix expression evaluator, **EvalPostfix**, to handle prefix expressions.

12 Boolean Algebra

Introduction

In this chapter, we will use Maple to model Boolean algebra. In the first section, we demonstrate the basic commands of the **Logic** package, which will be used extensively in this chapter. In the second section, we will focus on the disjunctive normal form of a logical expression. We will see how to use Maple’s command for finding a disjunctive normal form expression for a Boolean function, and we will develop a procedure for finding such a representation for a function defined by a table of values. In Section 3, we will see how Maple can be used to model logical circuits, including how to go about transforming a circuit diagram into a Maple expression. We also provide a procedure that will transform a logical expression into a model of a circuit. In the final section of the chapter, we consider simplification of logical expressions, and we develop an implementation of the Quine–McCluskey method.

In this chapter, we will be using the Maple package **Logic**. This package includes several commands related to Boolean algebra that will be useful. We load this package now.

```
> with(Logic):
```

12.1 Boolean Functions

In this section, we will introduce Maple’s **Logic** package, which can be used to explore Boolean algebra. In particular, we will see how Maple represents Boolean operators, how to work with Boolean expressions, and how to create Boolean functions. We will also use Maple to verify identities in Boolean algebra and to compute the dual of an expression.

Preliminaries

In Chapter 1 of this manual, we discussed Maple’s logical expressions. The Boolean values **true** and **false** are represented by the literal constants **true** and **false**. To Maple, these are constant values, like the numbers 2 or **Pi**.

We also saw in Chapter 1 the logical operators, including **and**, **or**, and **not**. These are similar to arithmetic operators like + and *. Combining Boolean values with the logical operators causes Maple to evaluate the resulting expression.

```
> true or (not(false) and false)  
true
```

(12.1)

These “ordinary” operators are a vital part of any programming language, as they are needed for controlling execution of procedures. Moreover, Maple recognizes a third value, **FAIL**, which is useful from the perspective of programming. (The help page for Boolean expressions provides tables showing how the operations are defined on **true**, **false**, and **FAIL**.)

The **Logic** package provides a second set of Boolean operators. The basic operators are **&and**, **&or**, and **¬**. These are supplemented by **&nor**, **&xor** (exclusive or), **&implies** (implication), and **&iff** (biconditional).

These operators are different from the ordinary operators in three ways. First, they are inert, as opposed to active. This means that when you enter an expression in Maple using the **Logic** package operators, they are not immediately evaluated. They are, however, displayed in symbol form in the output.

$$\begin{aligned} > \quad & (\text{true} \&\text{or} \&\text{not}(\text{false}))\&\text{and} \text{false} \\ & (\text{true} \vee (\neg \text{false})) \wedge \text{false} \end{aligned} \tag{12.2}$$

This makes it easier to explore and analyze Boolean expressions symbolically since Maple will not perform simplification until you explicitly tell it to do so.

The second difference is that the operators in the **Logic** package do not recognize **FAIL** as a logical value.

The third difference is that the **Logic** operators all have equal precedence, so you should fully parenthesize statements.

More on Precedence

Precedence and parentheses require a bit of explanation. Recall that the usual order of operations for logical operators is **not**, then **and**, then **or**, and finally implication. When you enter an expression with the **Logic** operators, this order of precedence is not respected.

For example, if you enter p **or** q **and** r , using the **Logic** operators, Maple will consider the operations from left to right.

$$\begin{aligned} > \quad & p \&\text{or} \ q \&\text{and} \ r \\ & (p \vee q) \wedge r \end{aligned} \tag{12.3}$$

Note that in the output for that expression, Maple has put $p \vee q$ in parentheses. This is consistent with the fact that the operators have equal precedence, meaning that the **&or** is applied first. That is, $(p \vee q) \wedge r$ is the interpretation of what was input.

Using the inert **Logic** operators, you must enforce the order of precedence yourself. To input $p \vee q \wedge r$ and have Maple interpret it in the correct order, you must use parentheses.

$$\begin{aligned} > \quad & p \&\text{or} \ (q \&\text{and} \ r) \\ & p \vee (q \wedge r) \end{aligned} \tag{12.4}$$

It is a good idea to always fully parenthesize your input with the **Logic** package or you may obtain erroneous results.

The 0–1 Form of Boolean Algebra

We conclude this subsection with a warning.

The textbook uses the objects 0 and 1 with operations +, ·, and – instead of *true*, *false*, \vee , \wedge , and \neg as we have done. Maple does, in fact, have commands available for using the 0–1 form. The **Logic** package contains commands, **Import** and **Export**, that can be used to change between expressions involving the **Logic** inert operators and either the active Boolean operators or a 0–1 form, called **MOD2**.

However, the **MOD2** form is in opposition to the conventions used in the textbook. Specifically, in Maple, **+** corresponds to **&xor**, rather than **&or**.

To avoid the confusion that this difference could create, we will always use the logical form in this manual. However, some readers may be interested to know that it is possible to create operators to mirror the kinds of Boolean algebra expressions used in the text. To do so, you can create definitions for the inert addition, multiplication, and negation operators. Recall that in Chapter 4, we made definitions for the inert arithmetic operators in order to emulate modular arithmetic.

For example, to define a Boolean **+**, we define **&+** as follows.

```
> '&+' := (a :: {0,1}, b :: {0,1}) → ifelse(a = 1 or b = 1, 1, 0)
      '&+' := (a :: {0,1}, b :: {0,1}) ↪ ifelse(a = 1 or b = 1, 1, 0) (12.5)
```

The type declarations will ensure that only bits are allowed to be addends.

With this definition, we can now compute $0 + 1$.

```
> 0 &+ 1
      1 (12.6)
```

Once the definitions of **&*** and a unary **&-** are in place, we could compute the value of $1 \cdot 0 + (0 + 1)$ (Example 1) by entering the following:

```
> (1 &* 0) &+ (&-(0 &+ 1))
```

Definitions of the other operations are left to the interested reader. In this manual, we will not use this approach, since the logical form of Boolean expressions is more naturally supported by Maple.

Boolean Expressions and Boolean Functions

We saw above how to create Boolean expressions using the inert **Logic** operators. Now, we will look at how to evaluate Boolean expressions and how to create Boolean functions.

Evaluating Boolean Expressions

Because the operators in **Logic** are inert, you must explicitly tell Maple to evaluate expressions involving them by applying the **BooleanSimplify** command.

Consider Example 1 from the text, which asks that we compute the value of $1 \cdot 0 + \overline{(0 + 1)}$. To perform this computation in Maple, we must first translate it into a logical statement. We do this by changing 1 into *true*, 0 into *false*, the multiplication into \wedge , the addition into \vee , and the bar into \neg .

This results in the Boolean expression $(\text{true} \wedge \text{false}) \vee \neg(\text{false} \vee \text{true})$. Note that we added parentheses since the **Logic** arguments have equal precedence.

Using Maple's active logical operators, as in Chapter 1, we would just enter the statement to evaluate it.

```
> (true and false) or not(false or true)
      false (12.7)
```

Using the inert operators, we apply the **BooleanSimplify** command. This command accepts only one argument, the logical expression that it is to simplify. In this example, it will simplify the expression down to a single truth value.

```
> BooleanSimplify ((true &and false) &or &not (false &or true))
false
```

(12.8)

As you might guess from the name, **BooleanSimplify** does more than evaluate an expression to a truth value. It will also simplify more general expressions involving unassigned names. For example, consider the Boolean expression $(x + y)(x + z)$. In logical terms, this is $(x \vee y) \wedge (x \vee z)$.

```
> BooleanSimplify ((x &or y) &and (x &or z))
x \vee (y \wedge z)
```

(12.9)

Note that Maple has simplified this in accordance with the distributive law.

Representing Boolean Functions

Defining a Boolean function is identical to defining any other function in Maple.

Consider, for example, the Boolean function shown below (written in the 0–1 notation).

$$F(x, y, z) = xy + yz + zx.$$

This can be modeled in Maple by the procedure **Fp** (p for procedure).

1	Fp := proc (x, y, z)
2	return (x &and y) &or (y &and z) &or (z &and x);
3	end proc:

It is often more natural, however, to model such functions as functional operators.

```
> F := (x, y, z) → (x &and y) &or (y &and z) &or (z &and x)
F := (x, y, z) ↪ ((x \wedge y) \vee (y \wedge z)) \vee (z \wedge x)
```

(12.10)

You can work with **F** (or **Fp**) in the usual way. The following applies **F** to **p**, **q**, and **r**.

```
> F(p, q, r)
((p \wedge q) \vee (q \wedge r)) \vee (r \wedge p)
```

(12.11)

Observe what happens when **F** is applied to truth values.

```
> F(true, false, true)
((true \wedge false) \vee (false \wedge true)) \vee (true \wedge true)
```

(12.12)

Because the operators **&and** and **&or** are inert, Maple does not automatically simplify to a truth value. We have to explicitly call **BooleanSimplify**.

```
> BooleanSimplify(F(true, false, true))
true
```

(12.13)

It can be easier to embed the call to **BooleanSimplify** in the function definition itself. We redefine **F** to include **BooleanSimplify**.

```
> F := (x,y,z) →  
  BooleanSimplify((x &and y) &or (y &and z) &or (z &and x))  
F := (x,y,z) ↪ BooleanSimplify(((x ∧ y) ∨ (y ∧ z)) ∨ (z ∧ x))
```

(12.14)

Now applying **F** to **p**, **q**, and **r** produces the same result as before, since the expression cannot be simplified any further.

```
> F(p,q,r)  
(p ∧ q) ∨ (q ∧ r) ∨ (r ∧ p)
```

(12.15)

But applying **F** to truth values will return a single truth value.

```
> F(true,false,true)  
true
```

(12.16)

You can also mix truth values and symbols. In this case, the inclusion of **BooleanSimplify** in the definition of **F** causes Maple to simplify the expression as much as possible, given the partial information.

```
> F(true,q,r)  
q ∨ r
```

(12.17)

The output indicates that if *p* is known to be true, then $(p \wedge q) \vee (q \wedge r) \vee (r \wedge p)$ is equivalent to $q \vee r$.

Values of Boolean Functions

Examples 4 and 5 of Section 12.1 illustrate how the values of a Boolean function, in the 0–1 format, can be displayed in a table. In the logical form, this is equivalent to a truth table for the Boolean function. In Chapter 1 of this manual, we created truth tables by looping through all the possible values and also mentioned **TruthTable**.

The **TruthTable** command requires two arguments. The first argument is a Boolean expression. The second is a list of the variable names that appear in the expression.

For example, we will display the table of values for the Boolean function **F** defined above. The first argument to the **TruthTable** command will be the Boolean expression obtained by **F(p,q,r)**. The second argument will be the list **[p,q,r]**.

```
> Fdataframe := TruthTable(F(p,q,r),[p,q,r])
```

	<i>p</i>	<i>q</i>	<i>r</i>	<i>value</i>
1	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>
2	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>
3	<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>
4	<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>
5	<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>
6	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>
7	<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>
8	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>

(12.18)

Note that the result of **TruthTable** is a **DataFrame** object. If you wish to access individual elements, you can do so by referring to the row and column labels. For example, the entries in the 4th row corresponding to *q* and to the result are:

```
> Fdataframe[4, 'q']
true(12.19)
```

```
> Fdataframe[4, 'value']
true(12.20)
```

Note that the single quotes are not necessary, unless the name has been assigned to something.

Perhaps surprisingly, if you use the bracket selection operator with a single entry, either by number or name, it will refer to a column, not a row.

```
> Fdataframe[4], Fdataframe['value']
[ 1 false ] [ 1 false ]
[ 2 false ] [ 2 false ]
[ 3 false ] [ 3 false ]
[ 4 true ] [ 4 true ]
[ 5 false ] [ 5 false ]
[ 6 true ] [ 6 true ]
[ 7 true ] [ 7 true ]
[ 8 true ] [ 8 true ](12.21)
```

To extract a row, give two dots as the second index in the selection operation, which Maple understands to be the entire range of values. Below, we also put the row number in a list. This can be used to select multiple rows; here, it has the effect of the output appearing as a row.

```
> Fdataframe[[4], ..]
[ p   q   r   value ]
[ 4 false true true true ](12.22)
```

The **TruthyTable** command allows you to specify the representation of the truth table with the output option, if you would prefer a **Matrix** or **table**. Tables can be a very useful alternative to a **DataFrame**, since the indices in the table will be tuples corresponding to the values of the independent variables and the corresponding values are the truth values of the expression.

```
> Ftable := TruthTable(F(p,q,r), [p,q,r], output = table)
Ftable := table([(true, false, false) = false, (false, false, true) = false,
                 (true, false, true) = true, (true, true, false) = true,
                 (true, true, true) = true, (false, true, true) = true,
                 (false, true, false) = false, (false, false, false) = false])(12.23)
```

We illustrate interacting with the table form by printing one entry at a time. Recall that **indices** applied to a **table** produces a sequence of lists.

```
> indices(Ftable)
[true, false, false], [false, false, true], [true, false, true], [true, true, false],
[true, true, true], [false, true, true], [false, true, false],
[false, false, false](12.24)
```

In order to use the indices with the selection operation, we must apply **op** to remove the added list structure.

```
> for i in indices(Ftable) do
    print(i, Ftable[op(i)])
end do
[true, false, false], false
[false, false, true], false
[true, false, true], true
[true, true, false], true
[true, true, true], true
[false, true, true], true
[false, true, false], false
[false, false, false], false(12.25)
```

Operations on Boolean Functions

As with functions on real numbers, Boolean functions can be combined using basic operations. The complement as well as the sum and product of Boolean functions are defined in the text.

To compute complements, sums, and products of Boolean functions, you must define a new function in terms of the original. For example, consider the function $G(x, y) = x \cdot y$. In logical notation, this is $G(x, y) = x \wedge y$.

```
> G := (x, y) → BooleanSimplify(x &and y)
G := (x, y) ↪ BooleanSimplify(x ∧ y)(12.26)
```

The complement of **G**, which we will call **notG**, is created as follows. The arguments of **notG** are the same as **G**. The formula that defines **notG** is **¬ G(x,y)**. That is, **¬** is applied to the result of evaluating **G** on the arguments. And once again, **BooleanSimplify** is called.

$$\begin{aligned}
 > \text{notG} &:= (x, y) \rightarrow \text{BooleanSimplify}(\&\text{not}(G(x, y))) \\
 & \text{notG} := (x, y) \mapsto \text{BooleanSimplify}(\neg G(x, y))
 \end{aligned} \tag{12.27}$$

Observe that if we evaluate **notG** at a pair of variables, the presence of **BooleanSimplify** ensures that De Morgan's law is applied.

$$\begin{aligned}
 > \text{notG}(x, y) \\
 & (\neg x) \vee (\neg y)
 \end{aligned} \tag{12.28}$$

Let us define another function, $H(x, y) = x \cdot \bar{y}$.

$$\begin{aligned}
 > H &:= (x, y) \rightarrow \text{BooleanSimplify}(x \&\text{and} (\&\text{not } y)) \\
 & H := (x, y) \mapsto \text{BooleanSimplify}(x \wedge (\neg y))
 \end{aligned} \tag{12.29}$$

To compute the Boolean sum $G + H$, we combine the functions with the **&or** operator. More precisely, we define a functional operator **GpH** with the formula **G(x,y) &or H(x,y)**.

$$\begin{aligned}
 > \text{GpH} &:= (x, y) \rightarrow \text{BooleanSimplify}(G(x, y) \&\text{or } H(x, y)) \\
 & \text{GpH} := (x, y) \mapsto \text{BooleanSimplify}(G(x, y) \vee H(x, y))
 \end{aligned} \tag{12.30}$$

Applying this to a pair of variables, we obtain the following formula for $G + H$.

$$\begin{aligned}
 > \text{GpH}(x, y) \\
 & x
 \end{aligned} \tag{12.31}$$

This result indicates that $x \cdot y + x \cdot \bar{y} = x$. This can also be verified using the identities in Table 5 of Section 12.1.

Identities of Boolean Algebra

The **Logic** package's **Equivalent** command makes checking identities and equivalence of Boolean expressions fairly straightforward. This command can also be used to check equality of Boolean functions.

Identities and Equivalence of Boolean Expressions

We will use the distributive law $x(y + z) = xy + xz$ as an example. First, we must translate the statement from the 0–1 form into a statement of logic: $x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$.

Now, we assign the expressions on either side of the equivalence to names. This is not necessary, but it will make later statements easier to read.

$$\begin{aligned}
 > \text{distributiveL} &:= x \&\text{and} (y \&\text{or} z) \\
 & \text{distributiveL} := (x \wedge (y \vee z))
 \end{aligned} \tag{12.32}$$

$$\begin{aligned}
 > \text{distributiveR} &:= (x \&\text{and} y) \&\text{or} (x \&\text{and} z) \\
 & \text{distributiveR} := (x \wedge y) \vee (x \wedge z)
 \end{aligned} \tag{12.33}$$

To confirm the equivalence of the two Boolean expressions, we use the **Equivalent** command. This command requires two arguments, the two Boolean expressions that are to be tested for equivalence. The command returns true if the expressions are equivalent and false if not.

```
> Equivalent(distributiveL,distributiveR)
true
```

(12.34)

This verifies the given distributive law.

The **Equivalent** command also accepts an optional third argument. In the case that the two expressions are not equivalent, if you provide an unevaluated name (a name in single right quotes), then that name will be assigned to a set of assignments of truth values to the variables in the expression that demonstrate that the expressions are not equivalent.

Consider the nonequivalence: $x + xy \neq y$. In logical form, this is $x \vee (x \wedge y) \not\equiv y$. Apply the **Equivalent** command with third argument '**P**'.

```
> Equivalent((x &or (x &and y)),y,'P')
false
```

(12.35)

The name **P** now stores a set indicating assignments for x and y .

```
> P
{x = false,y = true}
```

(12.36)

This output means that setting x equal to false and y equal to true provides a demonstration, by counterexample, that $x \vee (x \wedge y) \not\equiv y$. Indeed, substituting $x = \text{false}$ and $y = \text{true}$ on the left hand side produces

$$\text{false} \vee (\text{false} \wedge \text{true}) \equiv \text{false} \vee \text{false} \equiv \text{false}.$$

That is not the same as the right-hand side, y , which is assigned *true*.

Note that the single right quotes around the name in the third argument of **Equivalent** ensure that, should **P** have already stored a value, that value would be overwritten. If you omit the single quotes and **P** has a value already stored in it, an error will result.

Equality of Boolean Functions

Equality of Boolean functions can also be checked with the **Equivalent** command. You do this by applying the command to the two functions evaluated on the same variables.

Consider the following Boolean functions.

$$f_1(x,y) = \overline{(xy)}$$

$$f_2(x,y) = \bar{x} + \bar{y}.$$

Define the corresponding functional operators:

```
> f1 := (x,y) → BooleanSimplify(&not(x &and y))
f1 := (x,y) ↪ BooleanSimplify(¬(x ∧ y))
```

(12.37)

```

> f2 := (x,y) → BooleanSimplify ((&not x) &or (&not y))
f2 := (x,y) ↪ BooleanSimplify ((¬x) ∨ (¬y))
```

(12.38)

We can test the assertion that $f_1(x,y) = f_2(x,y)$ by applying the **Equivalent** command with arguments **f1(x,y)** and **f2(x,y)**.

```

> Equivalent (f1 (x,y),f2 (x,y))
true
```

(12.39)

Duality

We conclude this section by introducing the **Dual** command. This command accepts only one argument, a Boolean expression, and produces the dual of that expression.

For example, consider the expression $x \cdot \bar{y} + y \cdot \bar{z} + \bar{x} \cdot z$. As a logical expression, this can be written as $(x \wedge \neg y) \vee (y \wedge \neg z) \vee (\neg x \wedge z)$. We calculate the dual by applying the **Dual** command.

```

> Dual ((x &and &not (y)) &or (y &and &not (z)) &or (&not (x) &and z))
((x ∨ (¬y)) ∧ (y ∨ (¬z))) ∨ ((¬x) ∨ z)
```

(12.40)

Similarly, the dual of $\bar{x}y + \bar{y}z + x\bar{z}$ can be computed by

```

> Dual ((&not (x) &and y) &or (&not (y) &and z) &or (x &and &not (z)))
(((¬x) ∨ y) ∧ ((¬y) ∨ z)) ∨ (x ∨ (¬z))
```

(12.41)

Note that Exercise 13 of Section 12.1 asks you to prove that the expressions $x \cdot \bar{y} + y \cdot \bar{z} + \bar{x} \cdot z$ and $\bar{x} \cdot y + \bar{y} \cdot z + x \cdot \bar{z}$ are equivalent. The duality principle implies that the duals calculated above are also equivalent. This can be verified by the **Equivalent** command.

```

> Equivalent (12.40), (12.41)
true
```

(12.42)

12.2 Representing Boolean Functions

In this section, we will see how to use Maple to express Boolean functions in the disjunctive normal form (also called sum-of-products expansion). We will first look at the Maple command for turning an expression in Boolean algebra into the disjunctive normal form. Then, we will see how to write a procedure for finding an expression based on a table of values.

Disjunctive Normal Form from an Expression

Given an expression written in terms of the **Logic** operators, the **Canonicalize** command can be used to transform the expression into disjunctive normal form.

As an example, consider Example 3 of Section 12.2: $(x + y)\bar{z}$. In logical form, this is $(x \vee y) \wedge \neg z$. We assign this logical expression to a name.

```

> Example3 := (x &or y) &and &not (z)
Example3 := (x ∨ y) ∧ (¬z)
```

(12.43)

The **Canonicalize** command has several forms. To transform an expression into disjunctive normal form, two arguments are required. The first argument is the expression to be transformed. Second, you must provide a set or list containing the variables appearing in the expression.

$$> \text{Canonicalize}(\text{Example3}, \{x, y, z\}) \\ ((\neg z) \wedge x \wedge y) \vee ((\neg z) \wedge x \wedge (\neg y)) \vee ((\neg z) \wedge y \wedge (\neg x)) \quad (12.44)$$

Note that this agrees with the solution to Example 3.

The **Canonicalize** command also accepts a third argument used to specify the type of canonical form desired. The default behavior is to produce disjunctive normal form, but this can be emphasized by including the option **form=DNF**.

$$> \text{Canonicalize}(\text{Example3}, \{x, y, z\}, \text{form} = \text{DNF}) \\ ((\neg z) \wedge x \wedge y) \vee ((\neg z) \wedge x \wedge (\neg y)) \vee ((\neg z) \wedge y \wedge (\neg x)) \quad (12.45)$$

To produce conjunctive normal form instead, you use the **form=CNF** option.

$$> \text{Canonicalize}(\text{Example3}, \{x, y, z\}, \text{form} = \text{CNF}) \\ (x \vee y \vee z) \wedge (x \vee y \vee (\neg z)) \wedge (x \vee (\neg y) \vee (\neg z)) \wedge (y \vee (\neg x) \vee (\neg z)) \\ \wedge ((\neg x) \vee (\neg y) \vee (\neg z)) \quad (12.46)$$

There is a third option, **form=MOD2**, which results in the 0–1 canonical form. However, as we mentioned earlier, Maple interprets + as the exclusive or, and thus the result of this option will be different from what is described in the textbook.

The **Normalize** command can also be used to produce a disjunctive normal form expression. Note that an argument specifying the variables is not used by **Normalize**. Like **Canonicalize**, the default is disjunctive normal form, but **form=DNF** and **form=CNF** are both accepted by **Normalize**.

Here is the result of **Normalize** applied to Example 3.

$$> \text{Normalize}(\text{Example3}) \\ ((\neg z) \wedge x) \vee ((\neg z) \wedge y) \quad (12.47)$$

Note that the result is not the same as before. In fact, if you compare this result to the solution of Example 3, you will see that this expression is equivalent to the second line in the step-by-step expansion in the solution. In particular, it is the result of applying the distributive law to the original expression.

Normalize produces an expression in disjunctive normal form. That is, the result is a disjunction of terms with each term consisting of a conjunction of variables and their negations. It does not, however, produce canonical disjunctive normal form, like **Canonicalize** does.

The benefit of **Normalize** is that, as you can see, it is typically simpler than canonical disjunctive normal form. The benefit of **Canonicalize** is that the canonical disjunctive normal form is unique. That is, two equivalent expressions necessarily have the same canonical disjunctive normal form, while **Normalize** may express them differently. While the textbook does not emphasize this, the examples and techniques it describes are for producing the canonical sum-of-products expansion.

Disjunctive Normal Form from a Table

Example 1 of Section 12.2 describes how to find an expression for a Boolean function represented by a table of values. Here, we will show how to write a procedure to accomplish this task.

First, we must decide how we will represent the table of values in Maple. Rather than creating a Maple table object, we will represent the table as the set of those assignments of truth values to variables for which the function returns true.

For example, consider the function defined by the following table.

x	y	z	$F(x, y, z)$
<i>true</i>	<i>true</i>	<i>true</i>	<i>false</i>
<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>
<i>true</i>	<i>false</i>	<i>true</i>	<i>false</i>
<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>
<i>false</i>	<i>true</i>	<i>false</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>

There are four rows in the table for which the function returns true. We represent this table by forming the set consisting of the four lists of values for x , y , and z corresponding to those rows.

```
> exampleTable := {[false,false,true], [false,true,false],
[true,false,false], [true,true,false]}
exampleTable := {[false,false,true], [false,true,false],
[true,false,false], [true,true,false]} (12.48)
```

This will be the first argument to the procedure we create. The procedure will also need names for the variables; therefore, we also require a list of names. This will be the second argument to the procedure.

```
> exampleVariables := [x,y,z]
exampleVariables := [x,y,z] (12.49)
```

To form the Boolean expression that represents this function, we follow Example 1. For each row in the table for which the function returns true, that is, for each element in **exampleTable**, we produce the corresponding minterm.

To create a minterm associated with an element of **exampleTable**, we proceed as follows. Initialize a local variable to **NULL**. This variable will hold the sequence of variables or their negations. Then, begin a for loop with loop variable ranging from 1 to the number of variables. Within the loop, test whether or not the entry in the list of truth values is true or not. If the entry is true, then update the sequence by adding the name of the corresponding variable. If the entry is false, then update the sequence by appending the negation of the variable. Finally, apply **&and** to the sequence of variables or negations of variables. This approach, rather than applying **&and** at each step, will result in minterms without extra parentheses.

The following procedure accepts a single list of truth values (a single row) and the list of variables, and produces the minterm.

```

1 FormMinterm:=proc(row::list(truefalse), vars::list(symbol))
2   local mintermSeq, i;
3   uses Logic;
4   if nops(row) <> nops(vars) then
5     error "Incorrect number of variables";
6   end if;
7   mintermSeq := NULL;
8   for i from 1 to nops(row) do
9     if row[i] then
10       mintermSeq := mintermSeq, vars[i];
11     else
12       mintermSeq := mintermSeq, &not(vars[i]);
13     end if;
14   end do;
15   return &and(mintermSeq);
16 end proc:
```

This procedure, applied to a member of the **exampleTable**, produces the corresponding minterm.

> *FormMinterm* ([true, true, false], [x, y, z])
 $x \wedge y \wedge (\neg z)$ (12.50)

To create the Boolean expression for the function, all that remains is to form the disjunction of the minterms produced by **FormMinterm**. The procedure below accepts the table representation and list of variables as arguments. It first checks to see if it was passed the empty set, and if so, returns the expression **false**. Similar to **FormMinterm**, it initializes a sequence to **NULL**. For each element of the table representation, it calls **FormMinterm** and adds the output from that procedure to the sequence. At the end, the **&or** operator is applied to the sequence of minterms.

```

1 BooleanFromTable :=proc(T::set(list(truefalse)), V::list(symbol))
2   local mtSeq, row, mt;
3   uses Logic;
4   if T = {} then
5     return false;
6   end if;
7   mtSeq := NULL;
8   for row in T do
9     mt := FormMinterm(row, V);
10    mtSeq := mtSeq, mt;
11  end do;
12  return &or(mtSeq);
13 end proc:
```

Note that the structured type **set(list(truefalse))** ensures that the first argument is a **set** whose members are each a **list** with members of type **truefalse** (namely **true** or **false**).

Applying the procedure to our example table produces the desired Boolean expression.

```
> BooleanFromTable(exampleTable, exampleVariables)
((¬x) ∧ (¬y) ∧ z) ∨ ((¬x) ∧ y ∧ (¬z)) ∨ (x ∧ (¬y) ∧ (¬z))
∨ (x ∧ y ∧ (¬z))
```

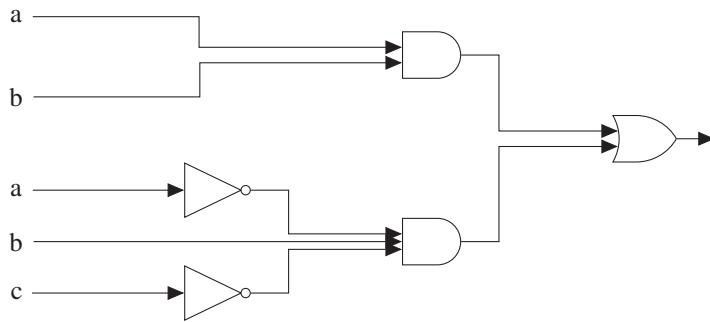
(12.51)

12.3 Logic Gates

In this section, we will use Maple to work with logic gates, particularly circuit diagrams. First, we will see how to use Maple to translate a circuit diagram into an expression using the **Logic** operators. Then, we will do the reverse and see how to transform a logical expression into a circuit diagram (modeled as a tree diagram).

Circuit Diagram to Logical Expression

Consider the circuit diagram shown below.



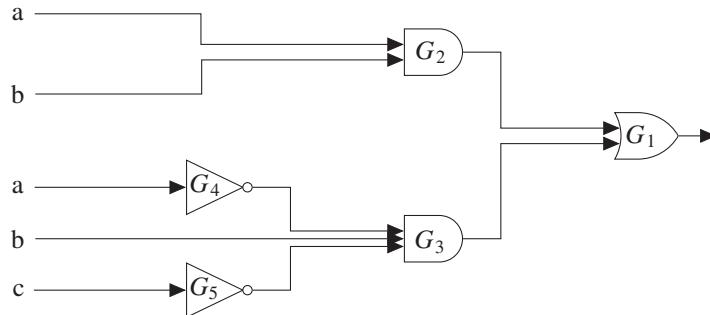
Our goal in this subsection is to use Maple to produce a logical expression for the output of this diagram.

To do this, we use the fact that the inert forms of the logical operators can be used in functional form. In particular, **&and** and **&or** can be used as functions applied to sequences of expressions, and **¬** can be applied to a single argument. For example, the following forms the disjunction of **x**, **y**, and **z**.

```
> &or(x,y,z)
x ∨ y ∨ z
```

(12.52)

We give each gate in the diagram a label. The specific names are not important. We chose to label the gates using the capital letter **G** with subscripts numbered from the right to the left.



Interpret the labels as names for both the gates themselves and for their outputs. Note that G_1 is the name for both the output of the final gate and also names the output of the circuit. First, we remove any existing assignments with **unassign**.

```
> unassign('G1', 'G2', 'G3', 'G4', 'G5', 'a', 'b', 'c')
```

The input of G_1 is the outputs from gates G_2 and G_3 . That is to say, $G_1 = G_2 \text{ OR } G_3$. We can write that in Maple.

```
> G1 := &or(G2, G3)
G1 := G2 ∨ G3
(12.53)
```

For each gate, do the same. Note that the order in which the gates are specified is irrelevant. The output G_2 is $a \text{ AND } b$.

```
> G2 := &and(a, b)
G2 := a ∧ b
(12.54)
```

The output of G_3 is the conjunction of G_4 , b , and G_5 .

```
> G3 := &and(G4, b, G5)
G3 := G4 ∧ b ∧ G5
(12.55)
```

And G_4 and G_5 are the results of inversion on a and c , respectively.

```
> G4 := &not(a)
G4 := ¬a
(12.56)
```

```
> G5 := &not(c)
G5 := ¬c
(12.57)
```

Once all of the gates have been specified, inspect the value for the final gate, G_1 .

```
> G1
(a ∧ b) ∨ ((¬a) ∧ b ∧ (¬c))
(12.58)
```

This tells us that the circuit's result in 0–1 form is $ab + \bar{a}b\bar{c}$.

The reason this works is that when we define the output of a gate in terms of unassigned names, such as **G2**, Maple accepts the definition. When **G2** is later assigned its own value and then the statement **G1** is executed, Maple resolves all assigned names into their definitions so that the expression for **G1** is in terms of unassigned names (**a**, **b**, and **c**) only.

Logical Expression to Circuit Diagram

We have just seen how to use Maple to transform a circuit diagram into a logical expression for the result of the circuit. Now we consider the reverse. Given a logical expression, such as that for **G1**, we will use Maple to transform the expression into a circuit diagram.

We will model a circuit diagram as an ordered rooted tree. While circuit diagrams are not necessarily ordered, making the tree ordered will allow for the possibility of including Boolean functions as subcircuits.

Vertices in the tree will correspond to gates in the circuit. One of the vertices is distinguished as the root, which will correspond to the final gate, whose output is the output of the circuit. Each vertex has a number of children vertices. The edge between a vertex and its child corresponds to the input to the gate. Each vertex other than the root has a parent, and the edge from the vertex to the parent corresponds to the output from the gate.

The assumption that a circuit can be modeled as a tree requires that the circuit satisfy the following properties. First, the circuit has only one output. Second, each gate has only one output. Third, there are no branches (e.g., an input cannot be used as input to more than one gate). Given that our goal is to begin with a logical expression and create a corresponding abstract circuit, these restrictions are of little importance. If we were actually building physical circuits, there would be efficiency concerns.

Recall that in Chapter 11, we wrote the procedure **InfixToTree** for converting an algebraic expression in terms of inert versions of the binary arithmetic operators into a tree representation. We will make use of the procedures from Chapter 11 here, and have included them in the package for this chapter as well. Thus, if you apply **with** to either “Chapter 11” or “Chapter 12,” the needed commands will be included.

Recall that the procedures written in each chapter of this manual are collected in packages. If you have not moved the library provided with the manual into Maple’s default library directory (you can obtain the directory by inspecting the value of **libname**), then you should tell Maple where to look for the manual’s library by executing a command like one of the two below.

```
> libname := libname, "/Users/danieljordan/DiscreteMath/"  
> libname := libname, FileTools[ParentDirectory]()  
Maplets[Utilities][GetFile]('title' = "Locate RosenMaplePackages.mla",  
'directory' = currentdir(homedir), filefilter = "mla",  
filterdescription = "Library Files"))
```

For the first of the commands above, you would need to replace the given directory path with a string appropriate to your system and location of the provided .mla file. The second will open a dialog that you can use to navigate to the directory. Note that these commands have been made nonexecutable so that they will not create errors or open dialogs if you decide to execute the entire worksheet. Once Maple is able to find the library file for the manual, the following will load the needed procedures and type definitions.

```
> with(Chapter12, VOrderComp, DrawORTree, NewExpressionTree,  
JoinTrees, InfixToTree) :
```

Please refer to the Introduction or Maple’s documentation if you need more instructions on how to use the packages on your system.

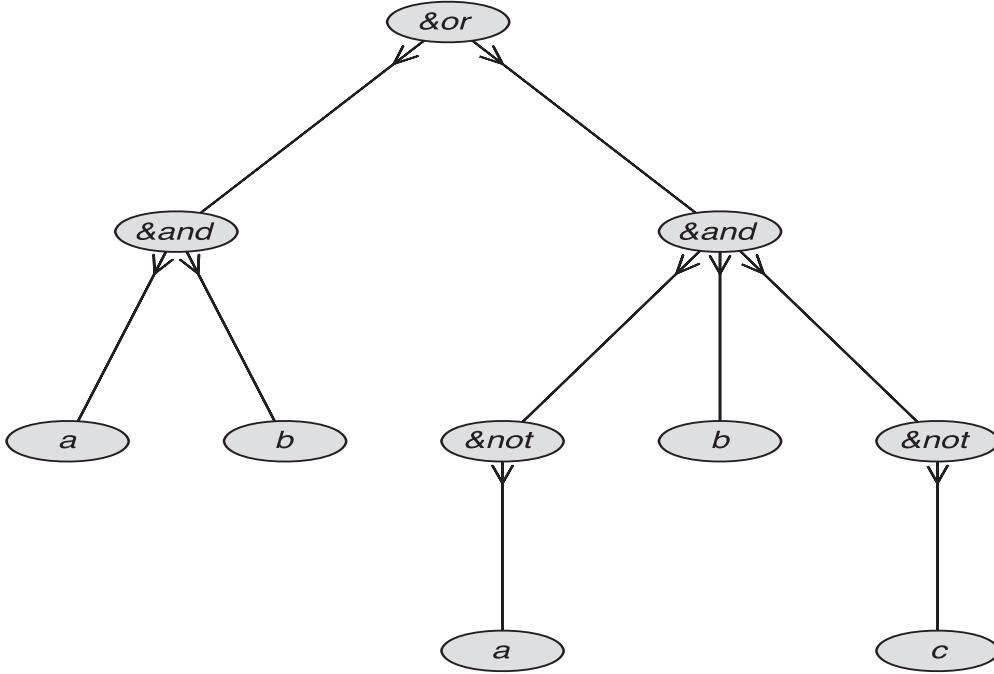
Observe what happens if we apply the **InfixToTree** command to the logical expression we obtained from the circuit diagram above.

> *InfixToTree (G1)*

Graph 1: a directed unweighted graph with 10 vertices and 9 arc(s)

(12.59)

> *DrawORTree (%)*



Observe that the original circuit diagram and the tree have the same structure. After reversing the arrows, rotating by 90°, and exchanging the symbols with the inert commands, the two are identical.

12.4 Minimization of Circuits

In this section, we will discuss the use of the **BooleanSimplify** command for minimizing circuits. Then we will create a brute force algorithm for handling don't care conditions. Finally, we will provide an implementation of the Quine–McCluskey method.

The BooleanSimplify Command

We described the **BooleanSimplify** command in the first section of this chapter. The command accepts only one argument, a Boolean expression.

For example, we apply **BooleanSimplify** to G1, the expression we obtained for the output of the circuit diagram at the beginning of Section 12.3.

> *BooleanSimplify (G1)*

(a \wedge b) \vee (b \wedge (\neg c))

(12.60)

The result indicates that $\neg a$ can be removed as an input to the second AND gate.

Note that the result of **BooleanSimplify** is guaranteed to be minimal. That is, it is not possible to reduce it further. It is not, however, guaranteed that the result is a minimum sum of prime

implicants. That is, while no simplification of the output from **BooleanSimplify** is possible, it may be the case that there is a simpler expression equivalent to the original input.

The reason for this lies in the final step of the Quine–McCluskey method. Once the essential prime implicants have been identified, you are left with prime implicants that are not essential and you must identify the best choice of those prime implicants that will complete the cover. To find the minimum expression, you use a backtracking approach (a depth-first search). Unfortunately, this requires exponential time.

The alternative is to use a heuristic approach, choosing the prime implicants that cover the most minterms. This is considerably more efficient, but does not guarantee that the resulting expression is the minimum. At the conclusion of this section, we will design an implementation of the Quine–McCluskey method using such a heuristic approach. Implementing a backtracking approach is left as an exercise.

Don't Care Conditions

Informally, a set of don't care conditions for a Boolean function F is a set of points in the domain of F whose images do not concern us.

If F is a function on n variables, then its domain is $\{true, false\}^n$. Let A be the subset of $\{true, false\}^n$ for which the values of F are specified. If we think of F as fully defined on this subset A , then we are interested in the family of all extensions of F to all of $\{true, false\}^n$. In other words, the set of all G defined on $\{true, false\}^n$ that agree with F on A . The goal is to choose the particular G that is “simplest.” That is, the G that has the smallest sum of products expansion.

We should pause to consider the size of this problem. If there are d don't care points, then there are 2^d possible extensions G . Considering every possible extension can become rather time consuming.

The procedure we write will make use of the **BooleanFromTable** procedure from Section 12.2. Recall that the **BooleanFromTable** procedure accepted a set consisting of those points for which the function returns true. The points are represented by lists of trues and falses. It also required a list of the names of the variables. **BooleanFromTable** returns the conjunctive normal form of the function that returns true on the specified points and false on all others.

In **DontCare**, we will loop through every possible extension G of the input function F . Specifically, the procedure will accept two sets of points. One representing those points for which the function must return true, and the second set of points representing the don't care conditions. It is understood that the function must return false on all points in neither set. **DontCare** will also accept a list of variable names.

Each extension G corresponds to a subset of the don't care conditions. We will use the **subsets** command, described in Section 6.1 of this manual. Recall that **subsets**, when applied to a set, returns a table with two elements. The **finished** element is a Boolean value set to true once all of the subsets of the given set have been listed. The **nextvalue** entry is a procedure that, when executed, produces the next subset.

For each subset of the don't care conditions, **DontCare** will apply **BooleanFromTable** to the union of the subset and the set of points for which F must be true. It will then apply **BooleanSimplify** to the result.

To determine the minimal expression, we compare expressions using the **length** command, which returns an integer representing the length of the expression. (This is a crude comparison, but it will suffice.) We use the standard approach of storing the simplest expression found so far, and replacing it each time a shorter expression is located. Note that the first set produced by **subsets** is always the empty set, so we initialize the temporary minimum to the function in which all don't care conditions are taken to be false.

```

1 DontCare := proc (T : :set (list (truefalse)) , DC : :set (list (truefalse)) ,
2   V : :list (symbol))
3   local minExpr, minLength, S, s, nextExpr;
4   uses Logic;
5   S := combinat [subsets] (DC);
6   s := S [nextvalue] ();
7   minExpr := BooleanSimplify (BooleanFromTable (T, V));
8   minLength := length (minExpr);
9   while not S [finished] do
10     s := S [nextvalue] ();
11     nextExpr := BooleanSimplify (BooleanFromTable (T union s, V));
12     if length (nextExpr) < minLength then
13       minExpr := nextExpr;
14       minLength := length (nextExpr);
15     end if;
16   end do;
17   return minExpr;
end proc;
```

Consider the Boolean function F defined by the following table of values, in which “d” in the final column indicates a don't care condition.

x	y	z	$F(x,y,z)$
<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>
<i>false</i>	<i>true</i>	<i>false</i>	<i>d</i>
<i>false</i>	<i>true</i>	<i>true</i>	<i>d</i>
<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>
<i>true</i>	<i>false</i>	<i>true</i>	<i>false</i>
<i>true</i>	<i>true</i>	<i>false</i>	<i>false</i>
<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>

The points that must evaluate to true are

$$\{[false, false, false], [true, false, false], [true, true, true]\},$$

and the don't care conditions are

$$\{[false, true, false], [false, true, true]\}.$$

We apply the procedure **DontCare** with these two sets as inputs along with the list of variables.

```
> DontCare ({[false,false,false], [true,false,false], [true,true,true]},  
 {[false,true,false], [false,true,true]} ,['x', 'y', 'z'])  
(y  $\wedge$  z)  $\vee$  (( $\neg$ y)  $\wedge$  ( $\neg$ z))
```

(12.61)

Before leaving don't care conditions, we should mention that the Quine–McCluskey method, which is the subject of the next subsection, provides a much more efficient solution than the procedure **DontCare**.

To take don't care conditions into account with the Quine–McCluskey method, you include them in the list of minterms that are used to generate prime implicants, but you do not include them in the list of minterms that need to be covered by the prime implicants. In terms of Example 9 of the text, don't care conditions appear in the first column of Table 3, but are omitted from the top row of Table 4.

Quine–McCluskey Method

We conclude with an implementation of the Quine–McCluskey method. This method is fairly involved and it will take considerable effort to implement it correctly.

It will be helpful to have an example that we can use to illustrate the method as we build the procedure. The expression we use for the example is

$$wx\bar{y}\bar{z} + w\bar{x}yz + w\bar{x}\bar{y}\bar{z} + w\bar{x}\bar{y}z + \bar{w}xyz + \bar{w}\bar{x}\bar{y}z + \bar{w}xy\bar{z} + \bar{w}\bar{x}yz + \bar{w}\bar{x}\bar{y}z + \bar{w}\bar{x}y\bar{z}.$$

We assign this to the name **F**.

```
> F := (w  $\wedge$  x  $\wedge$  ( $\neg$ y)  $\wedge$  ( $\neg$ z))  $\vee$   
(w  $\wedge$  ( $\neg$ x)  $\wedge$  y  $\wedge$  z)  $\vee$   
(w  $\wedge$  ( $\neg$ x)  $\wedge$  y  $\wedge$  ( $\neg$ z))  $\vee$   
(w  $\wedge$  ( $\neg$ x)  $\wedge$  ( $\neg$ y)  $\wedge$  ( $\neg$ z))  $\vee$   
( $\neg$ w  $\wedge$  x  $\wedge$  y  $\wedge$  z)  $\vee$   
( $\neg$ w  $\wedge$  x  $\wedge$  ( $\neg$ y)  $\wedge$  z)  $\vee$   
( $\neg$ w  $\wedge$  x  $\wedge$  y  $\wedge$  ( $\neg$ z))  $\vee$   
( $\neg$ w  $\wedge$  ( $\neg$ x)  $\wedge$  y  $\wedge$  z)  $\vee$   
( $\neg$ w  $\wedge$  ( $\neg$ x)  $\wedge$  ( $\neg$ y)  $\wedge$  z)  $\vee$   
( $\neg$ w  $\wedge$  ( $\neg$ x)  $\wedge$  y  $\wedge$  ( $\neg$ z)) :
```

Let us begin by (very) briefly outlining the approach. More details will be given as we proceed.

1. Transform the minterms into bit strings.
2. Group the bit strings by the number of 1s.
3. Combine bit strings that differ in exactly one location.
4. Repeat steps 2 and 3 until no additional combinations are possible.
5. Identify the prime implicants (those bit strings not involved in a simplification) and form the coverage table.
6. Identify the essential prime implicants and update the table.
7. Process the remaining prime implicants using a heuristic approach in order to achieve complete coverage.

Implementing this will require several different procedures that will come together to achieve the goal of minimizing the expression for F .

Modifying Arguments

Before we begin implementing the method, we take a moment to reiterate the use of **evaln** as a parameter modifier, which we first described in Section 11.2. This will be used later in the implementation to avoid the need to copy data structures that must be modified by a procedure.

Modifying a parameter with **evaln** means that, instead of sending a copy of the object to be worked on by the procedure, you send the name of the object. This allows the argument to be modified directly within the procedure. However, every time you want to access the value of the object, rather than the name, you must apply **eval** to the parameter.

The procedure below adds 3 to its argument, changing the value stored in the name it is passed.

```
1 add3 := proc(x::evaln(integer))
2     x := eval(x) + 3;
3 end proc:
```

Applying **add3** to a name that stores an integer will now alter the value stored in the name.

```
> two := 2
two := 2
```

(12.62)

```
> add3(two)
5
```

(12.63)

```
> two
5
```

(12.64)

Transforming Minterms into Bit Strings

The first task is to process the input. That is, F must be transformed into a list of bit strings. This is not strictly necessary, but it makes working with the minterms more convenient. We represent bit strings as lists of 0s and 1s.

We begin by creating a procedure to transform a single minterm into a bit string. We assume that the input to this procedure will be a properly formed minterm, that is, a conjunction of variables and negations of variables. We require that a list of variables be provided to the procedure, so that the bit string can be formed in the proper order.

Consider the following minterm, which is the fourth minterm in our example F .

```
> minterm := w &and &not(x) &and y &and &not(z)
minterm := ((w ∧ (¬x)) ∧ y) ∧ (¬z)
```

(12.65)

In order to transform this into the bit string [1, 0, 1, 0], we must first determine the variables and negations of variables that are conjoined. Our first goal, therefore, is to transform the minterm into the list [w , **not** x , y , **not** z].

For this, recall that the **op** command can be used to extract the operands of an expression. In this case, **op** will return the two operands of the final **and**.

```
> [op(minterm)]
[(w ∧ (¬x)) ∧ y, ¬z] (12.66)
```

In order to obtain a list of the conjoined variables and negations, we will repeatedly apply **op**.

We begin by initializing a list consisting of **minterm** as the only object and we set an index variable equal to 1.

```
> MTList := [minterm]
MTList := [(w ∧ (¬x)) ∧ y) ∧ (¬z)] (12.67)
```

```
> i := 1
i := 1 (12.68)
```

We create a while loop that will continue as long as the index variable is not greater than the length of **MTList**. Within the loop, we consider **MTList[i]**. Comparing **op(0,MTList[i])** to the operator '**&and**' will tell us whether or not the *i*th member of the list is a conjunction. If so, we apply **op** to it, using **subsop** to replace the **i** location in the list with **op(MTList[i])**. On the other hand, if **op(0,MTList[i])** is not '**&and**', then we increment **i**.

```
> while i ≤ nops(MTList) do
  if op(0, MTList[i]) = }&and} then
    MTList := subsop(i = op(MTList[i]), MTList)
  else
    i := i + 1
  end if
end do
```

This has transformed **MTList** into a list of the conjoined variables and negations of variables.

```
> MTList
[w, ¬x, y, ¬z] (12.69)
```

To complete the transformation into a bit string, we only need to check, for each variable, whether the variable or its negation is in the list. Recall that we will insist that the procedure be given the list of variables as an argument to maintain the proper order of the variables.

We first assign the list of variables to a name.

```
> variableList := [w, x, y, z]
variableList := [w, x, y, z] (12.70)
```

Now create a list, initialized to the proper length, for the bit string.

```
> Bitstring := [0 $nops(variableList)]
Bitstring := [0, 0, 0, 0] (12.71)
```

Finally, we use a for loop to check, for each variable, whether the variable is in **MTList**. If the variable is a member of **MTList**, then we change the bit to 1. Otherwise, we assume that the negation is in the minterm and we leave the value in the bit string as 0.

```
> for i from 1 to nops(variableList) do
    if variableList[i] in MTList then
        Bitstring[i] := 1
    end if
end do
```

This has created the bit string associated to **minterm**.

```
> Bitstring
[1, 0, 1, 0] (12.72)
```

We condense this process into a single procedure.

```
1 MTtoBitString := proc(minterm, variableList :: list(symbol))
2     local MTList, i, Bitstring;
3     uses Logic;
4     MTList := [minterm];
5     i := 1;
6     while i <= nops(MTList) do
7         if op(0, MTList[i]) = '&and' then
8             MTList := subsop(i=op(MTList[i]), MTList);
9         else
10            i := i + 1;
11        end if;
12    end do;
13    Bitstring := [0 $ nops(variableList)];
14    for i from 1 to nops(variableList) do
15        if variableList[i] in MTList then
16            Bitstring[i] := 1;
17        elif '&not'(variableList[i]) in MTList then
18            Bitstring[i] := 0;
19        else
20            error "Unrecognized object in MTList.";
21        end if;
22    end do;
23    return Bitstring;
24 end proc;
```

```
> MTtoBitString(minterm, [w, x, y, z])
[1, 0, 1, 0] (12.73)
```

Transforming the Original Expression into Bit Strings

Now that we have the means for transforming a single minterm into a bit string, we are ready to transform an expression in disjunctive normal form into a list of bit strings.

This works in nearly the same way as **MTtoBitString** did. Given an expression in disjunctive normal form, we break it into a list, **DNFList**, but this time, instead of looking for the operator to be ‘**&and**’, it must be ‘**&or**’. Once the list is formed, we apply **MTtoBitString** on each element of the list.

Here is the procedure. Notice that the first while loop is very similar to **MTtoBitString**, but after the while loop, we complete the procedure with an application of **map**. We use **variableList** as a third argument to **map**. When **map** is given more than two arguments, subsequent arguments are treated as additional arguments to the procedure. That is, **map(P,L,a)** applies the procedure **P** to **(L,a)** for each **L** in the list **L**. In this case, **MTtoBitString** requires that the list of variables be passed to it each time it is called.

```

1  DNFtoBitList := proc (dnfExpr, variableList :: list (symbol) )
2      local DNFList, i;
3      uses Logic;
4      DNFList := [dnfExpr];
5      i := 1;
6      while i <= nops (DNFList) do
7          if op (0, DNFList [i]) = '&or' then
8              DNFList := subsop (i=op (DNFList [i]), DNFList);
9          else
10             i := i + 1;
11         end if;
12     end do;
13     return map (MTtoBitString, DNFList, variableList);
14 end proc;
```

Apply this procedure to the example expression.

```

> Fbits := DNFtoBitList (F, [w, x, y, z])
Fbits := [[1, 1, 0, 0], [1, 0, 1, 1], [1, 0, 1, 0], [1, 0, 0, 0], [0, 1, 1, 1],
[0, 1, 0, 1], [0, 1, 0, 0], [0, 0, 1, 1], [0, 0, 0, 1], [0, 0, 0, 0]]
```

(12.74)

Transforming Bit Strings into Minterms

At the conclusion of the Quine–McCluskey process, we will want to display the result in disjunctive normal form. This will require that we turn bit strings back into minterms.

Note that since this procedure will be applied at the end of the process, it may be that some of the variables have been removed. We will be using the string “-” in a bit string to indicate the elimination of a variable.

This procedure will require the bit string and a list of variable names as its input. It operates in two stages. First, it processes the variable list based on the content of the bit string. We make a copy of the variable list and then modify it by applying **¬** when the entry in the bit string is 0 and by replacing the variable with “-” when that is the entry in the bit string.

```

> varList := [w, x, y, z]
varList := [w, x, y, z]
```

(12.75)

```

> bitstr := [0, 1, “-”, 0]
bitstr := [0, 1, “-”, 0]
```

(12.76)

```

> for i from 1 to nops(varList) do
    if bitstr[i] = 0 then
        varList[i] := &not(varList[i])
    elif bitstr[i] = “-” then
        varList[i] := “-”
    end if
end do

> varList
[¬w, x, “-”, ¬z] (12.77)

```

Once this processing has been done, we remove any occurrences of “-” by replacing them with **NULL**. The **subs** command accepts as its arguments a sequence of equations and a final expression. For example, **subs(x=a,expr)**. The result of this statement is that every occurrence of **x** in **expr** is replaced by **a**. With the equation **“-”=NULL** and the processed variable list as the arguments, **subs** will return the list with each **“-”** removed.

```

> varList := subs (“-” = NULL, varList)
varList := [¬w, x, ¬z] (12.78)

```

To form the conjunction, we simply apply **op** to extract the sequence of variables and negations from the enclosing list and then apply **&and**.

```

> &and (op (varList))
(¬w) ∧ x ∧ (¬z) (12.79)

```

We combine these steps into a procedure.

```

1 BitStringtoMT := proc (bitstring, variableList :: list (symbol) )
2   local varList, i;
3   uses Logic;
4   varList := variableList;
5   for i from 1 to nops (varList) do
6     if bitstring[i] = 0 then
7       varList[i] := &not (varList[i]);
8     elif bitstring[i] = “-” then
9       varList[i] := “-”;
10    end if;
11  end do;
12  varList := subs (“-”=NULL, varList);
13  return &and (op (varList));
14 end proc;

```

Applied to $[0, 1, 0, 1]$ and $[x, y, z, w]$, we see that **BitStringtoMT** reproduces the original minterm.

```

> BitStringtoMT([0, 1, 0, 1], [w, x, y, z])
(¬w) ∧ x ∧ (¬y) ∧ z (12.80)

```

And applied to $[0, 1, “-”, 1]$, it removes the y .

$$\begin{aligned} > \text{BitStringtoMT}([0, 1, “-”, 1], [w, x, y, z]) \\ & (\neg w) \wedge x \wedge z \end{aligned} \tag{12.81}$$

Initializing the Source Table

In order to form the coverage table in the second part of the method, we need to know which of the original minterms are covered by which of the prime implicants. Refer to Tables 3 and 6 in the text. Notice that each bit string in those tables is associated with either a single number, in the case of the original minterms, or lists of numbers, for the derived products.

We will store this information in a table whose indices are the bit strings and whose entries are sets of integers. Given the **Fbits** list, we initialize this table with the elements of **Fbits** as the indices. The corresponding entries will be the set consisting of the bit string's position in **Fbits**.

We will refer to this as the “coverage dictionary,” since it allows us to look up any bit string and determine all of the original minterms covered by it. The following procedure accepts the **Fbits** list as an argument and returns the coverage dictionary.

```

1  initCoverDict := proc (L : : list)
2    local coverDict, i;
3    coverDict := table();
4    for i from 1 to nops(L) do
5      coverDict[L[i]] := {i};
6    end do;
7    return coverDict;
8  end proc;
```

Applying this procedure to **Fbits** produces the initial coverage dictionary. In order to inspect the entries, we must apply **eval** to the name of the table.

$$\begin{aligned} > \text{coverageDict} := \text{initCoverDict}(\text{Fbits}) \\ & \text{coverageDict} := \text{coverDict} \end{aligned} \tag{12.82}$$

$$\begin{aligned} > \text{eval(coverageDict)} \\ & \text{table}([[0, 1, 0, 0] = \{7\}, [0, 0, 1, 1] = \{8\}, [0, 1, 0, 1] = \{6\}, \\ & [0, 1, 1, 1] = \{5\}, [1, 1, 0, 0] = \{1\}, [0, 0, 0, 1] = \{9\}, [1, 0, 1, 0] = \{3\}, \\ & [1, 0, 0, 0] = \{4\}, [0, 0, 0, 0] = \{10\}, [1, 0, 1, 1] = \{2\}]) \end{aligned} \tag{12.83}$$

Grouping by the Number of 1s

Step 2 in our outline is to group the bit strings by the number of 1s.

The reason for this step is to improve the efficiency of finding simplifications to make. Since two bit strings can be combined only when they are identical except for one location, the only possible combinations are when one bit string has n 1s and the other has $n - 1$.

After step 1 is concluded, we have a list of bit strings. That will be the starting point for the procedure we create for this step. The result of this step will be to turn the list of bit strings into a table

of sets of bit strings, which we call **groups**. Associated to index i in **groups** will be the set of all bit strings with i 1s.

For each member of **Fbits**, we need to count the number of 1s. The **Occurrences** function from the **ListTools** package can do this for us, as illustrated below. We will use **ListTools** fairly extensively, so we load it to use the short forms of its commands.

```
> with(ListTools):
> Occurrences(1, [1, 0, 1, 1, 0, 0, 1])
4
```

(12.84)

Observe that the first argument is the object being sought and the second is a list. The result is the number of occurrences of the first argument in the list. It is tempting to use the **add** command to add the bits in the list. We cannot do this, however, because after the first simplification, our bit strings will contain symbols that are not 1s or 0s.

To sort the members of **Fbits** into groups, based on the number of 1s, we will apply the **Classify** command, also in the **ListTools** package. This command requires as its first argument a function and a list as its second argument. It returns a **table** whose indices are the unique results of the function applied to the elements of the list, and, associated with each of those values is the set of members of the list which produce that value. For example, the following classifies a set of values based on the **floor** function.

```
> Classify(floor, [3, 5, 3.8, 6.2, 6.9, 3.02, 4.7, 5.5, π])
table([3 = {3, 3.02, 3.8, π}, 4 = {4.7}, 5 = {5, 5.5}, 6 = {6.2, 6.9}])
```

(12.85)

Here is the result of classifying **Fbits**.

```
> groups := Classify(bitstr → Occurrences(1, bitstr), Fbits)
groups := table([0 = {[0, 0, 0, 0]}, 1 = {[0, 0, 0, 1], [0, 1, 0, 0], [1, 0, 0, 0]}, 
2 = {[0, 0, 1, 1], [0, 1, 0, 1], [1, 0, 1, 0], [1, 1, 0, 0]}, 
3 = {[0, 1, 1, 1], [1, 0, 1, 1]}])
```

(12.86)

In practice, we will want to ensure that every possible index from 0 to the length of a bit string appears in the table. One way to do this is by using the **tablemerge** function applied to the table in which every possible index is associated with the empty set. When given two tables as the only arguments, the second table's entry is used when an index appears in both. For example, the following would add indices 4 and 5 to the **groups** table, although nothing is being modified by this command.

```
> tablemerge(table([seq(j = {}, j = 0 .. 5)]), groups)
table([0 = {[0, 0, 0, 0]}, 1 = {[0, 0, 0, 1], [0, 1, 0, 0], [1, 0, 0, 0]}, 
2 = {[0, 0, 1, 1], [0, 1, 0, 1], [1, 0, 1, 0], [1, 1, 0, 0]}, 
3 = {[0, 1, 1, 1], [1, 0, 1, 1]}, 4 = {}, 5 = {}])
```

(12.87)

We encapsulate this in a small procedure.

	sortGroups := proc (bitstringList)
	local groupTable, max, j;

```

3   uses ListTools;
4   groupTable := Classify(bitstr -> Occurrences(1, bitstr),
5     bitstringList);
6   max := nops(bitstringList[1]);
7   return tablemerge(table([seq(j={}, j=0..max)]), groupTable);
end proc:

```

Combining Bit Strings

Step 3 is to combine all of the bit strings that differ in exactly one location. We first write a procedure that takes as input two bit strings and either combines them if, in fact, they do differ in exactly one location, or returns false if they do not.

This procedure needs to do two tasks. First, it has to check to see whether or not the two bit strings differ in more than one location. Second, it needs to combine them if they are allowed to be combined.

Combining two bit strings is easy, provided we know the one location in which they differ. For example,

```
> bit1 := [1, "-", 0, 1, 1]
bit1 := [1, "-", 0, 1, 1] (12.88)
```

```
> bit2 := [1, "-", 0, 0, 1]
bit2 := [1, "-", 0, 0, 1] (12.89)
```

You can see that these are identical except in position 4.

To merge them, we take either one and replace position 4 with “-”.

```
> subsop(4 = "-", bit1)
[1, "-", 0, "-", 1] (12.90)
```

We determine that they differ only in position 4 as follows. Begin by initializing a name **pos**, for position, to 0. This will hold the position at which the difference occurs. Setting it to 0 indicates that we have not found a difference.

Now use a for loop to compare each pair of entries in **bit1** and **bit2**. If we find a difference, check the value of **pos**. If **pos** is 0, then we know that this is the first time a difference was found and we set **pos** to the position of the difference. If **pos** is not 0, however, then we know that this is the second time a difference was found. In this case, the bit strings cannot be merged and we return false. If the loop completes without having returned false, then the two bit strings can be merged at position **pos**.

Here is the procedure.

```

1 MergeBitstrings := proc(bit1::list, bit2::list)
2   local i, pos;
3   pos := 0;
4   for i from 1 to nops(bit1) do
5     if bit1[i] <> bit2[i] then

```

```

6      if pos = 0 then
7          pos := i;
8      else
9          return false;
10     end if;
11    end if;
12 end do;
13 return subsop (pos="-", bit1);
14 end proc:
```

We see that it works correctly on our two example bit strings.

```
> MergeBitstrings (bit1, bit2)
[1, "-", 0, "-", 1] (12.91)
```

Searching for Combinations to Make

The **MergeBitstrings** procedure will do the work of checking to see if bit strings can be merged and returning the result if they can. However, we need to give **MergeBitstrings** the bit strings to test.

Recall that, in our example, we have successfully grouped the minterms by the number of 1s they contain.

```
> eval(groups)
table ([0 = {[0, 0, 0, 0]}, 1 = {[0, 0, 0, 1], [0, 1, 0, 0], [1, 0, 0, 0]}, 
       2 = {[0, 0, 1, 1], [0, 1, 0, 1], [1, 0, 1, 0], [1, 1, 0, 0]}, 
       3 = {[0, 1, 1, 1], [1, 0, 1, 1]}]) (12.92)
```

Here, we will produce a list containing all the bit strings formed by merging two members of **groups**. Since there may be multiple ways to obtain the same bit string, we store these as a set. We initialize to the empty set.

```
> Fbits1 := {}
Fbits1 := ∅ (12.93)
```

Also recall that it is only possible to merge bit strings when one has n 1s and one has $n - 1$ 1s. This suggests a for loop with **n** ranging from 1 to the maximum index in **groups**. Within the body of the for loop, we will consider the sets with $n - 1$ 1s and the set with n 1s.

The loop is structured as follows:

```
> for n to max(indices(groups)) do
    A := groups[n];
    B := groups[n - 1]
end do :
```

After **A** and **B** have been defined, we need to compare every possible pair. We use two more for loops, one for each member of **A** and one for each member of **B**. Within the inner for loop, we use **MergeBitstrings** and store the result. If it is not false, we add it to the new list of bit strings, **Fbits1**.

```

> for n from 1 to max(indices(groups)) do
  A := groups[n];
  B := groups[n - 1]
  for a in A do
    for b in B do
      m := MergeBitstrings(a, b);
      if m ≠ false then
        Fbits1 := Fbits1 union {m};
      end if
    end do
  end do
end do;

> Fbits1 := [op(Fbits1)]
Fbits1 := [[0, 0, 0, “-”], [0, 0, “-”, 1], [0, 1, 0, “-”], [0, 1, “-”, 1],
[0, “-”, 0, 0], [0, “-”, 0, 1], [0, “-”, 1, 1], [1, 0, 1, “-”], [1, 0, “-”, 0],
[1, “-”, 0, 0], [“-”, 0, 0, 0], [“-”, 0, 1, 1], [“-”, 1, 0, 0]] (12.94)

```

This is close to the procedure we want, but we need to think ahead a bit. Recall from the description of the Quine–McCluskey process in the text that, in order to proceed with the second half of the method, we need to know which of the bit strings are prime implicants. That is, which bit strings are never used in a simplification.

We will track which bit strings are used as follows. Before the first loop, we create a set consisting of all of the bit strings in groups. We do this using the functional ‘union’ applied to the **entries** of the **groups** table. Note that use of the **nolist** option; by default, **entries** wraps each entry in a list.

```

> }union} (entries(groups, nolist))
{[0, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 1], [0, 1, 0, 0], [0, 1, 0, 1], [0, 1, 1, 1],
[1, 0, 0, 0], [1, 0, 1, 0], [1, 0, 1, 1], [1, 1, 0, 0]} (12.95)

```

Then, each time **MergeBitstrings** returns true, we remove the pair of bit strings from this set, using the **minus** set operator.

The procedure will return the sequence consisting of the next level of bit strings and the prime implicants from this stage. Here is our second attempt at the procedure.

```

1 NextBitListtry2 := proc(lastgroups)
2   local nextL, primeImps, n, A, B, a, b, m;
3   nextL := {};
4   primeImps := ‘union’(entries(lastgroups, nolist));
5   for n from 1 to max(indices(lastgroups)) do
6     A := lastgroups[n];
7     B := lastgroups[n-1];
8     for a in A do
9       for b in B do
10         m := MergeBitstrings(a, b);
11         if m <> false then
12           nextL := nextL union {m};

```

```

13      primeImps := primeImps minus {a, b} ;
14  end if;
15  end do;
16  end do;
17 end do;
18 nextL := [op(nextL)];
19 return nextL, primeImps;
20 end proc;

```

This still is not sufficient, however, because we also need to update the coverage dictionary as we create new bit strings. Recall that “coverage dictionary” is the name we gave to the table that records, for each bit string, which of the original minterms are covered by that bit string. The coverage dictionary was initialized with the bit strings formed from the minterms.

```

> eval(coverageDict)
table ([[0, 1, 0, 0] = {7}, [0, 0, 1, 1] = {8}, [0, 1, 0, 1] = {6},
       [0, 1, 1, 1] = {5}, [1, 1, 0, 0] = {1}, [0, 0, 0, 1] = {9},
       [1, 0, 1, 0] = {3}, [1, 0, 0, 0] = {4}, [0, 0, 0, 0] = {10},
       [1, 0, 1, 1] = {2}])

```

(12.96)

Within the **NextBitList** procedure, we need to update the coverage dictionary. We will make the dictionary a parameter. Note that it will not be necessary to return the updated dictionary, nor is it necessary to make a copy of the parameter. This is because, as we mentioned in Section 2.6 of this manual, tables are “reference types,” so unlike most arguments, they are altered by the procedure.

We update the dictionary within the **m <> false** if statement. When we form a new bit string **m**, we obtain the set of minterms it covers by taking the union of the sets of minterms covered by the two bit strings that were merged. That is,

coverDict[m] := coverDict[a] union coverDict[b];

Note that bit strings formed beyond the first step are typically generated multiple times. However, each time they are generated, they always cover the same set of original minterms.

Here is the final version of **NextBitList**.

```

1 NextBitList := proc(lastgroups, coverDict)
2   local nextL, primeImps, n, A, B, a, b, m;
3   nextL := {};
4   primeImps := 'union'(entries(lastgroups, nolist));
5   for n from 1 to max(indices(lastgroups)) do
6     A := lastgroups[n];
7     B := lastgroups[n-1];
8     for a in A do
9       for b in B do
10        m := MergeBitstrings(a, b);
11        if m <> false then
12          nextL := nextL union {m};
13          primeImps := primeImps minus {a, b};

```

```

14      coverDict[m] := coverDict[a] union coverDict[b];
15      end if;
16      end do;
17      end do;
18  end do;
19  nextL := [op(nextL)];
20  return nextL, primeImps;
21 end proc;

```

We apply it to **groups** to obtain **Fbits1** and **primes1**.

```

> Fbits1, primes1 := NextBitList(groups, coverageDict)
Fbits1, primes1 := [[0,0,0,"-"], [0,0,"-",1], [0,1,0,"-"],
[0,1,"-",1], [0,"-",0,0], [0,"-",0,1], [0,"-",1,1], [1,0,1,"-"],
[1,0,"-",0], [1,"-",0,0], ["-",0,0,0], ["-",0,1,1],
["-",1,0,0]], Ø
(12.97)

```

We see that there are

```

> nops(Fbits1)
13
(12.98)

```

bit strings in the second level, but no prime implicants coming from the first pass.

```

> nops(primes1)
0
(12.99)

```

In addition, almost as a side effect, the procedure has updated **coverageDict**.

```

> eval(coverageDict)
table([[0,1,0,0] = {7}, [0,"-",0,1] = {6,9}, [0,0,1,1] = {8},
[0,1,0,1] = {6}, [0,"-",0,0] = {7,10}, [1,0,"-",0] = {3,4},
["-",0,0,0] = {4,10}, ["-",0,1,1] = {2,8}, [0,0,"-",1] = {8,9},
[0,1,1,1] = {5}, [1,1,0,0] = {1}, [0,0,0,1] = {9},
[0,0,0,"-"] = {9,10}, [0,"-",1,1] = {5,8}, [1,"-",0,0] = {1,4},
[1,0,1,0] = {3}, [1,0,1,"-"] = {2,3}, ["-",1,0,0] = {1,7},
[0,1,0,"-"] = {6,7}, [1,0,0,0] = {4}, [0,0,0,0] = {10},
[0,1,"-",1] = {5,6}, [1,0,1,1] = {2}])
(12.100)

```

Repeating

Step 4 is to repeat steps 2 and 3.

The **Fbits1** list takes the place of **Fbits**. We apply **sortGroups** to produce **groups1**.

```

> groups1 := sortGroups(Fbits1)
groups1 := table([0 = {[0,0,0,"-"], [0,"-",0,0], ["-",0,0,0]} ,
1 = {[0,0,"-",1], [0,1,0,"-"], [0,"-",0,1], [1,0,"-",0],
[1,"-",0,0], ["-",1,0,0]}, 2 = {[0,1,"-",1], [0,"-",1,1],
[1,0,1,"-"], ["-",0,1,1]}, 3 = {}, 4 = {}])
(12.101)

```

Then applying **NextBitList** to **groups1** produces **Fbits2** and **primes2**.

```
> Fbits2, primes2 := NextBitList(groups1, coverageDict)
Fbits2, primes2 := [[0, “–”, 0, “–”], [0, “–”, “–”, 1], [“–”, “–”, 0, 0]],
{[1, 0, 1, “–”], [1, 0, “–”, 0], [“–”, 0, 1, 1]} (12.102)
```

We see that we have found three prime implicants. The coverage dictionary was further expanded to include the new bit strings.

```
> eval(coverageDict)
table([[0, 1, 0, 0] = {7}, [0, “–”, 0, 1] = {6, 9},
      [“–”, “–”, 0, 0] = {1, 4, 7, 10}, [0, 0, 1, 1] = {8}, [0, 1, 0, 1] = {6},
      [0, “–”, 0, 0] = {7, 10}, [1, 0, “–”, 0] = {3, 4}, [“–”, 0, 0, 0] = {4, 10},
      [“–”, 0, 1, 1] = {2, 8}, [0, 0, “–”, 1] = {8, 9}, [0, 1, 1, 1] = {5},
      [1, 1, 0, 0] = {1}, [0, 0, 0, 1] = {9}, [0, 0, 0, “–”] = {9, 10},
      [0, “–”, “–”, 1] = {5, 6, 8, 9}, [0, “–”, 1, 1] = {5, 8},
      [1, “–”, 0, 0] = {1, 4}, [1, 0, 1, 0] = {3}, [1, 0, 1, “–”] = {2, 3},
      [“–”, 1, 0, 0] = {1, 7}, [0, “–”, 0, “–”] = {6, 7, 9, 10},
      [0, 1, 0, “–”] = {6, 7}, [1, 0, 0, 0] = {4}, [0, 0, 0, 0] = {10},
      [0, 1, “–”, 1] = {5, 6}, [1, 0, 1, 1] = {2}]) (12.103)
```

Do the same thing again with **Fbits2**.

```
> groups2 := sortGroups(Fbits2)
groups2 := table([0 = {[0, “–”, 0, “–”], [“–”, “–”, 0, 0]}, 1 = {[0, “–”, “–”, 1]}, 2 = {}, 3 = {}, 4 = {}]) (12.104)
```

```
> Fbits3, primes3 := NextBitList(groups2, coverageDict)
Fbits3, primes3 := [], {[0, “–”, 0, “–”], [0, “–”, “–”, 1], [“–”, “–”, 0, 0]} (12.105)
```

This time, **Fbits3** was empty, which indicates that no more merging is possible and all prime implicants have been found.

This part of the process concludes by forming the list of all the prime implicants.

```
> allprimeImps := [op(primes1 union primes2 union primes3)]
allprimeImps := {[0, “–”, 0, “–”], [0, “–”, “–”, 1], [1, 0, 1, “–”],
[1, 0, “–”, 0], [“–”, 0, 1, 1], [“–”, “–”, 0, 0]} (12.106)
```

Forming the Coverage Table

Now that we have identified all of the prime implicants, we will use the coverage dictionary to create the coverage table.

Take a look at the final state of the coverage dictionary.

```
> eval(coverageDict)
```

```
table ([[0, 1, 0, 0] = {7}, [0, “–”, 0, 1] = {6, 9},
      [“–”, “–”, 0, 0] = {1, 4, 7, 10}, [0, 0, 1, 1] = {8}, [0, 1, 0, 1] = {6},
      [0, “–”, 0, 0] = {7, 10}, [1, 0, “–”, 0] = {3, 4}, [“–”, 0, 0, 0] = {4, 10},
      [“–”, 0, 1, 1] = {2, 8}, [0, 0, “–”, 1] = {8, 9}, [0, 1, 1, 1] = {5},
      [1, 1, 0, 0] = {1}, [0, 0, 0, 1] = {9}, [0, 0, 0, “–”] = {9, 10},
      [0, “–”, “–”, 1] = {5, 6, 8, 9}, [0, “–”, 1, 1] = {5, 8},
      [1, “–”, 0, 0] = {1, 4}, [1, 0, 1, 0] = {3}, [1, 0, 1, “–”] = {2, 3},
      [“–”, 1, 0, 0] = {1, 7}, [0, “–”, 0, “–”] = {6, 7, 9, 10},
      [0, 1, 0, “–”] = {6, 7}, [1, 0, 0, 0] = {4}, [0, 0, 0, 0] = {10},
      [0, 1, “–”, 1] = {5, 6}, [1, 0, 1, 1] = {2}])
```

(12.107)

Each bit string, and in particular each prime implicant, is an index in this table. The corresponding entry is the set of integers which are the indices to the original minterms in **Fbits**. Thus, to determine which of the original minterms are covered by each prime implicant, we look it up in the table.

We will model the coverage table as a matrix. Each row corresponds to a prime implicant, so there will be

```
> nops(allprimeImps)
6
```

(12.108)

rows. Each column corresponds to a minterm, so there are

```
> nops(Fbits)
10
```

(12.109)

columns. The entries in the matrix will be 0s and 1s with 1 in position (i, j) indicating that the prime implicant at position i in **allprimeImps** covers the minterm at position j in **Fbits**.

Recall that if the **Matrix** command is given two integers as its only arguments, it will create the matrix whose size is specified by the integers and has all 0 entries.

```
> Matrix(nops(allprimeImps), nops(Fbits))

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(12.110)

```

To enter 1s in the appropriate positions, we loop over the rows, considering each prime implicant in turn. For each prime implicant, we look up its entry in the coverage dictionary to obtain the set of minterms it covers. For each of those minterms, we place a 1 in the matrix.

The following procedure initializes the coverage table.

```

1 initCoverMatrix := proc (minterms, primeImps, coverDict)
2   local M, i, C, j;
3   M := Matrix(nops(primeImps), nops(minterms));
4   for i from 1 to nops(primeImps) do
5     C := coverDict [primeImps [i]];
6     for j in C do
7       M[i, j] := 1;
8     end do;
9   end do;
10  return M;
11 end proc;

```

Applied to our example, this produces the following coverage table.

> *coverageTable* := *initCoverMatrix*(*Fbits*, *allprimeImps*, *coverageDict*)

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

(12.111)

Manipulating the Matrix

Once the coverage table is set up, we move to steps 6 and 7, determining which prime implicants to include in the minimal expression. In step 6, we identify the essential prime implicants and in step 7 we identify which of the nonessential prime implicants we will include. We will see how to identify the prime implicants to use in a moment.

To aid in performing both steps 6 and 7, we will be manipulating the coverage table. Once we have decided to include a particular prime implicant in the minimal expression, we can take three actions.

First, record the decision by adding the prime implicant to a new list, say **minBits**, the list of bit strings to be included in the minimal expression.

Second, delete that prime implicant's row from the coverage table and delete the columns corresponding to the minterms it covered. We know the prime implicant will be in the expression, and, thus the minterms it covers are satisfied. Hence, there is no longer any need to keep track of that information.

Third, delete the prime implicant and the minterms it covers from the lists storing them (**Fbits** and **allprimeImps**). This is to ensure that the indices of **Fbits** and **allprimeImps** continue to match the row and column numbers of the matrix.

We write a procedure that implements these actions. Its input will be the index to the prime implicant that has been chosen. It will also accept the names of the coverage matrix, the list of minterms, and the list of prime implicants. All of these will be modified in the procedure (refer to

Modifying Arguments above). The procedure will return the bit string of the prime implicant that was chosen.

Our procedure will be called **UpdateCT**, for “update coverage table.” The **minBits** list, the list of chosen prime implicants, will be updated via the return value. This accomplishes the first task for this procedure.

Second, we must delete the row corresponding to the chosen prime implicant and the columns corresponding to the minterms covered by that implicant. Suppose, in our example, that we have decided to include the fourth prime implicant in the final result. This is

```
> allprimeImps[4]
[1, 0, “–”, 0] (12.112)
```

From **coverageTable**, we need to remove row 4 (since this corresponds to the prime implicant). We also need to remove the columns corresponding to the minterms covered by this prime implicant. To determine which columns are to be removed, we find the locations of the 1s in the row of the matrix.

To determine the locations of the 1s, we loop over the columns checking each position in row 4 to see if it is 1 or not. We use the **ColumnDimension** command from the **LinearAlgebra** package to determine the number of columns.

```
> with(LinearAlgebra):
> covered := {}
covered := Ø (12.113)
```

```
> for i to ColumnDimension(coverageTable) do
  if coverageTable[4, i] = 1 then
    covered := covered union {i}
  end if
end do:
covered
{3, 4} (12.114)
```

We now know that we need to remove row 4 and columns 3 and 4. To do this, we use a complicated selection.

We have seen that ranges can be used to select from lists. The same is true for matrices. For example, we can obtain the first three rows of this matrix as follows.

```
> coverageTable[1 .. 3, 1 .. -1]

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} (12.115)$$

```

The first range indicates rows 1 through 3, the second that we want all the columns, from the first to the last.

You can also give lists of the rows instead of ranges.

$$> \text{coverageTable}[[1, 2, 3], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (12.116)$$

In our example, we want all of the rows except for the fourth, which we can obtain from the pair of ranges **1..3** and **5..6**. More generally, if **newPI** is the index of the new prime implicant to be included in the minimal expression, we would use **1..(newPI-1)** and **(newPI+1)..-1**.

$$> \text{row4} := 4$$

$$\text{row4} := 4 \quad (12.117)$$

$$> \text{coverageTable}[[1 .. (\text{row4} - 1), (\text{row4} + 1) .. -1], [1 .. -1]]$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \quad (12.118)$$

For the columns we will take a different approach. Begin with the set of all column indexes.

$$> \text{colList} := \{\$1 .. \text{ColumnDimension}(\text{coverageTable})\}$$

$$\text{colList} := \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \quad (12.119)$$

Now remove from this set the columns that are to be removed by subtracting the **covered** set.

$$> \text{colList} := \text{colList} \text{ minus } \text{covered}$$

$$\text{colList} := \{1, 2, 5, 6, 7, 8, 9, 10\} \quad (12.120)$$

Then turn it into a list.

$$> \text{colList} := [\text{op}(\text{colList})]$$

$$\text{colList} := [1, 2, 5, 6, 7, 8, 9, 10] \quad (12.121)$$

Using this to select the columns, we obtain the desired matrix.

$$> \text{coverageTable}[[1 .. (\text{row4} - 1), (\text{row4} + 1) .. -1], \text{colList}]$$

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \quad (12.122)$$

The reader is encouraged to compare this to the original matrix. Note that in this example, we have not actually modified the table.

The last tasks are to remove the selected prime implicant from the list of prime implicants, and remove the covered minterms from **Fbits**. We use **subsop** to remove elements from lists, as usual. For instance, the following removes the fourth prime implicant.

```
> subsop(4 = NULL, allprimeImps)
[[0, "−", 0, "−"], [0, "−", "−", 1], [1, 0, 1, "−"], ["−", 0, 1, 1],
 ["−", "−", 0, 0]]
```

(12.123)

When removing the minterms, we remove them in the reverse order. For instance, to remove the minterms in locations 3 and 4, we first remove the minterm in position 4 and then the minterm in position 3. Otherwise, if we remove the minterm in location 3 first, then all the other minterms shift location by 1. That is, the minterm previously in location 4 is now in location 3. By removing them in reverse order, this is not a concern.

Here is the procedure. Remember that since we are using the **evaln** parameter option in order to make parameters modifiable, we must use **eval** when we need the values of those objects.

```

1 UpdateCT := proc(newPI, coverTable::evaln, minterms::evaln,
2   primeImps::evaln)
3   local newPIbits, numcols, covered, i, colList;
4   newPIbits := eval(primeImps)[newPI];
5   numcols := LinearAlgebra[ColumnDimension](eval(coverTable));
6   covered := {};
7   for i from 1 to numcols do
8     if eval(coverTable)[newPI, i] = 1 then
9       covered := covered union {i};
10    end if;
11  end do;
12  colList := [op({$1..numcols} minus covered)];
13  coverTable :=
14    eval(coverTable)[[1..(newPI-1), (newPI+1)..-1], colList];
15  primeImps := subsop(newPI=NULL, eval(primeImps));
16  for i from nops(covered) to 1 by -1 do
17    minterms := subsop(covered[i]=NULL, eval(minterms));
18  end do;
  return newPIbits;
end proc;
```

Finding Essential Prime Implicants

Next we write a procedure to identify the essential prime implicants. Recall that a prime implicant is essential when it is the only prime implicant to cover some minterm. In terms of the coverage table, this is equivalent to the existence of a column with only one 1.

We will locate the essential prime implicants as follows. First, we initialize the set of essential prime implicants to the empty list.

We proceed in a manner similar to the **MergeBitstrings** procedure. We use a for loop to step through the columns of the coverage table. Within this loop, we initialize a name, **rowhas1**, to 0.

We then enter a second for loop to step through the entries in the columns. When a 1 entry has been found, we check **rowhas1**. If that name is 0, then it is assigned to the current row number. If it is not 0, then we have found a second 1 in the column and we assign **rowhas1** to -1 and use **break** to terminate the inner loop. After the inner loop, we test **rowhas1**. If it is positive, then we know that only one 1 was located in that column, and hence the row the solitary 1 was found in corresponds to an essential prime implicant. In this case, we add the row number (**rowhas1**) to **essentials**.

The following procedure implements this algorithm and returns the list of essential prime implicants.

```

1 FindEssentials := proc(coverTable)
2   local essentials, i, j, rowhas1;
3   essentials := {};
4   for i from 1 to LinearAlgebra[ColumnDimension](coverTable) do
5     rowhas1 := 0;
6     for j from 1 to LinearAlgebra[RowDimension](coverTable) do
7       if coverTable[j,i] = 1 then
8         if rowhas1 = 0 then
9           rowhas1 := j;
10        else
11          rowhas1 := -1;
12          break;
13        end if;
14      end if;
15    end do;
16    if rowhas1 > 0 then
17      essentials := essentials union {rowhas1};
18    end if;
19  end do;
20  return essentials;
21 end proc;
```

We use this to determine the essential prime implicants of our example.

```
> essentialPIs := FindEssentials(coverageTable)
essentialPIs := {2,6} (12.124)
```

Now that we have the essential prime implicants, we can initialize **minBits** and apply **UpdateCT** to the essential prime implicants. Once again, we loop through the list backwards.

```
> minBits := []
minBits := [] (12.125)
> for i from nops(essentialPIs) by -1 to 1 do
  minBits := [op(minBits), UpdateCT(essentialPIs[i], coverageTable,
  Fbits, allprimeImps)]
end do :
```

```
> minBits
[[“,“, 0, 0], [0,“,“, 1]]
```

(12.126)

```
> coverageTable

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

```

(12.127)

Completing the Coverage

Provided that the essential prime implicants did not completely cover the original minterms, we must complete the coverage with nonessential prime implicants. First, we ensure that the coverage is not complete by checking the column dimension.

```
> evalb(LinearAlgebra[ColumnDimension](coverageTable) > 0)
true
```

(12.128)

As we mentioned earlier, we will use a heuristic approach to find a minimal set of prime implicants rather than using an exhaustive search to determine the minimum. The heuristic we use will be to choose the prime implicant with the most extensive coverage of the remaining minterms.

To find such a prime implicant, we will do the following. First, initialize **maxCoverage** and **bestImp** both to 0. Then loop over each row of the (modified) coverage table. For each row, we will compute the sum of the entries. If this sum is greater than **maxCoverage**, then set **maxCoverage** to the sum and set **bestImp** to the row number. Once the loop is complete, **bestImp** will be the index to a row with maximum coverage and will be the next prime implicant added to the **minBits** list.

Here is the procedure that implements this strategy.

```

1  findBestImp := proc(coverTable)
2      local maxCoverage, bestImp, i, j, sum;
3      maxCoverage := 0;
4      bestImp := 0;
5      for i from 1 to LinearAlgebra[RowDimension](coverTable) do
6          sum := 0;
7          for j from 1 to LinearAlgebra[ColumnDimension](coverTable) do
8              sum := sum + coverTable[i, j];
9          end do;
10         if sum > maxCoverage then
11             maxCoverage := sum;
12             bestImp := i;
13         end if;
14     end do;
15     return bestImp;
16 end proc;
```

As long as the coverage table is not empty, we apply this procedure to it to obtain the next implicant. We add the implicant to the list **minBits** representing the minimal expression and update the coverage table using **updateCT**.

```
> while LinearAlgebra[ColumnDimension](coverageTable) > 0 do
    nextPI := findBestImp(coverageTable);
    minBits := [op(minBits), UpdateCT(nextPI, coverageTable, Fbits,
        allprimeImps)]
end do
nextPI := 2
minBits := [[“, “, 0, 0], [0, “, “, 1], [1, 0, 1, “]]
```

(12.129)

All that is left is to translate **minBits** back into a logical expression. This can be done using **BitStringtoMT** applied to each element of **minBits** with the **map** command and then combined into one expression with **&or**.

```
> &or (op (map (BitStringtoMT, minBits, [w, x, y, z])))
((¬y) ∧ (¬z)) ∨ ((¬w) ∧ z) ∨ (w ∧ (¬x) ∧ y)
```

(12.130)

Putting It All Together

Finally, we assemble the pieces into a single procedure, which accepts a logical expression in disjunctive normal form and a list of its variables. It returns a minimal equivalent expression.

```
1 QuineMcCluskey := proc (F, variables)
2   local Fbits, FbitsL, coverageDict, groups, primes, i,
3       allprimeImps, j, coverageTable, essentialPIs, minBits,
4       nextPI;
5   uses Logic;
6   Fbits := DNFtoBitList (F, variables);
7   coverageDict := initCoverDict (Fbits);
8   i := 0;
9   FbitsL[0] := Fbits;
10  while FbitsL[i] <> [] do
11    i := i + 1;
12    groups[i] := sortGroups (FbitsL[i-1]);
13    FbitsL[i], primes[i] :=
14      NextBitList (groups[i], coverageDict);
15  end do;
16  allprimeImps := {};
17  for j from 1 to i do
18    allprimeImps := allprimeImps union primes[j];
19  end do;
20  allprimeImps := [op(allprimeImps)];
21  coverageTable :=
22    initCoverMatrix (Fbits, allprimeImps, coverageDict);
23  essentialPIs := FindEssentials (coverageTable);
24  minBits := [];
25  for i from nops (essentialPIs) to 1 by -1 do
```

```

22      minBits := [op(minBits), UpdateCT(essentialPIs[i],
23                      coverageTable, Fbits, allprimeImps)];
24      end do;
25      while LinearAlgebra[ColumnDimension](coverageTable) > 0 do
26          nextPI := findBestImp(coverageTable);
27          minBits := [op(minBits),
28                      UpdateCT(nextPI, coverageTable, Fbits, allprimeImps)];
29      end do;
30      return &or(op(map(BitStringtoMT, minBits, variables)));
31  end proc:

```

Define **Ex10** to be the expression in Example 10 from Section 12.4 of the text.

```

> E10 := (w &and x &and y &and &not(z)) &or
  (w &and &not(x) &and y &and z) &or
  (w &and &not(x) &and y &and &not(z)) &or
  (&not w &and x &and y &and z) &or
  (&not w &and x &and &not(y) &and z) &or
  (&not w &and &not(x) &and y &and z) &or
  (&not w &and &not(x) &and &not(y) &and z)
E10 := (w &and x &and y &and &not(z)) &or (w &and &not(x) &and y &and z) &or
(w &and x &and &not(y) &and z) &or (w &and &not(x) &and &not(y) &and z) &or
(w &and &not(x) &and y &and &not(z)) &or (w &and x &and &not(y) &and &not(z))
(12.131)

```

```

> QuineMcCluskey(E10, [w, x, y, z])
(w &and y &and &not(z)) &or ((&not(w) &and z) &or (w &and &not(x) &and y)
(12.132)

```

Note that this is the first of the two answers given in the solution to Example 10.

Solutions to Computer Projects and Computations and Explorations Computer Projects 2

Construct a table listing the set of values of all 256 Boolean functions of degree three.

Solution: The Boolean functions of degree three are in one-to-one correspondence with the subsets of $\{\text{true}, \text{false}\}^3$. This is because each subset S of $\{\text{true}, \text{false}\}^3$ can be identified with the unique Boolean function of degree three which returns true on the members of S and false on all other inputs.

Thus, we begin by constructing the set $\{\text{true}, \text{false}\}^3$ and its power set.

To construct $\{\text{true}, \text{false}\}^3$, we will use the **cartprod** command from the **combinat** package. (Refer to Section 2.1 of this manual for details.) Like many of the commands for generating combinatorial objects, **cartprod** produces a list with indices **finished** and **nextvalue**. We use it to form the set **TF3**.

```

> TF3 := {}
TF3 := ∅
(12.133)

```

```

> TF3iterator := combinat[cartprod]([[true,false]$3]):
> while not TF3iterator[finished] do
    TF3 := TF3 union{TF3iterator[nextvalue]()}
end do:
> TF3
{[false,false,false], [false,false,true], [false,true,false],
 [false,true,true], [true,false,false], [true,false,true],
 [true,true,false], [true,true,true]}(12.134)
```

To produce the subsets of $\{true, false\}^3$, we use the **subsets** command from **combinat**. This command also produces a table with indices **finished** and **nextvalue**.

```
> TF3subsets := combinat[subsets](TF3):
```

Now we will create a list of all of the Boolean functions. The subsets that are produced by **TF3subsets** are each valid inputs to the **BooleanFromTable** procedure we wrote in Section 12.2. We also apply **BooleanSimplify**, in order to have simpler representations, before adding the expression to the list.

```

> allFunctions := []
allFunctions := [](12.135)
> while not TF3subsets[finished] do
    nextTF3subset := TF3subsets[nextvalue]();
    nextTF3function := BooleanFromTable(nextTF3subset, [x, y, z]);
    nextTF3function := BooleanSimplify(nextTF3function);
    allFunctions := [op(allFunctions), nextTF3function];
end do :
```

The **allFunctions** list is lengthy, so we display only a few members.

```
> nops(allFunctions)
256(12.136)
```

```
> allFunctions[1..10]
[false, ( $\neg x$ )  $\wedge$  ( $\neg y$ )  $\wedge$  ( $\neg z$ ),  $z \wedge (\neg x) \wedge (\neg y)$ ,  $y \wedge (\neg x) \wedge (\neg z)$ ,  $y \wedge z \wedge (\neg x)$ ,
  $x \wedge (\neg y) \wedge (\neg z)$ ,  $x \wedge z \wedge (\neg y)$ ,  $x \wedge y \wedge (\neg z)$ ,  $x \wedge y \wedge z$ ,
 ( $\neg x$ )  $\wedge$  ( $\neg y$ )](12.137)
```

To obtain the values of the functions, we will use the **TruthTable** command. Recall from the first section that **TruthTable** requires two arguments, a Boolean expression and a list of variables that appear in the expression. The result is a **DataFrame** object. Using the **output** option, we can specify the output to be a **table** with indices lists of truth values and corresponding entries the value of the expression at the index.

We form the list of the truth tables associated to each function in the **allFunctions** list with the **map** command. Note that we use the third argument **[x,y,z]** and fourth argument **output=table** in **map** so that this list of variables and the option are passed to **TruthTable** each time it is applied to an expression from **allFunctions**.

```
> allTables := map(TruthTable, allFunctions, [x,y,z], output = table) :
```

Each entry in **allTables** is a table storing the truth table.

```
> allTables[136]  
table([(true, false, false) = false, (false, false, true) = true,  
       (true, false, true) = true, (true, true, false) = true,  
       (true, true, true) = false, (false, true, true) = false,  
       (false, true, false) = true, (false, false, false) = false])  
(12.138)
```

The output below indicates that the value of the 136th function on $(true, false, true)$ is *true*.

```
> allTables[136][true, false, true]  
true  
(12.139)
```

To display the entire truth table, loop through the members of **TF3**, each of which corresponds to a row of the truth table. Then, loop through members of **allTables** and obtain the value of the corresponding function. We must apply **op** to the member of **TF3**, since the index to a truth table is a sequence not a list. For demonstration purposes, we restrict to the functions 100 through 105.

```
> for i from 1 to nops(TF3) do  
    row := TF3[i];  
    for j from 100 to 105 do  
        row := row, allTables[j][op(TF3[i])]  
    end do;  
    print(row);  
end do:  
[false, false, false], true, true, true, true, true, true  
[false, false, true], true, true, true, true, true, true  
[false, true, false], false, false, false, false, false, false  
[false, true, true], true, true, true, false, false, false  
[true, false, false], false, false, false, true, true, true  
[true, false, true], true, false, false, true, false, false  
[true, true, false], false, true, false, false, true, false  
[true, true, true], false, false, true, false, false, true  
(12.140)
```

The output above indicates that on the input $(false, false, false)$, all six Boolean functions associated to the integers 100 through 105 output *true*.

Computations and Explorations 6

Randomly generate 10 different Boolean expressions in four variables and determine the average number of steps required to minimize them using the Quine–McCluskey method.

Solution: To solve this problem, we need to find a way to generate random Boolean expressions and, then, we must find a method of examining the minimization process so that we can count the number of steps.

In the **Logic** package, the command **Random** produces a random Boolean expression. The only required argument is a set or list specifying the symbols to be used. For example, to produce a random Boolean expression on the symbols u , v , w , x , y , and z , you enter the following.

$$\begin{aligned}
 & > \text{Random}([u, v, w, x, y, z]) \\
 & (u \wedge v \wedge w \wedge y \wedge (\neg x) \wedge (\neg z)) \vee (u \wedge w \wedge x \wedge y \wedge z \wedge (\neg v)) \vee \\
 & (u \wedge w \wedge (\neg v) \wedge (\neg x) \wedge (\neg y) \wedge (\neg z)) \vee \\
 & (v \wedge w \wedge (\neg u) \wedge (\neg x) \wedge (\neg y) \wedge (\neg z)) \vee \\
 & (v \wedge y \wedge (\neg u) \wedge (\neg w) \wedge (\neg x) \wedge (\neg z)) \vee \\
 & (v \wedge (\neg u) \wedge (\neg w) \wedge (\neg x) \wedge (\neg y) \wedge (\neg z))
 \end{aligned} \tag{12.141}$$

The **Random** command also accepts a second optional argument: **form=CNF**, **form=DNF**, or **form=MOD2**, specifying the form of the expression produced. The default form is disjunctive normal form.

Having determined how to generate random expressions, we need to find a way to count the number of steps taken during the minimization process. There are (at least) three approaches we could take to this part of the problem.

The first is to measure the time taken to execute the procedure. We have done this many times before.

The second approach is to modify the procedure to count the number of times certain operations are called. For example, we may be interested in the number of times that the **UpdateCT** procedure is executed. In this case, we can alter **UpdateCT** to include a global variable that is incremented at the start of every execution.

```

1 UpdateCT := proc(newPI, coverTable::evaln, minterms::evaln,
2   primeImps::evaln)
3   local newPIbits, numcols, covered, i, collist;
4   global countUpdateCT;
5   countUpdateCT := countUpdateCT + 1;
6   newPIbits := eval(primeImps)[newPI];
7   numcols := LinearAlgebra[ColumnDimension](eval(coverTable));
8   covered := {};
9   for i from 1 to numcols do
10     if eval(coverTable)[newPI, i] = 1 then
11       covered := covered union {i};

```

```

11      end if ;
12  end do;
13  colList := [op({$1..numcols} minus covered)] ;
14  coverTable :=
15      eval(coverTable) [ [1..(newPI-1), (newPI+1)..-1], colList] ;
16  primeImps := subsop(newPI=NULL, eval(primeImps)) ;
17  for i from nops(covered) to 1 by -1 do
18      minterms := subsop(covered[i]=NULL, eval(minterms)) ;
19  end do;
20  return newPIbits;
end proc:
```

We must initialize the variable to 0.

```
> countUpdateCT := 0
countUpdateCT := 0
```

(12.143)

Now execute **QuineMcCluskey** on 100 random expressions.

```
> for i to 100 do
    randExp := Random([u, v, w, x, y, z]);
    QuineMcCluskey(randExp, [u, v, w, x, y, z])
end do :
```

The **countUpdateCT** variable will now store the number of times **UpdateCT** was called. Dividing by 10 gives us the average.

```
> countUpdateCT
      100.
      4.290000000
```

(12.144)

The third approach is to make use of Maple's debugging facilities. We used **trace** in earlier chapters to get information about the workings of a procedure. Here, we will use the **showstat** command.

If you apply **showstat** to the name of a procedure, it will display the definition of the procedure with line numbers added on the left hand side. An integer or a range of integers can be given as a second argument to narrow the display to the desired lines.

```
> showstat(QuineMcCluskey, 5..8)
QuineMcCluskey := proc(F, variables)
    local Fbits, FbitsL, coverageDict, groups, primes, i, allprimeImps,
j, coverageTable, essentialPIs, minBits, nextPI;
    ...
5 while FbitsL[i] <> [] do
6 i := i+1;
7 groups[i] := sortGroups(FbitsL[i-1]);
8 FbitsL[i], primes[i] := NextBitList(groups[i], coverageDict)
end do;
...
end proc
```

You can get more information by telling Maple to track the procedure. You do this by calling the **debugopts** command with argument **traceproc=** and then the name of the procedure.

```
> debugopts(traceproc = QuineMcCluskey)
```

Now we apply **QuineMcCluskey** to 1000 random expressions. We suppress the output as it is not needed here.

```
> for i to 1000 do
    randExp := Random([u, v, w, x, y, z]);
    QuineMcCluskey(randExp, [u, v, w, x, y, z])
end do :
```

If we call the **showstat** command again, the output gives us more information than it did before.

```
> showstat(QuineMcCluskey)
```

```
QuineMcCluskey := proc(F, variables)
local Fbits, FbitsL, coverageDict, groups, primes, i, allprimeImps,
j, coverageTable, essentialPIs, minBits, nextPI;
|Calls Seconds Words|
PROC | 1000 0.771 11315189 |
1 | 1000 0.131 1769082| Fbits := DNFToBitList(F,variables);
2 | 1000 0.007 361562| coverageDict := initCoverDict(Fbits);
3 | 1000 0.001 0| i := 0;
4 | 1000 0.002 273000| FbitsL[0] := Fbits;
5 | 1000 0.013 14080| while FbitsL[i] <> [] do
6 | 1816 0.003 0| i := i+1;
7 | 1816 0.276 5184118| groups[i] := sortGroups(FbitsL[i-1]);
8 | 1816 0.075 897695| FbitsL[i], primes[i] := NextBitList(groups[i], cove
do;
9 | 1000 0.000 0| allprimeImps := {};
10 | 1000 0.002 0| for j to i do
11 | 1816 0.010 31134| allprimeImps := allprimeImps union primes[j] end
do;
12 | 1000 0.001 4000| allprimeImps := [op(allprimeImps)];
13 | 1000 0.036 554955| coverageTable := initCoverMatrix(Fbits,allprimeI
14 | 1000 0.031 330035| essentialPIs := FindEssentials(coverageTable);
15 | 1000 0.003 0| minBits := [];
16 | 1000 0.014 0| for i from nops(essentialPIs) by -1 to 1 do
17 | 41870.096 961581| minBits := [op(minBits), UpdateCT(essentialPIs[i]
do;
18 | 1000 0.004 20100| while 0 < LinearAlgebra[LinearAlgebra:-
ColumnDimension](coverageTable) do
19 | 5 0.000 430| nextPI := findBestImp(coverageTable);
20 | 5 0.000 895| minBits := [op(minBits), UpdateCT(nextPI, coverageTable,
do;
21 | 1000 0.066 912522| return Logic:-'&or'(op(map(BitStringtoMT,minBit
proc
```

In addition to the line numbers, you see three columns labeled Calls, Seconds, and Words. Note that above line 1 is line PROC. This refers to information for the procedure as a whole.

The Calls column reports the number of times the line was executed. That the Calls column contains 10 in the PROC row indicates that the procedure was called 1000 times. Note that lines 17 and 20 are the lines containing the calls to **UpdateCT**. The sum of the values in the Count column is the number of times **UpdateCT** was called.

The Seconds column reports the amount of CPU time that was spent executing the line.

The Words column indicates the amount of memory that was allocated as a result of the statement.

Together, the three columns give you a considerable amount of information about the computational complexity, performance, and memory requirements of a procedure. In **QuineMcCluskey**, we see that the most time and memory are used in line 7, the call to **sortGroups**. However, the **UpdateCT** procedure was executed more often in line 17.

You can also have Maple store this information in a table for you by calling `debugopts` with argument **traceprocable**= and the name of the procedure.

```
> traceTable := debugopts(traceprocable = QuineMcCluskey)
traceTable := 
$$\begin{bmatrix} 1 .. 22 \times 1 .. 3 \text{ Array} \\ \text{Data Type: integer}_4 \\ \text{Storage: rectangular} \\ \text{Order: C_order} \end{bmatrix}$$
 (12.145)
```

The entries in this table correspond to the information displayed. The first row stores information about the procedure as a whole.

```
> traceTable[1, 1 .. 3]
[ 1000 771 11315 189 ] (12.146)
```

Information on specific lines is stored in the row one greater than the line number. For instance, the data related to the calls of **UpdateCT**, in line 17 and 20, are contained in the table at 18 and 21.

```
> traceTable[[18, 21], 1 .. 3]
[ 4187 96 961 581 ]
[ 5 0 895 ] (12.147)
```

Note that the second column, containing the time measurement, has been multiplied by 1000.

Executing

```
> debugopts(traceproc = QuineMcCluskey)
```

a second time clears the information that was stored and toggles the option to have Maple record the Calls, Seconds, and Words information back off.

Exercises

Exercise 1. Use Maple to verify De Morgan's Laws and the commutative and associative laws. (See Table 5 of Section 12.1.)

Exercise 2. Construct truth tables for each of the following pairs of Boolean expressions and decide whether they are logically equivalent.

- a) $a \rightarrow b$ and $b \rightarrow a$
- b) $a \rightarrow \bar{b}$ and $b \rightarrow \bar{a}$
- c) $a + bc$ and $(a + b + d)(a + c + d)$.

Exercise 3. Write a Maple procedure that translates Boolean expressions or functions written in terms of 0, 1, and the inert addition, multiplication, and negation operators into expressions in terms of the logical operators and values. (Hint: the implementation of the InfixToTree procedure in Chapter 11 of this manual may be helpful.)

Exercise 4. Write a Maple procedure that, given a Boolean function, represents this function using only the **&nand** operator.

Exercise 5. Use the procedure in the previous exercise to represent the following Boolean functions using only the **&nand** operator.

- a) $F(x, y, z) = xy + \bar{y}z$
- b) $G(x, y, z) = x + \bar{x}y + \bar{y}z$
- c) $H(x, y, z) = xyz + \bar{x}\bar{y}z$

Exercise 6. Write a Maple procedure that, given a Boolean function, represents this function using the **&nor** operator.

Exercise 7. Use the procedure in the previous exercise to represent the Boolean functions in Exercise 5 using only the **&nor** operator.

Exercise 8. Write a Maple procedure for determining the output of a threshold gate, given the values of n Boolean variables as input, and given the threshold value and a set of weights for the threshold gate. (See the Supplementary Exercises of Chapter 12 for information on threshold gates.)

Exercise 9. Develop a Maple procedure that, given a Boolean function in four variables, determines whether it is a threshold function, and if so, finds the appropriate threshold gate representing this function. (See the Supplementary Exercises of Chapter 12.)

Exercise 10. A Boolean expression e is called self dual if it is logically equivalent to its dual e^d . Write a Maple procedure to test whether a given expression is self dual.

Exercise 11. Determine, for each integer $n \in \{1, 2, 3, 4, 5, 6\}$, the total number of Boolean functions of n variables and the number of those functions that are self dual.

Exercise 12. Write a Maple procedure that, given a positive integer n , constructs a list of all Boolean functions of degree n . Use your procedure to find all Boolean functions of degree 4.

Exercise 13. Use **DontCare** to compute a minimal sum of products expansion for the Boolean functions with don't care conditions specified by the Karnaugh maps shown in Exercises 30 through 32 of Section 12.4.

Exercise 14. How can you change exactly one character in the definition of the procedure **DontCare** so that it returns the last expression of minimum length that it encounters that is equivalent to the input function? (As written now, it returns the first.)

Exercise 15. Use the procedure you wrote in Exercise 10 to write a Maple procedure to generate random Boolean expressions in 4 variables and stop when it has found one that is self dual. Run the program several times and time it. Find the average time. Repeat for Boolean expressions in 5 and 6 variables. Can you make any conjectures from this information?

Exercise 16. Revise the procedure **DontCare** to return all minimal expressions that it finds, rather than just the first.

Exercise 17. Revise the procedure **DontCare** to use different measures of complexity of Boolean expressions, such as the number of Boolean operations, etc.

Exercise 18. Modify **QuineMcCluskey** to allow for don't care conditions. See the discussion at the end of the Don't Care Conditions subsection in Section 12.4 of this manual.

Exercise 19. Modify **QuineMcCluskey** to use backtracking instead of the heuristic approach in order to determine the expression with the minimum number of terms. Use a large number of randomly generated expressions to compare the old procedure with the new and determine how often the heuristic produces nonoptimal output.

13 Modeling Computation

Introduction

In this chapter, we will use Maple to create implementations of theoretical models of computation. We will see how to generate elements of a language from a type 2 phrase-structure grammar and how to implement finite-state machines with and without output. We will also examine Maple's support for regular expressions, and we will implement Turing machines.

13.1 Languages and Grammars

In this section, we write a procedure to generate elements of a language from a type 2 phrase-structure grammar. Recall that a type 2 grammar has productions only of the form $w_1 \rightarrow w_2$ with w_1 a single nonterminal symbol.

Our strategy for generating the language will be as follows. We initialize a set L to the empty set. In this set, we will store all words, that is, strings consisting only of terminal symbols. A list A is initialized to the set consisting of the starting symbol.

We process an element of A by removing it from the list and applying all possible productions to it. The results of the productions are either placed in L if they consist solely of terminal symbols, or placed at the end of A to be processed further.

In order to prevent the time taken from becoming excessive, we will stop processing elements of A either if A becomes empty or if a set number of words have been produced. This limit will be an argument to our procedure.

Representation

We first need to determine how we will model the elements of the grammar in Maple.

We insist that terminal symbols be represented as characters (strings), so that the words produced can be presented as strings. Nonterminal symbols can also be represented as strings, but, for convenience, we will use unassigned Maple names instead.

Strings containing nonterminal symbols will be represented as lists, but words will be presented as strings. Note that the **cat** (concatenation) command accepts any number of strings and concatenates them. Given a list consisting entirely of strings, we apply **op** followed by **cat** to obtain a single string.

```
> cat(op([“a”, “b”, “c”, “d”]))  
“abcd”
```

(13.1)

Productions will be stored in a **table**. The indices of the table will be the nonterminal symbols (recall that we are considering only type 2 grammars). The entry associated to a nonterminal symbol will be the set of all products derivable from that symbol.

In Example 12, $S \rightarrow AB$ is the only derivation from the starting symbol, so $\{[A, B]\}$ will be the entry associated to S in the table. On the other hand, $B \rightarrow Ba$, $B \rightarrow Cb$, and $B \rightarrow b$ are all productions from B . Thus, $\{[b], [B, a], [C, b]\}$ would be the entry associated to B .

Here is the production table for Example 12. We begin by ensuring the names used as the nonterminal symbols do not store values.

```

> unassign('S', 'A', 'B', 'C'):

> Ex12productions := table():

> Ex12productions[S] := {[A, B]}:

> Ex12productions[A] := {[C, "a"]}:

> Ex12productions[B] := {[["b"], [B, "a"], [C, "b"]]}:

> Ex12productions[C] := {[["b"], ["c", "b"]]}:

> eval(Ex12productions)
table ([A = {[C, "a"]}, C = {[["b"], ["c", "b"]]}, S = {[A, B]}, 
       B = {[["b"], [B, "a"], [C, "b"]]}])
```

(13.2)

Our procedure will require the following arguments: the set **V** defining the vocabulary, the set **T** of terminal symbols, the starting symbol **S**, the table of productions **P**, and the limit on the number of words to generate, **wordlimit**. Note that, with the exception of the limit on the number of words, this is the same information that makes up a grammar. If you wish, you could apply the **timelimit** function to limit the run of the procedure by time rather than number of words, but the procedure will run so quickly that imposing a limit on the number of words generated also controls the length of the output.

Implementation

The procedure begins by calculating **N**, the nonterminal symbols. Then, **L** is initialized to the empty set and **A** is initialized to the list **[S]**. Recall that these will store the words that have been produced and the list of strings with nonterminal symbols that still require processing. We also initialize **count** to 0. This will count the number of words that have been produced, which is more efficient than repeatedly calculating the size of **L**.

After the initializations are complete, we begin a while loop controlled by the conditions that **count** does not exceed the limit on the number of words and that **A** is nonempty. Within the while loop, we set **curString** (the “current string”) equal to the first member of **A** and remove it from **A**.

We need to find all the strings that are directly derivable from **curString**. We do this as follows. First, initialize a list **D** (for derivations) to the empty list. We will store all the strings derived from **curString** in this list and then later determine which should be added to **L** and which to **A**.

Remember that **curString** is represented as a list. Loop over the entries of **curString**. For each element, check to see if it is a member of **N**, the nonterminal symbols. If not, we move on to the next element. If the symbol is nonterminal, then look the symbol up in the production table **P**. For each associated production, we perform a substitution.

An example may be helpful to explain this step. Suppose we are processing the string $[c, b, a, B, a]$ as part of the grammar given in Example 12.

```
> curString := ["c", "b", "a", B, "a"]
curString := ["c", "b", "a", B, "a"]
```

(13.3)

After determining that the first three entries are terminal, we look at **curString[4]** and see that it is nonterminal. We obtain the derivations associated with it from the table.

```
> Ex12productions[curString[4]]
{["b"], [B, "a"], [C, "b"]}
```

(13.4)

For each of these derivations, we will substitute the derivation into **curString** in place of the fourth position. We use **subsop** to perform the substitution within a loop over the elements of the set obtained from the derivations table.

```
> for s in Ex12productions[curString[4]] do
    subsop(4 = op(s), curString)
end do
["c", "b", "a", "b", "a"]
["c", "b", "a", B, "a", "a"]
["c", "b", "a", C, "b", "a"]
```

(13.5)

Once **curString** has been completely processed, we turn to deciding whether each element we placed in **D** is a word or not. The most straightforward way to approach this is to consider whether or not the set of elements in the string is a subset of the terminal symbols.

```
> {op(["c", "b", "a", "b", "a"])} subset {"a", "b", "c"}
true
```

(13.6)

```
> {op(["c", "b", "a", B, "a", "a"])} subset {"a", "b", "c"}
false
```

(13.7)

Those that are words are concatenated into strings and added to **L** (and **count** is updated). Those that are not words are added to **A**.

Here is the procedure.

```

1 FormWords := proc(V, T, S, P, wordlimit)
2   local N, L, A, count, curString, D, i, s, d;
3   N := V minus T;
4   L := {};
5   A := [S];
6   count := 0;
7   while count <= wordlimit and A <> [] do
8     curString := A[1];
9     A := A[2..-1];
10    D := [];
11    for i from 1 to nops(curString) do
12      if curString[i] in N then
13        for s in P[curString[i]] do
14          D := [op(D), subsop(i=op(s), curString)];
```

```

15      end do;
16  end if ;
17 end do;
18 for d in D do
19   if {op(d)} subset T then
20     L := L union {cat(op(d))};
21     count := count + 1;
22   else
23     A := [op(A), d];
24   end if ;
25 end do;
26 end do;
27 return L;
28 end proc:
```

We use our procedure on the grammar defined by Example 12, up to 20 words.

> *FormWords({“a”, “b”, “c”}, A, B, C, S), {"a", "b", "c"}, S, Ex12productions, 20)*
{“bab”, “baba”, “babb”, “bacbb”, “cbab”, “cbaba”,
“cbabb”, “cbacbb”} (13.8)

Note that the procedure did not return a set of size 20. This is because **count** is incremented every time a word is derived, not every time a *new* word is derived. Since there is more than one way to derive the same word, we obtain fewer than 20 words.

13.2 Finite-State Machines with Output

Example 4 from Section 13.2 describes a finite-state machine with five states and with input and output alphabets both equal to $\{0, 1\}$. Example 6 describes how to implement addition of integers using their binary expressions with a finite-state machine with output. Here, we will use Maple to model those two finite-state machines.

A First Example

Recall from Definition 1 in Section 13.2 that a finite-state machine consists of six objects: a set S of states, an input alphabet I , an output alphabet O , a transition function f , an output function g , and an initial state s_0 .

We will write a procedure that, given data defining a finite-state machine and an input string, will return the associated output string. Specifically, we will give as an argument to the procedure a list of members of the input alphabet, and the procedure will return a list of members of the output alphabet such that the I th element in the output list is the output associated with the I th member of the input list.

Representation

As is typical, we must first describe how we will represent the necessary objects in Maple.

The states will be represented by nonnegative integers. For example, in Example 4, the states will be $\{0, 1, 2, 3, 4\}$. We will assume, for the sake of simplicity, that the initial state will always be state 0. Neither S nor s_0 are required as arguments to the procedure.

The input and output alphabets, I and O , can be represented by sets of Maple objects but will not be required arguments to the procedure. In Example 4, these are both equal to the set $\{0, 1\}$.

The transition function and output function will be represented by a single table. This will have the benefit of making the definition of the functions less cumbersome. The indices to the table will be pairs $[state, input]$ where $state$ is a nonnegative integer and $input$ will be a member of I . The entries of the table will be pairs $[newState, output]$, where $newState$ is the state transitioned to and $output$ is the output corresponding to the original state and the $input$.

Here is the definition of the transition-output table for Example 4. (Refer to Table 3 of Section 13.2 as the source of the values in the table.)

```
> Ex4Table := table():

> Ex4Table[0, 0] := [1, 1]:

> Ex4Table[0, 1] := [3, 0]:

> Ex4Table[1, 0] := [1, 1]:

> Ex4Table[1, 1] := [2, 1]:

> Ex4Table[2, 0] := [3, 0]:

> Ex4Table[2, 1] := [4, 0]:

> Ex4Table[3, 0] := [1, 0]:

> Ex4Table[3, 1] := [0, 0]:

> Ex4Table[4, 0] := [3, 0]:

> Ex4Table[4, 1] := [4, 0]:
```

Observe that the indices for the transition-output table consist of every possible state-input pair.

The Machine Modeling Procedure

The procedure we create is to accept as arguments the transition-output table and the input string. It will produce the output string.

The procedure is fairly straightforward. Initialize the current state of the machine, stored in **curState**, to 0, since we are insisting that 0 represent the starting state. Also initialize the output string, **outString**, to the list of all 0s of the same length as the input list. (It is more efficient, when the length of a list is known in advance, to initialize it to the correct length than it is to build it one element at a time.)

Begin a for loop from 1 to the length of the input string. For each index, look up the pair consisting of **curState** and the element in the input string in the transition-output table. The second element in the result is placed in the output string at the correct position, and the first element is used to update **curState**. Once the loop is complete, the output list is returned.

Here is the procedure.

```

1 MachineWithOutput := proc(transTable::table, inString::list)
2   local curState, outString, i;
3   curState := 0;
4   outString := [0 $ nops(inString)];
5   for i from 1 to nops(inString) do
6     outString[i] := transTable[curState, inString[i]][2];
7     curState := transTable[curState, inString[i]][1];
8   end do;
9   return outString;
10 end proc;
```

Example 4 asks to find the output string when the input is 101011.

> *MachineWithOutput(Ex4Table, [1, 0, 1, 0, 1, 1])*
[0, 0, 1, 0, 0, 0] (13.9)

A Finite-State Machine for Addition

Example 6 in Section 13.2 describes how a finite-state machine with output that adds two integers using their binary expansions can be designed. Figure 5 in the text gives a diagram illustrating the machine.

The input alphabet for this machine are the four bit pairs: 00, 01, 10, and 11. We will represent the pairs as strings. As described by the text, we assume that the initial bits x_n and y_n are both 0.

As an example, consider adding $7 = 0111_2$ and $6 = 0110_2$. We input these two numbers as pairs and in reverse order. Thus the input string will be [10, 11, 11, 0].

The transition-output table is obtained from the diagram shown in Figure 5.

```

> addTable := table():
> addTable[0,“00”] := [0,0]:
> addTable[0,“01”] := [0,1]:
> addTable[0,“10”] := [0,1]:
> addTable[0,“11”] := [1,0]:
> addTable[1,“00”] := [0,1]:
> addTable[1,“01”] := [1,0]:
> addTable[1,“10”] := [1,0]:
> addTable[1,“11”] := [1,1]:
```

Applying the **MachineWithOutput** procedure to this table and the input produces the sum of the integers.

```
> MachineWithOutput(addTable, [“10”, “11”, “11”, “00”])  
[1, 0, 1, 1] (13.10)
```

This corresponds to $1101_2 = 13$.

13.3 Finite-State Machines with No Output

In this section, we will see how to use Maple to represent finite-state automata and to perform language recognition.

Kleene Closure

We begin this section by writing procedures to compute the concatenation of two sets of strings and the partial Kleene closure of a set of strings. As in previous sections, we will model a string as a list.

Given two lists **listA** and **listB**, we can concatenate them as follows: [**op(listA),op(listB)**]. For example,

```
> listA := [1, 2, 3]  
listA := [1, 2, 3] (13.11)
```

```
> listB := [“a”, “b”, “c”]  
listB := [“a”, “b”, “c”] (13.12)
```

```
> [op(listA), op(listB)]  
[1, 2, 3, “a”, “b”, “c”] (13.13)
```

Note that **op** applied to a number or a string has no effect.

```
> op(5)  
5 (13.14)
```

```
> op(“abc”)  
“abc” (13.15)
```

Given two sets of strings, we can form all possible concatenations by using two for loops to concatenate each pair.

```
1 SetCat := proc (A::set, B::set)  
2   local C, x, y;  
3   C := {};  
4   for x in A do  
5     for y in B do  
6       C := C union { [op(x), op(y)] } ;  
7     end do;  
8   end do;  
9   return C;  
10 end proc;
```

Applying this function to the sets from Example 1 produces the same output as in the solution to that example.

```
> listA := {0,[1,1]}
listA := {0,[1,1]} (13.16)
```

```
> listB := {1,[1,0],[1,1,0]}
listB := {1,[1,0],[1,1,0]} (13.17)
```

```
> SetCat(listA, listB)
{[0,1],[0,1,0],[1,1,1],[0,1,1,0],[1,1,1,0],[1,1,1,1,0]} (13.18)
```

Given a set A , recall that A^0 is defined to be the set of the empty string, and that for $n > 0$,

$A^{n+1} = A^n A$. Also recall that the Kleene closure of A is $A^* = \bigcup_{k=0}^{\infty} A^k$. We define the partial Kleene closure to level n by $A^{[n]} = \bigcup_{k=0}^n A^k$.

We write the following procedure to produce the powers of A . The procedure is modeled on the recursive definition given in the text.

```
1 SetPow := proc (A::set, k::nonnegint)
2   if k=0 then
3     return {[]};
4   else
5     return SetCat (SetPow(A, k-1), A);
6   end if;
7 end proc;
```

For example, with $B = \{1, 10, 110\}$, we can compute B^3 as follows.

```
> SetPow(listB, 3)
{[1,1,1],[1,0,1,1],[1,1,0,1],[1,1,1,0],[1,0,1,0,1],[1,0,1,1,0],
 [1,1,0,1,0],[1,1,0,1,1],[1,1,1,0,1],[1,1,1,1,0],[1,0,1,0,1,0],
 [1,0,1,1,0,1],[1,0,1,1,1,0],[1,1,0,1,0,1],[1,1,0,1,1,0],
 [1,1,1,0,1,0],[1,0,1,0,1,1,0],[1,0,1,1,0,1,0],[1,1,0,1,0,1,0],
 [1,1,0,1,1,0,1],[1,1,0,1,1,1,0],[1,1,1,0,1,1,0],[1,0,1,1,0,1,1,0],
 [1,1,0,1,0,1,1,0],[1,1,0,1,1,0,1,0],[1,1,0,1,1,1,0]} (13.19)
```

To form the partial Kleene closure $A^{[n]}$, we must find the union of $1, A, A^2, \dots, A^n$. Building the A^k iteratively is more efficient than using **SetPow**.

```
1 Kleene := proc (A::set, n::posint)
2   local K, x, Ak;
3   K := {};
4   for x in A do
5     K := K union {[op(x)]};
6   end do;
7   Ak := K;
```

```

8   from 2 to n do
9     Ak := SetCat (Ak, A);
10    K := K union Ak;
11  end do;
12  return K;
13 end proc:

```

We compute the Kleene closure up to level 3 of $\{0, 1\}$.

> *Kleene*($\{0, 1\}$, 3)
 $\{[], [0], [1], [0, 0], [0, 1], [1, 0], [1, 1], [0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1],$
 $[1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]\}$ (13.20)

Extended Transition Function for a Finite-State Automaton

We now create a procedure that serves as the extension of the transition function of a finite-state automaton, as described following Example 4 in Section 13.3 of the text.

As in Section 13.2 of this manual, we model the transition function as a table. The indices to the table will be the pairs consisting of the current state of the automaton and the input. The corresponding entries will be the next state of the automaton.

For example, the transition function of the finite-state automaton M_1 in Example 5 is as follows.

> *Ex5ITable* := *table*($[(0, 0) = 1, (0, 1) = 0, (1, 0) = 1, (1, 1) = 1]$):

To model the extended function that takes a pair consisting of a state and a member of the Kleene closure of the alphabet and returns the final state, we write a procedure, **ExtendedTransition**. The arguments of this procedure will be a state number, a list representing the input string, and the transition function.

We will not use the recursive definition provided in the text, but will instead use an iterative approach to designing the procedure. Begin by initializing the current state to the input state. Then, loop through the list representing the input string and apply the transition function to update the current state. Once the loop is concluded, return the state.

```

1 ExtendedTransition := proc (state, input, transFunc::table)
2   local curState, i;
3   curState := state;
4   for i from 1 to nops(input) do
5     curState := transFunc[curState, input[i]];
6   end do;
7   return curState;
8 end proc:

```

We can use this procedure to see that applying the automaton M_1 from Example 5 to $[1, 0, 1, 1, 0]$ from initial state 0 ends in state 1.

> *ExtendedTransition*(0, $[1, 0, 1, 1, 0]$, *Ex5ITable*)
1 (13.21)

Language Recognition with Finite-State Automata

Recall that a string x is recognized by a finite-state automaton if the extended transition function applied to the initial state and the string x results in a final state.

We write a procedure that, given the transition table for a finite-state automaton with initial state **init**, the set of **final** states, and the string **x**, will return true or false indicating whether or not the string is recognized by the machine.

The procedure only needs to apply **ExtendedTransition** to the state 0, the transition table, and string, and then check to see whether or not the result is in the set of final states.

```
1 IsRecognized := proc (x, transFunc::table, init, final::set)
2   local endState;
3   endState := ExtendedTransition (init, x, transFunc);
4   return evalb (endState in final);
5 end proc;
```

The solution to Example 5 indicated that the only strings accepted by M_1 are those consisting of consecutive 1s.

```
> IsRecognized ([1, 1, 1, 1, 1], Ex51Table, 0, {0})
      true
(13.22)
```

```
> IsRecognized ([1, 1, 0, 1], Ex51Table, 0, {0})
      false
(13.23)
```

Using the **Kleene** procedure from the beginning of this section, we can partially determine the language recognized by a machine.

Given the transition table, the initial state, the set of final states, a set A , and a positive integer n , the following procedure will calculate the subset of $A^{[n]}$ recognized by the finite-state automaton defined by the transition table and set of final states.

This procedure operates by applying **Kleene** and then using **IsRecognized** to check each element of $A^{[n]}$. Note that we extract those members of $A^{[n]}$ that are recognized by applying the **select** command. This command requires its first argument be a procedure that returns true or false and the second argument a set, list, or other expression. The result is the operands in the expression given as the second argument, in this case, the members of the set, for which the procedure returns true. When the procedure given as the first argument requires more than one argument, additional arguments to the procedure can be given as the third and subsequent arguments of **select**. For example, below we use **select** to obtain the members of the list that are less than the number 5.

```
> select ((a, b) → evalb(a < b), [3, 9, 5, 2, 1, 5, 6, 0, 3, 8], 5)
      [3, 2, 1, 0, 3]
(13.24)
```

The members of the list are substituted in for a , while 5 is used for b .

Here is the procedure.

```

1 FindLanguage :=  

2   proc (transFunc: :table, init, final: :set, A: :set, n: :posint)  

3     local An;  

4     An := Kleene (A, n);  

5     return select (IsRecognized, An, transFunc, init, final);  

end proc;

```

Applying this procedure to our M_1 machine and $\{0, 1\}^{[10]}$, we see that the only strings in that set recognized by the finite-state automaton are those consisting only of 1s.

```

> FindLanguage (Ex51Table, 0, {0}, {0, 1}, 10)
{[], [1], [1, 1], [1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1]}                                         (13.25)

```

Nondeterministic Finite-State Automata

We conclude this section with an implementation of the constructive proof of Theorem 1 of Section 13.3. Given a nondeterministic finite-state automaton, our procedure will produce a deterministic finite-state automaton.

In particular, given the transition table for a nondeterministic automaton, its input alphabet, its starting state, and its set of final states, the procedure will produce the transition table for a deterministic automaton, its starting state, and its set of final states.

For a nondeterministic automaton, we will represent the transition function in the same way as for the deterministic automaton earlier, except the entries in the table will be sets of states, rather than individual states.

For example, here is the transition table for the nondeterministic automaton described in Example 10.

```

> Ex10Table := table():
> Ex10Table[0, 0] := {0, 2}:
> Ex10Table[0, 1] := {1}:
> Ex10Table[1, 0] := {3}:
> Ex10Table[1, 1] := {4}:
> Ex10Table[2, 0] := {}:
> Ex10Table[2, 1] := {4}:
> Ex10Table[3, 0] := {3}:
> Ex10Table[3, 1] := {}:

```

```
> Ex10Table[4, 0] := {3}:
```

```
> Ex10Table[4, 1] := {3}:
```

The final states are 0 and 4.

To determine the deterministic automaton's transition table, its starting state, and final states, we follow the proof of Theorem 1. The deterministic automaton's states are sets of states of the nondeterministic automaton.

We begin with the set consisting of the nondeterministic automaton's starting state. This is the starting state for the deterministic automaton. Given any state of the deterministic automaton, and any input, the deterministic transition is the union over all members of the state of the results of applying the nondeterministic automaton's transition with that input value.

In our procedure, we will create a table initialized to the empty table. We will also create two sets S and T . The set S will be initialized to the empty set and, at the conclusion of the procedure will be the set of all states of the deterministic automaton. The set T will be initialized to $\{s_0\}$, the set containing the initial state of the deterministic automaton.

As long as T is nonempty, we will move one of its members from T to S and apply the nondeterministic automaton's transition function with all possible input values. The results are the entries in the deterministic transition table and those that are not already members of S are added to T for further processing.

The final states of the deterministic automaton are those states which contain a final state of the nondeterministic automaton. That is, the final states are those whose intersection with the set of the original final states is nonempty. Before exiting, the procedure calculates the set of final states for the deterministic automaton.

Here is the procedure. Note that the procedure returns the sequence of the new transition table, the starting state, and the set of final states.

```
1 MakeDeterministic :=
2   proc (transFunc::table, Iset::set, init, final::set)
3     local newTable, S, T, state, i, s, x, newfinal;
4     newTable := table ();
5     S := {};
6     T := {{init}};
7     while T <> {} do
8       state := T[1];
9       T := T minus {state};
10      S := S union {state};
11      for i in Iset do
12        x := {};
13        for s in state do
14          x := x union transFunc[s, i];
15        end do;
16        newTable[state, i] := x;
17        if not (x in S) then
```

```

17   T := T union {x};
18   end if;
19   end do;
20 end do;
21 newfinal := {};
22 for state in S do
23   if state intersect final <> {} then
24     newfinal := newfinal union {state};
25   end if;
26 end do;
27 return newTable, {init}, newfinal;
28 end proc:
```

Applying this procedure to the Example 10 information produces the following.

```

> Ex10DTable, Ex10Dinit, Ex10Dfinal :=  

  MakeDeterministic(Ex10Table, {0, 1}, 0, {0, 4})  

  Ex10DTable, Ex10Dinit, Ex10Dfinal := newTable, {0},  

  {{0}, {4}, {0, 2}, {1, 4}, {3, 4}}  

(13.26)
```

We can inspect the transition table by applying **eval** to **Ex10DTable**.

```

> eval(Ex10DTable)
table([(0, 2), 0] = {0, 2}, ((1, 4), 1) = {3, 4}, ((3), 0) = {3},
      ((0, 2), 1) = {1, 4}, ((3), 1) = {}, ((), 0) = {}, ((4), 1) = {3},
      ((3, 4), 1) = {3}, ((0), 1) = {1}, ((4), 0) = {3}, ((0), 0) = {0, 2},
      ((3, 4), 0) = {3}, ((1), 1) = {4}, ((1), 0) = {3}, ((1, 4), 0) = {3},
      ((), 1) = {}])  

(13.27)
```

You can confirm that this agrees with Figure 8 from Section 13.3.

We use the output as the arguments to **FindLanguage**.

```

> FindLanguage(Ex10DTable, Ex10Dinit, Ex10Dfinal, {0, 1}, 10)
{[], [0], [0, 0], [0, 1], [1, 1], [0, 0, 0], [0, 0, 1], [0, 1, 1], [0, 0, 0, 0], [0, 0, 0, 1],
 [0, 0, 1, 1], [0, 0, 0, 0, 0], [0, 0, 0, 0, 1], [0, 0, 0, 1, 1], [0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 1], [0, 0, 0, 0, 1, 1], [0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 1],
 [0, 0, 0, 0, 0, 1, 1], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 1],
 [0, 0, 0, 0, 0, 0, 1, 1], [0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 1],
 [0, 0, 0, 0, 0, 0, 0, 1, 1], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
 [0, 0, 0, 0, 0, 0, 0, 1, 1]}  

(13.28)
```

This list of strings suggests that the language recognized by this automaton are those strings consisting of a positive number of 0s followed by no more than two 1s, together with the empty string and the string 11.

13.4 Language Recognition

In this section, we will introduce Maple's support for regular expressions for working with strings. We will also develop a procedure for calculating the concatenation of two nondeterministic automata.

Regular Expressions

Maple's commands related to regular expressions lie in the **StringTools** package, which we load now.

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

The command **RegMatch** is used to determine whether or not a given string can be matched to a given pattern. When using **RegMatch**, the first argument is a string describing the regular expression or pattern, and the second argument is the string you are attempting to match against.

Perhaps the most basic form of a regular expression is the concatenation of elements of the set. For example, “01” is a regular expression. This expression matches itself, of course.

```
> RegMatch("01", "01")  
true
```

(13.29)

The output indicates that yes, the string “01” matches the regular expression “01”.

Anchors

The next example illustrates a significant difference between regular expressions as described in the text and regular expressions in Maple.

```
> RegMatch("01", "so42301kkj91")  
true
```

(13.30)

It is clear that the string “so42301kkj91” is not a member of the regular set specified by the regular expression “01”, since the only member of that regular set is the string “01”. Maple returned true because the pattern “01” matched a substring of the second argument.

Maple, and most programming languages, use regular expressions primarily to search for patterns within strings and they interpret regular expressions as loosely as possible to make them flexible. To use regular expressions in Maple in the more formal sense described in the text, we will need to specify that the pattern we give as the first argument is to match the entire string, not part of it.

To do this, we use two special characters, \wedge and $\$$. These are referred to as anchors and they match the beginning and end of a string, respectively. Including them in the regular expression will ensure that we match only those strings completely described by the regular expression and that we do not match substrings.

```
> RegMatch("^01$", "01")  
true
```

(13.31)

```
> RegMatch("^01$", "so42301kkj91")  
false
```

(13.32)

Kleene Closure

The asterisk is a symbol used in a regular expression to represent the Kleene closure.

For example, the regular expression **10*** will match a 1 followed by any number of 0s.

```
> RegMatch("^10*$", "10000000")
true
```

(13.33)

```
> RegMatch("^10*$", "1")
true
```

(13.34)

```
> RegMatch("^10*$", "0111000")
false
```

(13.35)

Note that without the **^** and **\$**, the final example would have also returned true.

As in the text, parentheses can be used to group symbols. For example **(10)*** matches any number of copies of **10**.

```
> RegMatch("^(10)*$", "101010101010")
true
```

(13.36)

```
> RegMatch("^(10)*$", "1010101")
false
```

(13.37)

Maple, and most languages that support regular expressions, also recognizes **+** and **?**. These are used like ***** but with different meaning. The expression **A+** is used to match one or more copies of A. Essentially, it is the Kleene closure minus the empty string. For example, **1*0+** matches any number of 1s followed by at least one 0.

```
> RegMatch("^\1*0+$", "1111000")
true
```

(13.38)

```
> RegMatch("^\1*0+$", "00")
true
```

(13.39)

```
> RegMatch("^\1*0+", "111")
false
```

(13.40)

The **A?** expression is used to match 0 or 1 copies of A. For example, **1*0?** matches any number of 1s which may be followed by at most one 0.

```
> RegMatch("^\1*0?$", "111111")
true
```

(13.41)

```
> RegMatch("^\1*0?$", "1111110")
true
```

(13.42)

```
> RegMatch("^\1*0?$", "11111100")
false
```

(13.43)

Union

To represent union, the vertical line is used. A `|` placed between two expressions will match either of them. The `|` can take the place of the \cup symbol in an expression such as $0(0 \cup 1)^*$.

```
> RegMatch("^\wedge 0(0|1)^*\$","011010")
true
```

(13.44)

```
> RegMatch("^\wedge 0(0|1)^*\$","1011010")
false
```

(13.45)

This can also be done in more complicated expressions. For example, $2((10)^* \cup (01)^*)^2$ describes the set of strings beginning and ending with 2s with an alternating sequence of 0s and 1s in between.

```
> RegMatch("^\wedge 2((10)^*|(01)^*)^2\$","21010102")
true
```

(13.46)

```
> RegMatch("^\wedge 2((10)^*|(01)^*)^2\$","201012")
true
```

(13.47)

```
> RegMatch("^\wedge 2((10)^*|(01)^*)^2\$","210012")
false
```

(13.48)

In some circumstances, union can be replaced by “character classes.” By placing characters within a set of brackets, you indicate that any of the characters inside the brackets are allowed. For example, $0(0 \cup 1)^*$ can be expressed as follows.

```
> RegMatch("^\wedge 0[01]^*\$","011010")
true
```

(13.49)

Note that this cannot be used in more complicated expressions such as (13.46) and (13.48). It is only available when the options are single characters.

Ranges can also be used within a character class by separating the beginning and end of a range of characters with a hyphen. For example, the following matches strings beginning at least one lower-case letter and ending with a digit between 6 and 9.

```
> RegMatch("^\wedge [a-z]+[6-9]\$","test9")
true
```

(13.50)

```
> RegMatch("^\wedge [a-z]+[6-9]\$","Test9")
false
```

(13.51)

Observe that character classes are case sensitive, but multiple ranges can be used within a class.

```
> RegMatch("^\wedge [a-zA-Z]+[6-9]\$","Test9")
true
```

(13.52)

Character classes have a negated option. By beginning a character class with a caret, you indicate that any character other than those specified are allowed. For example, in the following, the

regular expression matches all strings beginning with 1, ending with 0, and which include no other 1s nor 0s.

```
> RegMatch("^1[^01]*0$", "169jwq0")
true
```

(13.53)

The special character dot (.) is used to match any character. For example, 1...0 will match any string beginning with a 1, followed by any three characters and ending with a 0.

```
> RegMatch("^1...0$", "12340")
true
```

(13.54)

```
> RegMatch("^1...0$", "1230")
false
```

(13.55)

```
> RegMatch("^1...0$", "1234567890")
false
```

(13.56)

Regular expressions in Maple are extremely flexible. The interested reader is referred to the help page on regular expressions for more information.

Concatenation of Automata

We will write a procedure that concatenates two nondeterministic finite-state automata, as described in the proof of Theorem 1 of the text.

Two Automata

We begin by defining two automata that our procedure will concatenate.

The first automata is the result of Example 3, for recognizing $1 * \cup 01$. Our implementation is based on the simple form shown in Figure 3b.

Note that the diagram in the text omits the results of transitioning from certain states via certain input values. For example, it does not show the result of the transition from state s_1 with input 0. This makes for a simpler and cleaner diagram, but the transition table will need to include this information. It will be assumed that all such omissions correspond to a transition to the state {}.

Here is the transition table corresponding to the automaton shown in Figure 3b.

```
> Atable := table():
> Atable[0, 0] := {2}:
> Atable[0, 1] := {1}:
> Atable[1, 0] := {}:
> Atable[1, 1] := {1}:
> Atable[2, 0] := {}:
```

```

> Atable[2,1] := {3} :

> Atable[3,0] := {} :

> Atable[3,1] := {} :

```

The final states for this automaton are $\{0, 1, 3\}$. We can confirm that it recognizes $1^* \cup 01$ by applying **MakeDeterministic** and **FindLanguage**.

```

> FindLanguage (MakeDeterministic (Atable, {0,1}, 0, {0,1,3}), {0,1}, 10)
{[], [1], [0,1], [1,1], [1,1,1], [1,1,1,1], [1,1,1,1,1], [1,1,1,1,1,1],
 [1,1,1,1,1,1,1], [1,1,1,1,1,1,1,1], [1,1,1,1,1,1,1,1,1],
 [1,1,1,1,1,1,1,1,1]}                                         (13.57)

```

As you can see, the language recognized by this machine includes the string 01 as well as 1^* .

The second automaton we create will recognize the language 101.

```

> Btable := table( ):

> Btable[0,0] := {} :

> Btable[0,1] := {1} :

> Btable[1,0] := {2} :

> Btable[1,1] := {} :

> Btable[2,0] := {} :

> Btable[2,1] := {3} :

> Btable[3,0] := {} :

> Btable[3,1] := {} :

```

The only final state is state 3.

Applying **FindLanguage**, we confirm that this models that machine that recognizes 101.

```

> FindLanguage (MakeDeterministic (Btable, {0,1}, 0, {3}), {0,1}, 10)
{[1,0,1]}                                         (13.58)

```

Concatenating the Machines

Our concatenation procedure will require the following arguments, for both machines: the transition table, the starting state, and the final states. It will also require that the two machines have a common input alphabet but that alphabet does not need to be an argument.

Recall the following elements of the construction of the concatenation as described in the proof of Theorem 1 of Section 13.4.

1. The states of the concatenation is the union of the states of the original machines, which are assumed to be disjoint.
 2. The starting state of the concatenation is the starting state of the first of the two machines.
 3. The final states of the concatenation include the set of final states of the second machine.
 4. The final states of the concatenation also include the starting state if the empty string is a member of both languages.
 5. All transitions of the original machines are transitions of the new machine.
 6. Additionally, for every transition in the first machine leading to a final state, we add a transition in the concatenation to the starting state of the second machine.
 7. Finally, if the starting state of the first machine is final, then for every transition from the starting state of the second machine, we add a transition from the starting state of the new machine.
- The assumption that the states of the original two machines are disjoint means that we will need to make them so. There are a variety of ways in which we could do this. Since we assume that states are designated by nonnegative integers, we can make the states distinct by multiplying each state by 10 and adding 1 if it is in the first machine and 2 if it is in the second machine.

Therefore, the starting state of the concatenation is found by $10 \cdot s_A + 1$ where s_A is the starting state of the first machine. In our case, this will be equal to $10 \cdot 0 + 1 = 1$.

Next, we find the final states of the concatenation. Let **Afinal** and **Bfinal** be the sets of final states for the original two machines. According to point 3 above, the final states of the concatenated machine include the final states of the second machine. We only need to update the names.

The final states of the machines we defined above are as follows.

```
> Afinal := {0, 1, 3}
Afinal := {0, 1, 3} (13.59)
```

```
> Bfinal := {3}
Bfinal := {3} (13.60)
```

We can obtain the final states of the concatenation by applying the function $f \rightarrow 10f + 2$ to the set of final states of the second machine.

```
> map(f → 10f + 2, Bfinal)
{32} (13.61)
```

Item 4 asserts that the starting state of the concatenated machine is a final state if and only if the empty string is a member of both languages. Another way to put this is that the starting state of the concatenated machine is a final state when both of the original machines have their own starting states as final states. This is not the case in our example. We will include this possibility in our general procedure by checking to see if the starting states are members of the sets of final states.

To form the transitions of the new machine, we begin with an empty table.

```
> ABtable := table():
```

Item 5 tells us that all of the original transitions are transitions in the new machine. Thus, for both of the original tables, we need to add corresponding entries to this table. Keep in mind that the state names are updated by multiplying by 10 and adding 1 or 2.

We proceed as follows. For **Atable**, use **indices** to obtain the list of all indices in the table. For each index **i**, the index in **ABtable** will be **[10*i[1]+1,i[2]]**. This computes the appropriate state name in the concatenated machine and keeps the same input value. The associated entry will be obtained by using **map** and the functional operator **f->10*f+1** applied to the previous entry. Note that **op** must be applied to **i** before it can be used as an index to **Atable**, since the **indices** command wraps indices in lists.

```
> for i in indices(Atable) do
    ABtable[10 * i[1] + 1, i[2]] := map( f → 10 * f + 1, Atable[op(i)])
end do
    ABtable11,1 := {11}
    ABtable21,1 := {31}
    ABtable1,1 := {11}
    ABtable11,0 := ∅
    ABtable31,1 := ∅
    ABtable31,0 := ∅
    ABtable1,0 := {21}
    ABtable21,0 := ∅
(13.62)
```

For the second machine, we do the same thing except adding 2 instead of 1.

```
> for i in indices(Btable) do
    ABtable[10 * i[1] + 2, i[2]] := map( f → 10 * f + 2, Btable[op(i)])
end do
    ABtable12,1 := ∅
    ABtable22,1 := {32}
    ABtable2,1 := {12}
    ABtable12,0 := {22}
    ABtable32,1 := ∅
    ABtable32,0 := ∅
    ABtable2,0 := ∅
    ABtable22,0 := ∅
(13.63)
```

Next, we must add transitions between the two components. As item 6 instructs, for each transition in the first of the two machines that leads to a final state, we must add a transition in the concatenated machine to the starting state of the second machine.

We will again loop through the indices of **Atable**, this time checking whether the image contains any states that are final for machine A. If so, we will add the transition to state 2 (the name of the starting state in the second machine in the concatenation). (Note that we must update the entry in the **ABtable** rather than replace it.)

```
> for i in indices(Atable) do
    if Atable[op(i)] intersect finalA ≠ { } then
        ABtable[10 * i[1] + 1, i[2]] := ABtable[10 * i[1] + 1, i[2]] union {2}
    end if
end do
```

We can see the current transition table by applying **eval**.

```
> eval(ABtable)
table([(32, 1) = ∅, (1, 1) = {2, 11}, (11, 0) = ∅, (2, 1) = {12},
       (22, 1) = {32}, (1, 0) = {21}, (21, 0) = ∅, (2, 0) = ∅, (31, 0) = ∅,
       (21, 1) = {2, 31}, (31, 1) = ∅, (32, 0) = ∅, (22, 0) = ∅, (12, 1) = ∅,
       (11, 1) = {2, 11}, (12, 0) = {22}])
```

(13.64)

Finally, since the starting state of the first machine is final, we must add transitions from the starting state of the concatenated machine for each of the transitions from the starting state of the second machine. The starting state of the second machine in this example is 0, and the starting state of the concatenation is 1.

```
> for i in indices(Btable) do
  if i[1] = 0 then
    ABtable[1, i[2]] := ABtable[1, i[2]] union map(f → 10 * f + 2, Btable[op(i)])
  end if
end do
```

Apply **eval** again.

```
> eval(ABtable)
table([(32, 1) = ∅, (1, 1) = {2, 11, 12}, (11, 0) = ∅, (2, 1) = {12},
       (22, 1) = {32}, (1, 0) = {21}, (21, 0) = ∅, (2, 0) = ∅, (31, 0) = ∅,
       (21, 1) = {2, 31}, (31, 1) = ∅, (32, 0) = ∅, (22, 0) = ∅, (12, 1) = ∅,
       (11, 1) = {2, 11}, (12, 0) = {22}])
```

(13.65)

Note that this modified the entry for (1, 1). (Recall that state 1 is the starting state for the combined machine.) Before, (1, 1) was associated with {2, 11}, the starting state of the second machine and state 1 of the first machine. Now, the entry for (1, 1) also includes 12, state 1 of the second machine.

That (1, 1) is associated with {2, 11, 12} means that from the starting state of the concatenation and input 1, there are three options. Going to state 2, the starting state of the second machine, corresponds to recognizing the string 1 followed by a string recognized by the second machine. Going to state 11, state 1 of the first machine, corresponds to building a string of all 1s, which is recognized by the first machine. And going to state 12, state 1 of the second machine, corresponds to the first machine contributing the empty string followed by 1 as the first character of a string recognized by the second machine.

Implementation as a Procedure

Here is the complete procedure.

```
1 CatAutomata :=
2   proc (Atable, Astart, Afinal, Btable, Bstart, Bfinal, Iset)
3     local ABtable, ABstart, ABfinal, i;
4     ABstart := 10 * Astart + 1;
5     ABfinal := map(f → 10 * f + 2, Bfinal);
6     if (Astart in Afinal) and (Bstart in Bfinal) then
7       ABfinal := ABfinal union {ABstart};
8     end if;
```

```

8 ABtable := table();
9 for i in indices(Atable) do
10    ABtable[10*i[1]+1, i[2]] := map(f->10*f+1, Atable[op(i)]);
11 end do;
12 for i in indices(Btable) do
13    ABtable[10*i[1]+2, i[2]] := map(f->10*f+2, Btable[op(i)]);
14 end do;
15 for i in indices(Atable) do
16    if Atable[op(i)] intersect Afinal <> {} then
17       ABtable[10*i[1]+1, i[2]] := ABtable[10*i[1]+1, i[2]] union
18          {Bstart*10+2};
19    end if;
20 end do;
21 if (Astart in Afinal) then
22   for i in indices(Btable) do
23     if i[1] = 0 then
24        ABtable[1, i[2]] := ABtable[1, i[2]] union
25           map(f->10*f+2, Btable[op(i)]);
26     end if;
27   end do;
28 end if;
return ABtable, Iset, ABstart, ABfinal;
end proc:
```

Applying this to our examples and passing the results on to **MakeDeterministic** and **FindLanguage** shows us that the result does indeed recognize $(1 * \cup 01) 101$.

$$\begin{aligned}
&> Ctable, CI, Cstart, Cffinal := \\
&\quad \text{CatAutomata}(Atable, 0, \{0, 1, 3\}, Btable, 0, \{3\}, \{0, 1\}) \\
&\quad Ctable, CI, Cstart, Cffinal := ABtable, \{0, 1\}, 1, \{32\}
\end{aligned} \tag{13.66}$$

$$\begin{aligned}
&> \text{FindLanguage}(\text{MakeDeterministic}(Ctable, CI, Cstart, Cffinal), \{0, 1\}, 10) \\
&\quad \{[1, 0, 1], [1, 1, 0, 1], [0, 1, 1, 0, 1], [1, 1, 1, 0, 1], \\
&\quad [1, 1, 1, 1, 0, 1], [1, 1, 1, 1, 1, 0, 1], [1, 1, 1, 1, 1, 1, 0, 1], \\
&\quad [1, 1, 1, 1, 1, 1, 1, 0, 1], [1, 1, 1, 1, 1, 1, 1, 1, 0, 1]\}
\end{aligned} \tag{13.67}$$

13.5 Turing Machines

In this section, we will create a model of a Turing machine. In our model, the tape will be represented by a list, with the assumption that all elements to the left and right of the bounds of the list are blanks. The blank symbol will be represented by the symbol B.

The Partial Function

The text uses the convention that the partial function that controls the operation of the Turing machine is defined by a set of five-tuples. It will be more convenient for our procedures to represent the function as a table from pairs (s, x) to triples (s', x', d) .

We create a procedure that will transform the set of five-tuples representation into the table representation.

```

1 TuplesToTable := proc (S :: set)
2   local T, x;
3   T := table ();
4   for x in S do
5     T [x [1], x [2]] := x [3..5];
6   end do;
7   return T;
8 end proc;
```

Applying this procedure to the set of tuples given in Example 1 provides us with an example of a partial function to work with.

```

> unassign ('B', 'R', 'L')

> Ex1 := TuplesToTable ({[0, 0, 0, 0, R], [0, 1, 1, 1, R], [0, B, 3, B, R],
1, 0, 0, 0, R], [1, 1, 2, 0, L], [1, B, 3, B, R], [2, 1, 3, 0, R]})

Ex1 := T
```

(13.68)

```

> eval (Ex1)
table ([ (0, B) = [3, B, R], (1, 1) = [2, 0, L], (2, 1) = [3, 0, R],
(0, 1) = [1, 1, R], (1, 0) = [0, 0, R], (1, B) = [3, B, R],
(0, 0) = [0, 0, R]])
```

(13.69)

Note that the symbols **B**, **L**, and **R** must all be unassigned names, otherwise they will be evaluated within the set of five-tuples and will produce unexpected results.

The Turing Machine Procedure

Our Turing machine procedure will accept as input a table representing the partial function, a list representing the status of the tape before running the machine, and the initial state. It will return the final tape and the state of the machine upon exit.

When the procedure begins, we initialize the name **pos** to 1, indicating that the control head is positioned at the leftmost element in the tape. We set the **state** of the machine to the initial state and copy the **tape** from the argument as well. We also compute the **domain** of the partial function by applying **indices** to the table. This will make it easier to check whether we have reached a halt.

The main work of the procedure will take place within a while loop controlled by the condition that the domain of the function includes the pair consisting of the current state and the entry on the tape at the current position.

Within the loop, we first obtain the values of the new state, new tape entry, and direction from the partial function. We then set the **state** to the new state, change the entry on the **tape**, and update the position **pos**. Note that when changing the position of the control head, we must take care not to exceed the bounds of the list representing the tape. If the previous position was location 1 in the list and the direction is left, then instead of changing the position, we extend the list by adding a blank on the left with the syntax **[B,op(tape)]**. On the other hand, if the previous position was the right

end of the tape and the direction is right, then we increase the position and extend the tape to the right via **[op(tape),B]**.

Here is the procedure.

```

1 Turing := proc (f :: table, T :: list, init)
2   local pos, state, tape, domain, Y;
3   pos := 1;
4   state := init;
5   tape := T;
6   domain := {indices(f)};
7   while [state, tape[pos]] in domain do
8     Y := f[state, tape[pos]];
9     state := Y[1];
10    tape[pos] := Y[2];
11    if pos=1 and Y[3]='L' then
12      tape := ['B', op(tape)];
13    elif pos=nops(tape) and Y[3]='R' then
14      tape := [op(tape), 'B'];
15      pos := pos + 1;
16    elif Y[3]='L' then
17      pos := pos - 1;
18    else
19      pos := pos + 1;
20    end if;
21  end do;
22  return tape, state;
23 end proc:
```

We use the procedure to run the Turing machine from Example 1 on the tape shown in Figure 2a.

> *Turing(Ex1,[0,1,0,1,1,0],0)*
[0,1,0,0,0,0], 3 (13.70)

Observe that this agrees with Figure 2 from Section 13.5 in the text.

We will create a verbose version of this procedure as well. The operation of the verbose version is identical to **Turing**, but it displays the status of the machine at every step.

```

1 VerboseTuring := proc (f :: table, T :: list, init)
2   local pos, state, tape, domain, Y, displayTape;
3   pos := 1;
4   state := init;
5   tape := T;
6   domain := {indices(f)};
7   displayTape := tape;
8   displayTape[pos] := cat('*', tape[pos]);
9   print(displayTape, state);
10  while [state, tape[pos]] in domain do
```

```

11      Y := f[state, tape[pos]];
12      state := Y[1];
13      tape[pos] := Y[2];
14      if pos=1 and Y[3]='L' then
15          tape := ['B', op(tape)];
16      elif pos=nops(tape) and Y[3]='R' then
17          tape := [op(tape), 'B'];
18          pos := pos + 1;
19      elif Y[3]='L' then
20          pos := pos - 1;
21      else
22          pos := pos + 1;
23      end if;
24      displayTape := tape;
25      displayTape[pos] := cat('*', tape[pos]);
26      print(displayTape, state);
27  end do;
28  return tape, state;
end proc;

```

> *VerboseTuring(Ex1, [0, 1, 0, 1, 1, 0], 0)*

```

[*0, 1, 0, 1, 1, 0], 0
[0, *1, 0, 1, 1, 0], 0
[0, 1, *0, 1, 1, 0], 1
[0, 1, 0, *1, 1, 0], 0
[0, 1, 0, 1, *1, 0], 1
[0, 1, 0, 1, 0, 0], 2
[0, 1, 0, 0, *0, 0], 3
[0, 1, 0, 0, 0, 0], 3

```

(13.71)

Applications of Turing Machines

We now apply our Turing machine procedure to two applications: recognizing strings in a language and computing functions.

Recognizing Sets

We will implement the Turing machine for recognizing $\{0^n 1^n \mid n \geq 1\}$.

The partial function was given in the solution to Example 3.

> *Ex3 := TuplesToTable({[0, 0, 1, M, R], [1, 0, 1, 0, R], [1, 1, 1, 1, R], [1, B, 2, B, L], [1, M, 2, M, L], [2, 1, 3, M, L], [3, 0, 4, 0, L], [3, 1, 3, 1, L], [3, M, 5, M, R], [4, 0, 4, 0, L], [4, M, 0, M, R], [5, M, 6, M, R]})*
Ex3 := T

(13.72)

To determine whether or not a string is in the language, we only have to apply the Turing machine to the string and check the exit state.

```
> Turing (Ex3, [0, 0, 0, 0, 1, 1, 1, 1], 0)
    [M, M, M, M, M, M, M, M, B], 6
```

(13.73)

The fact that the machine halted in state 6, the final state, indicates that it recognizes the string. On the other hand,

```
> Turing (Ex3, [0, 0, 0, 1, 1], 0)
    [M, M, M, M, M, B], 2
```

(13.74)

halted in state 2, indicating that the string is not in the language.

Adding Nonnegative Integers

Example 4 describes how to use Turing machines to perform addition.

The machine is described by the following tuples.

```
> adder := TuplesToTable ({[0, 1, 1, B, R], [1, 1, 2, B, R], [1, "*", 3, B, R],
    [2, 1, 2, 1, R], [2, "*", 3, 1, R] })
adder := T
```

(13.75)

We add two numbers a and b by using the unary representation tape consisting of $a + 1$ 1s followed by an asterisk and then $b + 1$ 1s. We create a small procedure to create the tape given a and b .

1	UnaryTape := proc (a : : nonnegint, b : : nonnegint)
2	return [1\$ (a+1), "*", 1\$ (b+1)];
3	end proc;

The tape used to add 3 and 4 is shown below.

```
> UnaryTape (3, 4)
    [1, 1, 1, 1, "*", 1, 1, 1, 1, 1]
```

(13.76)

Performing addition is accomplished by applying **Turing** to the transition function and the tape.

```
> Turing (adder, UnaryTape (3, 4), 0)
    [B, B, 1, 1, 1, 1, 1, 1, 1, 1]
```

(13.77)

You can see that this contains a string of eight 1s, indicating a result of 7.

Using the verbose form of Turing, you can see how the Turing adder operates.

```
> VerboseTuring (adder, UnaryTape (3, 4), 0)
    [*1, 1, 1, 1, "*", 1, 1, 1, 1, 1], 0
    [B, *1, 1, 1, "*", 1, 1, 1, 1, 1], 1
    [B, B, *1, 1, "*", 1, 1, 1, 1, 1], 2
    [B, B, 1, *1, "*", 1, 1, 1, 1, 1], 2
    [B, B, 1, 1, **, 1, 1, 1, 1, 1], 2
    [B, B, 1, 1, 1, *1, 1, 1, 1, 1], 3
    [B, B, 1, 1, 1, 1, 1, 1, 1, 1], 3
```

(13.78)

Solutions to Computer Projects and Computations and Explorations

Computer Projects 8

Given the state table of a nondeterministic finite-state automaton and a string, decide whether this string is recognized by the automaton.

Solution: One solution to this problem, the solution used earlier in this chapter, is to find the deterministic automaton that recognizes the same language and use it to decide whether the string is recognized or not. This is what we have been doing when we apply **FindLanguage** to the result of **MakeDeterministic**.

Here we will take a direct approach. For deterministic machines, we created two procedures: **ExtendedTransition** and **IsRecognized**. The **IsRecognized** procedure merely called **ExtendedTransition** and checked whether the result was a final state or not. The **ExtendedTransition** procedure took a state, an input string, and a transition table, and determined the state of the machine following the processing of the input.

Our approach for nondeterministic machines will be similar. We will create two procedures: **ExtendedTransitionND** and **IsRecognizedND**. The main difference between the deterministic machines and nondeterministic machines is that with nondeterministic machines, given the initial state and an input, we do not know the next state. Instead, there is a set of possible states.

ExtendedTransitionND will therefore take a set of possible states, an input, and a transition table as its arguments. For each member of the input string, it will apply the transition table to each of the possible states, producing a new set of possible states. It will return the set of possible states after processing each element in the input string.

```
1 ExtendedTransitionND := proc(states, input, transFunc)
2   local curStates, i, s, newStates;
3   curStates := states;
4   for i from 1 to nops(input) do
5     newStates := {};
6     for s in curStates do
7       newStates := newStates union transFunc[s, input[i]];
8     end do;
9     curStates := newStates;
10    end do;
11    return curStates;
12  end proc;
```

A nondeterministic machine recognizes a string if the result of running the machine from the starting state with the input string results in a set of possible ending states that includes at least one final state. We write **IsRecognizedND** to call **ExtendedTransitionND** and check to see if the result intersects the set of final states.

```
1 IsRecognizedND := proc(x, transFunc, init, final)
2   local endStates;
3   endStates := ExtendedTransitionND({init}, x, transFunc);
4   return evalb(endStates intersect final <> {});
5 end proc;
```

With **IsRecognizedND** in hand, we can create **FindLanguageND**. This is effectively identical to **FindLanguage**.

```

1 FindLanguageND := proc(transFunc, init, final, A, n)
2   local An, x, L;
3   An := Kleene(A, n);
4   L := {};
5   for x in An do
6     if IsRecognizedND(x, transFunc, init, final) then
7       L := L union {x};
8     end if;
9   end do;
10  return L;
11 end proc;
```

Applying this procedure to the machine defined by transition function **Ctable**, starting state 1, final state {32}, and alphabet {0,1} that was produced by **CatAutomata**, we see that the result is the same as when we applied **FindLanguage** and **MakeDeterministic** in (13.67).

> *FindLanguageND(Ctable, 1, {32}, {0, 1}, 10)*
{[1, 0, 1], [1, 1, 0, 1], [0, 1, 1, 0, 1], [1, 1, 1, 0, 1], [1, 1, 1, 1, 0, 1],
 [1, 1, 1, 1, 1, 0, 1], [1, 1, 1, 1, 1, 1, 0, 1], [1, 1, 1, 1, 1, 1, 1, 0, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 0, 1]} (13.79)

> *evalb(% = (13.67))*
true (13.80)

Computations and Explorations 1

Solve the busy beaver problem for two states by testing all possible Turing machines with two states and alphabet {1, B}.

Solution: The busy beaver problem, described in the preface to Exercise 31 in Section 13.5, asks: what is the maximum number of 1s that a Turing machine with n states on the alphabet {1, B} may print on an initially blank tape? This exercise asks us to solve the busy beaver problem with a brute force approach for $n = 2$. (Note: Several of the steps in this solution take a few seconds to complete; therefore, except for the procedure definitions, none of the input lines in this solution will autoexecute.)

We will construct all possible Turing machines on two states with the given alphabet. For each possible Turing machine, we will allow it to run until either it halts, or until it has reached a predefined limit on the number of steps it is allowed. This later condition is important, since some of the possible machines will not halt on their own.

Generating all possible Turing machines on {1, B} with two states is equivalent to finding all possible transition functions. The domain of a transition function is the set $S \times I = \{0, 1\} \times \{1, B\}$. The codomain is the set $\{0, 1, 2\} \times \{1, B\} \times \{L, R\}$, where we use state 2 as a halting state, that is, a state which will cause the machine to halt.

We create the domain and codomain using the **cartprod** command from **combinat**.

```

> dom := []
dom := [] (13.81)

```

```
> StimesI := combinat[cartprod]([[0, 1], [1, B]]) :
```

```

> while not StimesI[finished] do
    dom := [op(dom), StimesI[nextvalue]()
end do :
```

```

> dom
[[0, 1], [0, B], [1, 1], [1, B]] (13.82)
```

```

> codom := []
codom := [] (13.83)
```

```
> StimesItimesLR := combinat[cartprod]([[0, 1, 2], [1, B], [L, R]]) :
```

```

> while not StimesItimesLR[finished] do
    codom := [op(codom), StimesItimesLR[nextvalue]()
end do :
```

```

> codom
[[0, 1, L], [0, 1, R], [0, B, L], [0, B, R], [1, 1, L], [1, 1, R], [1, B, L],
 [1, B, R], [2, 1, L], [2, 1, R], [2, B, L], [2, B, R]] (13.84)
```

Now, each possible transition function is an assignment of each member of **dom** to one of the members of **codom**. We can think of this as a member of *codom*⁴, the Cartesian product of *codom* with itself four times. Each 4-tuple of *codom*⁴ corresponds to the function that maps the *i*th member of *dom* to the *i*th element of the tuple. The procedure below accepts a member of *codom*⁴ and produces the corresponding transition table.

1	MakeTable := proc (<i>t</i> : : list)
2	local <i>T</i> , <i>j</i> ;
3	<i>T</i> := table () ;
4	for <i>j</i> from 1 to 4 do
5	<i>T</i> [op(dom[j])] := <i>t</i> [<i>j</i>] ;
6	end do ;
7	return <i>T</i> ;
8	end proc :

We now apply this procedure to each member of *codom*⁴.

```

> allTFs := []
allTFs := [] (13.85)
```

```

> codom4 := combinat[cartprod]([codom $ 4]) :
while not codom4[finished] do
    allTFs := [op(allTFs), MakeTable(codom4[nextvalue]())]
end do :
```

> *nops(allTFs)*
 20 736 (13.86)

The list **allTFs** now stores all 20 736 potential transition tables.

Recall, from Chapter 12, that the **Occurrences** command in **ListTools** can be used to count the number of 1s that appear on a tape.

> *ListTools[Occurrences](1,[1,B,B,B,1,1,1,0,1])*
 5 (13.87)

We need to place a limit on the number of steps the Turing machine can take and avoid getting stuck in an infinite loop because of a machine that does not halt. For this, we create a version of **Turing** specifically for this problem. It includes an extra argument for the limit on the number of steps and incorporates this limit into the while loop. We remove the argument for the initial tape and initial state, and instead set these to 0 and [B] in the procedure. Rather than returning the tape, this procedure will return the number of 1s appearing on the tape, assuming the machine halted. If it did not halt, we return -1.

```

1 BeaverTuring := proc (f : :table, maxstep)
2   local pos, state, tape, domain, Y, numsteps;
3   pos := 1;
4   state := 0;
5   tape := ['B'];
6   domain := {indices(f)};
7   numsteps := 0;
8   while [state, tape[pos]] in domain and numsteps < maxstep do
9     Y := f[state, tape[pos]];
10    state := Y[1];
11    tape[pos] := Y[2];
12    if pos=1 and Y[3]='L' then
13      tape := ['B', op(tape)];
14    elif pos=nops(tape) and Y[3]='R' then
15      tape := [op(tape), 'B'];
16      pos := pos + 1;
17    elif Y[3]='L' then
18      pos := pos - 1;
19    else
20      pos := pos + 1;
21    end if;
22    numsteps := numsteps + 1;
23  end do;
24  if numsteps < maxstep then
25    return ListTools[Occurrences](1, tape);
26  else
27    return -1;
28  end if;
29 end proc;
```

Now, we apply **BeaverTuring** to each of the transition tables in **allTFs** with a step limit of 100, keeping track of the number of 1s along the way.

```
> onesList := []
onesList := []
```

(13.88)

```
> for i to nops(allTFs) do
    onesList := [op(onesList), BeaverTuring(allTFs[i], 100)]
end do :
```

```
> max(onesList)
4
```

(13.89)

Using the **Tally** command from the **Statistics** package, we can see how many of the Turing machines produced tapes with four 1s.

```
> Statistics[Tally](onesList)
[-1 = 10952, 0 = 4184, 1 = 4876, 2 = 704, 3 = 16, 4 = 4]
```

(13.90)

This shows us that 4184 of the machines halted with no 1s on the tape, 4 machines halted with four 1s, and 10952 of the machines failed to halt.

We can see the four machines that produced four 1s as follows. The **SearchAll** command in **ListTools** will, given an element and a list, return the sequence of indices in the list that contain the element.

```
> ListTools[SearchAll](4, onesList)
7729, 7741, 9314, 9326
```

(13.91)

These are the transition functions for the four machines.

```
> for i in [ListTools[SearchAll](4, onesList)] do
    eval(allTFs[i])
end do
table ([0, B = [1, 1, R], 1, 1 = [2, 1, L], 0, 1 = [1, 1, L], 1, B = [0, 1, L]])
table ([0, B = [1, 1, R], 1, 1 = [2, 1, R], 0, 1 = [1, 1, L], 1, B = [0, 1, L]])
table ([0, B = [1, 1, L], 1, 1 = [2, 1, L], 0, 1 = [1, 1, R], 1, B = [0, 1, R]])
table ([0, B = [1, 1, L], 1, 1 = [2, 1, R], 0, 1 = [1, 1, R], 1, B = [0, 1, R]])
```

(13.92)

The busy beaver problem becomes very time consuming very quickly. Beyond $n = 2$, it is imperative to use more efficient approaches than was done here.

Exercises

Exercise 1. Construct the unit-delay machine described in Example 5 of Section 13.2.

Exercise 2. Construct a Maple procedure for simulating the action of a Moore machine. (See the prelude to Exercise 20 in Section 13.2 for the definition of a Moore machine.)

Exercise 3. Develop Maple procedures for computing the union of two nondeterministic finite-state automata and for computing the Kleene closure of a nondeterministic finite-state machine, as described in the proof of Theorem 1 of Section 13.4 of the text.

Exercise 4. Develop Maple procedures for finding all the states of a finite-state machine that are reachable from a given state and for finding all transient states and sinks of the machine. (See Supplementary Exercise 16 for definitions.)

Exercise 5. Construct a Maple procedure that computes the star height of a regular expression. (See Supplementary Exercise 11 for the definition of star height.)

Exercise 6. Construct a Turing machine that computes $n_1 - n_2$ for $n_1 \geq n_2$. Test that this Turing machine produces the desired results for sample input values.

Exercise 7. Construct a Maple procedure that simulates the action of a Turing machine that may move right, left, or not at all at each step.

Exercise 8. Construct a Maple procedure that simulates the action of a Turing machine that may have more than one tape.

Exercise 9. Construct a Maple procedure that simulates the action of a Turing machine with a two-dimensional tape. Represent a machine for multiplying integers and test it with your procedure.