

The logo for Metrowerks CodeWarrior CD. It features a vertical black bar on the left with a series of diagonal grey and black stripes. To the right of this bar, the text "Metrowerks CodeWarrior™ CD" is written vertically in white, sans-serif font.

Metrowerks CodeWarrior™ CD

CodeWarrior **Principles of Programming**

Because of last-minute changes to CodeWarrior, some information in this manual may be out of date. Please read all the Release Notes files that come with CodeWarrior for the latest information.

Metrowerks CodeWarrior Copyright ©1993-1995 by Metrowerks Inc. and its Licensors. All rights reserved.

Documentation stored on the compact disc may be printed by licensee for personal use. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from Metrowerks Inc.

Metrowerks, the Metrowerks logo and Software at Work are registered trademarks of Metrowerks Inc. CodeWarrior, PowerPlant, and PowerPlant Constructor are trademarks of Metrowerks Inc.

All other trademarks or registered trademarks are the property of their respective owners.

ALL SOFTWARE AND DOCUMENTATION ON THE COMPACT DISC ARE SUBJECT TO THE LICENSE AGREEMENT IN THE CD BOOKLET.

Canada and International

Metrowerks Inc.
1500 du College, suite 300
St. Laurent, QC
H4L 5G6 Canada

voice: (514) 747-5999
fax: (514) 747-2822

U.S.A.

Metrowerks Corporation
Suite 310
The MCC Building
3925 West Braker Lane
Austin, TX 78759-5321

voice: 512-305-0400
fax: 512-305-0440

Metrowerks Mail Order

voice: (800) 377-5416 or (419) 281-1802
fax: (419) 281-6883

World Wide Web site (Internet): <http://www.metrowerks.com>

Registration information (Internet): register@metrowerks.com

Technical support (Internet): support@metrowerks.com

Sales, marketing, & licensing (Internet): sales@metrowerks.com

AppleLink: METROWERKS

America OnLine: goto: METROWERKS

CompuServe: goto: METROWERKS

eWorld: goto: METROWERKS

Table of Contents

Chapter 1 A Road Map

1.1	Preview.....	14
1.2	Introduction to The Principles of Programming.....	15
	A Global View.....	15
	The Road Ahead.....	16
	Signs Along the Road.....	17
1.3	Computers.....	18
	A Low Level View.....	18
	Systems and Their Environments.....	19
	History of Programming and the Earth.....	21
1.4	Review Top Ten Things to Remember.....	24
1.5	Glossary.....	25
1.6	Problems.....	27

Chapter 2 An Overview

2.1	Preview.....	30
2.2	Problem Solving and the Computer.....	31
	Step 1 Problem Definition.....	32
	Step 2 Solution Design.....	33
	Step 3 Solution Refinement.....	34
	Step 4 Testing Strategy Development.....	35
	Step 5 Program Coding and Testing.....	36
	Step 6 Documentation Completion.....	37
	Step 7 Program Maintenance.....	37
	Using the Problem Solving Method.....	38
	Problems and Plans Dividing and Conquering.....	38
2.3	Break-Out Diagrams.....	39
	More on Break-Out Diagrams.....	41
2.4	Algorithms and Their Representations.....	45
	Modifying Algorithms.....	47
	Alternative Algorithms.....	52
	Equivalence of Algorithms.....	53
	Testing.....	54
2.5	Programming Languages.....	56
	Communicating Algorithms.....	56
	Basic.....	56
	Fortran.....	57
	Pascal.....	57
	Modula-2.....	58
	C.....	58

	Ada.....	59
	Other programming languages.....	59
2.6	Life Cycles Stages of Programming.....	60
2.7	Review Top Ten Things to Remember.....	63
2.8	Glossary.....	65
2.9	Problems.....	67

Chapter 3 Algorithms

3.1	Preview.....	74
3.2	What Are Algorithms?.....	75
	Algorithm Definition.....	75
	General Properties of Algorithms.....	75
	Desirable Attributes for Algorithms.....	77
3.3	Different Algorithm Representations.....	78
	Verbal Representations.....	80
	Algebraic Representations (formulas and expressions).....	82
	Tabular Representations (tables, arrays, and matrices).....	86
3.4	Data-Flow Diagrams.....	89
	Black Boxes vs. Glass Boxes.....	89
	General Data-Flow Diagrams.....	91
	Integer Data-Flow Diagrams.....	94
	Logical Data-Flow Diagrams.....	98
	Data Flow Black Boxes.....	101
	More on Data Flow Components and Diagrams.....	105
3.5	Flow of Control Diagrams.....	107
	Flowcharts.....	107
	Larger Flowcharts Using Subprograms.....	110
	Flowblocks.....	113
	Pseudocode.....	115
3.6	Review Top Ten Things to Remember.....	118
3.7	Glossary.....	119
3.8	Problems.....	121

Chapter 4 Algorithm Structure

4.1	Preview.....	128
4.2	Building Blocks for Structured Programming.....	128
	The Four Fundamental Forms.....	128
	Connecting Several Forms Sequences and Nests.....	131
	Deep Nesting.....	133
4.3	Different Algorithms, Same Problem.....	135
	Equivalent Algorithms.....	135

	Alternative Algorithms.....	137
4.4	Top-down Algorithm Design.....	140
	Job Description Example.....	144
	Change Maker Example.....	145
	A Game Example, Fifty.....	148
4.5	How Data and Algorithms Fit Together.....	152
	Structured Data.....	153
	Chili Recipe Example.....	155
	Card Trick Example.....	158
	Binary Conversion Example.....	159
	Guessing Game Example.....	160
4.6	Review Top Ten Things to Remember.....	164
4.7	Glossary.....	166
4.8	Problems.....	167

Chapter 5 Algorithm Behavior

5.1	Preview.....	176
5.2	Programs.....	176
	Data for Programs.....	176
	Actions In Programs.....	179
5.3	Sequence Forms.....	182
	More Programs Using the Sequence Form.....	183
5.4	Selection Forms.....	185
	Simple Selection Forms.....	185
	Proofs of Equivalence for Simple Selection Forms.....	187
	Larger Selection Forms.....	189
	Proofs of Equivalence for Larger Selection Forms.....	192
	Nested Selections.....	194
	Logical Conditions.....	198
	Using Logical Conditions to Simplify Selections.....	201
5.5	Repetition Forms.....	203
	The Repeat-Until Form.....	203
	The Disadvantages of Using the Repeat-Until Form...	206
	The While Loop Form.....	209
	Getting Insights Using Traces and Invariants.....	212
5.6	Invocation Forms.....	219
	Seconds Example.....	221
	De-militarize Time Example.....	222
5.7	Improving Programs.....	224
	Nested Loops and Selections.....	224
	Using Invariants.....	229
5.8	Review Top Ten Things to Remember.....	234
5.9	Glossary.....	235

5.10	Problems.....	236
------	---------------	-----

Chapter 6 Bigger Blocks

6.1	Preview.....	252
6.2	Using External Data and Files.....	252
	End-Of-File Markers.....	255
6.3	More Building Blocks.....	259
	The Select Form.....	259
	The For Form.....	262
6.4	Using the For Forms in Nests.....	265
	Nesting Fors in Fors.....	265
	Nesting Selections in Fors.....	269
	Creating Plots with For Nests.....	273
6.5	More Data Types.....	275
	The Character Type.....	275
	The Logical Type.....	279
6.6	Some General Problem-Solving Algorithms.....	281
	Bisection Finding Values.....	281
	Maximum Power Optimizing Power Output.....	284
	Solver Solving Equations.....	287
6.7	Review Top Ten Things to Remember.....	290
6.8	Glossary.....	291
6.9	Problems.....	292

Chapter 7 Better Blocks

7.1	Preview.....	300
7.2	Subprograms.....	300
	How to Simplify Large Programs.....	300
	What are Parameters?.....	305
	Data Space Diagrams.....	309
7.3	Parameter Passing.....	312
	Passing Parameters In and Out.....	312
	Special Cases.....	316
	Some Examples.....	321
7.4	Bigger, Leaner, and Meaner Programs.....	329
	Using Structure Charts.....	329
	Contour Diagrams.....	333
	Parameter Crossing A Common Mistake.....	336
	Minimizing Coupling, Maximizing Cohesion.....	338
	Deeply Nested Subprograms.....	340
	Dates Example.....	341
7.5	More Types of Subprograms.....	343
	Recursion Self-Referencing Subprograms.....	343

	Functions.....	345
	Modules.....	346
7.6	Review Top Ten Things to Remember.....	348
7.7	Glossary.....	350
7.8	Problems.....	351

Chapter 8 Data Structures

8.1	Preview.....	358
8.2	Arrays.....	358
	What are Data Structures Anyway?.....	358
	One-Dimensional Arrays.....	360
	Performing Operations on One-Dimensional Arrays..	361
	Using One-Dimensional Arrays in Algorithms.....	364
	Two-Dimensional Arrays.....	367
	Performing Operations on Two-Dimensional Arrays	370
	Matrix Multiplication.....	373
	N-Dimensional Arrays.....	375
8.3	Records.....	377
	What are Records?.....	377
	Accessing Record Components the Dot Notation...	380
	Combining Arrays and Records.....	382
8.4	Sets.....	384
	What are Sets?.....	384
	A Difficult Sets Example (optional).....	386
8.5	Data Structure Building Tools.....	392
	Dynamic Variables.....	392
	Pointers.....	392
8.6	Abstract Data Types.....	397
	Strings.....	399
	Stacks.....	401
	Queues.....	405
	Trees.....	407
8.7	Review Top Ten Things to Remember.....	408
8.8	Glossary.....	409
8.9	Problems.....	410

Chapter 9 Algorithms To Run With

9.1	Preview.....	416
9.2	Sorting.....	416
	General Sorting Strategies.....	416
	The Four Methods of Sorting.....	418
	Count Sort.....	419
	Count Sort Analysis.....	421

	Swap Sort.....	422
	Swap Sort Analysis.....	424
	Select Sort.....	425
	Select Sort Analysis.....	427
	Insert Sort.....	428
9.3	More Complex Sorting.....	428
	Improving Sorts.....	428
	Recursive Sorts.....	432
9.4	Searching.....	435
	Linear Searching.....	436
	Binary Searching.....	437
	Pattern Matching.....	438
9.5	Implementing Abstract Data Types.....	441
	Stacks.....	441
	Implementing Stacks with Arrays.....	441
	Implementing Stacks with Pointers.....	444
	Queues.....	445
	Implementing Queues of Arrays.....	446
	Implementing Queues with Pointers.....	448
	Trees.....	449
9.6	Review Top Ten Things to Remember.....	451
9.7	Glossary.....	453
9.8	Problems.....	454

Chapter 10 The Seven-Step Method

10.1	Preview.....	458
10.2	The Seven-Step Method and Applications.....	458
	Step 1 Problem Definition.....	458
	Problem Definition Application.....	460
	Step 2 Solution Design.....	461
	Solution Design Application.....	461
	Step 3 Solution Refinement.....	464
	Solution Refinement Application.....	464
	Step 4 Testing Strategy Development.....	466
	Development of Testing Strategy Application....	468
	Step 5 Program Coding and Testing.....	471
	Step 6 Documentation Completion.....	471
	Documentation Completion Application.....	472
	Step 7 Program Maintenance.....	474
	Program Maintenance Application.....	474
10.3	An Advanced Case Study Building a Text Index.....	476
	Step 1 Problem Definition.....	476
	Step 2 Solution Design.....	476
	Step 3 Solution Refinement.....	478
	Binary Search Tree Unit.....	483
	Queues Unit.....	486

Step 4	Development of Testing Strategy.....	486
Step 5	Program Coding and Testing.....	488
Step 6	Documentation Completion.....	488
Step 7	Program Maintenance.....	489
10.4	Review Top Ten Things to Remember.....	490
10.5	Glossary.....	491
10.6	Problems.....	492
	Level 1 — Getting Started.....	492
	Level 2 — Getting Organized with Subprograms.....	495
	Level 3 — Getting Fancier with Parameters.....	498
	Level 4 — Getting Your Wings with Units.....	501

Appendix: Solutions

Chapter 1	Solutions 1-3.....	508
Chapter 2	Solutions 1-3.....	508
Chapter 3	Solutions 1-7.....	510
Chapter 4	Solutions 1-3.....	511
Chapter 5	Solutions 1-3.....	512
Chapter 6	Solutions 1-3.....	514
Chapter 7	Solutions 1-3.....	515
Chapter 8	Solutions 1-3.....	516
Chapter 9	Solutions 1-3.....	517

Index	519
--------------------	-----

Preface

The purpose behind The Principles of Programming is to provide the reader/student with a structured method to solve problems using a computer. Since the Principles book is language-independent (it provides the steps and tools to solve problems without centering on a specific computer programming language), a companion book, Programming Practice: Pascal, is also provided on this CD.

While using this book to learn the principles of programming, you may find that there are too many examples and that some topics have been over-explained. You are encouraged to select only the topics and examples that best suit your needs and work habits.

Chapter 1 A Road Map;

This is a text book—whose purpose is to teach you about computer programming. Before we can begin, this book’s objectives, and the path it will take, must be outlined. Thus, Chapter 1 introduces this book and also provides a brief introduction to computers.

Chapter Overview

1.1	Preview.....	14
1.2	Introduction to The Principles of Programming.....	15
	A Global View.....	15
	The Road Ahead.....	16
	Signs Along the Road.....	17
1.3	Computers.....	18
	A Low Level View.....	18
	Systems and Their Environments.....	19
	History of Programming and the Earth.....	21
1.4	Review Top Ten Things to Remember.....	24
1.5	Glossary.....	25
1.6	Problems.....	27

1.1 Preview

This is a text book—it is intended to teach you about computer programming. Learning is like a journey and, with any journey, it is helpful to know something about the terrain before setting out, for you can make proper preparations and know what will be expected of you. This chapter provides you with a road map of your journey. The coming pages will present a great deal of detailed matter and it is all too easy to get mired down in this detail and lose sight of how it fits in with the whole. Here is a very general view that will allow you to gain a picture of where the details are leading.

This chapter is presented in two major sections, starting with an introduction to The Principles of Programming which contains the following subsections:

- *A Global View* describes the layout of the text book's material in very general detail. This sub-section also offers an idea of the way in which Principles of Programming should be read and studied.
- *The Road Ahead* gives a very brief synopsis of each chapter so that you can visualize how all the chapters fit together. This provides a route that will take you from an introduction to computers (the last section in this chapter) to being able to write large useful programs.
- *Road Signs* describes the common pattern on which each of the chapters are based, tells you how to read the road signs, and gives a brief explanation of some of the visual aids that will be used throughout the book.

As you can see, the first section of this chapter is organized to go from the very general to the particular. This is no accident. A major technique presented in this book is that of starting with an overview and then filling in the details. This technique is called the “top-down approach” and you will be meeting it again and again on your journey.

Finally, a section introducing computers is provided. Computers are the physical machines that execute (or “run”) programs. These machines are capable of performing a relatively small number of different operations. Such operations, which can be combined to carry out complex tasks, include:

- Arithmetic operations, such as the addition, subtraction, multiplication, and division of numbers.
- Comparison operations, such as determining whether one number is greater than another.
- Input and output operations, such as accepting data from a keyboard or displaying data on the screen.

It is not necessary to know much about the detailed construction of computers in order to use, or program, them. At the end of this chapter, we provide only a brief introduction into the structure of a computer and a computing system. This section also ends with a brief glance at history. This view, of both computing

and world history, should put the brief lifetime of computers and computing into its proper perspective.

1.2 Introduction to The Principles of Programming

A Global View

The goal of this book is to teach you how to program computers. At the beginning, no previous knowledge of computers is assumed although, these days, it is almost impossible to grow up without gaining some familiarity with the vocabulary and concepts of computers. Nevertheless, to make sure that everybody starts at the same level, we have assumed nothing and describe everything we need. If you come to a section that you think you already know, please do not skip it, there may well be an important nugget of information that is new to you.

Learning to write computer programs is very much like learning any skill—you must first understand the underlying principles upon which the craft is based and then practice, practice, practice. For example, you can read all the books ever published about riding a bicycle but, until you actually get on one and start pedaling on your own, you will not know how to ride a bicycle. The same applies to computer programming! Here, we provide the first book, the *Principles of Programming*, which treats computer programming in general. A second book, the *Practice of Programming* describes the application of the principles to a particular language—the way in which a computer program is expressed so that it can be used on a computer.

We can make an analogy with natural language. There are certain principles of language, such as the distinction between nouns—names of things— and verbs—names of actions— that apply to all human languages. However, the way in which these principles are expressed and the detailed rules that apply to them varies considerably from language to language. For instance, the rules of French are different from the rules of German and they are both different from the rules of English. The situation is similar with programming languages; there are many different languages each with its own set of rules. For this reason, there is one *Principles* and several versions of *Practice*, one for each of the programming languages we cover.

This book emphasizes the adage, “it’s best to learn through examples.” Keeping this in mind, each chapter uses many examples to teach the same point. For this reason, you may find the number of examples overwhelming and you should feel free to pay attention to only the examples which you feel best describe the point that is being illustrated.

The Road Ahead

In this section, we give a brief summary of what is in each chapter of the Principles of Programming. A synopsis of each chapter is presented so that you can visualize the road ahead.

- **Chapter 2, Problem Solving and Computers**, contains an overview of problem solving with a computer. It introduces the idea of formulating the solution to a problem into a set of precisely defined actions—an *algorithm*. It also emphasizes that problem solving without method is doomed, and introduces a seven step problem solving method that will help you design, write, and run computer programs. An algorithm is expressed in a form that can be communicated with both people and computers through the use of a *programming language* and you are shown examples an algorithm expressed in different programming languages.
- **Chapter 3, Algorithms**, gives a more detailed introduction to algorithms and discusses their necessary and desirable properties. This chapter shows that algorithms can be represented in different ways: in English sentences, as formulas, tables, in various graphical forms, or in *pseudocode*—a stylized English that is similar to a programming language.
- **Chapter 4, Algorithm Structure**, is an introduction to the construction of algorithms using four basic blocks—*forms*—that are interconnected in a simple way. The Four Fundamental Forms, Sequence, Selection, Repetition, and Invocation are studied in some detail, including the way in which they can be put together to create complete algorithms. A number of examples based on everyday experiences is used to illustrate the various forms.
- **Chapter 5, Algorithm Behavior**, discusses the dynamic behavior of algorithms. A representation of the data manipulated by programs is used to illustrate the concept of *variables*. The execution of various algorithms is *traced* step by step in order to show how variables operate and how their data values are changed.
- **Chapter 6, Bigger Blocks**, expands on the concepts introduced in Chapter 5. In particular, this chapter shows how programs can be made more general by using data obtained from outside the computer. The construction of more complex algorithms, always based on interconnects of the Four Fundamental Forms, is also demonstrated.
- **Chapter 7, Better Blocks**, describes in detail the concept of *subprograms*. Invocations are examined to show how the various steps of the algorithm are executed and the various ways in which data are passed between calling program and called subprograms.
- **Chapter 8 Data Structures** shows that data are usually more than simple values. Most data are more complex than that and have some structure. In this chapter, three basic data structures are introduced:

arrays, records, and sets. Arrays are groupings of homogeneous items that are accessed by their position in the group. Records are groupings of possibly heterogeneous items that are accessed by using names for the individual items. Sets are collections of distinct homogeneous items whose only property is whether they are in the group or not. This chapter also explores the concept of an *abstract data type*, seen as a class of data, defined through its values and operations.

- **Chapter 9 Algorithms to Run With:** The primary purpose of this chapter is to provide more extensive examples of how to use the data structures introduced in Chapter 8. At the same time, the chapter introduces a number of standard algorithms that are used frequently in programs. These algorithms are particularly concerned with sorting—the arranging of data into a specific sequence—and searching—the retrieval of specific data from a collection. Other ways of arranging data (like stacks, queues, and trees) are also described. Finally, a way of building dynamic data structures by way of pointers, is introduced.
- **Chapter 10 Seven Step Method:** In this final chapter, we return to the seven step method for solving a problem on a computer that was introduced in Chapter 2. The method is completely revised with a complete example illustrating each step. Another complete example is developed from start to finish..

Signs Along the Road

Although each chapter constitutes a stage in our learning journey, introducing new topics and conventions while expanding on others, the same basic pattern is followed:

1. Each chapter begins with a *Preview* that gives a summary of the material that will be presented in the chapter. This will give you an idea of what to expect and introduce you to the major concepts in the chapter.
2. Next, the actual material of the chapter is provided, where the major topics are divided into sections. Since the emphasis through each chapter is that we learn best by example, each section has many examples illustrating the topic being presented.
3. Following this, a *Review* of the material contained in the chapter is presented in a “top ten” format. This will serve to remind you of what you have learned and nudge you to go back and reread anything that you have forgotten.
4. Next, a *Glossary* of the major terms introduced in that chapter is provided.
5. Finally, each chapter ends with a set of *Problems* to solve. This is one of the most important parts of each chapter. There are always a few problems for which the solutions are provided in an appendix at the

end of the book (Solutions Appendix). You are encouraged to try your hand at these before looking at the solutions. In any case, programming is an intensely practical skill that can only be acquired by practice. Remember: “Practice makes perfect!”

Apart from following a similar pattern, each chapter uses many *signs* to visually illustrate important points and concepts. The most frequently used visual aids are note (or tip) boxes. An example is provided below.

Note: This is a tip or note box. Inside this box, important information and tips can be found.

This is a note in margin which directs you to useful information.

In addition to tip boxes, when a paragraph briefly touches upon a subject which is covered in more detail in another chapter, a reference is provided in the left margin.

1.3 Computers

A Low Level View

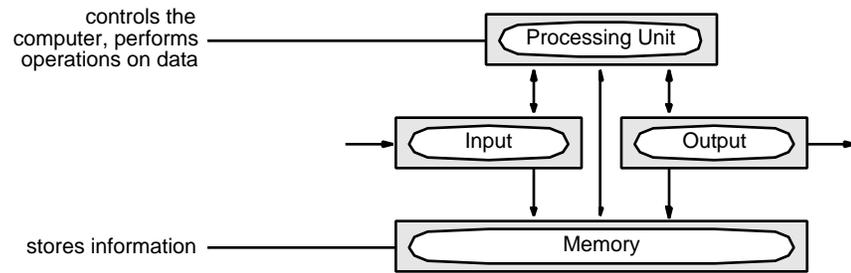
As you progress, you’ll use the computer to do more and more complex things. It’s always nice and sometimes even actually useful to know more about the tools you use, so we’ll take a look here at *computers*.

Most computers, big or small, have the same basic capabilities: they can input (“read”) data from an external device and store it for later use, they can manipulate the stored data by executing the instructions that constitute a program and they can output (“write”) results onto an external device.

You do not need to know many details on computers in order to program them, just as you don’t need to know many details about carburetors or transmissions in order to drive cars. Of course, knowledge of these details may be useful, but is not always necessary. It is only when trying to make the best and most efficient use of computers that these details become important.

Computers differ in details, but they have a common structure. They can be viewed as comprising four units: a processing unit, a memory unit, an input unit and an output unit. These units are not always physically separate, but it is useful to view them functionally this way as in Figure 1.1.

A *Memory Unit* stores information that can be retrieved, modified and processed. It not only stores the data that the program will process, but also the actual instructions of the program. This concept of a “stored program” is extremely important, for it makes the computer into a general purpose machine where changing the program in memory literally changes the behavior of the machine.

Figure 1.1 A simple computer

The memory can be thought of as a collection of pigeon holes, each referenced by a number, its “address”, just as a house number is used to reference a particular house in a street. Each of these pigeon holes can contain an item of data or an instruction from the program. If an item of data or an instruction is too big for a single pigeon hole, it is split across a group of adjacent pigeon holes.

The *Processing Unit* has two distinct functions:

- To “control” the behavior of the entire computer, and
- To perform operations on data such as arithmetic operations and comparisons of values.

The part of the processing unit that performs the operations on the data is often called an ALU, arithmetic and logic unit. The processing unit maintains control through the use of a few special purpose pigeon holes called *registers*. The control part has a program register, which contains the address of the program’s instruction that is currently being executed and an instruction register containing this instruction. The ALU also contains working registers where the actual operations on the data are performed.

The *Input* and *Output Units* serve either to input information (from keyboards or from files on disks) into the memory, or to output information (to printers or graphic displays) from the memory.

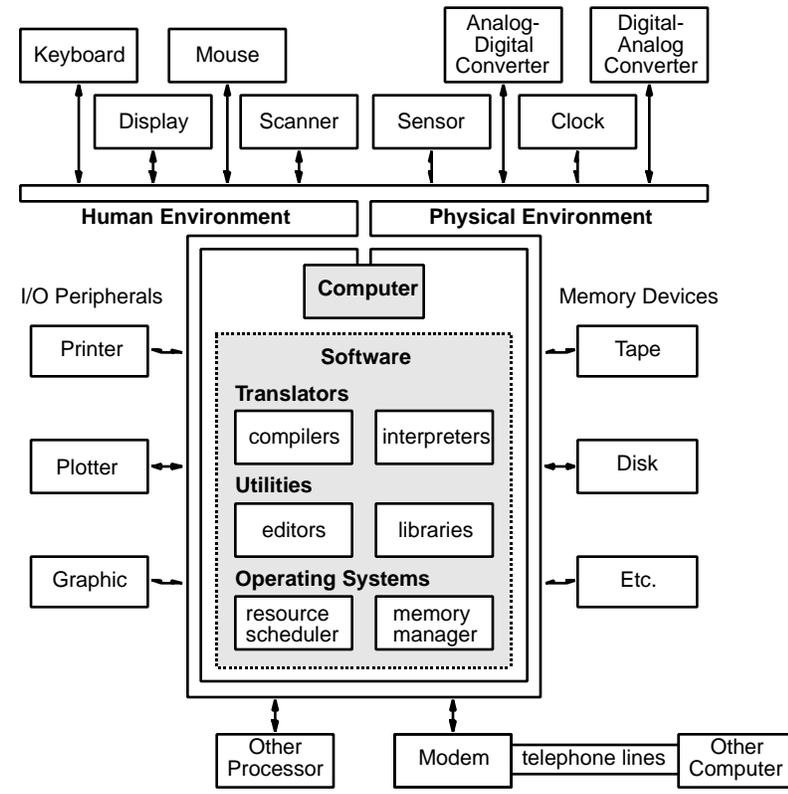
Systems and Their Environments

If the physical computer as represented in Figure 1.1 were all you had, you would find it very hard to do much with it! In order to be able to use a computer, you need a number of programs to make it work: these programs constitute the *software* part of the computer without which the *hardware* does not do anything. Also, the user of a computer is the one to tell the computer what task to do. For these reasons, a computer is not considered by itself, but always in relation to its environment.

In order to put computers into a proper perspective, let’s describe a *Computer System*. The system shown in Figure 1.2 is a large one so that we can introduce a lot of new terms; smaller personal computers are, of course, much simpler. In the large computing system, notice that the computer is one small component (in the center). This is generally referred to as the *central processor unit*, or *CPU*.

Here, we look at computing from the view of a computer interacting with both the human and the physical environment.

Figure 1.2 A complete computing system



Human Environments, such as in a business computer system, are shown to the left of Figure 1.2. People can supply data to the computer through input devices such as keyboards, mice and scanners that can, for example, sense bar-coded data, or sensors that can be touched on appropriate places of a display. People receive the results from the computer on video display terminals, printers, plotters or through audio speakers. These and other input-output, “I/O”, devices connect to the system through an input-output signal cable usually referred to as a “bus”.

Physical Environments, such as those found in a manufacturing plant, are shown at the top right of Figure 1.2. The physical environment communicates through converters that transform physical quantities (distance, pressure, time) from their continuous (*analog*) values into equivalent discrete (*digital*) values. These converters are known as *analog-to-digital* (or *A to D*) converters. The analog values are obtained from transducers, such as a sensor measuring temperature. Similarly, *digital-to-analog* (*D to A*) converters transform digital values into continuous values that could, for example, control a manipulator to regulate the source of heat.

The *memory* of a computer can be extended by auxiliary memory devices as shown to the right of Figure 1.2. Such devices include magnetic tape, disk, semiconductor memory, bubble memory, and any other type which can be “plugged onto” the memory bus. Other auxiliary devices may be “hung onto” this system, as shown on the bottom of Figure 1.2. These devices include other computers and “modems”, which are connections to communication lines such as the public telephone system.

Hardware is the term that refers to these many physical devices of the system. Since people and mechanical devices operate thousands of times slower than electronic devices, the computer can service all of the devices “polling” them hundreds of times a second to see, for example, if they have data ready for input. Thus, the system appears to operate all of the devices simultaneously, much like a busy chef cooking with several frying pans at once.

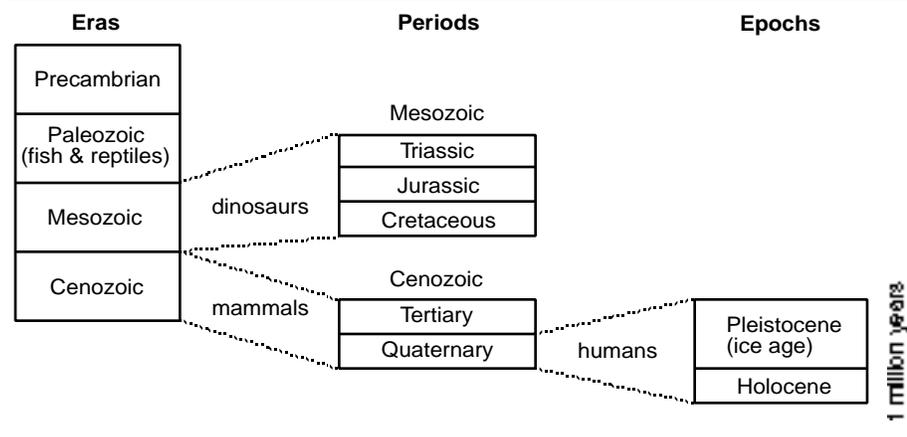
Software is the term that refers to all of the programs required to operate the system. This includes the translators (to convert high level languages into machine languages), utility programs (for convenience of editing, program storage and retrieval, and so on), and operating programs (for loading application programs, scheduling resources, and managing memory).

History of Programming and the Earth

Computers have been commercially available for only about forty years, so their history is recent. To put this history into perspective, let’s first view the complete history of the earth and end with the history of computing. This will allow us to show an example of a *break-out diagram*, which will be explained in more detail in Chapter 2.

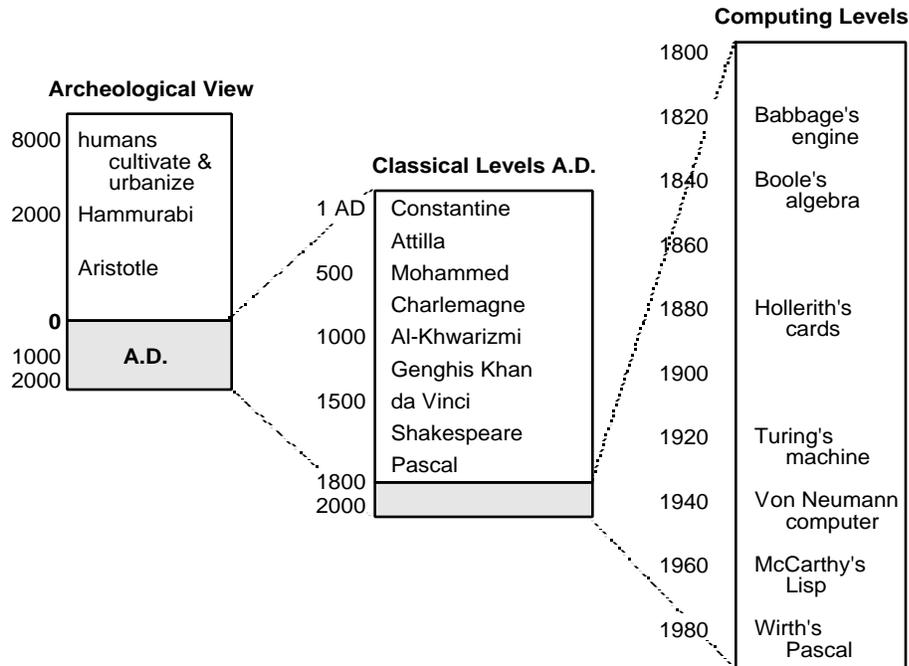
The early history of the earth could start with the geological levels called eras, which break up into periods, which in turn break up into epochs as shown in Figure 1.3.

Figure 1.3 Earth history Geological break-out diagram



Recent history is briefly shown in Figure 1.4. You may wish to add your own favorite historical events. Notice that the classical levels (the last 2000 years) are a very small part of the history of the earth. An even smaller part is the history of computing shown on the right-hand side of Figure 1.3.

Figure 1.4 Earth history, More recently



In the classical levels of Figure 1.4, going from AD 1 to 1800, we find a number of persons who had an influence on computing. For instance...

- Al-Khwarizmi, a ninth century mathematician, studied sets of rules which are now called algorithms (named after him).
- Blaise Pascal, a French mathematician and philosopher, designed the first mechanical adding machine in 1642.
- Based on Pascal's machine, Gottfried Wilhelm von Leibniz, a German mathematician, created a mechanical calculating machine that could perform both addition and multiplication in 1694.
- In the more recent period going from 1800 to 2000, we find others who influenced modern computers. In 1801, Joseph Jacquard, a Frenchman, developed a loom for weaving cloth with patterns controlled by punched cards.
- In the 1840s, George Boole discovered that the symbolism of mathematics could be applied to logic. His algebra forms a basis for both the hardware (interconnections of components) and also the software (programs with compound conditions).

- Charles Babbage, an English mathematician, designed in the 1850s the first general-purpose “analytical engine” with a sequential control using rotating wheels. His machine could not be completed, however, because of technological problems.
- Herman Hollerith, an American engineer, developed around 1890 punched card machines from Jacquard’s idea and applied it for processing census data. The size of punched cards, now obsolete, is based on his choice, the size of a dollar bill of that time.
- Alan Turing introduced in 1937 a conceptual model of computability and a theoretical device called a Turing Machine, which was used to define limits on what is computable.
- John Von Neumann, around 1945, introduced the concept of a stored program, where instructions and data are both stored in memory.

Later developments are too recent to be objectively put into historical perspective. For example, it was initially believed that the first electronic digital computer was designed and built by John W. Mauchly and J. Presper Eckert around 1945. However, after a court case ending in 1973, the father of the electronic computer was determined to be John V. Atanasoff who had created a small electronic special purpose computer in 1939 at Iowa State University. However, at the same time in Germany, Konrad Zuse was developing computers that were more general than Atanasoff’s!

1.4 Review Top Ten Things to Remember

1. The *Principles of Programming* is the first in a series of books. It teaches computer programming in general while a second book, the *Practice of Programming*, describes the application of the principles to a particular language.
2. Most computers, big or small, have the same basic capabilities: they can input (“read”) data from an external device and store it for later use, manipulate stored data by executing the instructions that constitute a program and they can output, or write, results onto an external device.
3. Computers can be viewed as comprising four units: a processing unit, a memory unit, an input unit and an output unit.
4. The *Processing Unit* has two distinct functions: to “control” the behavior of the entire computer and to perform operations on data such as arithmetic and comparisons between values. The part of the processing unit that performs the operations on the data is often called an ALU, arithmetic and logic unit.
5. A *Memory Unit* stores information that can be retrieved, modified and processed. It not only stores the data that the program will process, but also the actual instructions of the program.
6. Memory can be thought of as a collection of pigeon holes, each referenced by a number—an address. Each of these pigeon holes can contain an item of data or an instruction from the program.
7. The *Input and Output Units* serve either to input information (from keyboards or files on disks) into the memory, or to output information (to printers or graphic displays) from the memory.
8. *Hardware* is the term that refers to the many physical devices of a computer system.
9. *Software* is the term that refers to all of the programs required to operate a computer system.
10. Computers have been commercially available for only about forty years, so their history is recent; too recent to be objectively put into historical perspective.

1.5 Glossary

A to D converter: (Analogue to Digital converter) A physical device that takes an analog or continuous value (for instance, the intensity of an electric current) and converts it into a digital (numerical) value.

ALU: Arithmetic Logic Unit, a component of the computer Central Processing Unit that executes arithmetic and comparison operations.

Analog value: A directly measurable physical value (temperature, current voltage, pressure, and so on).

Bus: A signal transmission cable making it possible to interconnect various devices.

Compiler: A piece of software used to translate a program written in a high-level programming language into machine language.

CPU: Central Processing Unit, the heart of the computer that controls the behavior of its other components and performs the operations on the data.

D to A converter: Digital to Analog converter, a device to convert digital (numerical) values to their physical equivalent (current voltage, temperature, and so on).

Editor: A utility program that helps a user prepare data or programs for a later operation.

Execute: Perform the actions associated to a program.

Hardware: The collection of physical devices making up a computer system.

Input/Output: The devices designed to send external information into the computer memory and to produce external information from the memory.

Instruction register: A CPU register that holds the machine instruction being executed.

Machine Language: A programming language that is used directly by a computer; synonymous with low-level language.

Memory: The component of a computer used to store information (data and program instructions).

Modem: A device used to connect a computer to other computers through telephone lines.

Processing unit: CPU.

Program register: A CPU register containing the address of the next machine instruction to be executed (also called program counter).

Register: Special purpose computer memory cells used to store the pieces of information on which a machine instruction operates.

Utility program: A computer program in general support of the

processes of a computer, for instance,
an editor, a sort program.

1.6 Problems

1 Human Environments

Which of the following are considered human environments when it comes to the interaction between computers and humans.

- a. Keyboard
- b. Analogy-Digital Converter
- c. Scanner
- d. Mouse
- e. Clock

2 Physical Environments

Which of the following are considered physical environments when it comes to the interaction between computers and transducers.

- a. Display
- b. Sensor
- c. Digital-Analog Converter
- d. Scanner
- e. Clock

3 Influential People

Match the following list of names with their contributions to computing. Note: there is one more contribution than there are contributors!

George Boole	Al-Khwarizmi	Charles Babbage
John Von Neumann	Blaise Pascal	Alan Turing

- a. developed a machine used to define the limits on what is computable.
- b. studied sets of rules which are now called algorithms.
- c. designed and built the first electronic digital computer.
- d. discovered that symbolism of mathematics could be applied to logic.
- e. designed the first general-purpose “analytical engine”.

- f. introduced the concept of a stored program where instructions and data are both stored in memory.
- g. designed the first mechanical adding machine.

4 Memory

List the three functions of the Memory Unit. Besides data, what does the Memory Unit store? What happens if an item of data is too big for a single “pigeon-hole” of memory?

5 Hardware vs. Software

What is the difference between hardware and software? Give examples of both.

6. Think Big

Suppose that entire computers became as small as bugs (roaches or integrated circuits), and had the ability to move around and manipulate things (carry, measure, cut, and so on). Write a brief essay indicating possible uses and potential benefits of such programmable bugs (PRUGS).

For example, many functions could be performed differently. Lawns could be maintained, not by mowing, but with an "army" of PRUGS roaming around randomly, measuring each individual blade of grass and cutting it off at a precise length.

7. Think Well

Write another brief essay describing the potential negative consequences of the PRUGS in the previous problem.

8. Do Justice to History

Select some topic in the history of computing (person, machine, program, era, machine, and so on) and investigate it thoroughly. Relate this to other similar topics.

Chapter 2 An Overview

This chapter provides an overview of problem solving with a computer, showing a brief “bird’s-eye” view of the general ideas, and how they are related.

Chapter Overview

2.1	Preview.....	30
2.2	Problem Solving and the Computer.....	31
	Step 1 Problem Definition.....	32
	Step 2 Solution Design.....	33
	Step 3 Solution Refinement.....	34
	Step 4 Testing Strategy Development.....	35
	Step 5 Program Coding and Testing.....	35
	Step 6 Documentation Completion.....	37
	Step 7 Program Maintenance.....	37
	Using the Problem Solving Method.....	38
	Problems and Plans Dividing and Conquering.....	38
2.3	Break-Out Diagrams.....	39
	More on Break-Out Diagrams.....	41
2.4	Algorithms and Their Representations.....	45
	Modifying Algorithms.....	47
	Alternative Algorithms.....	52
	Equivalence of Algorithms.....	53
	Testing.....	54
2.5	Programming Languages.....	56
	Communicating Algorithms.....	56
	Basic.....	56
	Fortran.....	57
	Pascal.....	57
	Modula-2.....	58
	C.....	58
	Ada.....	59
	Other programming languages.....	59
2.6	Life Cycles Stages of Programming.....	60
2.7	Review Top Ten Things to Remember.....	63
2.8	Glossary.....	65
2.9	Problems.....	67

2.1 Preview

Using a computer to solve a problem frequently requires *programming*—the formulation of a plan for precisely defined actions to be performed by the computer to obtain the solution to the problem. Such a plan is an *algorithm*. In this chapter, we introduce a helpful method for solving a problem with a computer. We also show ways of breaking up a problem into smaller subproblems. Of course, in this introductory chapter, we will consider only some simple problems, such as the computation of part of a weekly payroll.

Algorithms are plans for performing actions. Common examples of algorithms are:

- Directions for getting somewhere: Directions for Getting to Fred's Pizza Parlor
 1. Take Route 116 until you come to the T-junction at Route 9.
 2. Turn onto Route 9 and go about three-quarters of a mile and it will be on your left. You can't miss it!

- A cooking recipe: Cream Horns

Roll out thinly puff or flaky pastry into an oblong approximately 12 inches long, and cut into 1-inch strips. Moisten one edge of each strip and roll the strip around a cream horn tin, starting at the pointed end of the tin and overlapping the pastry very slightly. Bake in a hot oven until crisp—10–15 minutes. Slip the horns off the tins. When they are cold, fill them with whipped cream and dredge with confectioners' sugar.

- A crochet pattern:

Row 2(afghan st): 1 sc in first dc, * (draw up loop in next st and retain loop on hook) 5 times (6 loops on hook), draw up loop in next st and draw this loop thru 1st loop on hook forming an upright st or bar (yo and thru 2 loops) 5 times *. There are 6 bars and 1 loop on hook. ** Retain loops on hook and draw up loop in each of next 5 bars (6 loops on hook), draw up loop in next st and thru first loop on hook (yo and thru 2 loops) 5 times. Rep from ** twice. Insert hook in 2nd bar, yo and thru bar and loop on hook (1st bound off). Bind off 4 more sts, 1 sc in next st. Rep from *, ending bind off 5 sts, sl st in top of turning ch. Ch 1, turn.

These algorithms require varying degrees of technical knowledge in order for someone to carry them out, or execute them. The first algorithm is expressed in a language that almost everybody can understand. The second requires some knowledge of pastry-making to understand, while the third requires considerable experience with crocheting and reading crochet patterns.

There is also a difference in the level of detail provided in each algorithm: the first is at a *high-level* and provides almost no detail, whereas the third is at a *low-level* and provides much detail—the making of each individual stitch is defined. Algorithms for use on computers are called *program* and can also be expressed at varying levels of detail. In this chapter, we will consider several examples of representations of simple algorithms.

Programming Languages are methods for representing and communicating algorithms, both to people and to computers. As with the algorithms we just viewed, different programming languages provide varying levels of detail. We will be mainly concerned with languages convenient for people, referred to as *high-level programming languages*. A *low-level programming language*, that expresses an algorithm in terms of its primitive operations, is so full of small details—like the crocheting pattern—that it is difficult for people to understand. Luckily, computers can easily translate an algorithm expressed in a high-level programming language into a low-level computer language. In this chapter, we will present a payroll program in a few different languages to provide an insight into the similarities and differences among programming languages.

2.2 Problem Solving and the Computer

This book is about programming, and it is important to know that programming exists solely to solve problems on a computer. Programming is, first and foremost, a *problem solving* activity.

The mathematician George Polya, an authority on problem solving, has divided¹ problem solving into a four step activity:

1. *understanding the problem*: This first step can be very difficult but is *absolutely crucial*. Although it happens all of the time, it is foolish to attempt to answer a question that is not fully understood. In general, one must find the given data, what is the unknown (the required result) and what is the relationship that links the given data with the unknown. It is also important to verify that the given information is sufficient to solve the problem.
2. *devising a plan*: Once the problem is understood, one must devise the plan of action to solve the problem. A plan is formed by proceeding from the data to the result according to the relationship that links them. If the problem is trivial, this step will not require much thinking. General techniques include the following:
 - Finding if there are known similar problems,
 - Reshaping the original problem so that it resembles a known problem

¹ George Polya, *How to Solve It*, Princeton, New Jersey: Princeton University Press, 1945.

- Restricting the problem to a shorter solvable form,
 - Generalizing a restricted problem, and
 - Finding existing work that can help in the search for a solution.
3. *executing the plan*: Once the plan is defined, it should be followed completely. Each element of the plan should be checked as it is applied. If parts of the plan are found to be unsatisfactory, the plan should be revised.
 4. *evaluating*: Finally, the result should be examined in order to make sure that it is valid and that the problem has been solved.

As we all know, there is usually more than one correct way of solving a problem. When several people are faced with the same problem, it is likely that each individual will reach a solution in a different manner. However, to be efficient, a person must adopt a systematic method of problem solving. When using a computer to solve a problem, the use of a systematic method is crucial. Without one, people tend to rush over the steps and find out too late that they should have spent more time in preparation and planning.

Based on the Polya problem solving method, we introduce here a seven-step problem solving method that can be adapted to each person's own problem solving style. This method is closely related to what is known as the *software life cycle* (the various stages in the life of a program).

The seven steps of the method are:

1. Problem Definition
2. Solution Design
3. Solution Refinement
4. Testing Strategy Development
5. Program Coding and Testing
6. Documentation Completion
7. Program Maintenance

Step 1 Problem Definition

Polya's first step is to fully understand the problem. The initial description of a problem is usually vague: it must be made precise. The poser of the problem—the user—and the problem-solver must work together in order to ensure that both understand the problem. This should lead to complete *specifications* of the problem including the precise definition of the input data—the given data—and of the desired output—the result.

Step 2 Solution Design

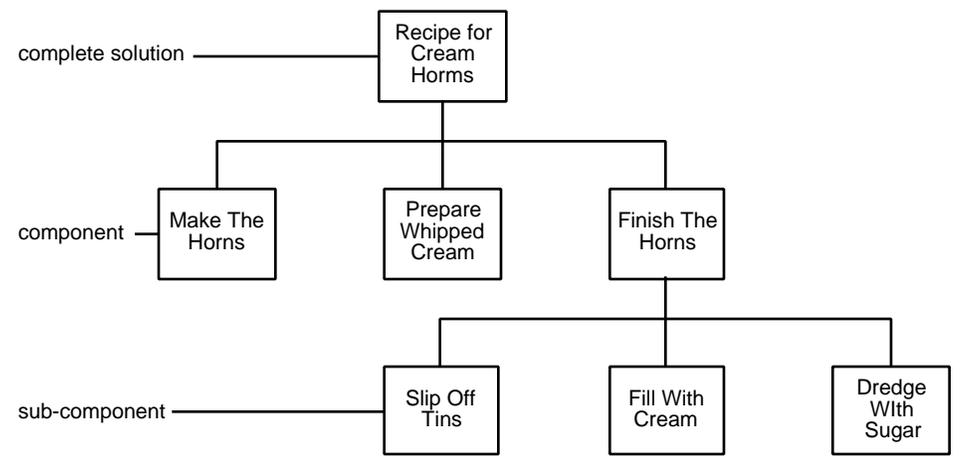
In this step, we want to define the outline of a solution. Starting from the original problem, we divide it into a number of subproblems. These subproblems, being necessarily smaller than the original problem, are easier to solve and their solutions will be the components of our final solution. The same “divide and conquer” method is applied in turn to the subproblems until the whole problem has been converted into a plan of well-understood steps.

To illustrate this step, let’s take a simple non-computer problem such as the recipe for Cream Horns given in the previous section. Based on this recipe, we can break down the problem into three subproblems:

1. Make the horns,
2. Prepare whipped cream, and
3. Finish the horns.

In turn, the subproblems themselves can be divided until we reach extremely simple subproblems. One of the products of this step is a chart that illustrates the structure of our solution (Figure 2.1). This chart is called a *structure chart* because it shows the structural components of the solution as well as the relationships between components.

Figure 2.1 Structure chart



The box at the top of the chart represents our complete solution, the three boxes at the next level below represent the components of that solution, and the three boxes at the lowest level represent the sub-components of the component Finish The Horns. Similarly, all components can be broken into smaller sub-components.

Step 3 Solution Refinement

The solution designed in the preceding step is in a very high-level form. It shows a general task divided into several sub-tasks with no indication given as to how all of these tasks are to be accomplished. This general solution must be refined by adding more detail.

Each box in the structure chart produced in Step 2 (Figure 2.1) must be associated with a precise method to accomplish each task. A precise method consists of a sequence of well defined steps which, when carried out, perform the corresponding task. This sequence of steps is called an *algorithm*.

Algorithms are usually developed in *pseudocode*—a language used to describe the manipulations to be performed on data. This language is a simple mixture of natural language and mathematical notation. Pseudocode is independent of any programming language.

For example the pseudocode for the task Prepare Whipped Cream might resemble Pseudocode 2.1.

Pseudocode 2.1 Prepare Whipped Cream

```

Put cream into mixing bowl
Put bowl on turntable of food-mixer
Select manufacturer - recommended speed for whipping cream
Start mixer
As long as the cream is not stiff
    Leave bowl in food-mixer
Stop food-mixer
Remove bowl from food-mixer

```

This particular solution is not complete because the level of detail is insufficient: the stiffness of the whipped cream has not been specified. Each pseudocode instruction might have to be expanded to refine the solution into a usable form. This process of refining the pseudocode instructions to avoid any misunderstanding is called *stepwise refinement*.

Had we been working instead on solving another problem, that of computing the weekly pay for the employees of the ACME company, we might have written the following pseudocode:

Pseudocode 2.2 Setting an employee's gross pay

```

If Hours greater than 40
    Set Gross Pay to Rate times 40 plus 1 and
    1/2 times the hours above 40
Else
    Set Gross Pay to Rate times Hours

```

Here we have used the verb Set to associate the name Gross Pay with a computed value. We could also have used a more mathematical notation like the following.

Pseudocode 2.3 Mathematical notation in pseudocode

```
If Hours > 40
  Set Gross Pay to Rate x 40 + 1.5 x Rate x (Hours - 40)
Else
  Set Gross Pay to Rate x Hours
```

Care should be taken to make the algorithm check the data values for reasonableness. If any data item is out of range, an error message should be produced instead of computing a meaningless result. This process of checking that the input data are acceptable is called data validation.

Step 4 Testing Strategy Development

The preceding step produces the algorithms that will be used to write the computer program that corresponds with our solution. This program will be executed by the computer and be expected to produce the correct solution for our problem. It is necessary to try this program with several different combinations of input data to make sure that the program will give the correct results in all cases

Each of these tests consist of different combinations of input data, each of which is referred to as a *test case*. To test the program thoroughly, we must define an assortment of test cases that cover, not only normal input values, but also extreme input values that will test the limits of our program. In addition, special combinations of input values will be needed to test particular aspects of the programs

For example, test cases for a payroll program should include values covering the full range of pay rates including limit values, a maximum number of hours worked, and so on. A typical test case would define the hourly rate, hours worked, and expected result as follows:

Rate = \$10.00 Hours = 35 Result = \$350.00

It is best to develop test cases before writing the program because the test cases can be used to check our algorithms at this stage, and because the pressure to complete a program might lead to a loss of objectivity when testing on the fly.

Note: For all test cases, the expected results must be computed and specified before proceeding to the next step. Complete test cases can then be used to check algorithms and programs efficiently.

Step 5 Program Coding and Testing

Pseudocode algorithms cannot be executed directly by the computer. Pseudocode must first be translated into a *programming language*, a process referred to as *coding*.

The pseudocode produced by solution refinement, in Step 3, is used as a blueprint for the program. Depending on the programming language used, this coding step may be somewhat mechanical; however, programming languages all have limitations. These limitations can sometimes make the coding step quite a challenge. Among the most commonly used languages are COBOL, Fortran, Pascal, C, Modula-2, and Ada. To give an example, the pseudocode shown earlier for the gross pay computation could be written in Pascal in as illustrated in Figure 2.2

Figure 2.2 Pascal version of Pseudocode 2.2

```
IF Hours > 40 THEN
    GrossPay := Rate * 40 + 1.5 * Rate * (Hours - 40)
ELSE
    GrossPay := Rate * Hours;
```

Notice that the pseudocode *Set...to* has been replaced by “:=” and that multiplication is indicated by an asterisk. Most programming languages use a notation similar to this one. Names like `GrossPay` are called *variables* and can be thought of as named “information holders” whose contents can be used and changed by statements in the program. The above Pascal code would only be a small part of a complete Pascal program that calculates a weekly payroll

Once the algorithms have been coded, the resulting computer program must be tested using our testing strategy. The program must be run and, for each test case developed in Step 4, the results produced must match those computed during the development of the test strategy. If there is a discrepancy, the produced results must be analyzed in order to discover the source of the error. The error may lie in one of four places:

- The coding: The error could have been formed when the algorithm was translated into the programming language.
- The algorithms: The error could have existed in the algorithm and has never been noticed.
- The design of the program: The program’s design may be flawed and lead to errors during execution.
- The computation of the expected test results: The results for a test case may have been calculated wrong.

Once the error has been discovered, the appropriate revision must be made and the tests rerun. This is repeated until we are sure that our program produces a correct solution to the problem under all circumstances. This process of coding and testing is called *implementation*.

Step 6 Documentation Completion

Documentation begins with the first step of program development and continues throughout the lifetime of the program. A program's documentation must include the following:

- Explanations of all of the steps of the method,
- The design decisions that were made, and
- The problems that were encountered during the writing and testing of the program.

The final documentation must also include a printed copy of the program, called a *program listing*. This listing should be made readable by careful layout as one does with a technical report—which is what the written form of the program really is: a written description of the computer solution of the problem. Programs can be made easier to understand by the reader if all of their complex parts include explanations in the form of comments. Later, when the program must be changed, these explanations will make the changes easier to implement.

To be complete, the documentation must include some user instructions. For programs that will be used by people unfamiliar with computers, it is necessary to include a user's manual that explains, without using technical jargon, how to use the program, how to prepare the input data and how to interpret the program's results.

Step 7 Program Maintenance

Program maintenance is not directly part of the original implementation process. It includes all activities that occur after a program becomes operational. The term "maintenance", when applied to programs, has a somewhat different meaning from normal usage. Houses, cars and other objects need maintenance because they deteriorate through use. Programs, on the other hand, have no physical properties, only logical properties—they do not need repainting and they do not wear out through use.

When we run the same program a thousand of times, some of the hardware components of the computer may wear out and need maintenance but the program can't wear out. However, a program can work properly nine hundred and ninety-nine times and then fail during the thousandth run. This may give the appearance that the program has worn out, but actually the program encountered an unusual set of circumstances. In other words, the program did not fail because it had worn out, it failed because it did not work properly to begin with; it was never tested for this particular set of circumstances. If the unusual circumstances had occurred on the first run instead of the thousandth, the program would have failed on the first run. Thus, program maintenance is largely completing the process of implementation.

Program maintenance also includes implementing improvements discovered while using the program. Program maintenance includes:

- eliminating newly detected program errors
- modifying the current program
- adding new features to the program
- updating the documentation

Maintenance documentation, produced specifically to help the “maintainers” in their work, may include design documents, program code, and information about testing. Often, the cost of program maintenance over the lifetime of a program surpasses the development costs for the original program.

Using the Problem Solving Method

The preceding sections describe the seven steps in a nice, sequential manner. However, it should be understood that, in practice, all of these steps are *not strictly sequential*. The boundaries between various steps are somewhat fuzzy and some steps might have to be reworked as a result of a later step.

In particular, you may have already noticed that documentation takes place throughout the whole process. Sometimes, the problem definition step and the first stages of the solution design step overlap, as design decisions require more precision from the problem definition. Also, the end of the solution design step may overlap with the beginning of the solution refinement step; this explains why the refinement step is sometimes called *detailed design*. We have also indicated that the definition of a testing strategy sometimes uncovers missing pieces in the earlier steps.

The point is that you should get used to the idea of revising and reworking some of your earlier steps. This going back and forth makes it easier to progress when solving a problem becomes difficult. The important thing is for you to adopt a problem solving method that is well adapted to your way of working.

You should also be aware that the problem definition step is often difficult because the person with the problem may not be able to state it clearly. Most often, a vague perception of the problem or a weak technical understanding are sufficient to lead to this situation. In these cases, an extended period of work with the user may be required in order to produce a more precise problem definition and to determine whether a computer solution is possible.

Problems and Plans Dividing and Conquering

As we have seen above, planning and problem solving are the main concerns of this book. The problems may be large and complex (like controlling a factory or managing an inventory) or they may be smaller (such as making engineering calculations, or computing statistics). In all cases, we’ll need to use our seven step method.

Even in problems that seem simple, such as computing a weekly payroll, there could be considerable complexity. For example, actions that depend on various changing conditions, such as income tax rules, can turn computing a weekly payroll into a complex task.

When faced with complexity, use Step 2, Solution Design, of our seven step method to develop a systematic way of thinking about such problems. As we have already mentioned, the best way to go about solving a problem is to break the problem into smaller subproblems: the “divide and conquer” approach.

But breaking up a problem into smaller subproblems is not always as easy as it seems. There are many ways to break up anything with some ways being better than others. In addition, the number of subproblems required for computing are often so numerous that they create another problem: the complexity of quantity. Our challenge is to organize this complexity while avoiding confusion.

To avoid confusion, a complex problem can only be viewed by looking at a small number of its subproblems at any one time and seeing how they fit into the “bigger problem.”

Note: Solving a problem is essentially the same as the age-old method for cooking a mammoth...

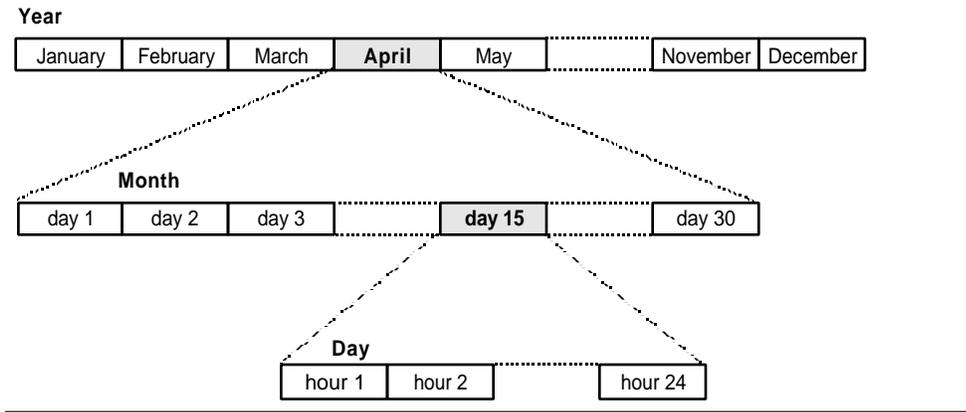
...Break it into smaller pieces!

2.3 Break-Out Diagrams

One useful way of making the complexity of problem solving manageable is to use a tree-like or hierarchical skeleton for viewing problems in levels, as illustrated by the structure chart in Figure 2.1. The following figures (Figures 2.3 through 2.6) show four hierarchical representations called *break-out diagrams*. These are another form of structure charts.

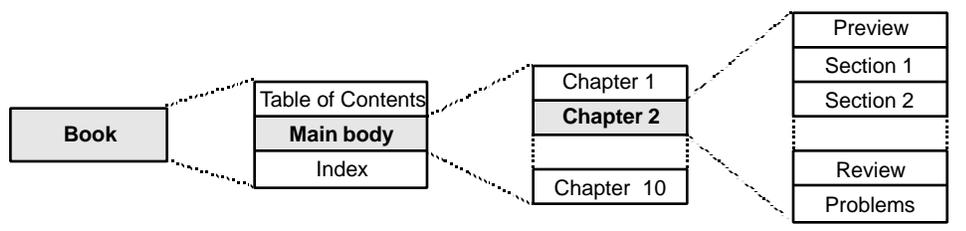
Time can be broken down into years, months, days (and further) as suggested in Figure 2.3. This break-out diagram, if completely expanded, would create one sequence of 365 (or 366) days at the second level, and one sequence of 8760 (or 8784) hours at the next level.

Figure 2.3 Time break-out diagram, vertical



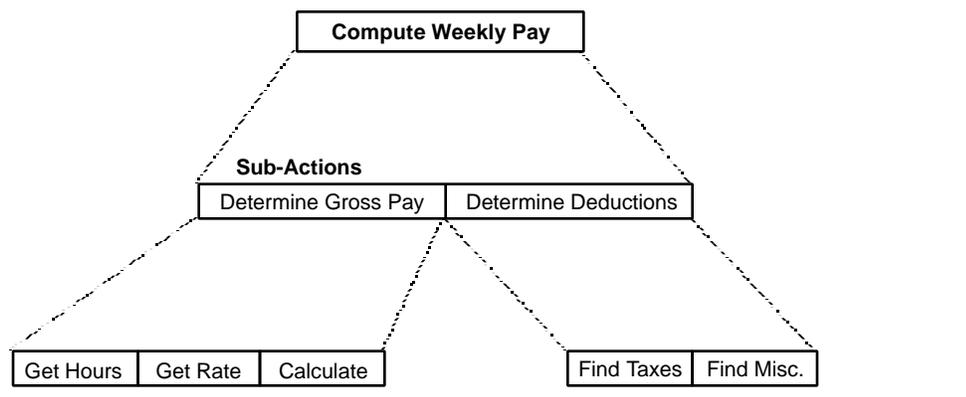
The space in this book, represented in Figure 2.4, is broken down horizontally. First the book is broken into chapters, then into sections. This book space could be broken down further by including sentences, words and finally letters.

Figure 2.4 Book space break-out diagram, horizontal



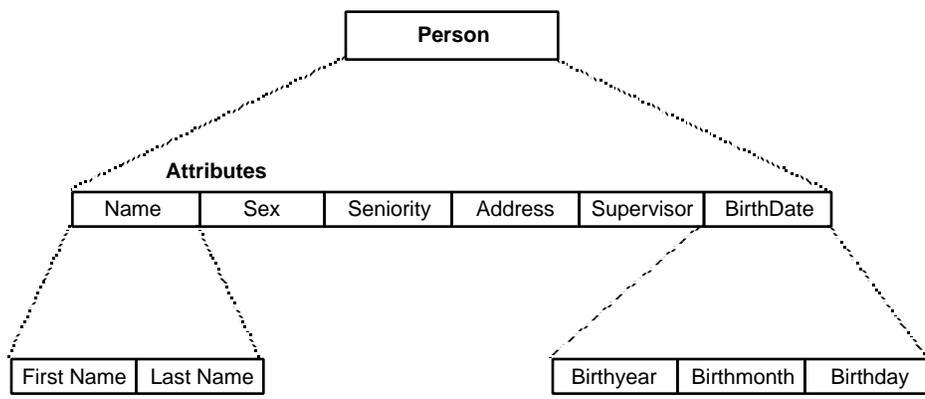
Actions, such as computing a weekly payroll, can be broken down into smaller actions (determine the gross pay, determine the deductions) as shown in Figure 2.5. Each of these sub-actions may also be broken down further. In fact, the rest of this chapter will mainly be concerned with the subproblem of computing an employee's gross pay

Figure 2.5 Actions break-out diagram



Data, such as the various attributes describing a person, are also easily described by break-out diagrams (Figure 2.6).

Figure 2.6 Data break-out diagram



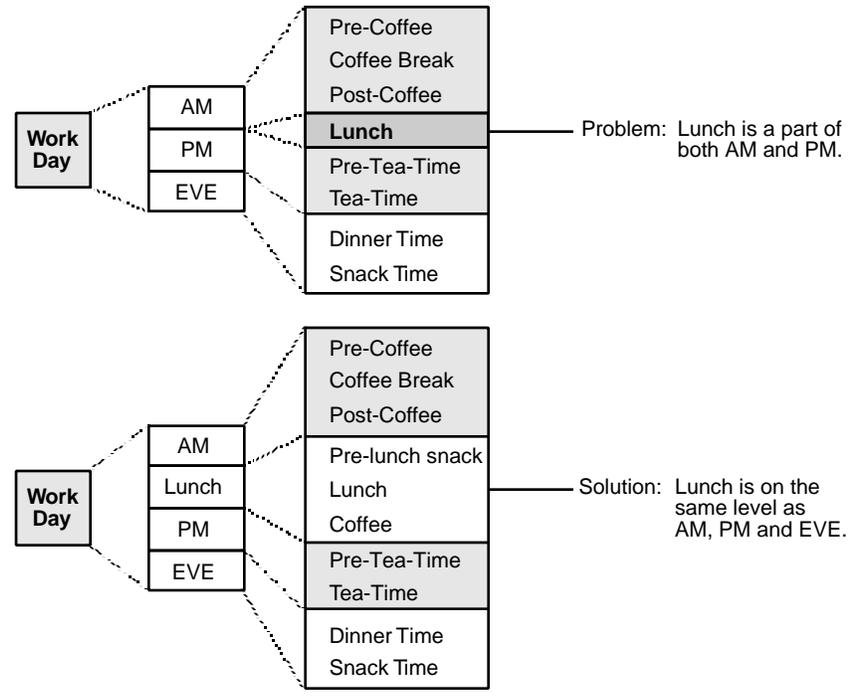
Notice how break-out diagrams can describe four entities as varied as time, space, actions and data. These diagrams essentially show how the long, linear list of small “leaves” at the right, or at the bottom, is organized or grouped together (into branches) forming a two-dimensional tree. More importantly, these structure charts show how to break a problem into its subproblems.

More on Break-Out Diagrams

Break-out diagrams or BODs are very useful tools for showing the structure of many kinds of systems. However, not everything that looks like a BOD is actually a BOD. To be useful, BODs must have a certain structure as well as being:

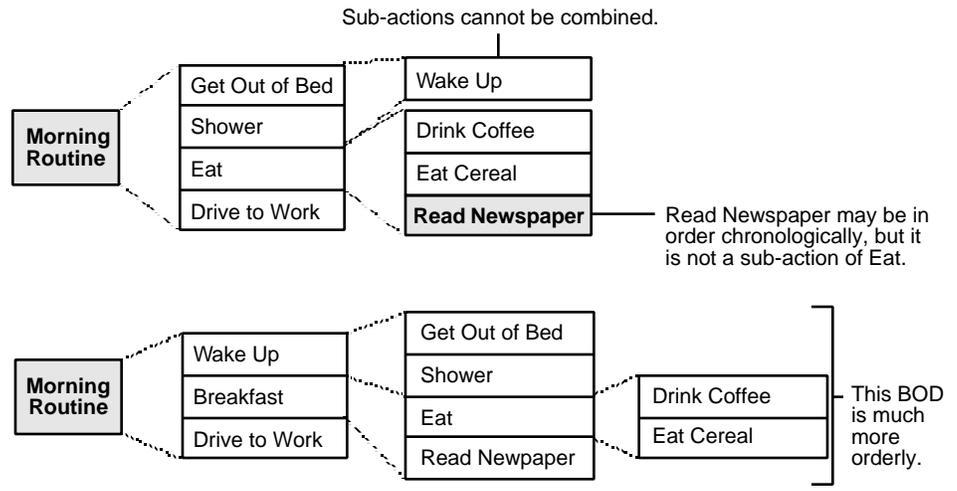
- *Consistent*: Each break-out must be of the same kind. For example, if the first box involves time (as in Figure 2.3), then all of the other boxes in the break-out should involve time. In other words, since time is being refined, then all break-outs should involve time, but at a different level of detail.
- *Orderly (independent or exclusive)*: all blocks at the same level must be separate or independent; there should be no overlapping of two break-outs. For example, the first break-out of Work Day in Figure 2.7 shows Lunch as part of the morning and of the afternoon; it should be either part of only one of these, or be a separate box on the same level as AM and PM.

Figure 2.7 An incorrect and a correct break-out of Work Day

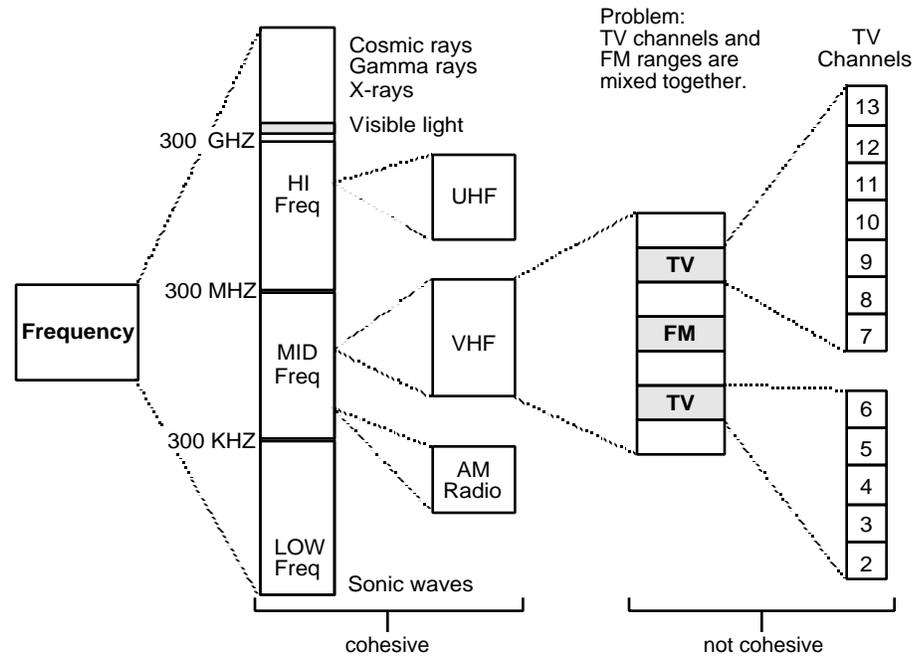


- *Refined*: each box of a given level must be a break-out of a box at the previous level. This means that all of the boxes at the right of a BOD must fit back into the boxes at their left. There can be no boxes introduced that do not fit into previous ones. For example, in the second impossible break-out of Work Day in Figure 2.8, the box labeled Read Newspaper does not fit into the Eat box. Also, as the refinement continues, there should be no merging or rejoining of boxes.

Figure 2.8 An incorrect and correct break-out of Morning Routine



- Cohesive:** All of the items within a breakout box must fit together. Frequencies, shown in the break-out diagram in Figure 2.9, range from a few cycles per second, or Hertz (Hz), to thousands of cycles per second (kHz), to millions of cycles per second (MHz) and beyond. At the left of this diagram, there is great cohesion. However there is some lack of cohesion in the part at the right. The VHF frequencies have a mixture of TV channels and FM ranges. There is a gap between TV Channels 6 and 7 in which some FM channels are used. These mixtures and gaps show a lack of cohesion.

Figure 2.9 The electromagnetic frequency spectrum

Alternative forms of break-out diagrams are often used and variations are possible. Some common alternatives to BODs are called: Warnier Diagrams², Orr Charts³, or SADT diagrams⁴. Some of these involve boxes in three dimensions, others involve parentheses and other notations

² Jean Dominique Warnier, *Logical Construction of Programs*, 3rd Edition, New York: Van Nostrand Reinhold Co., 1974.

³ Kenneth R. Orr, *Structured Systems Development*, New York: Yourdon Press, 1977.

⁴ Douglas Ross, "Structured analysis (SA): a language for communicating ideas", *IEEE Transactions on Software Engineering*, vol. SE3, no. 1, January 1977.

2.4 Algorithms and Their Representations

In the third step of our method, solution refinement, we refine the solution whose structure was defined in the preceding step. This means that we must give a precise definition of the actions to be performed to accomplish each task defined in our structure chart. This is done by defining an *algorithm* for each task. It is these algorithms that will be carried out to produce the desired solution.

As an example, let's consider the computation of the weekly gross pay of an employee. Here, the data are numbers while the actions are arithmetic operations. The algorithm we'll consider is very simple and will be used to illustrate many concepts.

Algorithms may be represented in a number of ways, as discussed below.

- *Verbal* representations of an algorithm can be given as statements in any natural language. A verbal description of a common pay algorithm is shown in Figure 2.10. The pay rate (of \$10 an hour) was chosen simply to make multiplication easy. Usually, more important reasons determine an employee's pay rate.

Figure 2.10 A verbal algorithm

Gross Pay

If the hours worked are less than (or equal to) 40, the pay is the product of the number of hours worked and the rate (of \$10.00 an hour). Also, if more than 40 hours are worked, the pay is \$15.00 an hour (time-and-a-half) for each of the hours over 40.

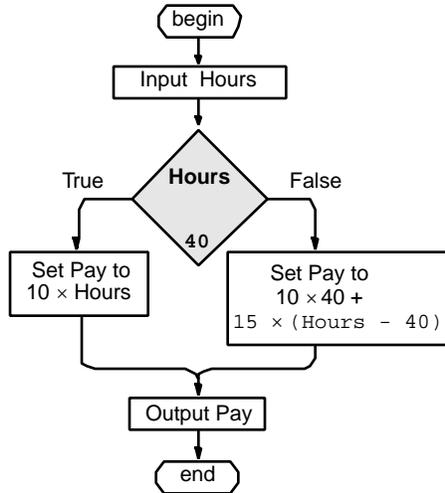
With this algorithm, if, for example, the number of hours worked is 25, then the pay is simply determined by multiplying the rate by the hours ($10 \times 25 = 250$). But if the hours are 50, then the pay is the sum of two parts, a regular part and an overtime part. The pay for the first 40 hours is the regular rate multiplied by 40 which is ($10 \times 40 = 400$). For the 10 hours over 40 we use the higher rate of 15 (time-and-a-half) to get $(50 - 40) \times 15 = 150$. The total pay is the sum of these regular and overtime amounts ($400 + 150 = 550$). So a total pay of \$550 is the final output of the algorithm.

- *Flowchart* representations of an algorithms are made of various boxes joined by arrows as in Figure 2.11. In flowcharts the following symbols are used:
 - Oval boxes indicate the start and the end.
 - Square boxes represent actions, while
 - Diamonds, or boxes with pointed ends, represent decisions to be made. Decision boxes contain the conditions that determines which arrow will be followed out of them.

For more information on the different possible algorithm representations, see Chapter 3.

This graphical form makes it easy for people to follow the sequence of actions, or *flow of control*, provided the flowchart is small. Figure 2.11 represents the pay algorithm we just described. Notice that in the computation of the pay for more than 40 hours (the box on the right of the diagram), the pay for the basic 40 hours is shown as 10×40 and not as 400. To have shown 400 in the algorithm, would have hidden the two “components” of the product. The 400 would appear as an anonymous “magic” number and the algorithm would be more difficult to understand.

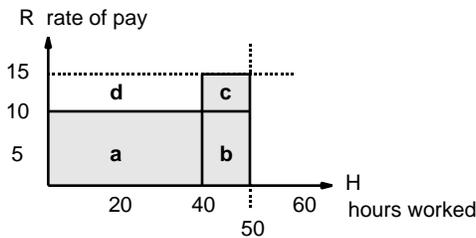
Figure 2.11 The pay algorithm expressed as a flowchart



- *Graphs (or plots)* are diagrammatic representations that help people understand the execution of algorithms. The graph of the pay algorithm in Figure 2.12 shows how the rate of pay depends on the number of hours worked. In this graph, the total pay for any number of hours is actually the shaded area. For example, for 50 hours, the total pay corresponds to the shaded area consisting of the three smaller rectangles labeled a, b and c. The total pay is:

$$\text{PAY} = 10 \times 40 + 10 \times 10 + 5 \times 10 = 550.$$

Figure 2.12 Graph of pay rate versus hours worked

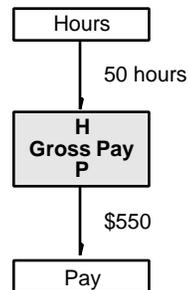


- *Data-flow diagrams* represent algorithms as “black boxes” where you can’t see what happens inside. Only the data input and output are

shown as in Figure 2.13. In this case, the data—50 hours—“flows” into the Gross-Pay computation box, and the resulting data (pay of \$550) flows out the bottom. These diagrams hide the details of an algorithm, but they will be useful later to describe interaction and flow of data between algorithms.

Note: Data-flow diagrams indicate what is being done, whereas flowcharts indicate how it is done.

Figure 2.13 Data-flow diagram



- As we have already seen, *pseudocode* representations of an algorithm are short descriptions using a notation that is a mixture of mathematics, logic, and natural language. Our pay algorithm in pseudocode is given in Figure 2.14.

Figure 2.14 A pseudocode algorithm

```

Input Hours
If Hours > 40
  Set Gross Pay to Rate × 40 + 1.5 × Rate × (Hours - 40)
Else
  Set Gross Pay to 10 × Hours
Output Pay
  
```

More representations of algorithms, such as flowblocks, tables, and trees, are possible and will be considered in the next chapter.

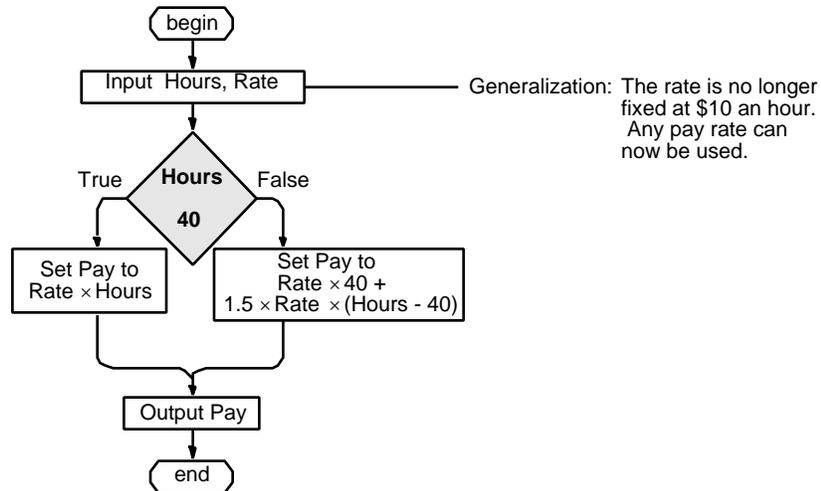
Modifying Algorithms

Once a useful algorithm has been defined, it often goes through many changes in its “lifetime.” It may be made more powerful, more useful, more convenient, more efficient, or more foolproof. Sometimes, it is used as part of larger algorithms. The five processes for modifying are detailed below.

- *Generalizing* algorithms is the process of making them apply to more cases. For example, the previous pay algorithm, shown in Figures 2.10 and 2.13, applies only to people paid at the same constant basic hourly rate of \$10. This algorithm could be *generalized* by requiring the input

of the hourly rate in addition to the number of hours worked. By implementing this generalization, the algorithm would work not only for \$10, but for any hourly pay rate.

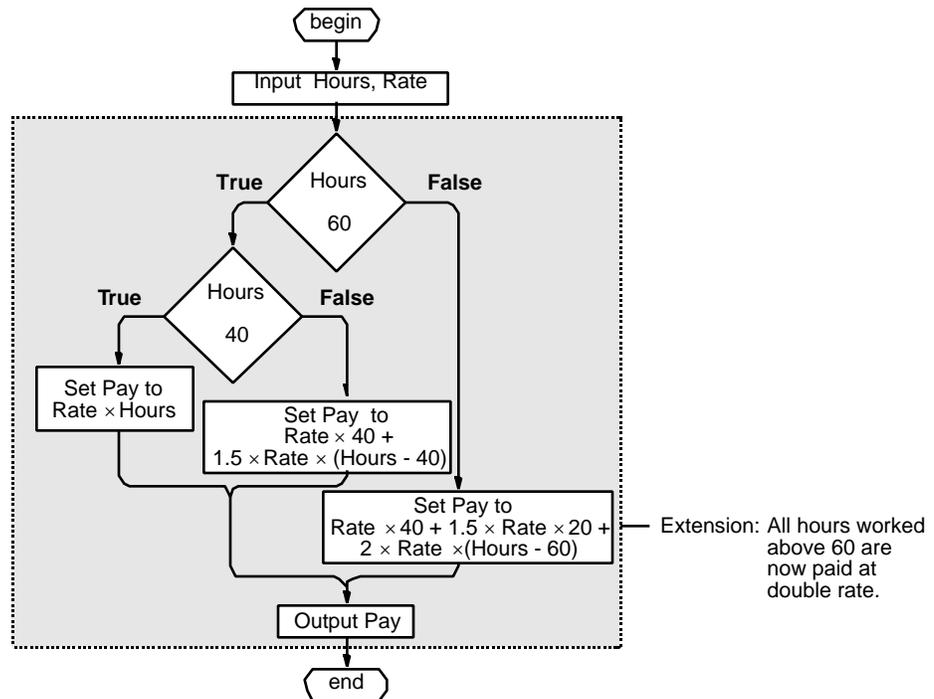
Figure 2.15 Generalized pay algorithm



This modified algorithm, Figure 2.15, can be used to compute the pay for people working at different pay rates, and is thus more general and more widely useful. Actually the overtime limit of 40, which is used three times, could be replaced by a variable *Maximum-Hours* indicating the overtime threshold, thus making the algorithm even more general. Thus, if *Maximum-Hours* is set to 35, the overtime rate is applied to all hours worked over 35, and not over 40 as in the algorithms of Figure 2.14 and 2.14.

- *Extending* algorithms to include more cases is also very common. For example, the original algorithm pays “time-and-a-half” for hours worked over 40. Often, the overtime rate becomes even larger (up to twice the regular rate) when more than a certain number of hours have been worked (usually 60). The original algorithm of Figure 2.15 can be *extended* to include a second rate of overtime. In this case, the original pay algorithm applies for the first 60 hours, and the hours over 60 are paid at double rate. An extended version of the original algorithm, that allows for a double rate, is shown in Figure 2.16.

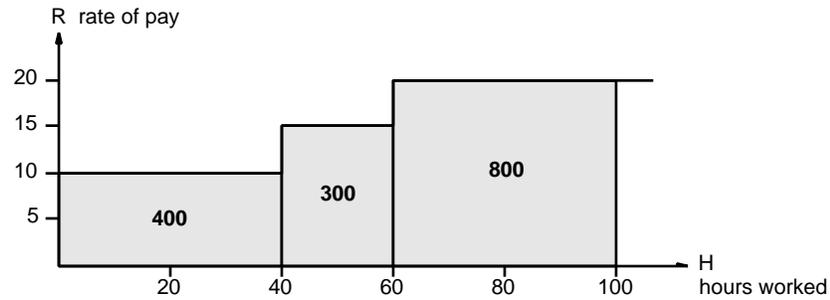
Figure 2.16 Extended pay algorithm



Let's execute or "trace" this algorithm with an input value of 100 hours and an hourly rate of \$10.00. First the number of hours and rate are input, then the number of hours is compared to 60 and the rightmost path is taken out of the decision box into the following computation.

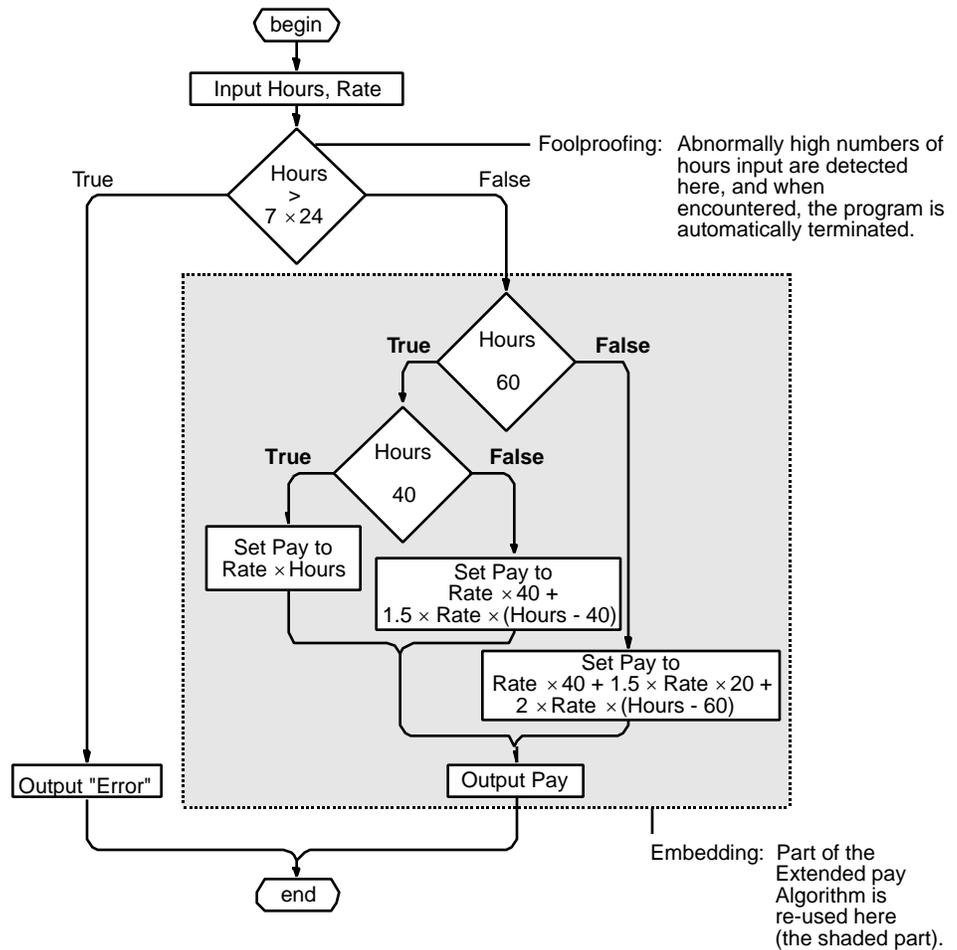
$$\begin{aligned}
 \text{Pay} &= \text{Rate} \times 40 + 1.5 \times \text{Rate} \times 20 + 2 \times \text{Rate} \times (\text{Hours} - 60) \\
 &= \$10.00 \times 40 + 1.5 \times \$10.00 \times 20 + 2 \times \$10.00 \times (\text{Hours} - 60) \\
 &= \$400.00 + \$15.00 \times 20 + \$20.00 \times 40 \\
 &= \$400.00 + \$300.00 + \$800.00 \\
 &= \$1500.00
 \end{aligned}$$

This formula (and others) can be derived from finding the area of various rectangles under a new graph of Rate versus Hours, an extension of Figure 2.12 shown in Figure 2.17. Notice that by doubling the number of hours worked (from 50 to 100), the resulting Pay more than doubles (from \$550 to \$1500). In fact, the Pay almost triples.

Figure 2.17 Extended graph of Rate versus Hours

- *Foolproofing* is the process of making an algorithm more reliable, fail-safe, or robust, by anticipating erroneous input or other difficulties. For example, if the number of hours input is more than the number of hours in a week ($7 \times 24 = 168$), then an error message should be produced. The resulting foolproofed, or robust, algorithm is given in Figure 2.18. It could be improved further to recognize the input of a negative number of hours, and output another error message.

Figure 2.18 Foolproofed pay algorithm



- *Embedding* an algorithm is the process of re-using that algorithm within another algorithm. For example, the extended pay algorithm from Figure 2.16 is shown in the shaded box embedded in Figure 2.18. If the number of hours input is smaller than the maximum number of hours permitted in a week, this algorithm computes the pay.

Notice that the original algorithm from Figure 2.11 also appears in slightly modified form embedded in Figures 2.14 and 2.15. This shows that, when algorithms are well structured, they can be modified without destroying already existing parts. On the other hand, when algorithms are structured poorly, small modifications can cause great problems. Reuse of existing algorithms is also efficient and usually saves coding and testing time and effort.

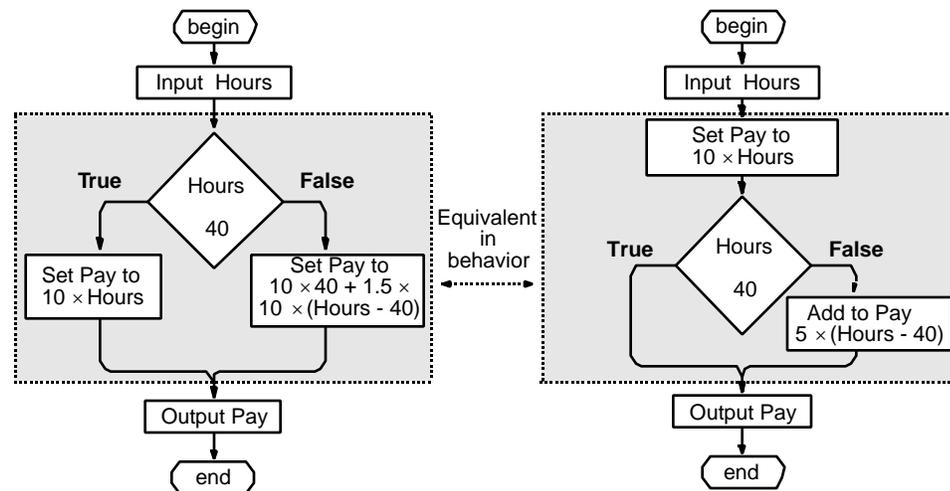
Modification of a finished product is not common in other disciplines. For example, painters do not try to touch up another painter's painting, and engineers do not add a few extra wheels to an existing car. However, in computer science, algorithms are modified all the time. It is therefore

important to keep in mind that the algorithms that you write are likely to be modified during their lifetimes. In other words, create algorithms with the intent to make them easy to modify. When algorithms are well designed, their modification can lead to better algorithms. Otherwise, a modification may lead to disaster.

Alternative Algorithms

In computing, as is the case with many disciplines, there are often many ways of accomplishing the same thing. This means that a choice must be made between the various possibilities. For example, let's consider the previous simple pay algorithm from Figure 2.11, which is repeated on the left side of Figure 2.19.

Figure 2.19 Equivalent algorithms



The algorithm next to it, on the right side of Figure 2.19, shows an alternative way to compute the gross pay.

First, the hours are input and the pay is computed by multiplying these hours by the regular hourly rate (\$10). For instance, in the case of 50 hours of work, this would yield a value of $50 \times 10 = \$500$.

Next, the hours are compared to 40 and, if they are greater than 40, the hours in excess of 40 (overtime) are multiplied by 5 (half the regular rate) and the result added to the first value. If the hours are not greater than 40, then nothing is added to the first value. In the case of 50 hours of work, the overtime is $(50 - 40)$, or 10 hours, multiplied by the extra \$5 per hour, which yields \$50. This overtime pay is then added to the first \$500 for a total of \$550 dollars.

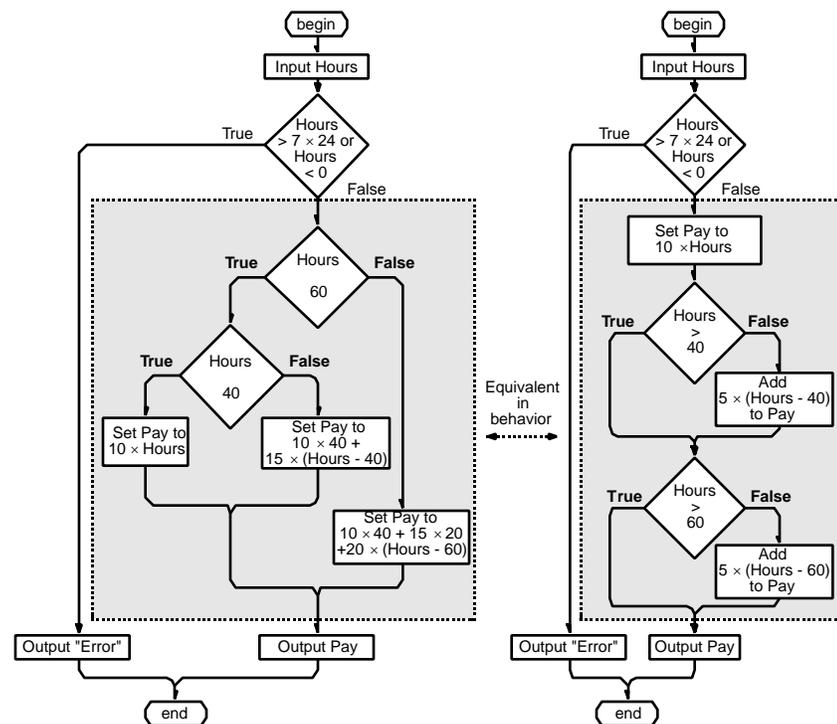
Equivalence of Algorithms

The two algorithms of Figure 2.19 are different in structure, but they are equivalent in behavior. In other words, for identical input data, they will produce identical results. Which do you prefer? Why?

In this example (Figure 2.19), there is no serious reason to prefer one algorithm over the other. However, in some cases, one algorithm may be considerably smaller, faster, simpler, or more reliable than another. The interesting thing is that one embedded algorithm can be replaced by another with the same behavior, like a spare part or a module, without changing the behavior of the whole program. This “plug-in” capability of modules or building blocks can be very useful.

Let us consider a more complex algorithm; the extended gross pay algorithm of Figure 2.16 is repeated on the left side of Figure 2.20 with an equivalent algorithm on the right.

Figure 2.20 More equivalent algorithms



Because the equivalence of these two algorithms may be less obvious, let us try the algorithm on the right for an input of 100 hours. After the first condition is False, the pay is computed from $10 \times \text{HOURS}$, giving \$1,000. Then the second decision ($100 > 40$) adds the amount $5 \times (100 - 40)$ or \$300. The third decision ($100 > 60$) adds the amount $5 \times (100 - 60)$ or \$200. Finally, the output is the

sum of these three amounts ($1000 + 300 + 200$) or \$1500. This output is identical to the previously computed output for the algorithm on the left.

Note: It is extremely important to realize that comparing outputs for one single input value is insufficient to determine an equivalence for all values! Several different values must be tried before equivalence can be proven. Once you encounter an input value that produces different outputs for both algorithms, you have proven that the two algorithms are not equivalent.

Testing

The fourth step of our problem solving method, Testing Strategy Development, leads us to define the following:

1. A strategy, and then
2. a collection of test cases for the particular problem being solved.

In the example seen in Figure 2.20, there are only a few ranges of values that could be used to test the algorithms. The left side of Figure 2.21 shows how the input values could be split into the following five distinct ranges:

- Input values less than zero and those greater than 168 indicate errors.
- Values from 0 to 40 (inclusive) belong to the regular range,
- Values above 40 and up to 60 belong to the time-and-a-half range, and
- Values above 60 to 168 belong to the double-time range.

To compare these two algorithms, we must test them with input data taken from each of these ranges. Only one value from each range is required since all values within a particular range will follow the same path through the algorithms (check it for yourself!).

Figure 2.21 Range of input values for testing

Hours	H		Input	Test Value	Output	
			Data range		Left	Right
		Out of Range	$H > 168$	200	Err	Err
168		Double Time	$60 < H \leq 168$	100	1500	1500
60		Time-and-a-half	$40 < H \leq 60$	50	550	550
40		Regular Time	$0 \leq H \leq 40$	20	200	200
0		Out of Range	$H < 0$ Err	-20	Err	

So, to compare these algorithms, we must take one typical test value from each of the ranges. Let us try -20, 20, 50, 100 and 200. These five values will exercise all possible paths in the flow charts. Other equally good test values could be -50, 30, 55, 150 and 170, or -5, 5, 45, 65 and 205. In addition, it is always useful to test critical values, or limit values, such as -1, 0, 40, 41, 60, 61, 168, and 169.

The right side of Figure 2.21 displays a table of test values with the corresponding output of each algorithm. This table shows identical outputs for all of the input values. Verify for yourself that the two algorithms behave identically for the limit values.

We can now say that the algorithms in Figure 2.20 are equivalent because the test values were chosen carefully to cover all possible cases and the results of the test values come out the same for both algorithms. It should be realized that it is not always easy to test algorithms in this way. In some algorithms, there may be hundreds or thousands of possible paths and test cases. Testing on this scale will be discussed in Chapter 5.

Now that the two algorithms have been shown to be equivalent, which of them is better? The difference between them is not great, but most people feel that the second (the one on the left) is simpler and clearer because it consists of a long and thin series of smaller decisions. Algorithms with a long and thin form usually appear simpler than a “nesting” of short-and-fat decisions.

Because the two algorithms of Figure 2.20 are equivalent, there is a single description of *what* they both do; however, there are two different descriptions of *how* they do it. This illustrates the difference between Step 2, Solution Design, and Step 3, Solution Refinement. Solution Design gives us a description of *what* each sub-task must do to solve the problem, while Solution Refinement gives us descriptions of *how* each sub-task is to be done. This distinction is extremely important because the definition of *what* is to be done should be made independently from *how* it will be done. Defining *what* generates the plan for the solution, while defining *how* comes with the application of the plan.

2.5 Programming Languages

Communicating Algorithms

The subject of this book is programming, and as you progress, you'll write programs to run on a computer. To do this, you'll need to use a specific *programming language*.

Programming languages are notations used for communicating algorithms between people and computers. There are now hundreds of such languages; to give you a taste of these, Figures 2.20 to 2.25 show the same payroll algorithm (taken from Figure 2.11) expressed in six different programming languages, Basic, Fortran, Pascal, C, Modula-2 and Ada.

If you don't understand all of these examples perfectly, don't worry! These examples are here only to suggest similarities and differences. All of these programs have a similar meaning (semantics) but differ greatly in the details of their form (syntax). For example, they all input the number of hours, but each program specifies this input differently: Basic uses the command `INPUT`, Fortran and Pascal use `READ`, C uses `scanf`, Ada uses `Get` and Modula-2 uses `ReadInt`.

Basic

BASIC (*Beginner's All-Purpose Symbolic Instruction Code*) was developed by John Kemeny at Dartmouth College around 1967. It is a simple programming language, designed to be easy to learn and to use. Many versions (or dialects) of BASIC have appeared since and are still in use.

Figure 2.22 A simple pay program in Basic

```

100 REM ] SIMPLE PAY IN BASIC
110 REM ] H IS THE NUMBER OF HOURS
120 REM ] P IS THE GROSS PAY
130 INPUT H
140 IF H > 40 THEN 170
150 LET P = 10 * H
160 GOTO 180
170 LET P = 10 * 40 + 15 * (H-40)
180 PRINT 'GROSS PAY IS ', P
190 END

```

The REM keyword introduces comments.

Single-letter variable name

Symbol for multiplication

In Basic, the numbers at the beginning of each line are reference numbers for use in the program. The word `REM` is used to introduce a “remark”, a comment to help people understand the program. Each of the programming languages shown here have their own way of marking comments. Some versions of Basic limits the names of variables to a single letter, possibly followed by a single digit. Here, we have used `H` for HOURS and `P` for PAY. Also notice that the asterisk, `*`, is used as the symbol for multiplication instead of the \times from mathematics. This use of the asterisk as a symbol for multiplication is common to most programming languages.

Fortran

Fortran (*FORmula TRANslation*), developed by John Backus around 1957, was intended for engineering and scientific computations. Revised versions of Fortran are still extensively used for numerical work.

Figure 2.23 A simple pay program in Fortran

```

      |----- This symbol, at the beginning of a
      | line, introduces a comment.
*     SIMPLE PAY IN FORTRAN 77
INTEGER HOURS, PAY ----- Proper variable names.
READ *, HOURS
IF (HOURS .LE. 40) THEN
    PAY = 10 * HOURS
ELSE
    PAY = 10 * 40 + 15 * (HOURS - 40)
ENDIF
PRINT *, 'Gross pay is ', PAY
END

```

In Fortran, a line that starts with an `*` is a comment, corresponding to a line that starts with `REM` in Basic. Fortran also uses the `*` as the symbol for multiplication, and to refer to the keyboard and screen in the `READ` and `PRINT` statement.

Pascal

Pascal, created by Niklaus Wirth around 1970, was based on an international language called Algol 60. It is somewhat similar to both Basic and Fortran, but incorporates many refinements in structure. It was designed as a teaching language for beginning students. Comments in Pascal are enclosed in braces, { and }.

Figure 2.24 A simple pay program in Pascal

```

PROGRAM SimplePay(INPUT, OUTPUT);
{ Simple pay in Pascal } ----- Comments begin and
VAR                                     end with braces, {}.
    hours, pay: INTEGER;

BEGIN
    Read(hours);
    IF hours <= 40 THEN
        pay := 10 * hours
    ELSE
        pay := 10 * 40 + 15 * (hours - 40);
    Writeln('Gross pay is ', pay:6);
END.

```

Modula-2

Modula-2 was also created by Niklaus Wirth in the late 1970s. Not only is it an extension of Pascal, but a totally new language that retained Pascal's good features while adding some other important features. Its clarity, simplicity and unity make it suitable for first time programmers and programming professionals alike. Comments in Modula-2 start with `(*` and end with `*)`.

Figure 2.25 A simple pay program in Modula-2

```

MODULE SimplePay;
(* Simple pay in Modula-2 *)
FROM InOut IMPORT WriteString, ReadCard, WriteCard;

VAR hours, pay: CARDINAL;
BEGIN
  ReadCard(hours);
  IF hours <= 40 THEN
    pay := 10 * hours;
  ELSE
    pay := 10 * 40 + 15 * (hours - 40);
  END;
  WriteString("Gross Pay is ");
  WriteCard(pay, 6);
END SimplePay;

```

Comments begin and end with parentheses and asterisks.

C

C is a programming language created at the Bell Laboratories in the early 1970s when low level access to the machine was considered important. Comments in C start with `/*` and end with `*/`.

Figure 2.26 A simple pay program in C

```

/* Simple pay in C */
#include <stdio.h>
main()
{
  int hours, pay;

  scanf("%d", &hours);
  if(hours <= 40)
    pay = 10 * hours;
  else
    pay = 10 * 40 + 15 * (hours - 40);
  printf("gross pay is %d", pay);
}

```

Comments begin and end with slashes and asterisks.

Ada

Ada is a language created to the specifications of the U.S. Department of Defense. It is a large, complex language named after Ada Lovelace, who is said to have been the first programmer, and was a colleague of Charles Babbage as well as the daughter of Lord Byron. Comments in Ada are introduced by the characters `--` and continue to the end of the line.

Figure 2.27 A simple pay program in Ada

```
-- Simple pay in Ada ----- Comments begin
with Text_IO;                with two dashes.
procedure Simple_Pay is
  hours, pay: integer;
begin
  Text_IO.Get(hours);
  if hours <= 40 then
    pay := 10 * hours;
  else
    pay := 10 * 40 + 15 * (hours - 40);
  endif;
  Text_IO.Put("Gross pay is ");
  Text_IO.Put(pay);
end Simple_Pay;
```

Other programming languages

Many other interesting programming languages have been developed. Here are some important landmarks:

- *Lisp (LISt Processor)*, developed by John McCarthy about 1960, is a language based on mathematical concepts. Its objective is the processing of data represented as lists of items. It is mainly used in artificial intelligence applications.
- *COBOL (COmmon Business Oriented Language)*, developed by a committee (CODASYL) of representatives from various computer manufacturers in the late 1950s, is a language particularly suited for business applications. Like Fortran, COBOL has been revised several times.
- *APL (A Programming Language)*, developed by Kenneth Iverson in 1962, is a programming language that uses a very esoteric mathematical notation.
- *Algol 60*: for scientific computation and for the communication of algorithms between computer scientists,
- *GPSS and Simscript*: for the simulation of discrete events such as the functioning of a set of elevators or the service provided by a group of bank tellers,
- *Logo*: for the introduction of the principles of computer programming to children through graphical manipulations,
- *PL/I*: a large language intended to be suitable for all applications,
- *Prolog*: for logic programming used, for example, to automate the proving of theorems,
- *Simula 67*: for the simulation of networks of discrete events,
- *Smalltalk*: for a style of programming where data takes an active rather than passive role; this is known as object programming,

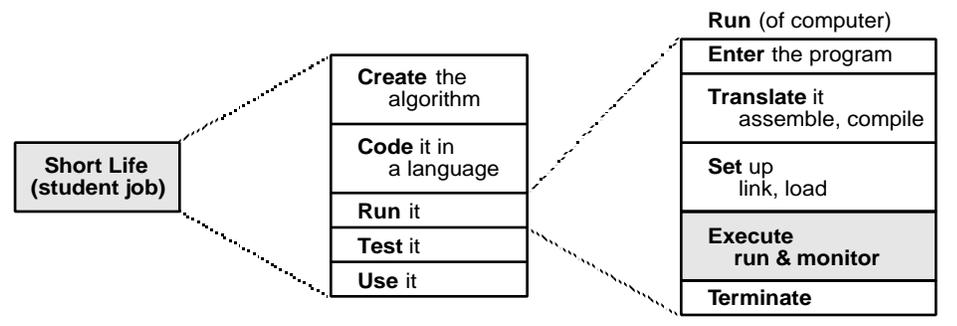
- Snobol 4: for the processing of text strings.

2.6 Life Cycles Stages of Programming

We have mentioned that our seven-step method was related to what is known as the software life cycle. Programs, like all dynamic things, are created, live and then ultimately die. The three most common sizes of programs are presented below, with a comparison of their problems and life-cycles.:

- *Small programs*, created simply for learning purposes, have a very simple life, as shown in Figure 2.28. First the algorithm is created, and only then is it coded or translated into a programming language. It is run and tested (sometimes repeatedly) on a computer. Finally, small programs are used and ultimately thrown away. Sometimes, they are modified and included within larger programs.

Figure 2.28 A small program



The *run* (or *execution*) of a program is shown in the last break-out diagram of Figure 2.28. First the program—a *task or job* is entered or input. It is then translated or compiled into a language that the computer understands. This version is loaded into memory and linked to other programs from a library of programs. The complete program is then run and monitored for time duration, space access and errors. It finally terminates with some results: either data output or error messages.

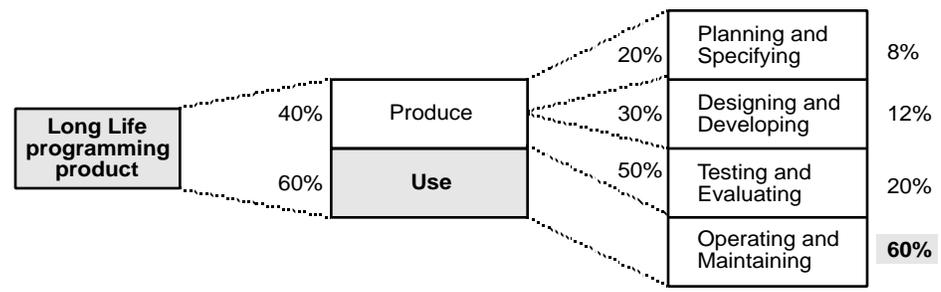
- *Medium-sized programs* are larger than small programs that can still be created by one person. The life cycle of medium-sized programs depends heavily on the programmer's style. Individual programmers have differing styles, with differing consequences. Here are a few "programming personalities.":
 - *The Programming Pervert* who spends no time on design or algorithm creation, but starts coding immediately. This leads to frequent throwing away of code and re-starting. The program is seldom finished in time.
 - *The Poor Programmer* who spends little time on design and planning the algorithm, rushes to coding, and spends most of the time testing.

This programmer claims to be “90% finished” for over 90% of the time.

- *The Persistent Plodder* who spends more time designing and planning the algorithm, starts slower on the coding, requires less time for testing, and finishes just in time.
- *The Perpetual Planner* who starts very slowly, spends much time in design, and never gets far into coding before the deadline.
- *The Perfect Programmer* who spends a reasonable amount of time in design and creating the algorithm, which results in fast coding and testing, and may be finished before the deadline.
- *Large programs*, or programming systems, are those programs that require more than a single person. Not only are such programs usually complex, but the task of creating them is made more complex by the difficulties inherent in maintaining good communication between members of the programming team. The larger the program, the larger the team and the greater the possibility of the communication problems.

After large programs are completed, they are frequently modified throughout their lives. This modification, or maintenance, is shown in Figure 2.29 as taking 60% of all of the time, money and effort spent on a program. Often, maintenance takes over 70%!

Figure 2.29 A large project



Life Cycles of large programming projects have the typical form shown in Figure 2.29. The four steps in this figure are explained as follows:

1. *Planning and specifying* involves studying the problem, analyzing the requirements, specifying functions (actions) and data. It encompasses Steps 1 and 2 of our seven step method.
2. *Designing and Developing* involves decomposing the whole project into parts, refining the parts, coding them, and documenting (describing) the design. It comprises Steps 3, 4, part of 5, and 6 of our seven step method.
3. *Testing and Evaluating* involves verifying the function, testing the performance, optimizing (improving time or space), and

validating usefulness. It makes up part of Step 5 of our seven step method.

4. *Operating and Maintaining* involves the training of users, operating the program, and correcting its errors. It also includes extending the uses, optimizing, and modifying to transport the program to another computer or system. It represents Step 7 of our seven step method.

2.7 Review Top Ten Things to Remember

1. Problem solving with a computer is best done by following a method. A seven step method was introduced to help you solve problems. This method can be adapted to your way of working. It includes the following steps:
 1. Problem Definition
 2. Solution Design
 3. Solution Refinement
 4. Testing Strategy Development
 5. Program Coding and Testing
 6. Documentation Completion
 7. Program Maintenance
2. The Seven-Step Method is not strictly sequential. You may find it necessary to go back and rework previous steps as a result of a later step. For example, sometimes the problem definition step and solution design step may overlap because design decisions may require more precision than has been provided by the problem definition.
3. Problems are solved by breaking them into subproblems, a method called divide and conquer. Break-out diagrams are particularly useful for breaking up problems. Their hierarchical structure reduces complexity by showing how sub-parts relate to the whole. Break-out diagrams can be used to represent space, time, objects, actions, and data, as well as the stages in a program's life-cycle.
4. Break-out diagrams can be drawn vertically or horizontally. Vertically, the main problem is the top-most box, followed by each level. Horizontally, the main problem is the left-most box with each break-out level placed to its right.
5. Break-out diagrams, or BODs, must possess four properties to be correct representations of problems and subproblems:
 - Consistency: Each break-out must be of the same kind. If the first box involves time, then the whole diagram must involve time.
 - Orderliness: All the blocks at the same level cannot overlap. Each block must be separate or independent.
 - Refinement: Each box on a given level must be able to fit back into the boxes at the previous level.
 - Cohesion: All of the boxes within a breakout box must fit together.

6. Algorithms may be represented in a variety of ways. This chapter discussed the following five different representations:
 - Verbal algorithms use any natural language.
 - Flowcharts use different shaped boxes joined by arrows indicating the flow of control.
 - Graphs (or plots) express algorithms in a tabular structure.
 - Data-flow diagram represent algorithms as “black boxes” which hide the tasks involved in obtaining the output from the input.
 - Pseudocode uses a notation that is a mixture of mathematics, logic, and natural language.
7. There are many ways to modify an algorithm. The following methods were covered in this chapter:
 - Generalizing involves making the algorithm apply to many cases.
 - Extending makes the algorithm include more cases.
 - Foolproofing makes an algorithm more reliable by anticipating erroneous input or other difficulties.
 - Embedding reuses an algorithm within another algorithm.
8. Alternative algorithms are equivalent methods of achieving a correct solution to a particular problem. To prove that alternative algorithms are equivalent, both algorithms must be tested using the all the test cases that were defined in Step 4, Testing Strategy Development. If the outputs are identical, then the algorithms are equivalent. Remember, comparing the outputs using one single input value is insufficient to determine equivalent algorithms.
9. Many programming languages can be used to code an algorithm. Some languages are more suited to certain needs than others. For example, Fortran is best suited for scientific applications while Pascal is used for teaching purposes.
10. The seven-step method is based on the life-cycle of programs. For large programs, maintenance—what happens to the program after it has been completed and put into operation—takes as much as 60% of the life cycle.

2.8 Glossary

Action: An operation performed in an algorithm or program.

Algorithm: A finite set of well-defined rules for the solution of a problem in a finite number of steps; e.g., a full statement of an arithmetic procedure for evaluating an employee's weekly pay.

Artificial intelligence: A field of study in computer science that aims at designing systems that exhibit "intelligent" behavior.

BOD: Break-out diagram.

Bottom-up design: A method of design by which the definition of a system starts with its innermost components.

Break-out diagram: A diagram used to represent a hierarchical decomposition.

Coding: The actual writing of a computer program.

Computer: A complex device that can input, store, process, and output information.

Continuous value: A value corresponding to some physical phenomenon.

Data-flow diagram: A diagram showing what happens to the data.

Data validation: Verification that the data conform to some given constraints.

Design: The activities preceding the actual code writing.

Detailed design: The elaboration of an outline solution into a set of algorithms.

Digital value: Numerical value.

Discrete value: Digital value.

Divide and conquer: A method by which a complex problem is divided in smaller and easier subproblems

Documentation: An important and necessary part of any program development

Embedding: The use of an already existing algorithm as part of another algorithm.

Execute: Perform the actions associated with an algorithm.

Flow of control: The sequence of actions in an algorithm or the order of execution of the various instructions of a program.

Flowchart: A graphical representation of an algorithm.

Foolproofing: Making an algorithm resistant to errors.

High-level language: A programming language specially designed for a particular field of applications and not for a particular computer.

Implementation: Practical realization and installation of an abstract design.

Low-level language: A programming language designed for a particular computer and specially adapted to that computer.

Maintenance: The last part of the life cycle of a program: whatever happens to the program once it has been developed, tested and released to the users.

Program: *n.* A plan for precisely defined actions to be performed by the computer to obtain the solution to the problem; *v.* To formulate such a plan.

Program listing: A printed list of the instructions in a program.

Programming Language: An artificial language established for writing computer programs.

Pseudocode: An informal language used to define algorithms; pseudocode is usually a mixture of natural language and mathematical notation.

Reuse: see Embedding.

Run: Execute.

Software: The collection of programs needed to operate a particular computer hardware.

Software life cycle: A description of the various phases of a program's life.

Specification: The precise and complete description of a problem to be solved on a computer.

Stepwise refinement: A method by which an algorithm is developed step by step, each step elaborating the previously defined steps.

Structure chart: A hierarchical diagram showing the structure of a computer solution to a problem.

Task: A part of a solution to solve a problem on a computer.

Test: An attempt to verify that a program operates in conformity with its specifications.

Test case: A set of input data together with the corresponding expected results.

Top-down design: A manner of designing a computer solution in which one starts from the global problem and decomposes it into smaller subproblems.

Trace: The execution of an algorithm with some specific values and the representation of the various variables at various steps of the execution.

Variable: An "information holder" whose contents can be used or changed by the program.

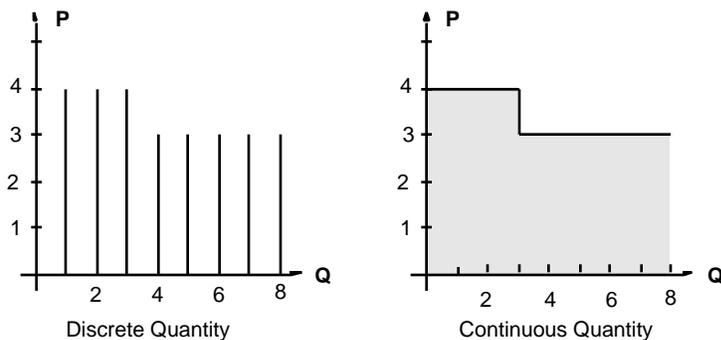
2.9 Problems

1. Price Break Problem

Suppose that the selling price P of something depends on the quantity Q that is purchased. If the quantity is less than or equal to 3 then the price is \$4 for each one, and if the quantity is over 3 then the price is \$3 for each one. For example, when the quantity $Q = 4$, the price $P = 3$ and the total cost T is

$$T = P \times Q = 3 \times 4 = \$12$$

Graphs of the unit price P versus quantity Q are shown below. The first graph describes items which are discrete or non-divisible (such as pens, pumpkins or puppies) and the second one describes items which are continuous or divisible (such as electric power or pounds of peanuts) which can have fractional or decimal values.



- a. Represent: Show

Draw a flowchart corresponding to this algorithm where the quantity Q is input and the total cost T is output.

- b. Analyze: Observe

Compute the total cost T when the quantity Q varies from 0 to 8. Draw this as a table with three columns, Q , P and T . Then, draw a graph of total cost T versus the quantity Q in both the discrete and continuous cases. Observe this graph for surprising insights.

Hint: How many items can be purchased for 12 dollars?

- c. Extend: Grow

Modify this algorithm if the price is further reduced to 2 dollars a unit when the quantity purchased is more than 5. Draw the flowchart for the extended algorithm. Draw also a graph of T versus Q for the discrete and continuous cases both shown on one graph.

d. Foolproof: fail-safe

Modify the above extended flowchart to include any foolproofing that would be necessary.

e. Embed: Repeat

Modify the above flowchart to repeat the process for as long as the input quantity Q is not zero.

f. Integrate: Generalize

Redraw all of the above features onto one flowchart, and generalize all numeric constants to named constants $P1$, $P2$, $Q1$, $Q2$, and so on.

g. Test: Evaluate

Select a set of test values to evaluate the final algorithm.

2. Scissors Search

Indicate which of the following verbal algorithms is better for finding an object, such as a pair of scissors, and explain why:

- a. Look on the rightmost end of the lower shelf of the middle cabinet in the garage.
- b. Look in the garage, in the middle cabinet, on the lower shelf, at the rightmost end.

3. Break-Out Problems

Create break-out diagrams describing four of the following:

- a. a telephone number
- b. the arrangement of five books on a shelf
- c. performing some process (laundry, cooking)
- d. your entire past life
- e. the plan for your present day
- f. the plan for the next five years
- g. the layout of newspaper sections
- h. anything else of interest to you.

4. Language Look

Even without knowing any programming languages, you can now make meaningful comparisons among the six programming languages presented in this chapter.

- a. Is multiplication represented by an asterisk (*) in all of these languages?
- b. What different symbols are used to represent the relation “is greater than”?
- c. What different verbs describe the output instruction?
- d. When some statements (formulas, etc.) are too long to fit on a line and continue on to another line, how is this indicated (if at all)?
- e. Write a prompt “Enter hours and rate” for each language.
- f. Describe any other differences and similarities that you find.

5. More Pay

- a. If the condition for overtime in the original payroll problem were changed, from (Hours \geq 40) to (Hours $<$ 40), would this change the amount of pay?
- b. If Hours is input as a negative amount (say -50 by mistake), is the output correct except for its sign?
- c. Modify the extended pay algorithm of Figure 2.16 to allow for triple-pay when the hours are greater than 80. Draw the corresponding graph and flow chart.

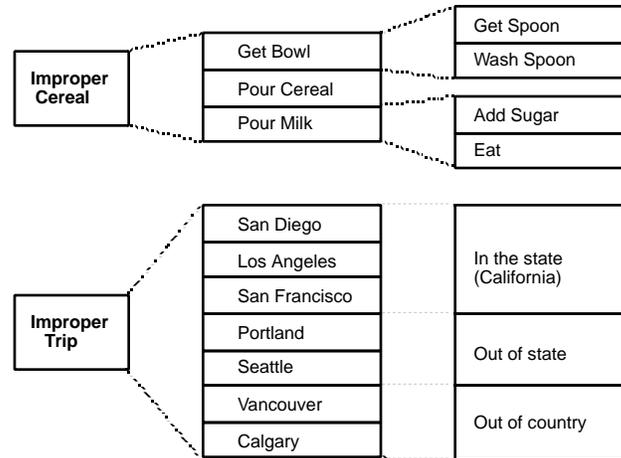
6. Constant Vehicle Velocity

Suppose that a vehicle travels at a constant velocity of 40 miles an hour for one hour, then at 20 miles per hour for another hour and finally at 60 miles per hour for the last hour. The distance D covered at constant velocity V for a given time T is the product $D = V \times T$. The total distance covered is the area under the V versus T curve.

- a. Draw a graph of the above Velocity versus Time curve.
- b. Create a flowchart to compute the distance traveled over the three ranges of time.
- c. Draw a graph of D versus T .

7. Beware Of Improper BODs

The following BODs are not proper. Why not? Redraw each of them in a better form.



8. Energy Rates

a. Not encouraging excessive use

Electrical energy rates are set in such a way that the rates decrease for higher usage. However, the rates should not encourage excessive use of energy, so the lower rates apply only to the higher energy amounts; not all of the energy used is at the low rate.

For example, the rate is constant \$4 per unit for the first 4 units and drops to \$3 a unit for more than 4 units. So when 6 units are used only the last 2 units are at the \$3 rate. Energy is a continuous quantity; it is not used in discrete steps.

(i) Represent, show

Draw a graph showing the price per unit versus the number of units used.

(ii) Analyze, observe

Create an algorithm to determine the total cost of energy, given the number of units used. Draw the graph of cost versus units.

(iii) Extend: grow

Modify this algorithm if the price per unit drops to \$2 a unit for more than 8 units. reuse. Draw the graph of cost versus units.

(iv) Foolproof, embed, and generalize

Complete this system in any way you see fit and draw the final algorithm in all of its detail.

(v) Test, evaluate

Select a set of test values to evaluate the final algorithm.

b. Discouraging Excessive Use

To discourage high energy use the rates could be set in such a way that the rates increase for higher usage. For example, the rate is a constant \$4 a unit for the first 4 units, and then increases to \$5 a unit for those amounts over 4 units.

Show the algorithm for this system, analyze it, foolproof it, and generalize it.

Draw the graph of cost versus units again.

c. Severe Discouraging of Excessive Use

An even more severe pricing policy would be a modification that when the higher quantity is used then the higher rate would be charged for all of the energy consumed, even the amounts at the low levels.

Show the algorithm and draw the corresponding graph again.

d. Comparison of Pricing Policy

Compare the above three policies by computing the cost for using 6 units of energy. Then compare by putting all policies onto one graph.

9. Ideal Weight

A man should weight 106 pounds for the first 5 feet, and 7 pounds for every inch above that. A woman should weigh 100 pounds for the first 5 feet, and 6 pounds for every inch over that.

- Create an algorithm in flowchart form that outputs the ideal weight when input the sex and height (feet and inches).
- Represent this algorithm as two tables from 5 feet to 6 feet 5 inches.
- Represent this algorithm in a graphic form, with 2 graphs on one grid.

10. Dog's Life

An algorithm that relates a dog's age to the corresponding human's age follows:

A one year old dog is equivalent in age to a 15 year old human. In the second year, the dog grows 10 human years older, and each year after that, it grows 5 human years.

- Create an algorithm as a flow chart to show the human age for any given dog age.
- Represent this algorithm as a table, with the dog's age varying from 1 to 10.

- c. Represent this algorithm as a graph both discrete and continuous.

11. Sales Commission

A salesman receives a commission or rate of profit of 4% for sales up to \$300K (where K is \$1000); then the rate doubles to 8% only for the sales at or above this level. Create an algorithm in flowchart form which describes this profit policy. Provide also a table and graph of profit versus rate for the rate varying from 0 to 1000K in steps of 100K.

Another profit policy begins at a higher rate of 5% and changes to 7% for sales over 600K. Create a table of this algorithm and draw it on the above graph. What is the significance of the intersection of these two graphs?

12. Telephone Rates

The rate for use of a telephone depends on the time of call, which is measured as MPMs or minutes past midnight (ranging from 0 to 60×24). The “day rate” from 6am to 6pm is given in dollars per minute, and the “night rate” as a proportion of this day rate.

If the rate is determined at the beginning of the call and remains fixed at that value, create an algorithm that computes the total cost for any call given the start time and the terminating time, both in MPMs.

If the rate can change during a call (when it lasts past the 6 o'clock times) then create the new algorithm to compute the cost.

Chapter 3 Algorithms

In this chapter, we study algorithms in some depth, discuss their necessary and desirable properties, provide examples of them and show various ways of representing them.

Chapter Outline

3.1	Preview.....	74
3.2	What Are Algorithms?.....	75
	Algorithm Definition.....	75
	General Properties of Algorithms.....	75
	Desirable Attributes for Algorithms.....	77
3.3	Different Algorithm Representations.....	78
	Verbal Representations.....	80
	Algebraic Representations (formulas and expressions).....	82
	Tabular Representations (tables, arrays, and matrices).....	86
3.4	Data-Flow Diagrams.....	89
	Black Boxes vs. Glass Boxes.....	89
	General Data-Flow Diagrams.....	91
	Integer Data-Flow Diagrams.....	94
	Logical Data-Flow Diagrams.....	98
	Data Flow Black Boxes.....	101
	More on Data Flow Components and Diagrams.....	105
3.5	Flow of Control Diagrams.....	107
	Flowcharts.....	107
	Larger Flowcharts Using Subprograms.....	110
	Flowblocks.....	113
	Pseudocode.....	115
3.6	Review Top Ten Things to Remember.....	118
3.7	Glossary.....	119
3.8	Problems.....	121

3.1 Preview

Our problem solving method requires the development of algorithms as part of its third step, Solution Refinement. To be able to complete this step, we need to know a lot more about algorithms. This chapter will help us do just that, for we will:

- Study algorithms in some depth
- Discuss their necessary and desirable properties
- Provide many examples of algorithms, and
- Show various ways of representing algorithms.

Algorithm is a plan for performing a sequence of actions on data to achieve an intended result. To be useful, the algorithm must be precise, unambiguous, and specify, for all possible cases, a unique and finite sequence of actions that achieve a predictable result. In addition, an algorithm should be applicable to a number of problems rather than to a single instance, have a simple structure, and be easy to use efficiently.

Algorithms can be *represented* in many different ways. For any algorithm, there may be many different representations, some much better than others. No one representation is best for all algorithms. The representations considered in this chapter are as follows:

- *Verbal*: The algorithm is expressed in words.
- *Algebraic*: The algorithm is expressed mathematically with symbols and formulas.
- *Tabular*: The algorithm is represented by one or more tables
- *Hierarchical*: The algorithm is presented as a break-out diagram.
- *Data-flow diagram*: The algorithm is shown as a set of boxes that show the actions to be performed. These boxes are linked by lines showing how data flows from one action to another. This is referred to as the flow of data.
- *Flowchart*: The algorithm is represented in the form of a diagram with action boxes linked by lines showing the order in which they are executed, or the sequence of actions. This is referred to as the flow of control.
- *Flowblocks*: Like flowcharts, the algorithm is represented by action boxes. Instead of connecting the boxes with lines, the flow of control is illustrated by stacking boxes on top of each other, or by nesting boxes within other boxes.
- *Pseudocode*: The algorithm is presented as a set of instructions written using a mixture of natural language and mathematical notation. The

form of instructions in pseudocode is similar to that of programming languages.

Flow of data and flow of control provide the so-called *black box* and *glass box* representations of algorithms. The black box view, corresponding to flow of data, represents an action as accepting inputs and producing outputs. This view hides the internal details and concentrates on *what* the action is. The glass box view, on the other hand, corresponds to flow of control and shows the details of *how* an action is performed. Both black box and glass box views can be broken out in a top-down manner.

Many examples of small algorithms are shown in this chapter. You should scan all the algorithms and concentrate on those that interest you.

Although the many representations of algorithms seem overwhelming, not all forms are equally important; the forms of lesser importance are included for completeness and are only described briefly. Ultimately, you will come to prefer some representations over others.

3.2 What Are Algorithms?

Algorithm Definition

An *algorithm* is a precise plan for performing a sequence of actions to achieve the intended purpose. Each action is drawn from a well-understood repertoire of actions on data. Here are some examples of algorithms:

- Prepare breakfast.
- Decide how much to charge for admission to a cinema.
- Calculate the average of a group of values.
- Change a car tire
- Play a dice game

Notice that each of the above algorithms specify two things.

1. An action specified by verbs such as “prepare”, “change”, and “play”
2. The data to be acted on, specified by a noun or noun phrase such as “breakfast”, “a car tire”, and “a dice game”.

Many of these algorithms will be considered in detail later, for now it is sufficient to understand that algorithms are common to everyday life, and do not necessarily involve computers.

General Properties of Algorithms

An algorithm must possess the following four properties:

- *Complete*: For an algorithm to be complete, all of its actions must be exactly defined. Consider the “algorithm” from Chapter 2 for getting to Fred’s Pizza Parlor, for example:

Directions for Getting to Fred’s Pizza Parlor

1. Take Route 116 until you come to the T-junction at Route 9.
2. Turn onto Route 9 and go about three-quarters of a mile and you will come to it on the left side. You can’t miss it!

When your get a set of directions that finishes with “You can’t miss it!”, you know you are in trouble. For starters, in which direction should we drive on Route 116, north or south? If we had some knowledge of the area, it might be reasonable to expect us to know. But the directions, as they stand, are not precise, and **do not** represent an algorithm!

- *Unambiguous*: A set of instructions will be unambiguous if there is only one possible way of interpreting them. For example, even if we accept the directions to the pizza parlor as being precise enough, they are still ambiguous about the direction to turn when we reach Route 9. We can not assume local knowledge because, if we knew which way to turn, we would know how to get to Fred’s. The ambiguity can only be resolved by trial and error. When we get to the junction of Route 116 and Route 9, we can try going left to see if we find the parlor. If Fred’s is not left, it must be to the right.
- *Deterministic*: This third property means that if the instructions are followed, it is certain that the desired result will always be achieved. The following set of instructions does not satisfy this condition:

Algorithm for becoming a Millionaire

1. Take all your money out of the bank and change it into quarters.
2. Go to Las Vegas
3. Play the slot machines until you either win four million quarters or you go broke.

Clearly, this “algorithm” does not always achieve the desired result of becoming a millionaire. It is most likely that you will finish with no money. However, you just might achieve the goal of becoming a millionaire, winning four million quarters. The point is that we cannot be certain what the result will be. This makes the set of instructions non-deterministic, and thus does not constitute an algorithm.

- *Finite*: The fourth requirement means that the instructions must terminate after a limited number of steps. The “algorithm” for becoming a millionaire fails this criterion too. It is possible that you will never reach either the stage of having four million quarters or of going broke. In other words, if you follow the instructions, you might end up playing the slot machines forever.

The finite requirement not only means termination after a finite number of steps, it also means that the instructions should use a finite number of variables to achieve the desired result.

Desirable Attributes for Algorithms

Although an algorithm may satisfy the criteria of the previous section by being precise, unambiguous, deterministic, and finite, it still may not be suitable for use as a computer solution to a problem. The basic desirable attributes that an algorithm should meet are explained below.

- *Generality*: An algorithm should solve a class of problems, not just one problem. For example, an algorithm to calculate the average of four values is not as generally useful as one that calculates the average of an arbitrary number of values.

Among its other failures, our “algorithm” to get to Fred’s Pizza Parlor lacks generality for it does not help us get to all pizza parlors. For example, it would not get us to Romano’s Pizza Palace.

- *Good Structure*: This attribute applies to the construction of the algorithm. A well-structured algorithm should be created using good building blocks that make it easy to...
 - Explain,
 - Understand,
 - Test, and
 - Modify it.

The blocks, from which the algorithm is constructed, should be interconnected in such a way that one of them can be easily replaced with a better version to improve the whole algorithm, without having to rebuild it.

- *Efficiency*: an algorithm’s speed of operation is often an important property, as is its size or compactness. Initially, these attributes are not important concerns. First, we must create well-structured algorithms that carry out the desired task under all conditions. Then, and only then, do we improve them with efficiency as an objective.
- *Ease of Use*: This property describes the convenience and ease with which users can apply the algorithm to their data. Sometimes what makes an algorithm easy to understand for the user, also makes it difficult to design for the designer (and vice versa).
- *Elegance*: This property is difficult to define, but it appears to be connected with the qualities of harmony, balance, economy of structure and contrast, whose contribution to beauty in the arts is prized. Also, as with the arts, we seem to be able to “know beauty when we see it” in algorithms. Sometimes the term *beauty* is used for this attribute of an algorithm.

Other desirable properties, such as *robustness* (resistance to failure when presented with invalid data) and *economy* (cost-effectiveness), will only be touched upon in this chapter. Not because these properties are unimportant, but because the basic attributes of an algorithm should be understood first.

To see an example of elegance, see Figure 3.21.

3.3 Different Algorithm Representations

Algorithms may be specified in many forms—these are called *representations* or *notations*. The representation of an algorithm is extremely important because it must rapidly convey the algorithm's meaning with the least amount of effort by the reader. No one representation is suitable for all algorithms. The best representation for one algorithm may be the worst for another.

A representation should serve as a mental beast of burden. It serves to relieve the brain of unnecessary work by reducing the perceived complexity of what is being described. Since we all have a limit to the amount of complexity we can handle, a good representation sets us free to concentrate on more advanced problems.

The significance of proper representation becomes obvious when you consider the many ways that numbers may be represented. For example, the following forms all refer to the year 1984:

- 1984: Hindu-Arabic notation, base 10
- Nineteen eighty four: English words
- MCMLXXXIV: Roman numerals
- 1111100000: Binary, base 2
- 3700: Octal, base 8
- '84: Shortened form

Each of these forms is useful for a specific purpose. The binary form is used by computers and the shortened form ('84) is useful when writing a check and so on. For some purposes, certain forms are much better than others. For example, the Hindu-Arabic numerals are a much better representation for arithmetic than Roman numerals, as the following example attests.

$$\begin{array}{rcl}
 63 & \text{versus} & \text{LXIII} \times \text{VII} = \text{L} \times \text{V} + \text{L} + \text{L} + \text{X} \times \text{V} + \text{X} + \text{X} + \text{V} + \text{I} + \text{I} + \text{V} + \\
 \times 7 & & \text{I} + \text{I} + \text{V} + \text{I} + \text{I} \\
 441 & & = \text{L} + \text{X} + \text{X} + \text{X} + \text{X} + \text{X} + \\
 & & \text{X} + \text{X} + \text{V} + \text{V} + \text{V} + \text{I} + \text{I} + \text{I} + \text{I} + \text{I} + \text{I} \\
 & & = \text{C} + \text{C} + \text{C} + \text{L} + \text{L} + \text{X} + \text{X} + \text{X} + \text{V} + \text{V} + \text{I} \\
 & & = \text{C} + \text{C} + \text{C} + \text{C} + \text{X} + \text{X} + \text{X} + \text{X} + \text{I} \\
 & & = \text{CDXLI}
 \end{array}$$

Before we look down on the Romans for their number system, we must remember that their numerals were not developed for arithmetic. When they came into existence, almost the only use for numbers was counting and the Roman numerals I, V and X were very simple to notch on tally sticks. Carpenters to this day still form these numerals with their axes when they number the beams and timbers they have fashioned. The other Roman numerals for larger numbers, M, C, D

and \mathbb{L} , did not have much place in early counting. They are thought to have evolved from these counting marks.

Normally, for arithmetic purposes, there is one standard form or notation (Hindu-Arabic, base 10 form). Unfortunately, in computing, there is no such standard when it comes to representing algorithms.

Selecting the proper representation for algorithms is not a simple task for there are no guidelines which we can follow. The only way to determine the best representation is to try at least two forms and see which one you like the best. After a few tries, you'll find a representation that you prefer, or you'll have developed your own by combining a couple of the representations introduced here. In any case, you should develop your algorithms using the representation that best fits your style.

Without a good representation, it is very difficult to express an algorithm. For example, we probably know the algorithm for making change for 12 cents when given a dollar. We would add from the 12 cents up to the dollar. For our specific example, we would first give 3 pennies (to make 15 cents), then a dime (to make 25 cents), and then 3 quarters to finally add up to the dollar tendered.

Although we know how to add up to a dollar from 12 cents without the help of a cash register, we would have difficulty describing how to make change for an arbitrary amount. To describe this algorithm precisely to a computer would be even more difficult for us. The point is that knowing how to do something is very different from being able to describe precisely how to do it! To use the words of St. Augustine,

“If no one asks me, I know what it is. If I wish to explain it to he who asks, I do not know.”

If we possess a way of representing algorithms, then we will be able to express ourselves. In fact, representation can be viewed as a powerful tool that helps us think of things that we would have not have been able to think of without it.

We must also realize that there may be many ways of creating an algorithm that makes change. This might help us find a method that is simpler in some ways than the one we usually use. This will lead us to the change-making algorithm that we will see later in this chapter. It is a convenient algorithm for us to communicate to a computer, but at the same time, its an algorithm that we would not want to used for ourselves. In this book, we will meet about half a dozen different algorithms that accomplish this same change-making task.

In the rest of the chapter we will consider many algorithm representations, and for each representation, we will give several example algorithms. Look briefly at all the examples, but consider in detail only a few that are of interest to you. These algorithms are meant to communicate to people. Later some may be modified to run on machines. Please realize that it is not required that you understand all the algorithms. On the other hand, you should be able to follow them, for that is precisely what computers do.

Verbal Representations

The following figures (3.1 to 3.5) present some very common examples of algorithms. They appear here in a simple verbal form (representation) and will be transformed into other forms later. Remember, For any particular algorithm, one representation may be much better than another; therefore, it is worth getting familiar with a number of alternative forms.

The algorithm Charge admission, of Figure 3.1, specifies an admission charge that depends on the age of the person being admitted; it consists of a procedure to determine the charge and some examples showing the results of applying the procedure.

Figure 3.1 First verbal algorithm Charge admission

<p>Kids under 12 pay \$2</p> <p>All others pay \$3</p>
<p>For example:</p> <p>12 year olds pay \$3.</p> <p>21 year olds pay \$3.</p> <p>2 year olds pay \$2.</p>

Figure 3.2 shows another algorithm expressed in a verbal form which specifies a method to determine which years are leap years. It is usually sufficient to check if 4 divides evenly into the year, but if the year marks a century, then the algorithm is more complex. For example, the year 1900 was not a leap year, while the year 2000 will be a leap year.

Figure 3.2 Second verbal algorithm Leap Year

<p>A leap year is divisible by 4 but, if it is divisible by 100, then it is not a leap year, unless it is also divisible by 400.</p>
<p>For example:</p> <p>1984 was a leap year.</p> <p>1900 was not a leap year.</p> <p>2000 will be a leap year.</p> <p>2100 will not be a leap year.</p>

Our third verbal algorithm example, Dice Game (Figure 3.3), specifies how a simple game is played, using two dice having one to six dots on each side.

Figure 3.3 Third verbal algorithm Dice Game

Two dice are thrown. If their sum is 7 or 11, you win, and if the sum is 2, 3 or 12, you lose. If neither rolls come up, remember the sum, the “point count”, and keep throwing until either:	
the sum equals the point count, or	
the sum equals 7 and you lose..	
For example, consider these sequences:	
7	you win
6, 7	you lose
4, 2, 11, 3, 4	you win
9, 3, 4, 12, 2, 8, 3, 7	you lose

Figure 3.4 shows a fourth example, Ideal weight, which describes one view of the relationship between height and weight.

Figure 3.4 A fourth verbal algorithm Ideal Weight

A man should weigh 106 pounds for the first 5 feet of height plus 7 pounds for every inch above that; a woman should weigh 100 pounds for the first 5 feet of height plus 6 pounds for every inch above that.
For example:
A woman 5ft. 10in tall should weigh $100 + 6 \times 10 = 160\text{lbs.}$
A man 6ft. 0in. tall should weigh $106 + 7 \times 12 = 190\text{lbs.}$

Our final example for the verbal representation of algorithms, ISBN, shown in Figure 3.5, describes the way to determine the last digit for the International Standard Book Number. Most recent books have on their back covers a special ten-digit ISBN such as:

0-06-500871-5 (the hyphens are not important)

The first digit, 0, represents the book’s area of origin. 0 means that it was published in an English-speaking country. The second group represents the publisher; 06 means that it was published by Harper Collins. The third group, 500871, represents the book’s title and is assigned by the publisher. The last digit, 5, is a check digit that is computed from the nine preceding digits as shown in the algorithm.

Figure 3.5 A verbal algorithm ISBN Checksum Code

<p>Find the sum of:</p> <p style="padding-left: 40px;">the first digit and</p> <p style="padding-left: 40px;">two times the second digit and</p> <p style="padding-left: 40px;">three times the third digit and</p> <p style="padding-left: 40px;">...so on to ...</p> <p style="padding-left: 40px;">nine times the ninth digit.</p> <p>Divide this sum by 11 and find the remainder.</p> <p>If the remainder is less than 10, then the remainder becomes the checksum, otherwise the checksum is the character 'X'.</p>
<p>For example:</p> <p style="padding-left: 40px;">0-06-500871 has the checksum 5.</p> <p style="padding-left: 40px;">0-387-96939 has the checksum X.</p>

For example, the “weighted sum” of the first number in Figure 3.5 is...

$$1 \times 0 + 2 \times 0 + 3 \times 6 + 4 \times 5 + 5 \times 0 + 6 \times 0 + 7 \times 8 + 8 \times 7 + 9 \times 1 = 159$$

Dividing this sum by 11 yields a remainder of 5, which is the checksum. This code is used for error checking when ordering a book, for example. If the computed checksum does not equal the last digit of the ISBN, then an error has been made in copying this book number. This is a very useful error detecting method because it detects the most common copying error, that of transposing adjacent digits.

Algebraic Representations (formulas and expressions)

Many algorithms, especially in mathematics and engineering, are expressed in a mathematical form as a formula or algebraic expression. This is often a concise form, convenient for computers. Some examples of this algebraic form follow. Again, look quickly at all of these, but concentrate on just a few. The first examples are simple and become more and more complex as you go on.

The algorithm Charge Admission, that we saw earlier in a verbal representation (Figure 3.1), is redefined in Figure 3.6. This example computes the total admission charged, given the number of adults A and the number of kids K.

Figure 3.6 A formula algorithm version of Charge Admission

$C = 3 \times A + 2 \times K$ <p>where</p> <p>A is the number of Adults</p> <p>K is the number of Kids</p>
<p>For example:</p> <p>For 2 Adults and 3 Kids</p> $C = 3 \times A + 2 \times K = 3 \times 2 + 2 \times 3 = \12

As a second example of a formula algorithm, let's look at Time Conversion in Figure 3.7. This is an algorithm for converting days, hours, minutes, and seconds into seconds. This conversion is done in two ways: the first way seems natural, while the second way is derived from the first by factoring out common terms, which results in half as many multiplication operations. Check that the second way produces the same result for the example shown.

Figure 3.7 A second formula algorithm Time conversion

$T = S + 60 \times M + 60 \times 60 \times H + 24 \times 60 \times 60 \times D$ <p>or alternatively</p> $T = S + 60 \times (M + 60 \times (H + 24 \times D)),$ <p>where</p> <p>S is the number of Seconds,</p> <p>M is the number of Minutes,</p> <p>H is the number of Hours,</p> <p>D is the number of Days.</p>
<p>For example:</p> <p>For 1 day, 2 hours, 3 minutes, 4 seconds</p> $T = 4 + 60 \times 3 + 60 \times 60 \times 2 + 24 \times 60 \times 60$ $= 93\,784 \text{ seconds}$

Another conversion algorithm is shown in Figure 3.8. Temperature Conversion shows two alternative pairs of formulas for converting between Celsius and Fahrenheit temperatures. You may wish to check that -40° Celsius converts to -40° Fahrenheit and back.

The alternative formulas present another way of performing the conversions. These alternative formulas also present an elegant symmetry that avoids the necessity of remembering when to add or subtract the 32.

Figure 3.8 A third formula algorithm Temperature Conversion

$C = \frac{5}{9} \times (F - 32)$ $F = \frac{9}{5} \times C + 32$ <p style="text-align: center;">or alternatively</p> $C = \left(\frac{5}{9} \times (F + 40)\right) - 40$ $F = \left(\frac{9}{5} \times (C + 40)\right) - 40$
<p>For example:</p> <p style="padding-left: 40px;">For F = 212°F</p> $C = \frac{5}{9} \times (212 - 32) = 100^\circ\text{C}$ <p style="padding-left: 40px;">For C = 20°C</p> $F = \frac{9}{5} \times 20 + 32 = 68^\circ\text{F}$

The next example, Figure 3.9, shows two different algorithms that compute the square of any positive integer number N. In the first formula, the number N is added to itself N times. Alternatively, in the second formula, the square of an integer N is determined by summing the first N odd integers. This Square example illustrates again that there may be many ways to do the same thing.

Figure 3.9 A Square formula algorithm

$S = N + N + N + \dots + N \text{ (N times)}$ <p style="text-align: center;">or alternatively</p> $S = 1 + 3 + 5 + 7 + \dots + (2N - 1)$
<p>For example, if we denote the Square of 7 as Square(7):</p> $\text{Square}(7) = 7 + 7 + 7 + 7 + 7 + 7 + 7 = 49$ $\text{Square}(7) = 1 + 3 + 5 + 7 + 9 + 11 + 13 = 49$

Statistical measures like mean and variance are shown as formulas in Figure 3.10. The mean of N given values, M, is found by summing all the values and dividing this sum by N. For example, here is the Mean of the 4 values 10, 20, 30 and 40:

$$M = (10 + 20 + 30 + 40) / 4 = 25$$

For other methods of finding the variance, see Problem 18 in Section 3.8 and see Chapter 8.

The variance, V , is a measure of the amount of variation of the values about this mean value. It is computed by taking the average of the square of the differences of the values from the mean, M , as shown in Figure 3.10.

Figure 3.10 A Mean and Variance formula algorithm

$$M = (X_1 + X_2 + X_3 + \dots + X_N) / N$$

$$V = [(X_1 - M)^2 + (X_2 - M)^2 + \dots + (X_N - M)^2] / N$$

For example, for the four values 10, 20, 30, 40:

$$M = (10 + 20 + 30 + 40) / 4 = 25$$

$$\begin{aligned} V &= [(10 - 25)^2 + (20 - 25)^2 + (30 - 25)^2 + (40 - 25)^2] / 4 \\ &= 125 \end{aligned}$$

The Factorial of a number N , denoted by $N!$, may be computed by the algorithm shown in Figure 3.11. The factorial $N!$ is computed as the product of N with all the positive numbers less than N . For example:

$$7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 5040$$

$$\begin{aligned} 10! &= 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 \\ &= 10 \times 9 \times 8 \times 7! \\ &= 362800 \end{aligned}$$

Notice that as N increases, Factorial N increases very quickly.

Figure 3.11 A Factorial formula algorithm

$$N! = N \times (N - 1) \times (N - 2) \times \dots \times 2 \times 1$$

For example:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Figure 3.12 shows a formula to compute the trigonometric sine of an angle X expressed in radians. This sine formula refers to the factorial formula shown in Figure 3.11.

Only a few terms of the series are required since their values decrease very quickly (because of the rapidly increasing value of the factorial in the denominator). If the value of X is 1, then the first two terms of this series yield this approximation:

$$\text{SIN}(X) = X - X^3 / (3 \times 2 \times 1) = 1 - 1/6 = 5/6 = 0.83333$$

Comparing this approximate value of 0.83333 to the actual value of 0.842 shows that using only two terms still yields a good approximation. The value of the next term in the series is 0.00833: if we add this, we get 0.8416, a very close

approximation. The more terms of the series we use for our computation, the more accurate the approximation we obtain.

Figure 3.12 A Sine formula algorithm

$\text{SIN}(X) = X - X^3/3! + X^5/5! - X^7/7! - \dots$
<p>For example:</p> $\begin{aligned} \text{SIN}(1) &= 1 - 1/(3 \times 2 \times 1) + 1/(5 \times 4 \times 3 \times 2 \times 1) \\ &= 1 - 1/6 + 1/120 - \dots \\ &= 0.8416 \end{aligned}$

Figure 3.13 shows a formula to convert binary numbers (base 2) into decimal numbers (base 10). For example, the binary number 11001 can be converted to the decimal number 25:

$$\begin{aligned} (11001)_2 &= 2^4 \times 1 + 2^3 \times 1 + 2^2 \times 0 + 2^1 \times 0 + 2^0 \times 1 \\ &= 16 + 8 + 0 + 0 + 1 = 25 \end{aligned}$$

This process can be generalized to any other base B by replacing the base value of 2 by B. For example, this algorithm can convert octal to decimal numbers simply by changing the base from 2 to 8.

Figure 3.13 A Base conversion formula algorithm

<p>Binary to Decimal Conversion</p> $(B_n \dots B_3 B_2 B_1 B_0)_2 = 2^n B_n + 2^{n-1} B_{n-1} + \dots + 2^2 B_2 + 2B_1 + B_0$
<p>For example:</p> $\begin{aligned} (110)_2 &= 4 + 2 + 0 = 6 \\ (1101)_2 &= 8 + 4 + 0 + 1 = 13 \\ (1000000)_2 &= 2^6 \times 1 = 64 \end{aligned}$

Tabular Representations (tables, arrays, and matrices)

Tables are rectangular grids which store values. They are often convenient for representing algorithms. Tables of various kinds are encountered in mathematics (matrices), in business (decision tables or spreadsheets), in logic (truth tables) and in computing (arrays).

An algorithm for computing the number of days in a month is shown as a table in Figure 3.14. Here the months are represented by integers from 1 to 12. Corresponding to each integer is a number indicating the number of days in that month.

In the square for February there is a break-out diagram because, depending on whether or not it's a leap year, February may consist of 28 (normal year) or 29 (leap year) days. By using a break-out diagram to illustrate these two possible values, an important point is presented: one algorithm (a *sub-algorithm*) may be used within another algorithm.

Notice that almost every second month has 31 days; the exceptions occur in July and August. This came about because Julius Caesar could not stand that the month named after him had fewer days than the month named after Caesar Augustus.

Figure 3.14 First “table” algorithm Days in a month

**Days in a Month
(1 Index)**

Month	Days
1	31
2	Leap
3	31
4	30
5	31
6	30
7	31
8	31
9	30
10	31
11	30
12	31

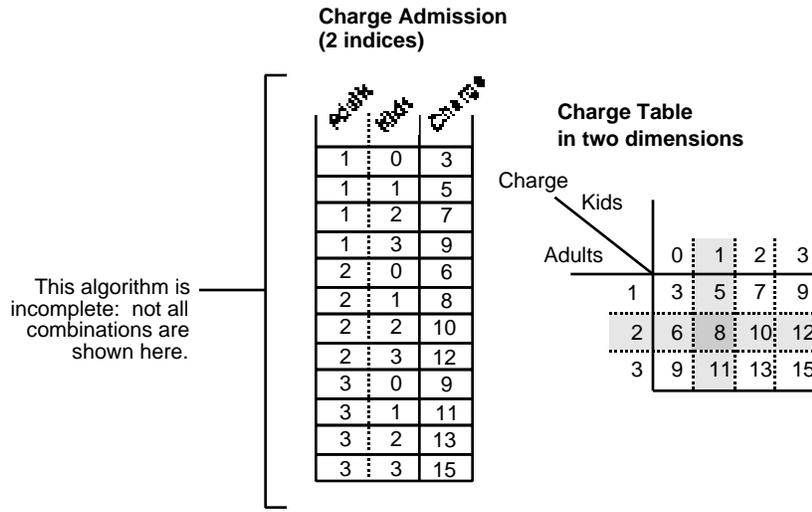
Leap Sub-table

Leap Year	Feb Days
NO	28
YES	29

For more on the property of completeness, consult Section 3.2 under General Properties of Algorithms.

The Charge Admission algorithm, which we have encountered twice previously in Figures 3.1 and 3.6, is shown as a table in Figure 3.15. In this algorithm, admission is \$3 per Adult and \$2 per Kid, some frequent combinations of Adults and Kids are computed once and then can be referred to later to save further computation. Although the property of completeness is not satisfied (not all combinations are shown), this table is still convenient and useful since it specifies the charge for the most common combinations of Adults and Kids.

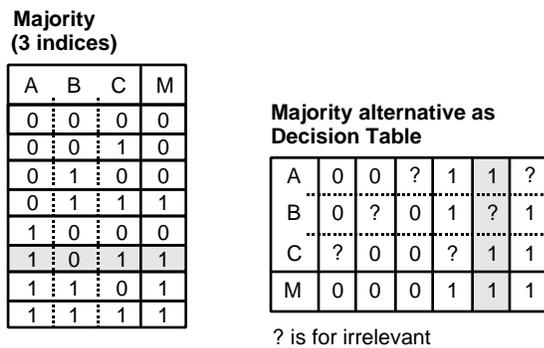
Figure 3.15 Second “table” algorithm Charge admission



As an alternative, the two-dimensional version of this algorithm—with the same combinations, is shown at the right of Figure 3.15. In this table, for example, the Charge corresponding to 2 Adults and 1 Kid can be determined by moving along row 2, and down column 1. The point where the row and column intersect (Charge = 8) indicates the amount to charge for this combination.

The next algorithm, Majority, shown in Figure 3.16, compares three variables A, B, and C. This algorithm determines which of two values (0 or 1) appears the majority of the time. For example, if A = 1, B = 0 and C = 1, then the Majority value is 1, as indicated in the third from last row. In Logic, similar tables are called *truth tables*.

Figure 3.16 Third “table” algorithm Majority



Decision tables are alternative forms of truth tables. They are used in many business applications that involve complex combinations of decisions because decision tables provide an easy way to check for completeness and consistency, since all combinations are listed.

A decision table for Majority is shown at the right of Figure 3.16. This table is smaller than the table on the left of Figure 3.16, because it has only 6 rules (combinations of the conditions A, B, C). For example, consider the first column (rule); if A and B conditions are both 0, then regardless of condition C the Majority action is 0. Often the conditions are labeled with values of Yes and No, or True and False instead of 0 and 1.

Indices refer to a position in a table. For example, the tables corresponding to Days in a Month and Leap, shown in Figure 3.14, only have one index each (Month and Leap Year). The table for Charge Admission, shown in Figure 3.15, has two indices labeled Adults and Kids. The tables for Majority in Figure 3.16 have three indices labeled A, B, C. Indices are also called *subscripts*.

Note: In the decision table in Figure 3.16, the “?” indicates a value that has no effect on the outcome and, therefore, that does not need to be examined.

3.4 Data-Flow Diagrams

Black Boxes vs. Glass Boxes

Algorithms can be considered from two points of view, both involving flows: data flow and control flow. Data flow emphasizes the flow of data between the actions, while control flow emphasizes the sequence of actions.

In *Data-Flow Diagrams* (or DFDs), the actions that constitute the algorithms are represented as black boxes and the lines show data flow between them. The temporal sequence of actions is not represented. This view is in direct contrast with flowcharts, which emphasize the sequence of actions. Data-flow diagrams describe mainly the function of an algorithm: *what* it does, rather than *how* it does it.

Figure 3.17 shows this difference with a Divide algorithm that divides a Numerator by a Denominator and produces a Quotient and a Remainder. The data-flow diagram is at the left of the figure and provides no indication as to how the Divide operation is done. This diagram only shows the following three things:

- What the Divide algorithm does,
- What Divide takes as input, and
- What Divide produces, or outputs.

On the right side of Figure 3.17, a flowchart for the same Divide operation shows how the division is done, hence the name “glass box”.

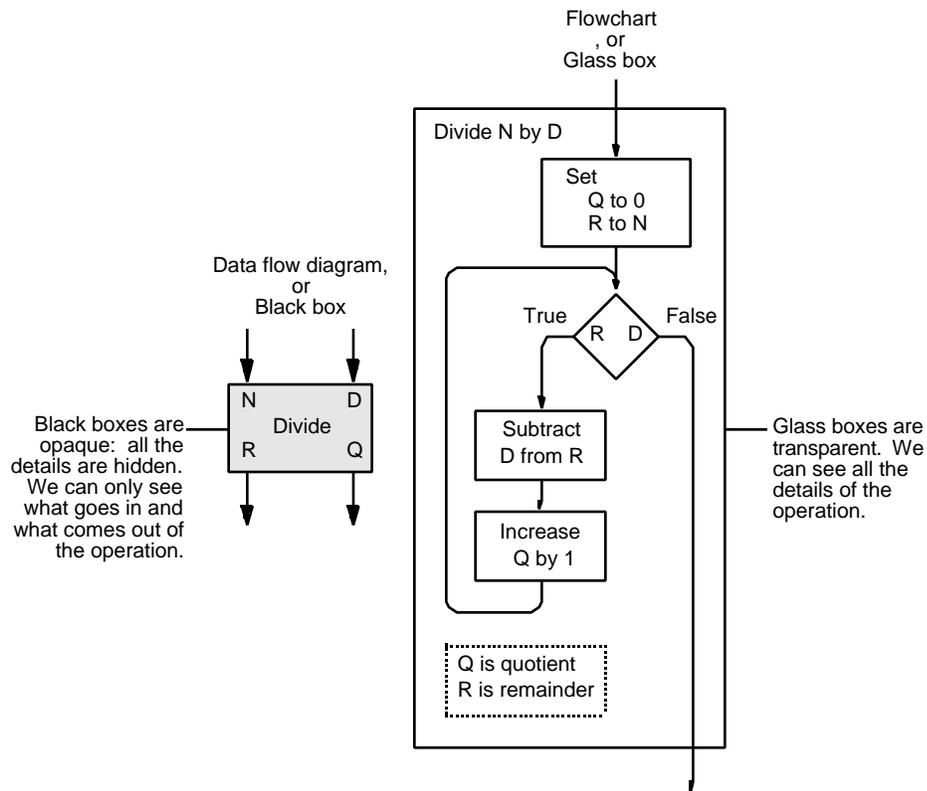
Examples of the “black box” view are shown later in this section.

Examples of the “glass box” view are shown in section 3.5, using three different representations: flowcharts, flowblocks and pseudocode.

A *Black box* describes a system that accepts inputs, and produces outputs, while hiding the internal details of the transformation. This view shows what is being done by each box and how the boxes interconnect. It provides a higher level “bird’s eye” view, as the low level details are not shown. This is the “black box” view provided by data-flow diagrams which use arrows to indicate the flow of data in and out of each box.

A *Glass box* describes the inner details of a system. This view shows how things are done within a box. It provides a lower level “worm’s eye” view. This view is represented by flowcharts and flowblocks, with arrowhead lines showing the flow of control between various boxes.

Figure 3.17 Black box vs. glass box



It is important to understand that these two views, black box and glass box, are complementary, and that each has its appropriate place. The black boxes are generally used first to give an initial, high-level specification of a system. The black boxes prevent us from filling our minds with details too early. The glass boxes are used, at lower levels, to deal with the details.

Choosing the right type of boxes to use is not simple. Sometimes it is very difficult to put an algorithm into one form, yet very easy to put it into the other form. Sometimes the black box data flow method is best; at other times, the

glass box control flow method is best. Sometimes we need to switch back and forth between these two views.

Top-down break up of systems into boxes is an important process. The larger boxes (usually black boxes) are broken-out into collections of smaller boxes. The smaller boxes, in turn, are broken down further until all boxes are sufficiently simple and can be transformed into glass boxes. Much more will be done with these two views later.

Black boxes hold a special significance. Each black box may be viewed as a sub-algorithm with input arrows representing values “passed in” and output arrows representing some result “passed out”. This allows us to consider some important concepts without having to get into great detail. For now, the boxes simply allow us to break out and manage the complexity of systems. Later, we will explore important concepts like sub-algorithms

General Data-Flow Diagrams

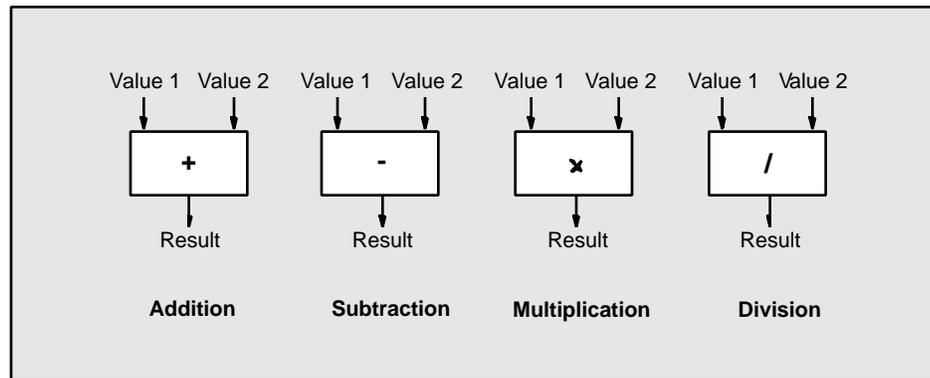
Let’s now concentrate on the black box approach and look at examples of data-flow diagrams.

Data Types describe the kind of data items that are considered: a data type encompasses values and the operations that can be applied to those values. It is not sufficient to say that the data are numbers, since in computer science, a big distinction is made between *Integers* (or whole numbers) and *Real Numbers* (those with a decimal point). The main reason for the distinction is that Integers and Real Numbers are represented differently by computers. We can say that Integers arise from counting, while Real Numbers come from measuring.

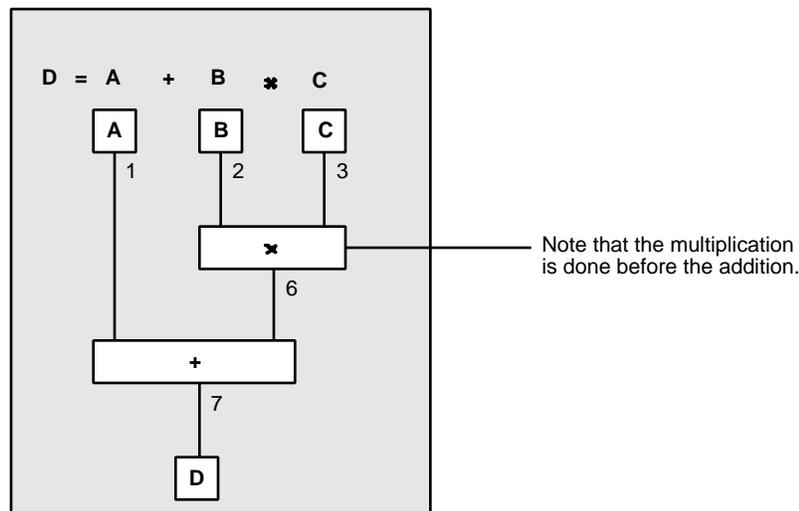
Real Numbers usually arise from measuring and are normally expressed with a decimal point such as 3.1415, -2.5 and 0.7. Sometimes, when the values are either very large or very small, they are expressed in a scientific or exponential notation such as 3.2E12, which means 3.2×10^{12} (which is 32 000).

The *operations* on Real Numbers are the four arithmetic operations shown in Figure 3.18: addition, subtraction, multiplication and division. Each operation is applied to Value1 and Value2 and produces a Result.

In Figure 3.18, the operations are considered at a high level to indicate what they do and not how they do it. Later we will consider how division is really done. For now, we will use the division operator to create larger functions. Remember, it often helps to hide details!

Figure 3.18 Real Number data flow components

Formulas, or arithmetic expressions, specify how numbers are grouped and manipulated. Data-flow diagrams present this information graphically. For example, Figure 3.19 presents a data-flow diagram for the expression $A + B \times C$.

Figure 3.19 Formula data-flow diagram

In Figure 3.19, the arithmetic operators addition and multiplication, are shown as black box operators with two inputs and one output each. The two values at the inputs are transformed by the operator into one value at the output. The “flow” of these values leads to one resulting output value, at the bottom of this diagram.

The *precedence* of an operator is a common convention that helps specify the order in which the operations of an expression are to be performed. This corresponds to the rule that we learned in high-school: that multiplication and division are done before addition and subtraction.

For example, the formula $1 + 2 \times 3$ should be evaluated as $1 + (2 \times 3) = 7$ instead of $(1 + 2) \times 3 = 9$ because multiplication has precedence over addition. Parentheses can be used in an expression to show the order of evaluation, as we did above.

Another way of describing precedence is to say that multiplication and division *bind their operands* more tightly than addition and subtraction. This idea of “binding” of operands is clearly shown in Figure 3.19. When multiplication and division occur together in an expression. It is evaluated from left to right as shown in Figure 3.20.

Figure 3.20 Temperature conversion data-flow diagram

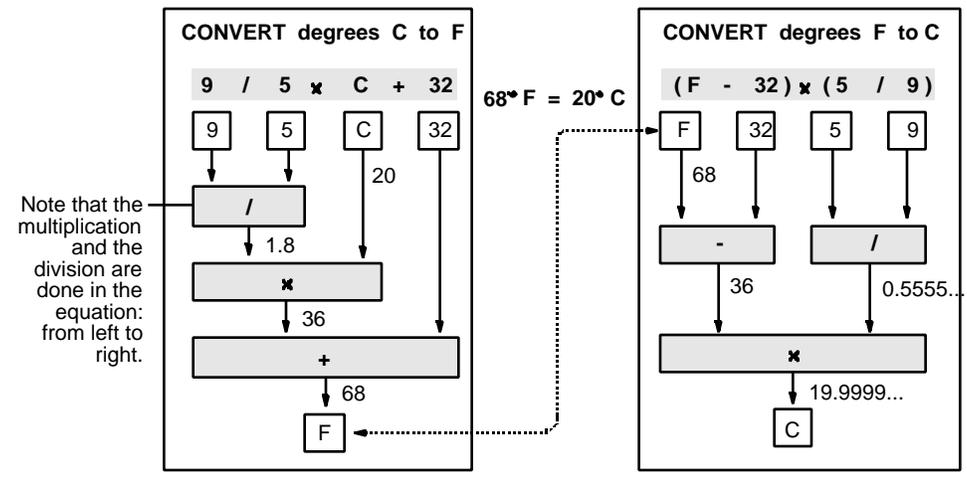


Figure 3.20 shows the data-flow diagrams for the conversion of temperatures between Fahrenheit and Celsius scales. The diagram on the left of Figure 3.20 shows the conversion of 20°C to 68°F , and on the right of the same figure, the conversion of 68°F back to $19.9999\dots^{\circ}\text{C}$. In other words, we did not finish up with what we started with! This illustrates a problem of possible inaccuracy when dealing with repeating decimals, such as $5/9$ ($0.5555555555\dots$). Because it is impossible to store an infinite number of decimals in a computer, only approximations to such numbers can be stored and used, which leads to a loss of precision in the computations.

Polynomials are formulas (arithmetic expressions) that involve a single variable, say X , taken to certain powers and multiplied by various coefficients as in the following example:

$$Y = A + B \times X + C \times X^2 + D \times X^3 + E \times X^4$$

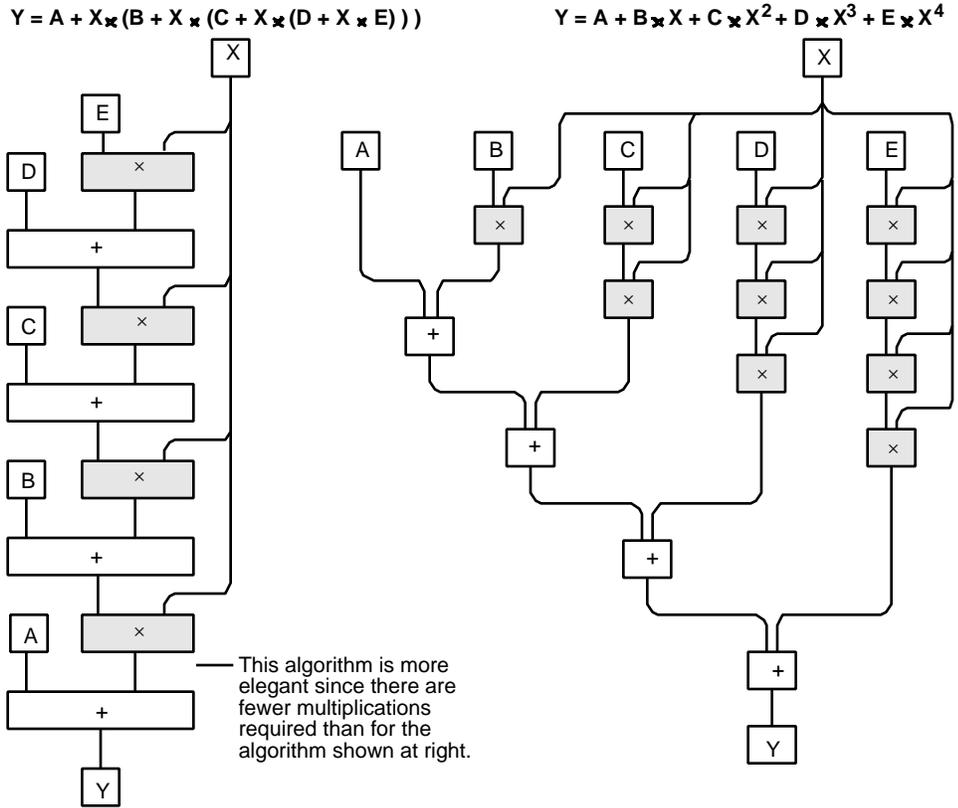
Such polynomials can usually be factored to yield a formula which, in the case of our example above, will look like this:

$$Y = A + X \times (B + X \times (C + X \times (D + X \times E)))$$

The first way of writing the polynomial may seem like a natural way to think about the evaluation of the formula, but it involves much more multiplication

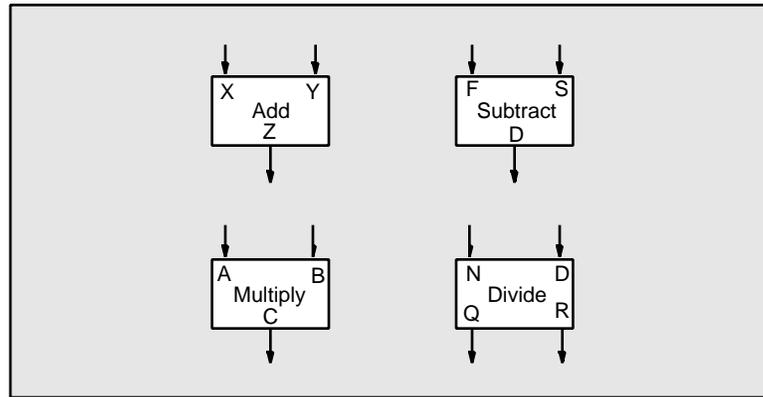
than the second way, which only requires four instances of multiplication. The difference is shown in the two data-flow diagrams of Figure 3.21. The second method, shown on the left, has a more “elegant” data flow structure.

Figure 3.21 Polynomial data-flow diagram



Integer Data-Flow Diagrams

Integers are whole numbers (such as 7, 0, -32, 365, 12 345), which usually arise from counting. Integers are represented differently from Real Numbers in computers and, although they have similar arithmetic operations, these operations are often handled differently. The four fundamental functions on Integer numbers are shown as data-flow diagrams in Figure 3.22.

Figure 3.22 Integer components

In particular, you should note that the division of Integers is different from the division of Real Numbers. Dividing an Integer N by another Integer D yields an Integer quotient Q and an Integer remainder R : for example, dividing 23 by 7 yields a quotient of 3 and a remainder of 2. There are two outputs (Q and R) from Integer division, whereas there is only one output from the division of Real Numbers. In the case of the division of 23 by 7, the Real Number operation yields this never-ending, repeating decimal:

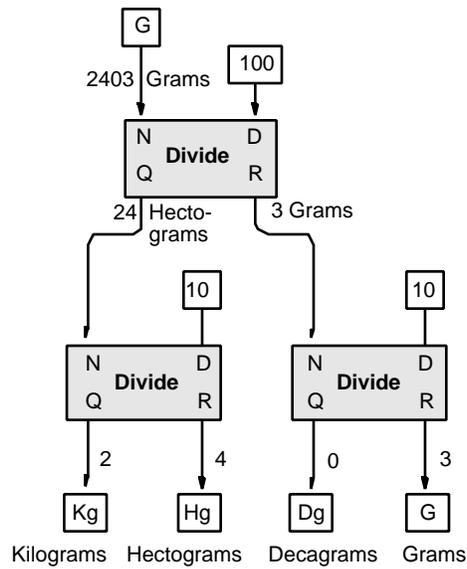
$$23/7 = 3.285\ 714\ 285\ 714\ 285\ 714\ 285\ 714\dots$$

The conversion of any number of grams to the corresponding number of kilograms, hectograms, decagrams and grams, illustrates the use of Integer Divide data flow boxes. The various conversion factors are shown in Figure 3.23.

Figure 3.23 Conversion factors

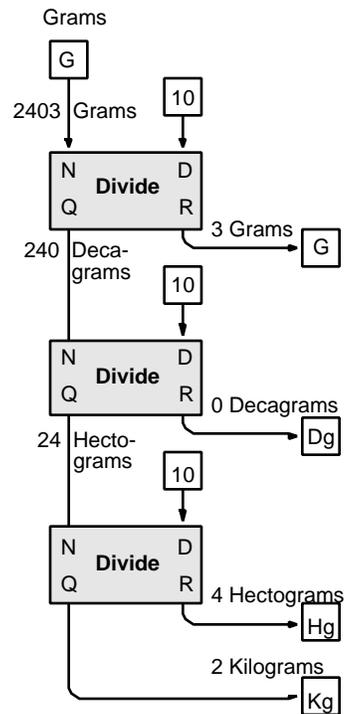
1 decagram	=	10 grams	
1 hectogram	=	10 decagrams	= 100 grams
1 kilogram	=	10 hectograms	= 1000 grams

Because the metric system has the same base as our decimal counting system (base ten), we are able to make conversions between these measures in our head without difficulty. However, to make these conversions on a computer, we need an algorithm. Figures 3.24 to 3.26 provide three different ways of expressing something that we can compute in our heads, as an algorithm.

Figure 3.24 Convert grams first conversion method

We'll illustrate three different ways of performing the conversion. First, consider the conversion method shown in Figure 3.24, where values are shown for one typical example of 2403 grams converted to 2 kilograms, 4 hectograms, 0 decagrams and 3 grams. These typical values help make the result concrete; they do not prove anything.

The first Divide block divides the number of grams by 100 (100 grams = 1 hectogram), yielding a quotient of hectograms and a remainder of grams. The second Divide, at the left, divides the hectograms by 10 (10 hectograms = 1 kilogram), yielding a quotient of kilograms and a remainder of hectograms. The third Divide, at the right, divides the number of grams in the remainder from the first division by 10 (10 grams = 1 decagram), yielding a quotient of decagrams and a remainder of grams.

Figure 3.25 Convert grams second conversion method

The second conversion method, Figure 3.25, begins by dividing the number of grams by 10, yielding a quotient of 240 decagrams and a remainder of 3 grams. Next the 240 decagrams are divided by 10 again (10 decagrams = 1 hectogram) yielding a quotient of hectograms (24) and a remainder of decagrams (0). Finally the 24 hectograms are divided by 10 (10 hectograms = 1 kilogram), yielding a quotient of kilograms (2) and a remainder of hectograms (4).

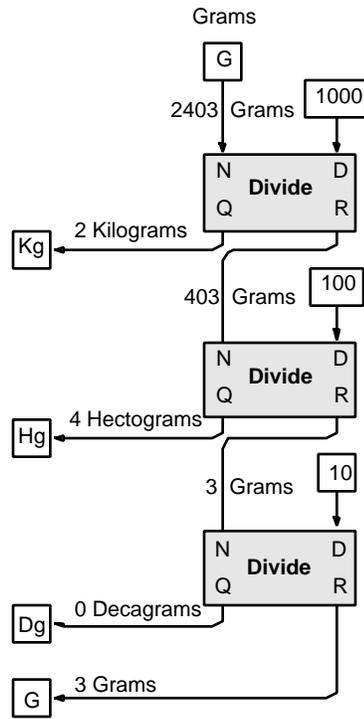
Figure 3.26 Convert grams third conversion method

Figure 3.26 shows yet another conversion method that works in the opposite direction from Figure 3.25.

First, it divides the number of grams by 1000, yielding a quotient of 2 kilograms and a remainder of 403 grams. This number of grams is then divided by 100, yielding a quotient of 4 hectograms and a remainder of 3 grams. For this example, the task is already complete at this point and it is tempting to stop now. However, in general, we need to proceed further. The number of grams is divided by 10 yielding a quotient of 0 decagrams and a remainder of 3 grams.

Oftentimes, it is possible to solve a problem in many ways. Our rather simple conversion problem had three different solutions. Larger problems may have many more solutions. None of the above solutions seem much better than the others for they all involve three divisions. In some problems; however, the solutions may be very different and some will be highly preferred over others.

Logical Data-Flow Diagrams

George Boole made significant contributions to the field of logic around 1850, showing in particular how it was possible to combine logical relationships into expressions just as we can combine Integer or Real Number values into expressions. Such “logical” expressions are called *Boolean expressions*. Logical

quantities (constants, variables and expressions) are used very often in programming.

Propositions, or logical statements, can have one of only two values: either True or False, often abbreviated to T and F. Here are a few examples of some propositions:

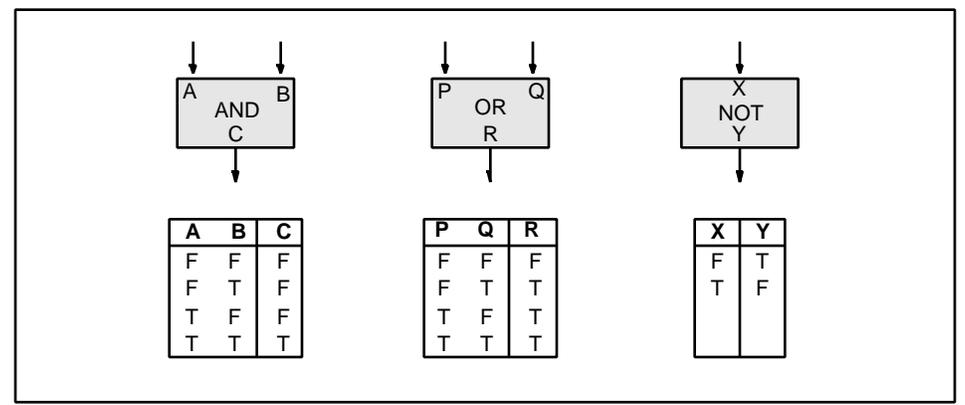
$2 + 2 = 4$	(True)
7 is even	(False)
It is raining here now	(True)

Predicates are logical functions that, for given values of their arguments, become propositions with values True or False. Here are a few examples of predicates:

($X < 10$):	True when X is 7;	False when X is 11.
($X = Y$):	True when $X=7$ and $Y=7$;	False when $X=7$ and $Y = 11$.
(It is raining there):	Value depends on what "there" is referring to.	

Symbolic logic is the study of logic based on the use of symbols. As arithmetic possesses operators (addition, subtraction, multiplication and division), symbolic logic has its own set of operators, which include AND, OR and NOT as shown in Figure 3.27.

Figure 3.27 Logic components



Truth tables for each operation (bottom of Figure 3.27) describe the output behavior for all given combinations of input values. These truth tables can be viewed and used as tiny arithmetic operation tables, somewhat like multiplication tables.

The AND operator, applied to two propositions A and B, has output True when both A and B are True. Otherwise its output is False. For example:

$(2 + 2 = 4)$ AND (7 is even) is False.

The OR operator, applied to two propositions P and Q, has output True when either P or Q or both are True. It only has the output False if both P and Q are False. For example:

$(2 + 2 = 4)$ OR (7 is even) is True.

The NOT operator (or negative), applied to a False proposition, gives True, and NOT, applied to a True proposition, gives False. For instance:

NOT (7 is even) is True.

Logical operators can be combined to build logical expressions or new logical functions as shown in Figures 3.28 and 3.29.

Figure 3.28 NOR logical operator

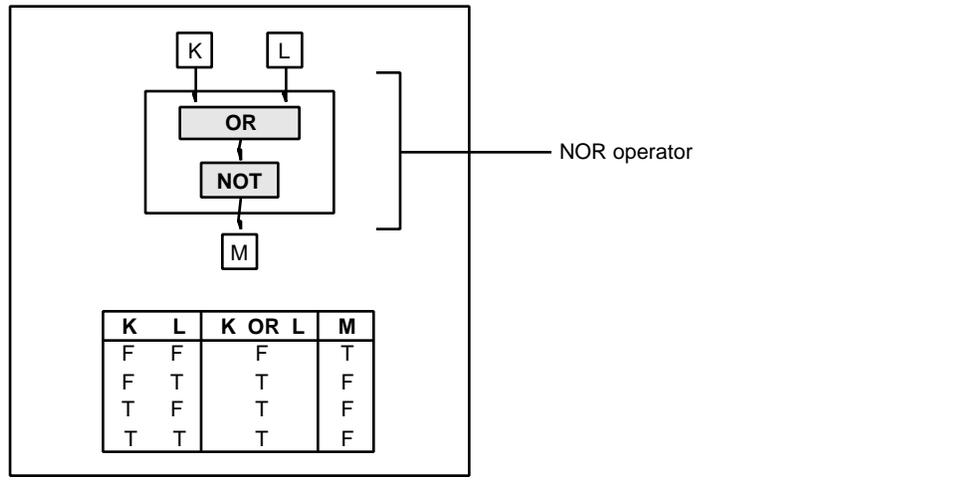


Figure 3.28 shows the complement (NOT) of the OR of propositions K and L, and the corresponding truth table. Notice that the resulting output M is True only when neither K nor L is true; this is called the NOR operator.

Complex logical expressions that involve the negation operator NOT can often be simplified by the application of *DeMorgan's Laws*. DeMorgan's First Law has this form:

$$\text{NOT}(P \text{ OR } Q) = \text{NOT}(P) \text{ AND } \text{NOT}(Q)$$

This states that, to negate the OR of two propositions, you negate each proposition and change the OR to an AND. Consider this statement:

I will go out, if it is not raining or freezing.

This is equivalent to

I will go out, if it is not raining and not freezing.

As another example of DeMorgan's First Law, let's take the negative of an expression involving a baseball game with inning I and scores S1 and S2:

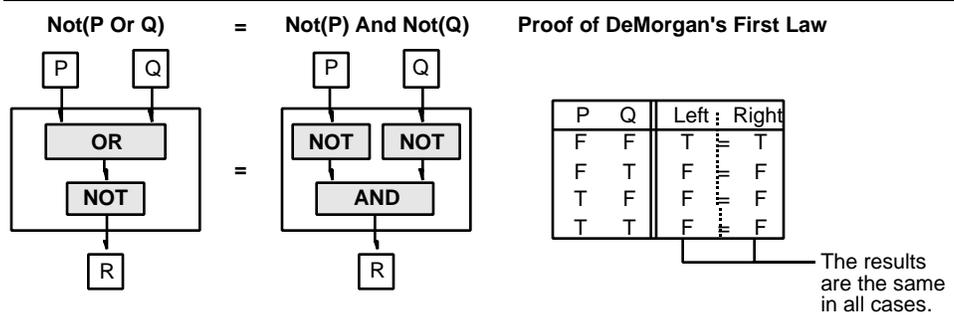
$$\text{NOT} \{ (I = 9) \text{ OR } (S1 = S2) \}$$

This is equivalent to

$(I > 9) \text{ AND } (S1 \ S2)$

Data-flow diagrams of DeMorgan's First Law are shown in Figure 3.29.

Figure 3.29 DeMorgan's first law



A proof of the first law is given at the right of Figure 3.29; the logical expressions corresponding to the diagrams at the left and right are the same in all four possible cases.

DeMorgan's Second Law is:

$$\text{NOT}(P \text{ AND } Q) = \text{NOT}(P) \text{ OR } \text{NOT}(Q)$$

This law will be further discussed in Chapter 5.

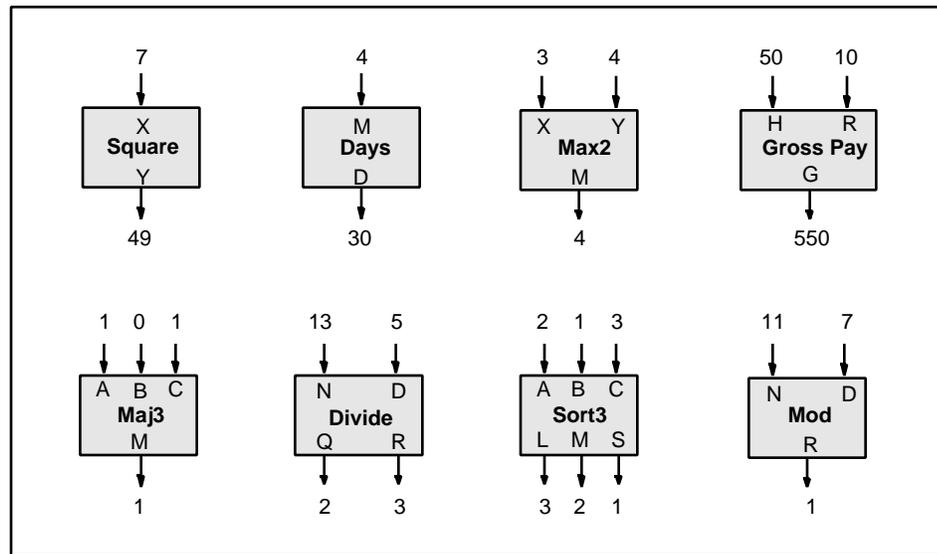
Data Flow Black Boxes

In data-flow diagrams, all actions to be performed on data are represented by black box components. Some of these components are shown in Figure 3.30 and will be used in the following chapters. Some of these components are directly available in programming languages, others can easily be created by programmers and used to create yet larger components and systems.

In Figure 3.30, notice the differing numbers of data inputs and outputs. Remember that the data flow view is concerned with what is done, not how it is done. The emphasis is on the use and reuse of these components. But, we first need to know what they do, not how.

Also notice that the first two components in this figure (Square and Days) have a single input (arrow pointing in) and single output (arrow leaving). The other examples have more inputs and outputs.

Figure 3.30 Data flow components



The data flow components in Figure 3.30 are explained as follows:

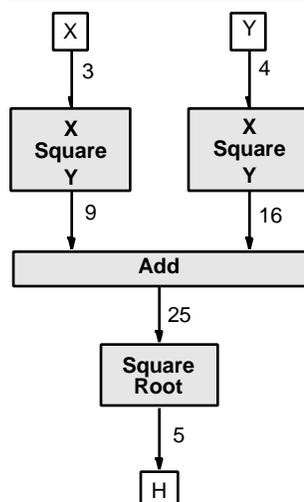
- *Square*: The first component in Figure 3.30 is a simple example shown with an input value of 7 “flowing” into the box, and the resulting data value of 49 being output. It does not indicate how the square of X was computed within the box. It could have been done by X successive additions of X , or by summing the first X odd integers, or by multiplication.
- *Days*: The second component is another very common component that, given a month number M returns the number of days D in that month, as previously seen in table form in Figure 3.14. In the example shown, the input value is 4 for April, and the output value is 30.
- *Max2*: The third component takes two numbers X and Y as input and returns the maximum value of these two numbers, M . With inputs 3 and 4, the returned result is 4.
- *Gross Pay*: The next component takes two input values, a number of hours worked H and a pay rate R , and it returns the gross pay G calculated from the input values. The example shows an output of \$550, for 50 hours of work and a pay rate of \$10 an hour. Based on what we saw in Chapter 2, you can probably guess how the computation was done.
- *Maj3*: The first component in the bottom row of Figure 3.30, *Maj3* takes three inputs, A , B and C with values 0 or 1, and returns the value of the majority of the inputs. For example, with inputs 1, 0, and 1 the majority value returned is 1.

- *Divide*: The next component has two inputs (Numerator N and Denominator D) and two outputs Q, the quotient of the division and R, which is the remainder of the division of N by D. For example, with inputs 13 and 5, the results are 2 and 3.
- *Sort3* shows three input values, A, B and C, and three output values L, M and S. These are the sorted input values, with L having the largest of A, B and C as its value, and S the smallest. M holds the middle value.
- *Mod*: The last component in Figure 3.30 takes two inputs: the numerator N and the denominator D and produces one output value: the remainder, R, when N is divided by D.

Data-flow diagrams are mainly used to show the data interaction among a number of algorithms. Since they hide the inner details of individual algorithms, they simplify the study of complex interconnections of algorithms. Let's now look at some examples of such interconnected components.

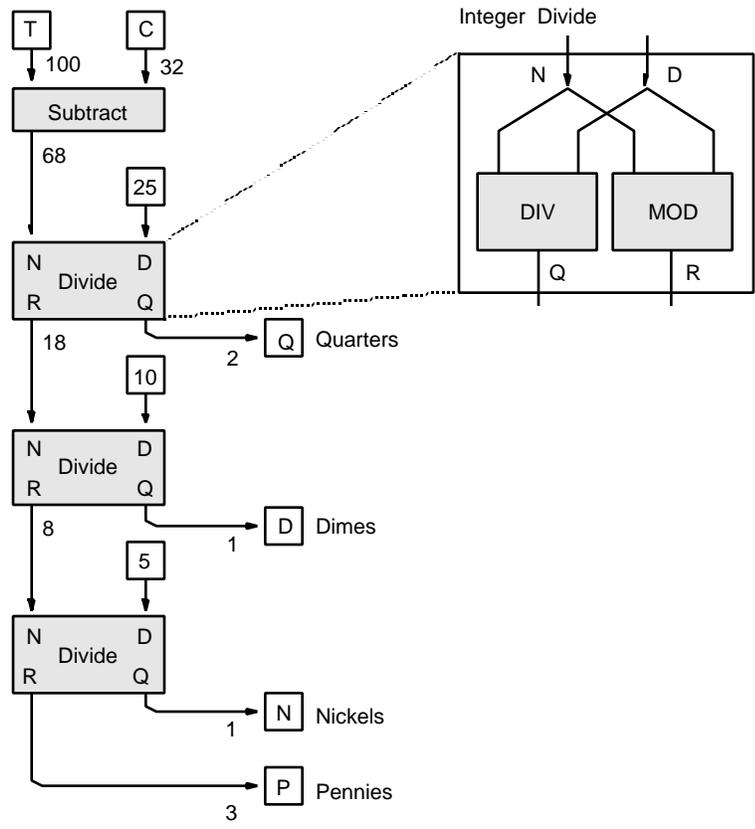
Figure 3.31 shows the computation of the hypotenuse H of a right triangle (whose right sides are X and Y). If the details of the Square and Square Root algorithms were shown, this Hypotenuse algorithm would look much more complex.

Figure 3.31 Data flow connections



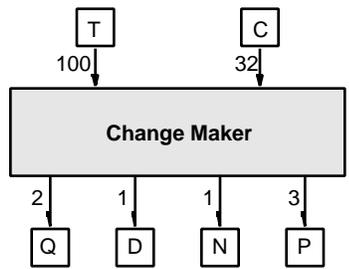
Another example shown in Figure 3.32 illustrates an algorithm to make change using Integer Division. It begins by subtracting the cost C from the amount tendered T (in the example here, 32 cents is subtracted from a dollar leaving a remainder of 68 cents). This remainder is then divided by 25 to determine the number of Quarters, the resulting remainder divided by 10 to determine the number of Dimes, and the result divided by 5 to determine the number of Nickels and Pennies.

Figure 3.32 Change Maker



We would not normally use such an algorithm for making change because the division is too difficult. We would probably choose to make change by avoiding even subtraction. We often do things differently than computers. Notice too that this Change Maker algorithm could be represented as a black box with two inputs and four outputs (as shown in Figure 3.33, a slight variation from what we saw in Figure 3.30).

Figure 3.33 Change Maker as a black box

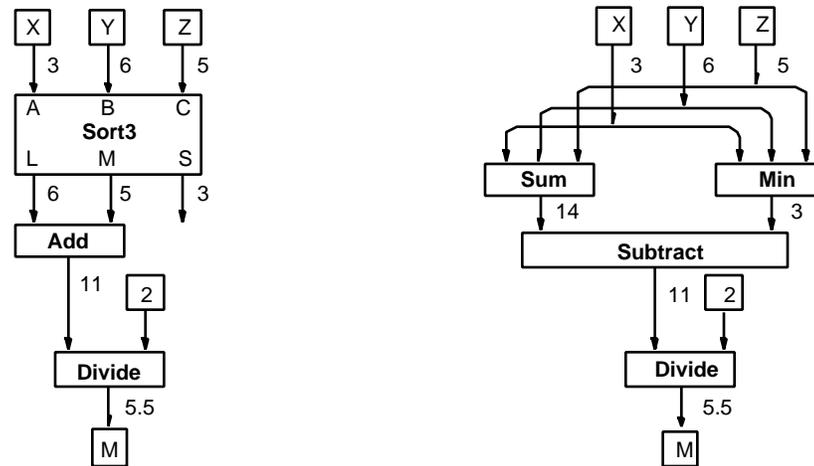


More on Data Flow Components and Diagrams

One of the most common problems encountered when creating algorithms is that we get into details too quickly and become bogged down in their complexity. The use of data-flow diagrams prevents this tendency, or at least postpones it. Data-flow diagrams allow us to think in bigger blocks.

Let's look at an algorithm to average three exam grades of a student where the lowest grade is excluded. We could do this in two different ways, as Figure 3.34 indicates.

Figure 3.34 Forgiven Mean two ways



The left side of Figure 3.34 shows an algorithm that begins by sorting the three exam grades X, Y, Z into order with the largest labeled L, the middle value labeled M and the smallest called S. Then the highest two values L and M are added, and this sum divided by 2 to get the resulting mean M.

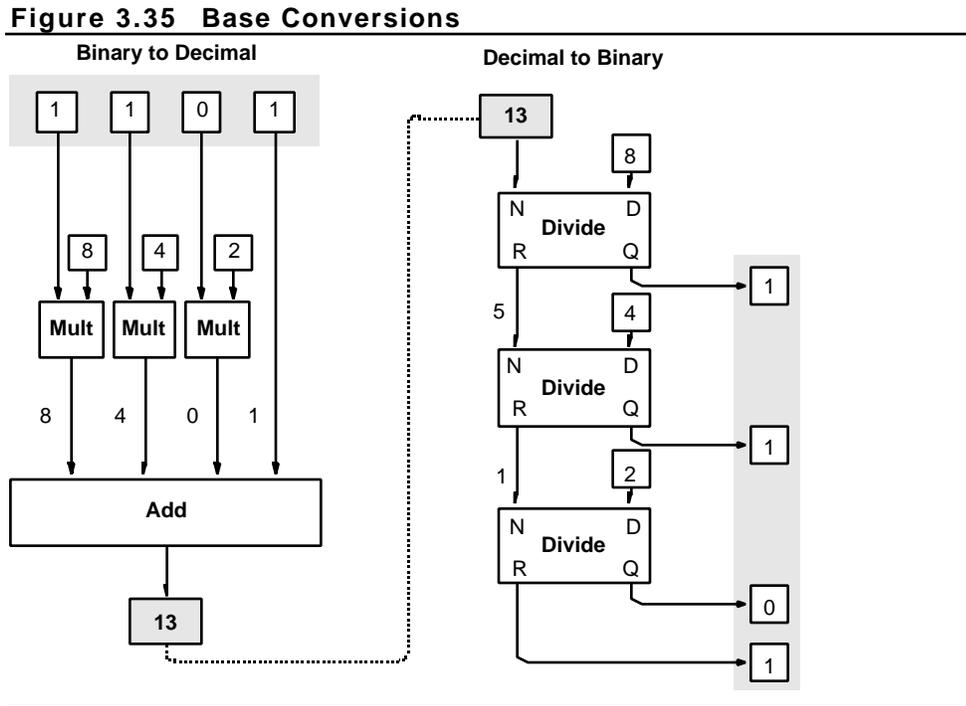
The Forgiven Mean algorithm at the right of Figure 3.34 finds the sum of the three grades as well as the minimum value of the grades. It then subtracts the minimum from the sum and divides this result by 2 to get M, the resulting mean.

At this point, it is not important to recognize which of these two methods is preferable. However, it is important to realize that there are often a number of ways to do anything. The first method may seem simpler, but it may be slower or more costly. The choice between methods depends on more knowledge of cost, speed, availability, and so on. These topics will be discussed in detail later in this book.

In computer science, *concurrency* is the ability of actions to be done at the same time, or in parallel. Data-flow diagrams often reveal the potential for concurrency. For example the operations of Sum and Min in Figure 3.34 could be done at the same time. Most of our present machines and languages do not take advantage of this; they wait for one to be done before doing the other. In the future, our systems may find this parallelism to be useful and efficient.

Figure 3.35 shows two more data-flow diagrams, Base Conversions, that represent the algorithms for converting a number from binary (base 2) into decimal (base 10), and then back into binary again.

The formula algorithm for base conversion is shown in Figure 3.13



The two algorithms in Figure 3.35 are described from left to right, as follows:

- **Binary to Decimal:** The algorithm at the left of Figure 3.35 shows how a four-bit binary number (1101) is converted to a decimal value by multiplying the left-most digit by 8, the next digit by 4, the next by 2 and the right-most digit by 1. Summing these values gives the decimal value of 13.

For longer binary numbers, the leftmost digit is multiplied by successive powers of 2. For example, if the binary number was (11101), the leftmost digit would be multiplied by 16 or 2^4 .

Instead of speaking of the left-most digit of a number, we refer to it as the *most significant digit*. The term “significant” is used here in the sense that since the left-most digit has the greatest value, it is the most significant digit in computing the value of the complete number. For example, in the decimal number 1234, the 1 has value one thousand, the 2 has value two hundred, and so on. Similarly, the right-most digit is referred to as the *least significant digit*.

- **Decimal to Binary:** The data-flow diagram at the right of Figure 3.35, shows how the decimal value of 13, obtained from the left, is converted back into the binary form by successively dividing by 8 then 4 then 2

and so on, to arrive at the original binary sequence 1101. In general, the first value to be used as the first divisor must be the largest power of two (2, 4, 8, 16,...) that is just less than the decimal number. For example, since the decimal number in Figure 3.38 is 13 and since 8 is the largest power of two that is just less than 13, 8 is used as the first divisor.

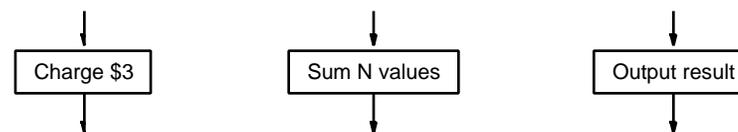
3.5 Flow of Control Diagrams

Flowcharts

Flowcharts are one of the most common ways of showing the sequence of actions in an algorithm. They consist of boxes joined by lines with arrows showing the flow of action sequences. The boxes are of different shapes depending on whether they represent actions, decisions or assertions:

- *Actions* denote processes (such as input, output, calculate) and are represented as rectangular boxes, as shown in Figure 3.36. Actions may change the value of the data on which they operate. For example, the action of incrementing a counter changes the value of the counter by adding 1 to it.

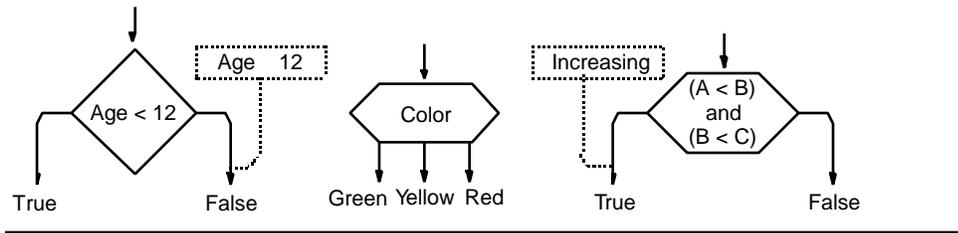
Figure 3.36 Action representations



The actions that can be performed on data depend upon what kind of data are involved. For example, Real Numbers may be added, subtracted, multiplied and divided to yield Real Numbers. On the other hand, the logical operators, AND, OR and NOT do not apply to Real Numbers or Integers; they only apply to logical quantities which have the value of True or False.

- *Decisions* usually take the form of tests or questions such as “Age < 12?” or “Color?” that have two or more results (mostly, but not always True or False), depending upon the values being tested. Decisions are represented by diamonds or boxes with pointed ends. Each decision box must have an outgoing arrow labeled for every possible result, as shown in the examples of Figure 3.37. These arrows lead to the next action or decision to be performed. If the possible results are True and False, you might find it easier to follow a certain convention, like having a True result always take the left path out of the decision box as we do in our flowcharts.

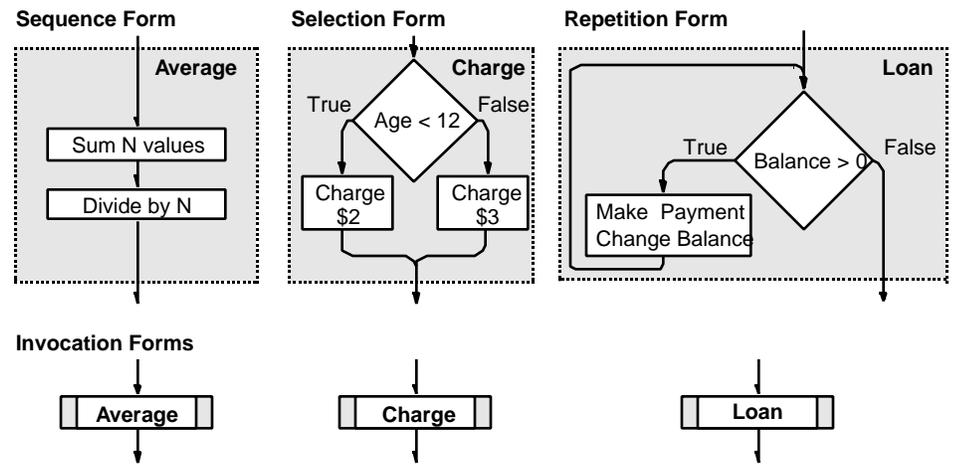
Figure 3.37 Condition and assertion representations



- *Assertions* are statements or facts (such as “Age = 12” or “Increasing”) about the values of variables at some point in a flowchart. They are shown in Figure 3.37 by dotted boxes pointing to the places where the assertions hold true.

Boxes representing actions, decisions, and assertions are combined to represent a complete algorithm. In fact, they can be grouped into specific forms that indicate the ways actions and objects may be connected. There are only Four Fundamental Forms, as illustrated in the examples of Figure 3.38.

Figure 3.38 Four basic building block forms for flowcharts



Each of the forms in Figure 3.38 are described in detail below:

- *Sequence*: The algorithm Average (also known as Mean) performs two actions in sequence: one after the other.
- *Selection*: The algorithm Charge involves a choice of actions depending on whether the condition “Age < 12” is True or False. If the age is less than 12, then the line labeled True is followed and the charge is \$2, otherwise it is \$3.
- *Repetition*: The algorithm Loan involves the repetition of some actions that depends on the condition “Balance greater than 0” (abbreviated as “Balance>0”). As long as this condition is True, a payment is made and the Balance is changed accordingly. This means that as long as money

is still owed, the actions are repeated once for each month. Only after the Balance has been finally reduced to zero is this *loop* terminated.

This form is referred to as a loop because if you look at the lines that indicate repetition, you will see that they resemble a loop. The condition “Balance>0” is referred to as the *termination condition*. The action of making the payment and reducing the Balance is known as the *body of the loop*. We will see the details of the actions within this loop later.

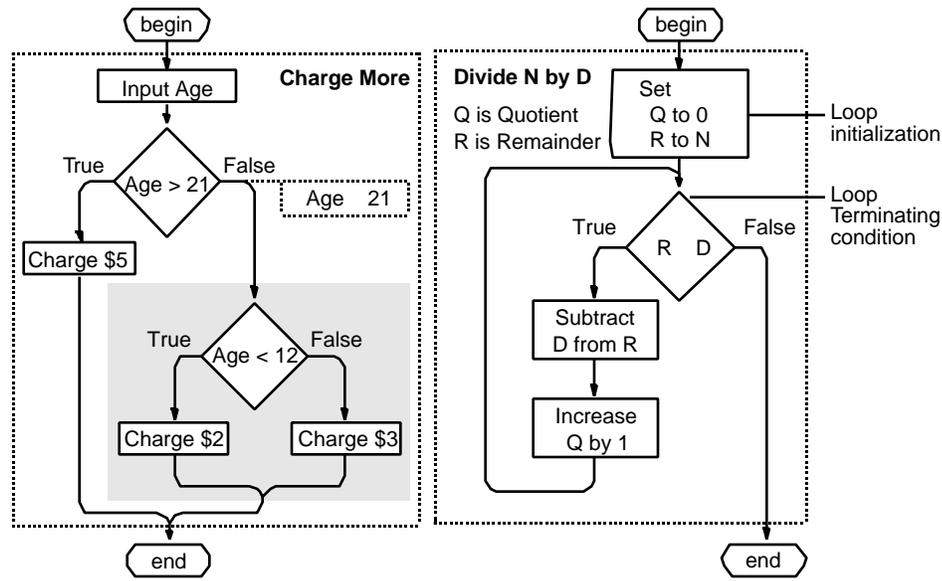
- *Invocation*: Each of the three above examples could be put into a box and labeled for future reference. These boxes are referred to as *sub-algorithm*, and they can be invoked whenever needed. The lower part of Figure 3.38 shows the invocation of the three corresponding sub-algorithms above. A sub-algorithm is marked as a box with double lines on each side. The invocation of a sub-algorithm is considered a single action within a flowchart.

The examples of Figure 3.38 illustrate the Four Fundamental Forms: Sequence, Selection, Repetition, and Invocation, which constitute the basic building blocks from which all algorithms can be made.

Notice that each of these four forms have a single entry and a single exit. Remember, we are representing flow of control here, not data flow. In data-flow diagrams, multiple entries and exits are possible for data, as we have seen for instance in the Change Maker algorithm of Figure 3.33.

Figure 3.39 shows a slightly more complex algorithm, Charge More. It is an extension of the previous Charge algorithm, shown in Figure 3.38, used to calculate the price of admission to a movie. The old algorithm has been modified to include a third category: persons over 21 years old. Notice that the previous Charge algorithm is embedded in this larger algorithm and is shaded. An assertion is delineated by a dotted box.

Figure 3.39 Combined forms



At the right of Figure 3.39 is another algorithm, Divide, that shows how computers can be used to divide positive integers in a way that is very different from how we do division. Since we plan to revisit this algorithm later, you may wish to avoid it for now.

Divide, the algorithm at the right of Figure 3.39, divides one Integer (N , the numerator) by another non-zero Integer (D , the denominator) to produce a quotient Q and remainder R . This is done in a loop where each time the loop is traversed, an *iteration*, 1 is added to the quotient, Q , and the divisor, D , is subtracted from the remainder, R . This continues until R is less than D .

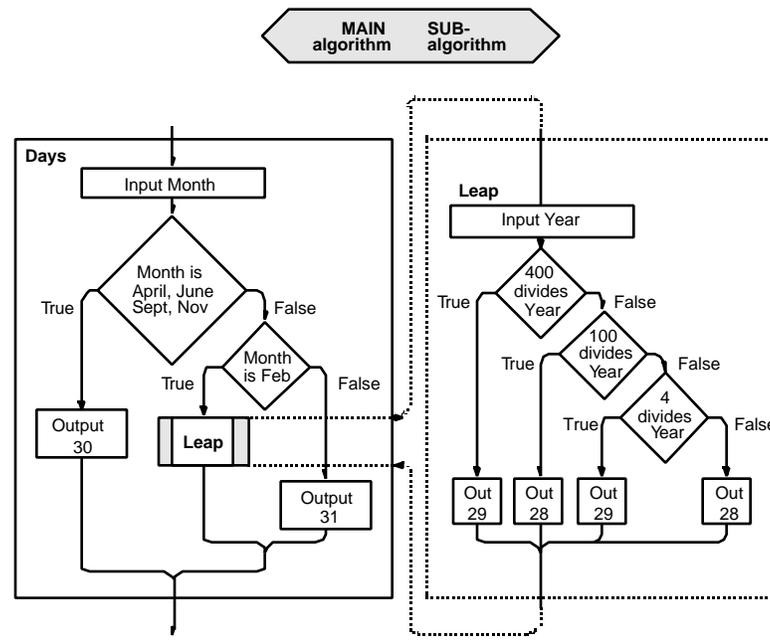
The body of the loop comprises the actions of subtracting the divisor from the remainder and adding 1 to the quotient. The condition “ $R < D$ ” is the loop’s terminating condition: when it is no longer true, the loop terminates.

Before the loop begins, the quotient Q is set to 0 and the remainder R is set to N . This is called *initializing the loop*. This verbal description is very concise, but is not as descriptive as the graphic flowchart. However beware! If your flowcharts are not properly structured, they may also be confusing.

Larger Flowcharts Using Subprograms

As algorithms become larger, it is often convenient to split them into smaller connected algorithms. Each sub-algorithm may then be considered separately, making the entire algorithm more manageable. Two such “decomposed” algorithms are shown in Figures 3.40 and 3.41.

Figure 3.40 Decomposed algorithms



This Days algorithm was previously seen in Figures 3.2 and 3.14.

Days is an algorithm that determines the number of days in any month. It does this first for all the months except February. Finding the number of days in February requires determining whether a leap year is involved. Since this is a fairly complex operation, this part of the Days algorithm is separated from the main algorithm as a sub-algorithm named Leap. The Leap sub-algorithm is broken out at the right of Figure 3.40.

Leap decides if a year is a leap year by determining if the year can be divided evenly by various numbers (400, 100 and 4). Notice that the sub-algorithm Leap may seem more complex than the “main” algorithm Days.

Decomposing, or breaking up this algorithm into two parts— a main algorithm connected to a sub-algorithm— is not necessary in this case because the algorithm is simple. However, adopting this break-out habit early, will be beneficial when you attempt to develop more complex systems.

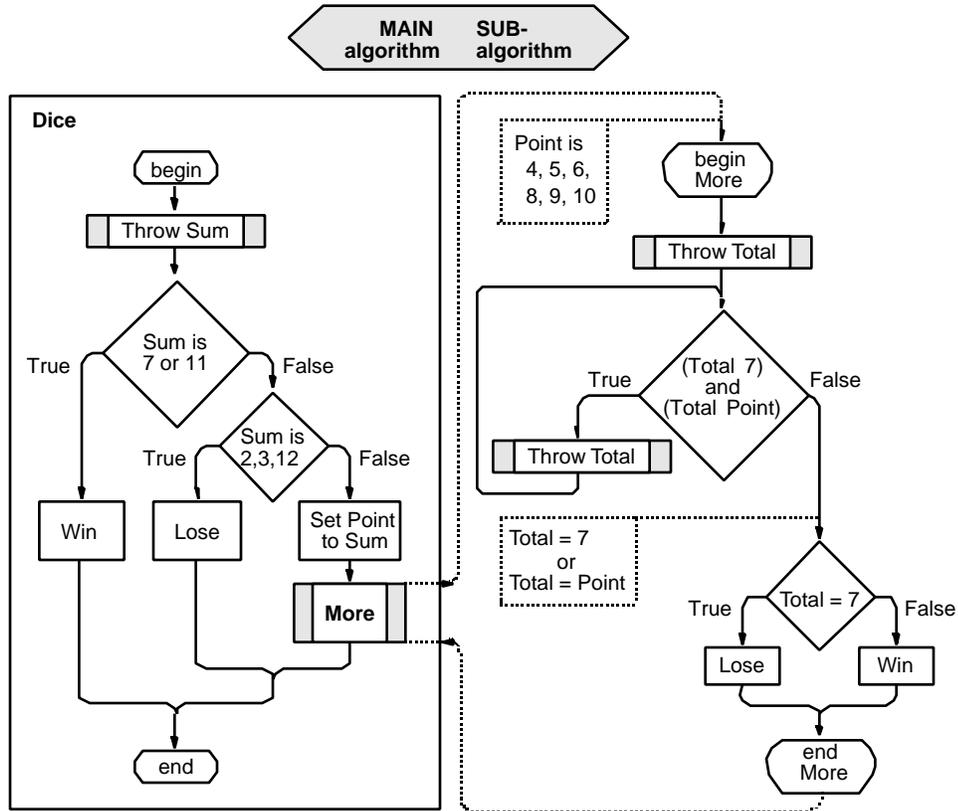
Let’s develop an algorithm to play the simple dice game introduced in Figure 3.3. Here are the rules for this game:

First, two dice are thrown.
 If their sum is 7 or 11, you win, and if the sum is 2, 3 or 12, you lose.
 Otherwise remember the sum, the “point count”, and keep throwing until either:
 the sum equals the point count (you win), or
 the sum equals 7, (you lose)..

The two dice used for this game have each side marked with 1 to 6 dots. When the two dice are thrown, the resulting sum of dots is a value between 2 and 12.

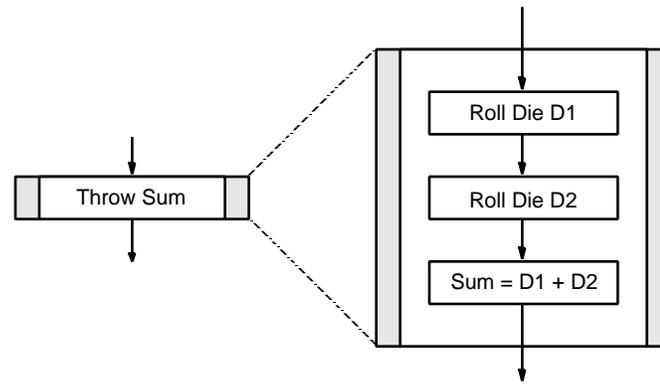
The flowchart of Figure 3.41 describes this game. Notice that “point count” is referred to by the variable Point.

Figure 3.41 Dice flowchart



The main algorithm describes the first throw and the sub-algorithm More describes all subsequent throws (if any). A third sub-algorithm, Throw, is used several times and is shown in Figure 3.42.

Figure 3.42 Throw flowchart



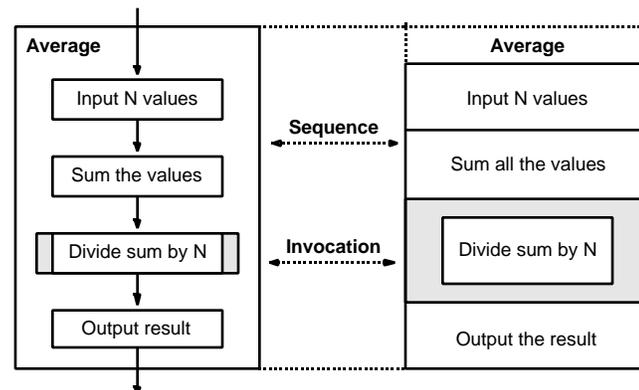
Flowblocks

Flowblock diagrams are an alternative to flowcharts as a way of representing the flow of control in an algorithm. Flowcharts can be useful when creating and communicating algorithms; however, flowcharts often do not flow! The lines joining each box often meanders in complex paths. This makes an algorithm difficult to understand which destroys the simple beauty of the graphic form. Also, when creating flowcharts, many dangling lines or arrows can easily get connected to the wrong boxes, resulting in error-prone algorithms.

Flowblocks (or Nassi-Shneiderman diagrams) are graphic alternatives to flowcharts. They consist of a series of rectangular boxes which are easier to draw than flowcharts. The boxes can be placed (connected) only in certain patterns (usually one above the other so the single exit of one is the single entry of another), thus preventing the creation of bad structures. So, flowblocks flow!

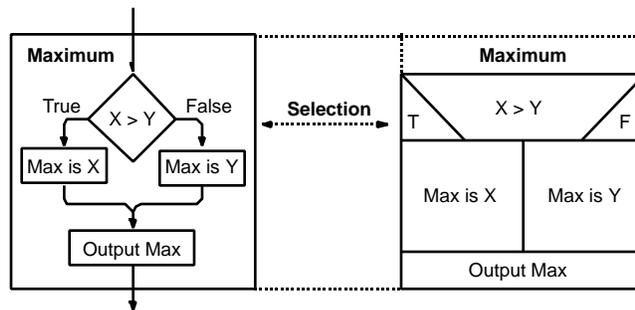
Figures 3.43 to 3.45 show algorithms illustrating the four basic flowblock forms of Sequence, Selection, Repetition and Invocation. They are shown next to their equivalent flowcharts for comparison.

Figure 3.43 Sequence form in flowcharts and flowblocks

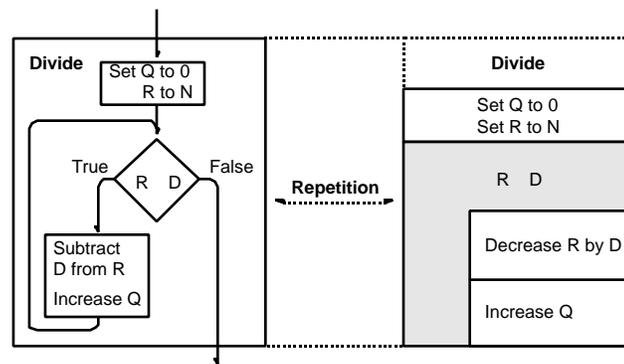


- *Sequence*: Figure 3.43 illustrates the flowblock representation of the Sequence form using the Average algorithm as an example. The flowblock version is essentially the same as the corresponding flowchart with the action boxes sitting on top of one another and the flow lines removed. Note that the boxes can be enlarged because room is not needed for lines and arrows.
- *Invocation*: The Divide blocks in Figure 3.43 provide a comparison of the flowchart and flowblock subprogram forms. In the flowblock form, subprograms are denoted by a box within a box, which is easier to see than the two vertical lines of the flowchart form.
- *Selection*: The flowblock selection form, illustrated by the Maximum algorithm in Figure 3.44, is derived from the flowchart selection box by removing its upper half and lower tip. The block is entered at the top and then, depending on the condition specified in the trapezoidal part, the exit is by one of the two sides into one of the corresponding boxes below. Finally, the two paths join at the bases of the two boxes, in the “output Max” box. The flow continues from here.

Figure 3.44 Selection form in flowcharts and flowblocks



- *Repetition*: The flowblock repetition form (illustrated by Integer Divide in Figure 3.45) consists of an “inverted L-shaped” box that encompasses the body of the loop: the part of the algorithm that is repeated while the condition is true. As with the selection form, the exit flows into the top of the next block of the diagram (not shown).

Figure 3.45 Repetition form in flowcharts and flowblocks

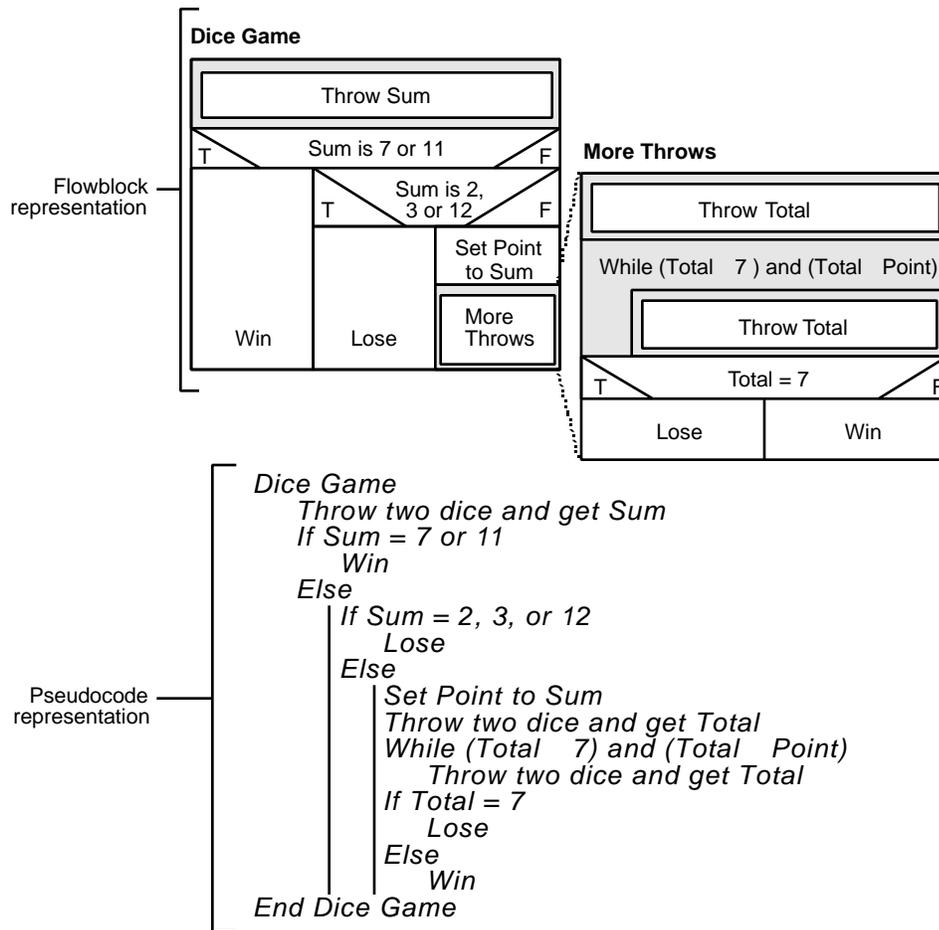
As algorithms become more complex, flowcharts become harder to draw and to follow. By comparison, the complexity of the equivalent flowblocks does not increase as much.

Pseudocode

Another way of representing the flow of control in an algorithm is through *pseudocode*, a mixture of natural language and mathematical notation independent of any programming language. Figure 3.46 shows a comparison between the flowblock representation of an algorithm and its pseudocode equivalent using the Dice Game algorithm as an example.

Figure 3.46 Dice Game flowblock and pseudocode

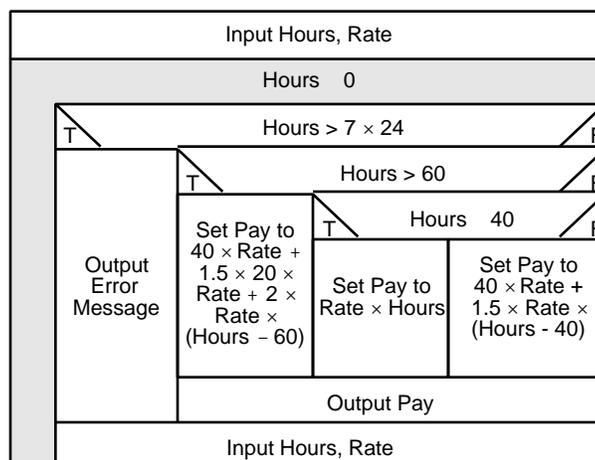
The flowchart version of this algorithm is shown in Figure 3.41.



The Payroll algorithm, shown as a flowblock diagram in Figure 3.47, is an extension of the flowchart in Figure 2.17, modified to continue computing employee's pay until a negative number of hours (Hours) is entered.

It is interesting to see the nesting of the blocks in this rather complex algorithm. The flowblock version is more compact, easier to draw, simpler to understand and easier to modify. This last advantage is important during the development of an algorithm, as there is likely to be considerable modification during this process. Pseudocode is also easy to modify. As another comparison or representation, the equivalent pseudocode is also shown.

Note: In both Figure 3.46 and 3.47, the vertical lines in the pseudocode are not essential. They are merely present to help us notice the levels of indentation (nesting) of the algorithm.

Figure 3.47 Payroll flowblock and pseudocode

```

Payroll
Input Hours, Rate
While Hours > 0
  If Hours > 7 * 24
    Output "Error"
  Else
    If Hours > 60
      If Hours > 40
        Set Pay to Hours * Rate
      Else
        Set Pay to 40 * Rate + 1.5 * Rate * (Hours - 40)
      Else
        Set Pay to 40 * Rate + 1.5 * 20 * Rate +
          2 * Rate * (Hours - 60)
    End If
    Output Pay
  End If
Input Hours, Rate
End Payroll
  
```

3.6 Review Top Ten Things to Remember

1. Algorithms are plans for performing a sequence of actions on data. The actions or data need not have anything to do with computers.
2. All algorithms must be:
 - *Complete*, all actions exactly defined,
 - *Unambiguous*, to allow only one interpretation,
 - *Deterministic*, to always produce a predictable result, and
 - *Finite*, restricted to limited time and space,In addition, algorithms should be:
 - *General*: applicable to a class of problems,
 - *Well-structured*: built from standard building blocks,
 - *Efficient*: economical of resources, and
 - *Elegant*: showing harmony of elements and economy of structure.
3. Representations of algorithms are forms for describing, denoting or presenting them. There are many such ways, some better than others, so it is important to try various representations.
4. *Verbal forms* involve words, in sentences and paragraphs. This representation is usually very verbose, long, and often inaccurate.
5. *Algebraic forms* involve mathematical symbols in expressions or formulas. This is usually a very concise representation.
6. *Tabular forms* involve rectangular grids (tables, arrays or matrices) with entries in the grids. This method is useful for summarizing large selections.
7. *Flowchart forms* involve boxes joined by lines, describing the flow of control of actions. They are very clear for smaller algorithms, but may become confusing if not structured well.
8. *Flowblock forms* involve only rectangular boxes, with very limited (but significant) ways of connecting them. They also describe only the flow of control of actions.
9. *Data flow forms* also involve boxes, but describe the flow of data. This form is most useful at higher levels when dealing with sub-algorithms. The flowblocks are also useful for concurrent programming.
10. *Pseudocode* is a mixture of natural language and mathematical notation independent of any programming language.

3.7 Glossary

Action: operation or process.

Algorithm: a plan to perform some actions on some data.

Array: a collection of memory cells to store some data.

Assertion: a statement which is either true or false.

Black box: the representation of what a process does, the input(s) it takes and the output(s) it produces, while hiding the internal details of the process.

Complete: a property of algorithms specifying that all actions must be precisely defined.

Data type: a description of the kind of data including the values and the operations that can be used on them.

Data-flow diagram: a diagram representing the flow of the data through various processes.

Decision table: a table listing all the possible combinations of the various conditions in a problem.

Deterministic: a property of algorithms by which an algorithm always produces a predictable result.

Elegance: a quality of algorithms showing harmony between its elements.

Finite: a property of algorithms specifying that they must terminate in a finite time and use finite space.

Flow of control: sequence of the actions of an algorithm.

Flowblock: a diagram to represent the flow of control of an algorithm.

Flowchart: a diagram to represent the flow of control of an algorithm.

Generality: a quality of algorithms that are applicable to a class of problems.

Glass box: a representation showing the inner details of a system.

Hierarchical: a presentation of an algorithm in the form of a breakout diagram.

Integer: whole number.

Logical expression: an expression involving quantities that can have only two values, True or False.

Operand: the data element on which is applied an operator.

Operator: a symbol describing the operation to apply to one or several operands.

Precedence: priority of an operator with respect to another one.

Predicate: a logical function.

Proposition: a logical expression having a value of True or False.

Pseudocode: a representation of algorithms based on a mixture of English and mathematical notation.

Real Number: a number having an integer part and a fractional part.

Repetition form: a basic form of algorithms indicating the repetition of a number of actions.

Robustness: a desirable property of an algorithm that is resistant to failure.

Selection: a choice between alternatives.

Selection form: a basic form of algorithms indicating a choice between a number of actions.

Sequence form: a basic form of algorithms which indicates the performance of a number of actions, one after the other.

Sub-algorithm: an algorithm that is used by another algorithm.

Symbolic logic: a system of logic based on the use of symbols.

Table: two-dimensional grid used to store data.

Unambiguous: a property of algorithms requesting that there is only one possible interpretation for the algorithm.

Well-structured: a desirable quality of algorithms requiring them to be built from simple basic blocks.

3.8 Problems

1. Binary Number Drill

Convert the following binary numbers to decimal numbers (base 10):

- a. 10 b. 1010 c. 101010

and then convert the following decimal numbers to binary (base 2):

- d. 7 e. 17 f. 170

2. Small Binary Numbers

Create a table of the binary numbers from 0 to 15, and notice the alternating structure of the columns.

3. Time Base

Draw a data-flow diagram showing how to break up a Military time into hours and minutes past midnight, and then convert this into minutes after midnight. Military time is in 24 hour notation, where 1430 means 2:30pm. Use Divide, Multiply and Add data flow components. For example, the military time 1430 is:

$$14 \times 60 + 30 = 870$$

or 870 minutes after midnight.

4. Decimal to Binary: Another way

Converting decimal numbers into binary can be done by dividing successively by 2 with the resulting remainders forming the binary number. Draw a data-flow diagram showing this process for the decimal number 13 converted to the binary number 1101.

5. Octal Numbers

Octal numbers have a base of 8, with digits 0, 1, 2, 3...7.

Convert the following octal numbers to decimal:

- a. 11 b. 23 c. 132

6. Hex Numbers

Hexadecimal numbers have a base of 16, with the values being 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F (where A is 10, B is 11, .. F is 15).

Convert the following hex values to decimal:

- a. 11 b. CC c. F00

7. Other Bases

Other bases have been used, including 3 (ternary), 5 (quinary), 12, 20, and 60 (sexagesimal). Where and why would they have been used?

8. Bad Min

What is wrong with the following definition of Min, the minimum of two values X and Y?

If X is less than Y then Min is X and
if Y is less than X then Min is Y.

9. ISBN Check

Check whether the following ISBN numbers are proper numbers:

- a. 0-387-90144-2 b. 0-13-215871-X
c. 0-574-21265-4 d. 0-88236-115-5
e. 3-540-90144-2 e. 1-1-1111111-1

10. Resistor Color Code

Electrical resistors have three colored bands, where each color represents an integer: Black is 0, Brown is 1, Red is 2, Orange is 3, Yellow is 4, Green is 5, Blue is 6, Violet is 7, Gray is 8 and White is 9. The resistance is determined by taking ten times the code of the first band, adding to this the code of the second band and adding a number of zeros equivalent to the code of the third band.

For example, if the bands are Red, Yellow and Orange (with values 2,4,3) the resistance is: 24000. Find the resistance of a “patriotic” resistor having band colors of Red, White, and Blue.

11. Charge Tree

For the previous Charge algorithm, with at most 3 adults and 3 kids, create a representation in the form of a tree. Do this in two ways, first beginning with adults.

12. Simpler Binary

Find the decimal equivalents of these binary numbers:

- a. 11111111 b. 10000000

Use the above to find a simple way to obtain the decimal equivalent of a binary number consisting of any number N of ones in succession.

13. Binary Octal

Draw a data-flow diagram showing how binary numbers can be converted to octal numbers by “bunching” each group of three binary numbers (from the right or least significant bit) and replacing this by the octal equivalent. Show it for the decimal value of 299 which is 100 101 011 in binary and 453 in octal.

14. ISBN Data Flow

Draw a data-flow diagram describing the ISBN algorithm. The inputs are the individual digits and the output is the check digit.

15. Expression Trees

Create a tree (data flow) diagram corresponding to the following expressions, and compare them:

- | | |
|---|---|
| a. $(X - Y) \times (X + Y)$ | b. $X \times X - Y \times Y$ |
| c. $S + 60 \times M + 60 \times 60 \times H$ | d. $S + 60 \times (M + 60 \times H)$ |
| e. $A \times B + B \times 4 + C \times 2 + D$ | f. $2 \times (2 \times (2 \times A + B) + C) + D$ |

16. Leaping Again

Represent the Leap algorithm of Figure 3.40 as a table, with three conditions (4 divides Y , 100 divides Y , and 400 divides Y) and eight rules (combinations). Then create another table with fewer rules.

17. Change Change

Modify the data-flow diagram describing Change Maker in Figure 3.32, to allow for half-dollars.

18. Variation On Variance

Another way to compute the variance of N values is to subtract the square of the averages of the values from the average of the squares of the values. Compute this for the four values: 10, 20, 30 and 40.

19. Hot One

Since humans hate division, they often create algorithms to avoid it. For example, instead of converting temperatures by this formula:

$$F = \frac{9}{5} C + 32$$

the following algorithm is sometimes used.

First, multiply C by 2 and subtract from this amount its first (most significant) digit. Then add 32 to this and the result should be the Fahrenheit value.

For example, 20°C becomes: $40 - 4 + 32 = 68$

Check this algorithm for some numbers and indicate any limitations, problems or inaccuracies.

20. Find Algorithms

There are many algorithms guiding everyday things, but we may not be aware of them. Investigate and determine some similar to the following. Attempt to express them using different methods (verbal, tabular, data-flow diagrams, flowblocks, and so on):

- a. Sales commission (For the first thousand sold, pay at the rate of ...)
- b. Numbering of state highways (Odd numbers go south to north, increasing)
- c. Rental agreement of items (For the first 24 hours charge ...)
- d. Library fines for late books (For every day after 5 days ...)
- e. Labeling of rooms in a building, campus, etc. (Floors, even/odd)
- f. Directions to a party (If you are heading North on ...)
- g. Price of Movies (Bargain prices Monday through Friday before 6 and ...)
- h. Schedules of planes or buses, (Every hour, on the half hour until ...)
- i. Meeting times (Every first Friday, except ...)
- j. Weight (Men should weight 106 pounds plus 6 pounds for every inch over 5 feet)
- k. Temperature (The high for the day is 18 degrees above the temperature at 6 am)
- l. Dog's age equivalent to a human (First year is 15 years, second is 10, each year after is 5)
- m. Proportional rule of two (Twice around the wrist is once round the neck)
- n. Ideal Athlete (Weight in pounds is twice the height in inches)

- o. Mortgage check (Limit of 30% of total income on principal interest and insurance).
- p. Date of Easter (Involves much division)
- q. Postal regulations (For sizes of envelopes, girth of packages, etc.)

Many other such algorithms can be found in the book: "Rules of Thumb" by Tom Parker, published by Houghton Mifflin.

Chapter 4 Algorithm Structure

The main concern of this chapter is the structure of algorithms. Even for small algorithms, good structure is important. Algorithms that are well-structured are usually simpler to understand, explain, modify, test, analyze and verify. For large algorithms, good structure is crucial for clarity.

Chapter Outline

4.1	Preview.....	128
4.2	Building Blocks for Structured Programming.....	128
	The Four Fundamental Forms.....	128
	Connecting Several Forms Sequences and Nests.....	131
	Deep Nesting.....	133
4.3	Different Algorithms, Same Problem.....	135
	Equivalent Algorithms.....	135
	Alternative Algorithms.....	137
4.4	Top-down Algorithm Design.....	140
	Job Description Example.....	144
	Change Maker Example.....	145
	A Game Example, Fifty.....	148
4.5	How Data and Algorithms Fit Together.....	152
	Structured Data.....	153
	Chili Recipe Example.....	155
	Card Trick Example.....	158
	Binary Conversion Example.....	159
	Guessing Game Example.....	160
4.6	Review Top Ten Things to Remember.....	164
4.7	Glossary.....	166
4.8	Problems.....	167

4.1 Preview

Now that we have just finished learning what algorithms can look like (chapter 3), you may feel you are ready to begin solving problems by writing algorithms. However, before you can do that, you need to learn more about the structure of algorithms.

Structured Programming is a method of building algorithms from a small number of different basic blocks (or forms) with simple interconnections.

The *Four Fundamental Forms* (called Sequence, Selection, Repetition and Invocation) are the building blocks from which all well-structured algorithms are constructed. Initially, we will use flowcharts to describe these structures; however, the emphasis will soon shift to flowblocks and then pseudocode. These last two representations make it impossible to create poorly structured algorithms.

Top-Down Design is another significant concept that is introduced in this chapter. It is the process of creating algorithms in stages, by successively refining them into smaller sub-algorithms, each refinement providing more details.

The *Data* and the *Actions* described by our algorithms in this chapter are common to everyday experience which shows that algorithms need not involve computers or mathematics. In the chapter that follows this one, computing algorithms (or programs) will be considered and their data (numbers) will actually be simpler than the “common” data treated here. For this reason, we make our data very detailed and explicit here.

4.2 Building Blocks for Structured Programming

Step 3 of the problem solving method, introduced in Chapter 2, is called Solution Refinement. It refines the definition of the various algorithms included in the structure chart of the solution. To elaborate these algorithms, we first need to know what building blocks are available.

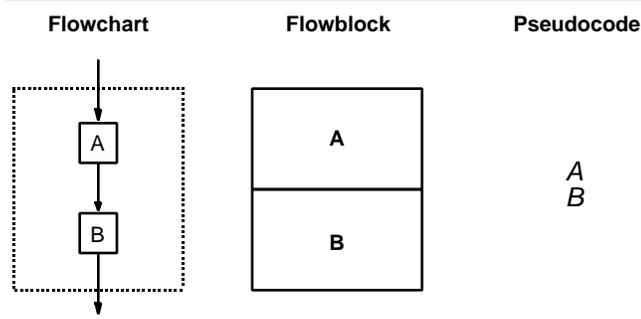
The Four Fundamental Forms

Structured programming is a method of organizing algorithms using a small number of different kinds of building blocks (forms), with simple interconnections.

Basically, there are four building blocks or forms (Sequence, Selection, Repetition and Invocation) and, with them, we can construct all algorithms. Each of the fundamental forms has a single entry and a single exit, making their flow of control very clear. Figures 4.1 to 4.4 represent the Four Fundamental Forms as flowcharts, flowblock diagrams, and pseudocode.

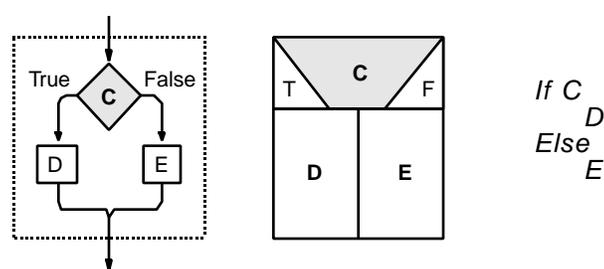
- The *Sequence* form (also referred to as the Series or Concatenation form) indicates the linear temporal sequence in which actions are to be performed. Although we can find examples where the order of actions is unimportant, this is usually not the case. The Sequence form fixes actions to be performed sequentially (one after the other). In Figure 4.1, A and B are performed one after the other.

Figure 4.1 The Sequence form



- The *Selection* form (also referred to as the Conditional, Alternative, or Decision form) specifies a condition that determines which action is to be done next. For example, in Figure 4.2, if condition C is true, the path leading to action D is taken. If C is false, the path to action E is followed. The actions D and E themselves may be sequences of many actions.

Figure 4.2 The Selection form



After the actions on either path have been performed, the two paths rejoin to provide a single exit from the form. This form is sometimes called the *if-then-else* form because it is expressed in many programming languages as:

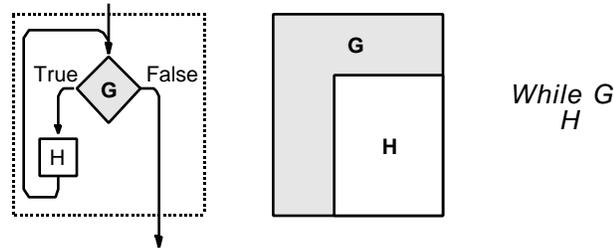
```
IF C THEN D ELSE E.
```

- The *Repetition* form (also referred to as the Iteration or Loop form) indicates that one or more actions must be repeated a number of times. As shown in Figure 4.3, this form begins by specifying a condition, G. If this condition is true, action H, the *body* of the Repetition form, is performed and condition G is re-tested. In other words, action H is repeated until condition G is evaluated as false. At this point, the

repetition stops and the form is exited. This form is also called the *While-loop* form for it is expressed in many programming languages as:

```
WHILE G DO H.
```

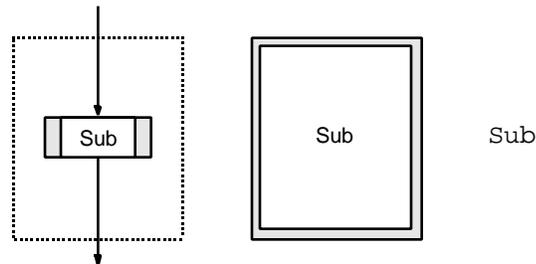
Figure 4.3 The Repetition form



- The *Invocation* form (also referred to as the Abstraction form and as a sub-algorithm) corresponds to a group of actions which have been given a name. This group of actions is invoked using its given name as if it were a single action. This enables an algorithm to be defined once, and then, whenever it is needed, its invoked or called by its name. For example, we could group the actions that calculate the average of some data together under the name *Average*. These actions could then be performed by invoking or calling *Average* as if it were a single action. Programming languages invoke such sub-algorithms with statements such as:

```
CALL Average, or
PERFORM Average, or simply
Average.
```

Figure 4.4 The Invocation form



Forms other than these basic four are possible and are sometimes useful, but they are neither necessary nor fundamental as they can be built from the four basic forms. These additional forms will be introduced in the next chapter.

Connecting Several Forms Sequences and Nests

It is easy to interconnect the four basic forms in order to construct a complete algorithm: to do this, we only have to replace an action in one of the forms

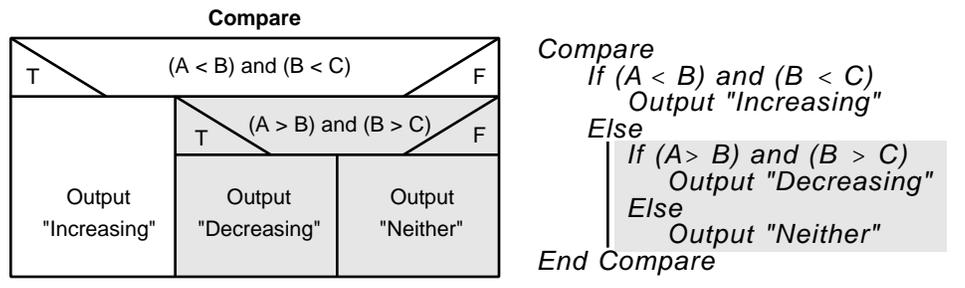
mentioned above with another form. Thus, interconnection of the four forms is possible using one of the following two methods:

- *Serial*, where one form follows another, like simple actions, or
- *Nested*, where one form is within the other.

Such interconnections yield a composition where all blocks have a single entry and a single exit.

Of these two methods for combining forms, the method of nesting is the more complex. Figures 4.5 through 4.8 show different nests as both flowblocks and pseudocode. These different nests can easily be identified because the inner form is shaded in each figure.

Figure 4.5 A Selection form nested in a Selection form



The algorithm *Compare*, in Figure 4.5, is an algorithm that compares three values A, B, and C to determine if the values are:

- Increasing (such as 1, 2, 5 or -1, 0, 3),
- Decreasing (such as 5, 2, 1 or 3, 0, -4),
- Neither (such as 1, 0, 2 or 1, 1, 1).

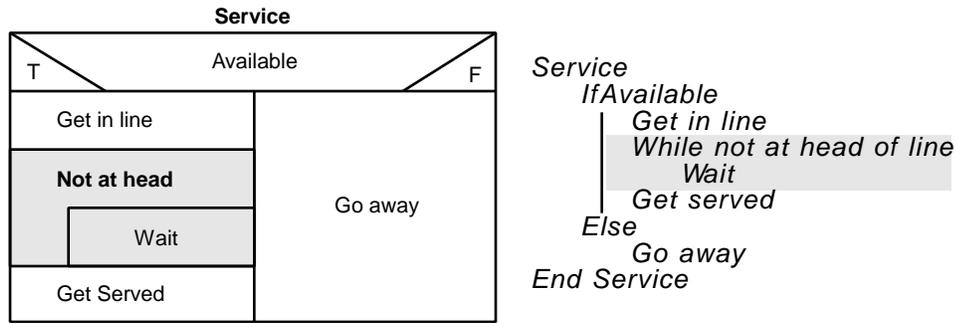
This algorithm should not be seen as five boxes (two conditions and three actions), but should instead be seen as two forms: a Selection nested within a Selection. Viewing algorithms as forms, rather than as the parts of forms, reduces the apparent complexity of the algorithm, thus keeping it simple.

In this case (Figure 4.5), the reduction of complexity is from 5 boxes to 2 forms. Although this may not seem significant, in other cases a reduction from 50 boxes to 20 forms could prove very helpful.

Extending this algorithm is very easy. The last box (Output “Neither”) could be expanded by further nesting to determine whether the values are “constant” (such as 7, 7, 7) or “non decreasing” (such as 2, 2, 3) or “undulating” (such as 3, 0, 3 or 3, 5, 3). A problem asking you to extend this algorithm can be found at the end of this chapter.

The *Service* algorithm, of Figure 4.6, describes a queue or waiting in line. It illustrates a Repetition form that is nested within a Selection form.

Figure 4.6 A Repetition form nested in a Selection form



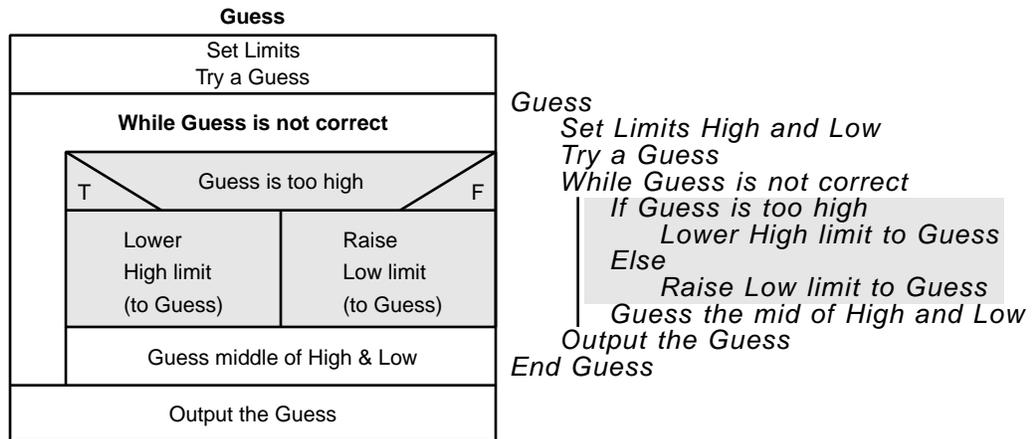
In Figure 4.6, if the service is available, then the following sequence of actions are performed:

- Get in line.
- While its not your turn, wait (Repetition form).
- When its your turn, get served.

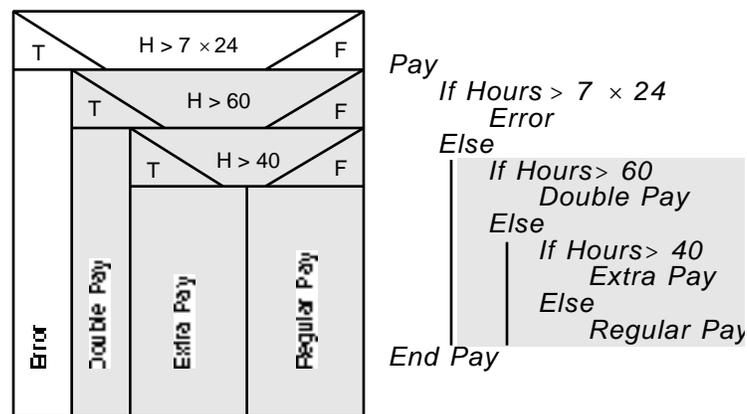
If the service is not available, the action is to go away.

The *Guess* algorithm of Figure 4.7 determines an unknown value by a series of guesses, each guess getting closer to the final correct result. It illustrates a Selection form nested within a Repetition form. This guessing method will be used later in the Bisection or Binary Search algorithm.

Figure 4.7 A Selection form nested in a Repetition form



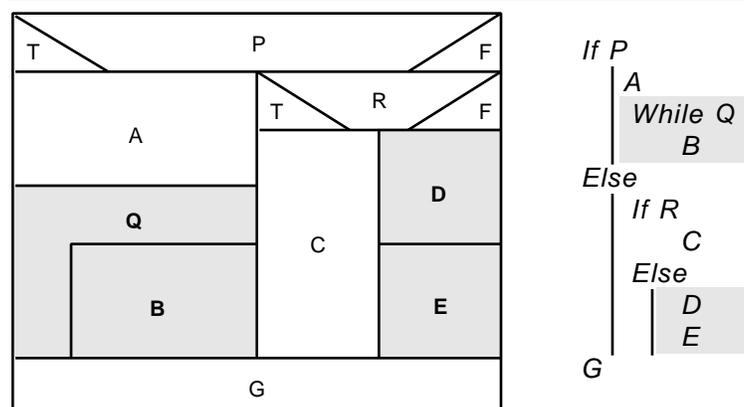
The *Pay* algorithm, shown in Figure 4.8, illustrates a deeper level of nesting, where a Selection contains a Selection within a Selection.

Figure 4.8 Nested Selection forms

Note: an algorithm should be analyzed by looking at how many forms are used, and how they are interconnected. An algorithm should not be analyzed by looking at how many instructions are used.

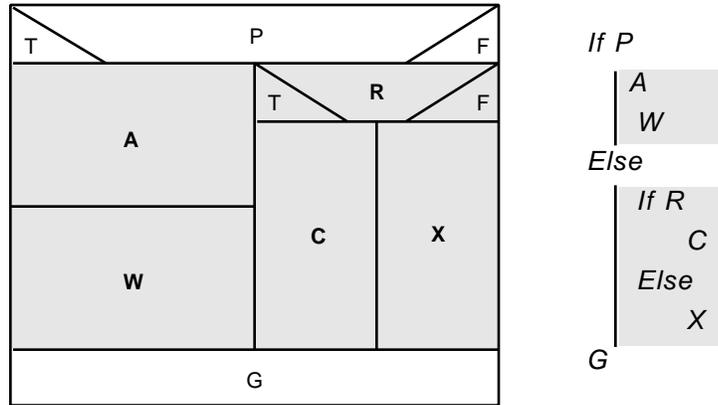
Deep Nesting

When complex algorithms are represented as either flowblocks or pseudocode, they are easy to split into their nested components. Figure 4.9 shows an example algorithm of moderate complexity, represented as a flowblock diagram with its pseudocode equivalent. We are only concerned here with the structure of the algorithm and not with the details of the actual conditions or actions. We have therefore symbolized them by single letters, (A, B, C, ..., G) for the actions and (P, Q, R) for the conditions. Both representations show two nested components in shaded areas.

Figure 4.9 A complex algorithm showing components

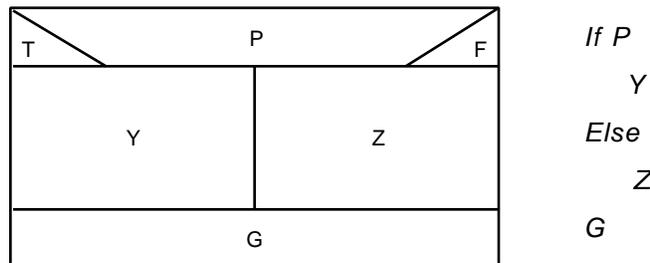
Each of these shaded areas could be given an action name, say W and X, and be shown as simple actions. This makes for a less complex representation of the algorithm, as in Figure 4.10.

Figure 4.10 Simpler version of algorithm of Figure 4.9



This process of replacing nested components by simple named actions to produce simpler representations of the algorithm can be continued to produce, for example, the version shown in Figure 4.11, where the shaded areas of Figure 4.10 have been renamed Y and Z.

Figure 4.11 Simpler version of algorithm of Figure 4.10

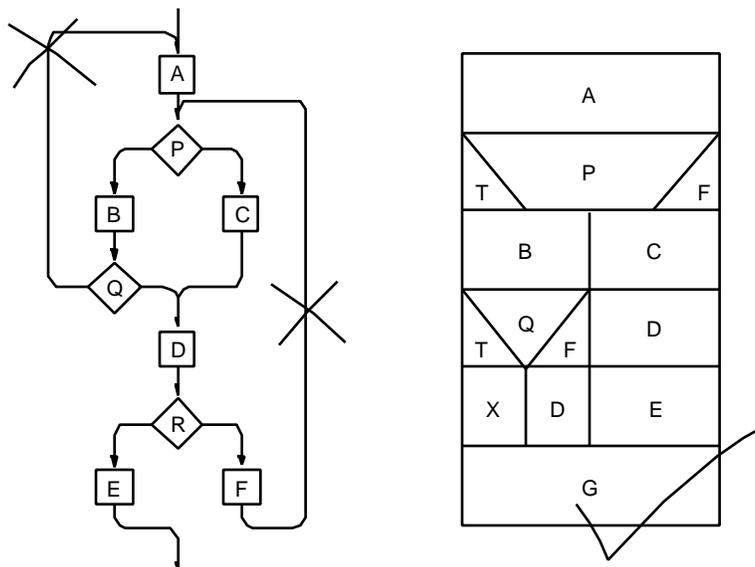


In this process, simplification is achieved by hiding details inside a box and giving it a new name. For the name to be more understandable, it should describe what the action does. For example, if Y or Z sorts a group of numbers or calculates change, they should be named either *Sort* or *Give Change*. The process of reducing the complexity by hiding the details is known as *abstraction*. Eventually, the whole algorithm could be represented by a single named action box through simplification and abstraction.

Successive simplification of algorithms represented as flowblocks or pseudocode is always possible because such algorithms are always well structured. Why? Because it is impossible to produce a badly structured algorithm in either of these representations while using the four basic forms. However, this does not always hold true for algorithms represented as flowcharts.

Figure 4.12 shows an improperly structured flowchart, which cannot be represented either as a flowblock or in pseudocode without having its structure changed. To find out why the flowchart is badly structured, remember that our basic forms have only one way in and one way out; try to cut the flowchart into blocks that have this property. A corrected and different version of this flowchart, as a flowblock, is shown on the right of Figure 4.12.

Figure 4.12 Avoiding badly structured flowcharts with flowblocks



Any process can be represented as a well-structured algorithm. For this reason, we will represent algorithms as either flowblock diagrams or pseudocode in the rest of this book.

4.3 Different Algorithms, Same Problem

Equivalent Algorithms

Before going any further, it is important to realize that two algorithms may behave the same way, but be structured differently. For example, consider the problem of determining whether or not a given year is a leap year.

The algorithm *Leap1* was considered previously in Figure 3.40, and is shown in Pseudocode 4.1. It begins with a Selection asking if the year is divisible by 400. The algorithm has two other Selections nested within it.

All paths in this algorithm can be tested by the four test cases: 2000, 1900, 1984, and 2001. For example, the year 2001 follows the Else path at each of the three

Selections. The paths taken by the four test cases are indicated by the assertions at the right of the pseudocode.

Pseudocode 4.1 Algorithm Leap1

```

Leap 1
  If Year is divisible by 400
    Year is a leap year           {Year 2000 will finish here}
  Else
    If Year is divisible by 100
      Year is not a leap year     (Year 1900 will finish here)
    Else
      If Year is divisible by 4
        Year is a leap year       {Year 1984 will finish here}
      Else
        Year is not a leap year   {Year 2001 will finish here}
  End Leap 1

```

A typical year (such as 2001) must go through 3 selections.

The algorithm *Leap2*, shown in Pseudocode 4.2, works in the opposite direction to *Leap1*. It begins with the Selection asking if Year is divisible by 4. Notice that in this version, the year 2001 requires only one Selection. The assertion on the right show where each path for the four test cases finish.

Pseudocode 4.2 Algorithm Leap2

```

Leap 2
  If Year is divisible by 4
    If Year is divisible by 100
      If Year is divisible by 400
        Year is a leap year       {Year 2000 will finish here}
      Else
        Year is not a leap year   (Year 1900 will finish here)
    Else
      Year is a leap year         {Year 1984 will finish here}
  Else
    Year is not a leap year       {Year 2001 will finish here}
  End Leap 2

```

A typical year must go through only 1 selection.

These two algorithms (*Leap1* and *Leap2*) are identical in behavior because they produce the same results in all possible cases. In other words, they are equivalent. They are, however, different in structure as is shown by the fact that the same test cases encounter different numbers of selections in the two algorithms. Which algorithm do you prefer?

The important thing now is not which one you prefer, but that you have a choice of one or the other. If two algorithms are equivalent in one sense, this is an opportunity for selecting the optimal one in another sense. The selection depends on your goals.

For efficiency reasons (minimizing time by having fewer Selections), you might prefer *Leap2* because in most common cases (when the year is not divisible by 4

which occurs 75% of the time), the number of Selections is smaller. Only one Selection is encountered in *Leap2*, compared to three Selections in *Leap1*.

Efficiency, or speed, is not always the best goal. Other goals include convenience, elegance, ease of communication, robustness and some others that will be considered in later chapters.

We can define yet another algorithm that is equivalent to the *Leap1* and *Leap2* algorithms. Pseudocode 4.3 illustrates such an algorithm, *Leap3*, which begins with the Selection checking if the year is divisible by 100. The structure is quite different from the others, because all test cases go through exactly two Selections; there are no short paths here.

Pseudocode 4.3 Algorithm Leap3

```

Leap 3
  If Year is divisible by 100
    If Year is divisible by 400
      Year is a leap year      {Year 2000 will finish here}
    Else
      Year is not a leap year  {Year 1900 will finish here}
  Else
    If Year is divisible by 4
      Year is a leap year      {Year 1984 will finish here}
    Else
      Year is not a leap year  {Year 2001 will finish here}
End Leap 3

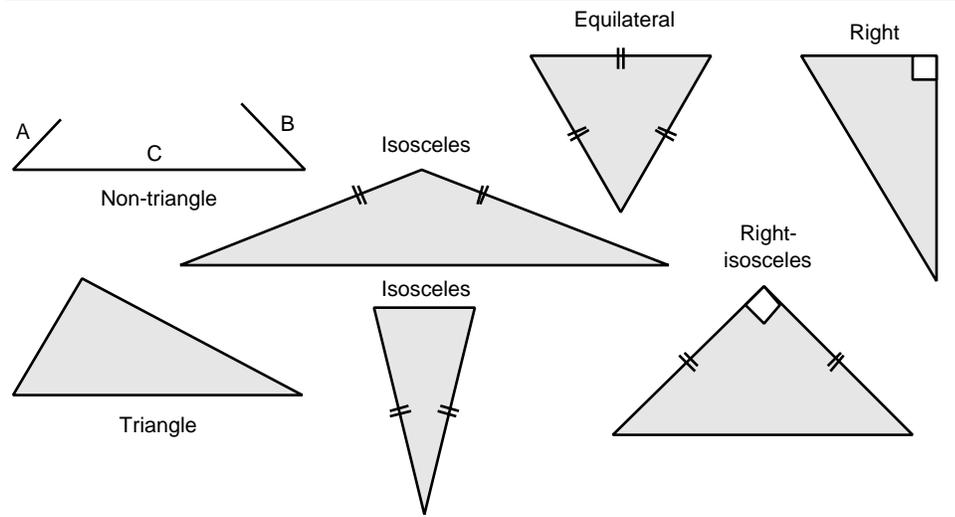
```

Any type of year must go through 2 selections.

There are still more algorithms that determine whether or not a year is a leap year, some of these will be considered later. For now, the important idea is not to concentrate on optimization, but simply to realize that there can be many different ways of writing an algorithm to solve the same problem.

Alternative Algorithms

There are often alternative ways of creating algorithms, some ways more convenient or better than others. Figure 4.13 illustrate algorithms that are not equivalent. Alternative algorithms are not equivalent when the output is not the same for all possible cases.

Figure 4.13 Various triangles

In Figure 4.13, the problem is to determine whether three given numbers A, B, C (representing lengths of sides) could create a triangle and, if so, whether it is one of the following three types:

- Isosceles: two sides are equal, or
- Equilateral: all three sides equal, or
- A right triangle: one 90° angle.

Triangles of various kinds are shown in Figure 4.13. From the non-triangle, we can see that a triangle is formed only if the sum of the two shorter sides exceeds the longest side. If the sides are given in increasing order (say $A \leq B \leq C$), the condition for a triangle is:

$$(A + B > C)$$

Similarly, for an equilateral triangle, the general condition is:

$$(A = B) \text{ AND } (B = C)$$

But if A, B, and C are in order, this condition becomes simply:

$$(A = C)$$

For an isosceles triangle, the general condition is:

$$(A = B) \text{ OR } (B = C) \text{ OR } (A = C)$$

which becomes the following when sides are ordered:

$$(A = B) \text{ OR } (B = C)$$

The point is that if the data values are structured, then this structure could be used to simplify the algorithm. In the triangle example, if the data has the structure of being ordered, then simpler tests can be used. There are various ways to put A, B, and C in increasing order.

One way is to request that the values be entered in increasing order, but we would have to check to see if these values were entered as instructed, and we

would have to repeat this process until the values are entered correctly. Another way would be to create a sub-algorithm that puts any three input values into increasing order. We will use the latter method.

Two alternative algorithms for classifying the triangles are shown in Pseudocode 4.4 and Figure 4.14. In each, the action of sorting the three numbers A, B, C is labeled as:

Sort sides so that A ≤ B ≤ C

This algorithm, also called *Sort3*, will be developed later.

Pseudocode 4.4 Algorithm Triangle Classification 1

```

Triangle Classification 1
  Input sides A, B, C
  Sort sides so that A ≤ B ≤ C
  If A + B ≤ C
    Output "Not a triangle"
  Else
    If A = C
      Output "Equilateral triangle"
    Else
      If (A = B) OR (B = C)
        Output "Isosceles triangle"
      If A2 + B2 = C2
        Output "Right triangle"
      Else
        Output "Triangle"
  End Triangle Classification 1

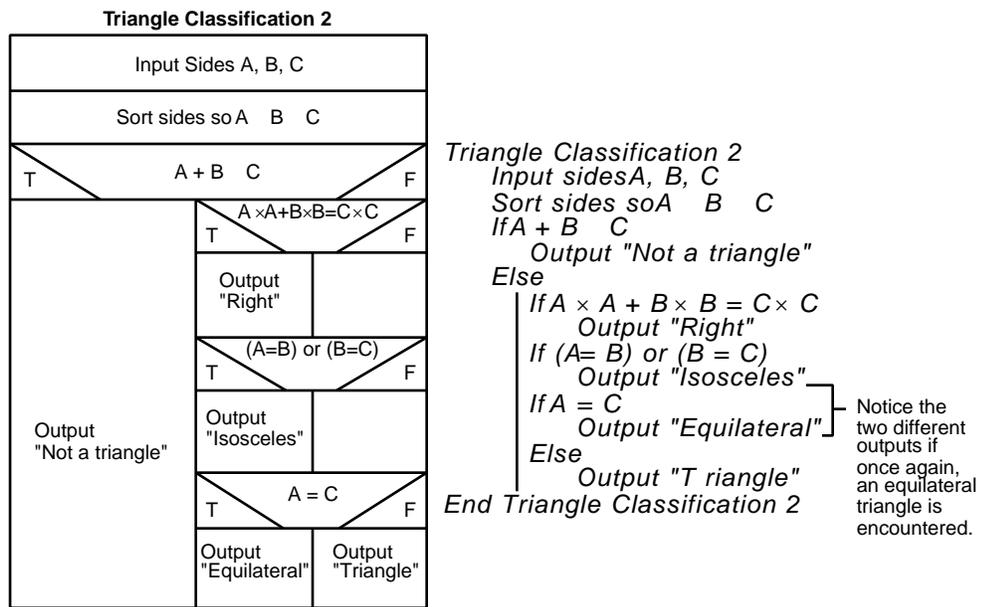
```

Notice the single output if an equilateral triangle is encountered.

The algorithm in Pseudocode 4.4 tests first for the equilateral property, and if it holds, terminates without indicating that the triangle is also isosceles although all equilateral triangles are isosceles. In the other cases, the algorithm checks for the isosceles property, and then for the right triangle property.

The second *Triangle Classification* algorithm, shown in Figure 4.14, appears in two representations: a flowblock diagram and pseudocode to remind us of the equivalence of these two notations. This algorithm first tests for isosceles triangles and, if successful, tests for the equilateral property. Therefore, an equilateral triangle will be shown as having both the isosceles and equilateral properties. This may be redundant, but sometimes it is clearer than having to recall that one property implies (or covers) another property.

Figure 4.14 Algorithm Triangle Classification 2

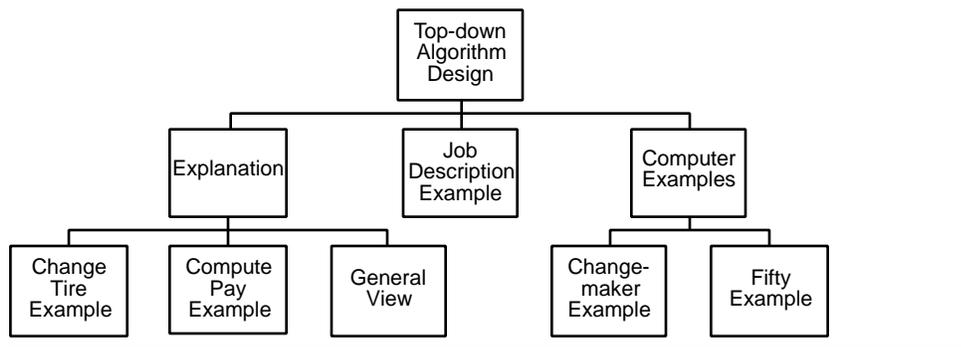


Remember that in the latter case of Pseudocode 4.4, an equilateral triangle would only show the equilateral property. For this reason, these two *Triangle Classification* algorithms are not equivalent. However, either algorithm can be used for they both classify triangles.

4.4 Top-down Algorithm Design

In step 2 of our problem-solving method, Solution Design, we use a top down approach to build a structure chart to define a solution. For example, we could build a structure chart to help us plan out this section (Figure 4.15).

Figure 4.15 Structure chart for this section



This chart, Figure 4.15, was built top-down: we started with the section title, then decided that we needed an explanation of the main concept, Explanation, an illustration that uses a general description of a person's job Job Description Example, and some computer-related examples.

The explanation itself was defined as including two small examples, Change Tire and Compute Pay, and a general view. As you can see, the top-down approach is useful for planning a solution to any problem. When it comes to creating algorithms (the Solution Refinement step of our problem-solving method), we use a similar approach.

Although there are two general approaches to algorithm design, top-down and bottom-up, we will focus on the top-down approach.

Top-down is an important method for designing algorithms. Simply stated, it starts at the top, with the most general view—a bird's-eye view—and then proceeds to lower levels by successively splitting the larger blocks into smaller, more manageable blocks. Finally, at the lowest levels it treats the fine details. The motto of this process is *Divide and conquer*.

Conversely, the Bottom-up method starts at the bottom with the details—the worm's eye view—and then proceeds to higher levels by combining smaller blocks. Unfortunately, by concentrating on the details first, without the context provided by the bird's eye view, the building process may quickly become mired in the details and unmanageable.

Other names for top-down design are: stepwise refinement, iterative multi-level modeling and hierarchical programming. It is often pictured with break-out diagrams as shown in the Figures 4.16 to 4.19.

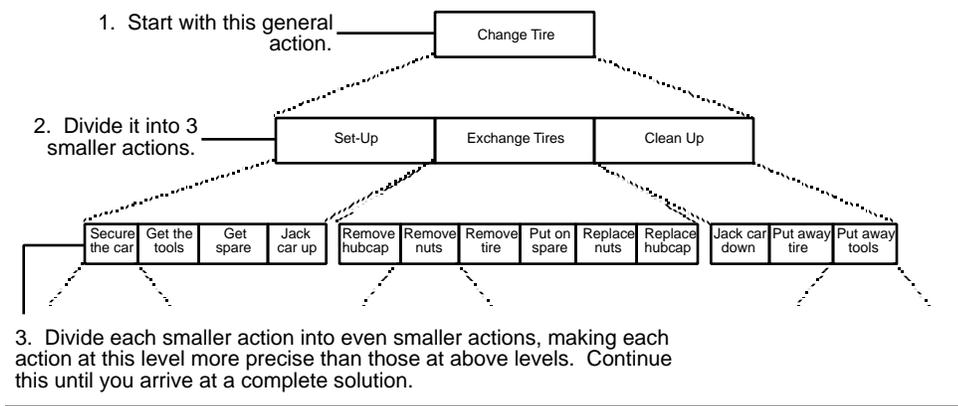
Tip Use the top down approach to create algorithms. Start from the general view and progressively refine this view until the level of detail becomes simple.

Design begins at the top as a single action. Then, the action is broken out or refined into a small number of sub-actions. These sub-actions are independent of one another and are not very detailed. This process continues by further refining each sub-action into sub-sub-actions, gradually including more details.

For example, Figure 4.16 shows the single action Change Tire, broken into the following three sub-actions:

- Set-up,
- Exchange Tires, and
- Clean Up.

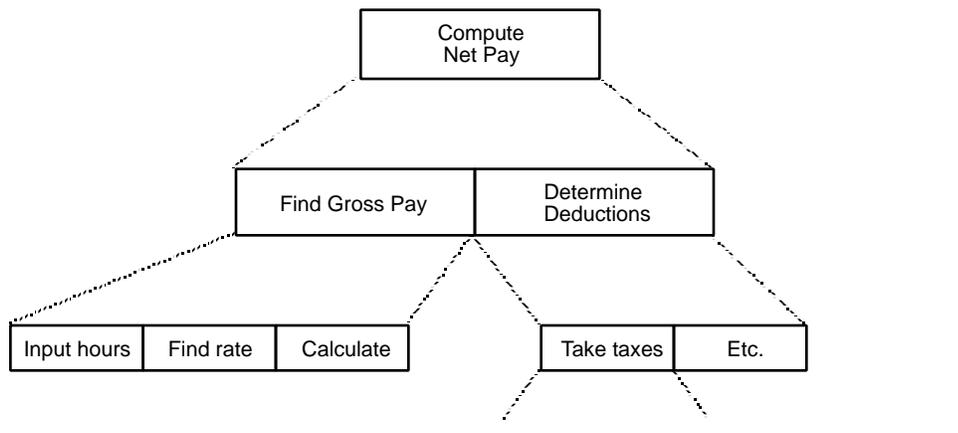
Figure 4.16 Breaking down Change Tire



At the third level, each of the previous sub-actions are refined further. If any of these smaller actions were seen as being too complex to be understood, they too could be broken out into further details. Ultimately, a stage is reached where every action is small enough to be understood without any further simplification: this is the complete solution.

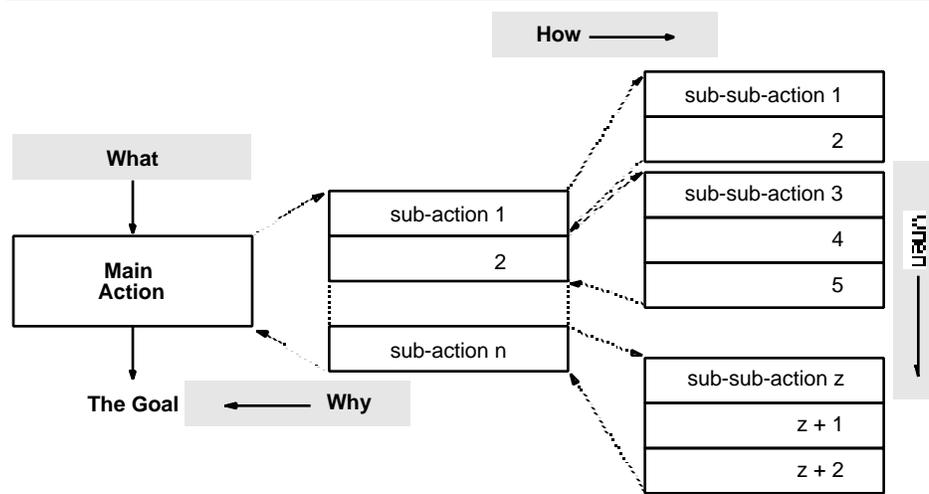
Figure 4.17 shows a more computer-oriented example, Compute Net Pay. Once the last stage of stepwise refinement is reached, we are ready to express the algorithms in pseudocode, and from there, in some programming language.

Figure 4.17 The top-down view of Computing Net Pay



The two diagrams shown in Figures 4.16 and 4.17 have been drawn as “true” top-down diagrams—they start from the top and, as one moves down the page, they become more detailed. They are just as much break-out diagrams as are the left-to-right versions. The orientation is incidental; clarity is the real criterion. In Figure 4.18, we have reverted to the more familiar left-to-right form to illustrate the top-down design method in a generic manner.

Figure 4.18 A generic break-out diagram



The above figure illustrates the general break-out process and shows how answers to four important questions (what, how, when and why) are found using break-out diagrams:

- **What** main action is being done? This is shown furthest to the left.
- **How** is an action done? This is shown broken out at the right of an action.
- **When** are the actions done? This is specified by the sequence along the far right of the diagram.
- **Why** is a sub-action done? This is found to the left of the action.

Although, sometimes, only the step-by-step sequence of actions is required (the When at the right), it is important to see the rest of this structure. It is also very important to realize that each level drawn on the break-out diagram must be complete. This means that at each level drawn, all of the sub-actions (or sub-sub-actions) must be present.

Make each level of your break-out diagram complete. If, for example, you decide to only develop it two levels deep, make sure that the second level contains all of the actions necessary (even if they are very general) to solve the problem posed. For example, in Figure 4.17, if we were to only have the first 2 levels, we would need both definitions Find Gross Pay and Determine Deductions to make the BOD complete.

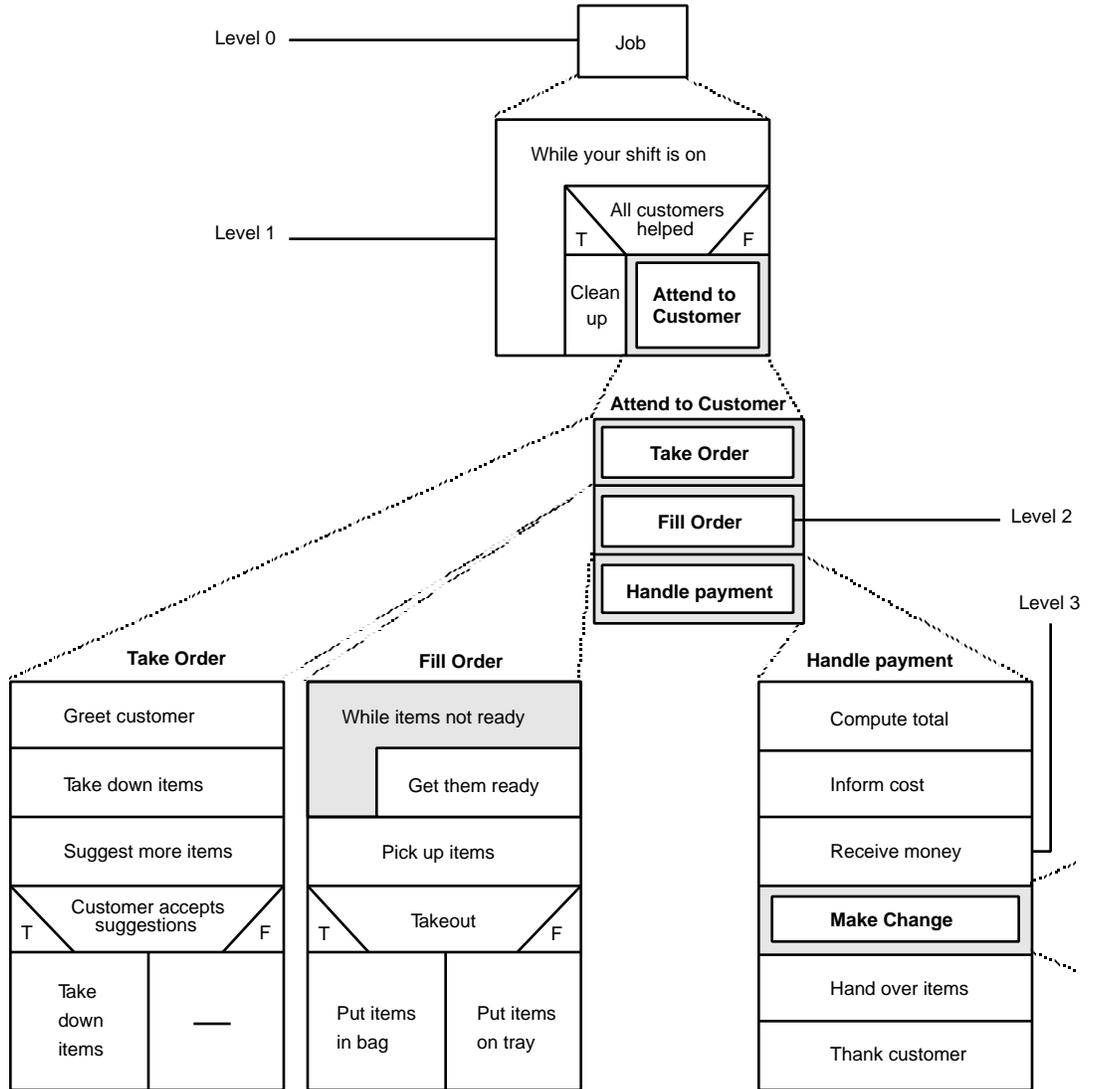
Don't leave out any actions at any of the levels of the break-out diagram. In Figure 4.17, notice that the word "etc." at the right indicates that this break-out diagram is incomplete!

The next few pages contain different examples of algorithms created using top-down design.

Job Description Example

Figure 4.19 shows the algorithm that an employee at a fast food restaurant follows. This algorithm is described in a top-down manner. It is developed in levels to show the convenience of “sub-” blocks in the top-down view.

Figure 4.19 Algorithm for an employee at a fast food restaurant



Level 1, in Figure 4.19, is a high level that shows no details, but gives a general view of how to proceed and refers to the sub-block Attend to Customer at a second lower level. Then, Attend to Customer is further broken down into three sub-blocks (or sub-algorithms) Take Order, Fill Order, and Handle Payment. Finally these three sub-blocks are refined in level 3.

If the detail is still not sufficient, then more levels must be created. For example, the last sub-algorithm, Handle Payment, refers to another sub-algorithm Make Change, which could be further refined at level 4 (not shown in Figure 4.19). To reveal how the change would be made, this Make Change sub-algorithm will be discussed in detail later.

The top-down method forces us to devise a general overview of a system before getting into its details. It also shows the segmenting of a larger system into smaller, independent modules such as Take Order, Fill Order, and Handle Payment. It is this segmenting that makes the complexity more understandable and manageable.

Change Maker Example

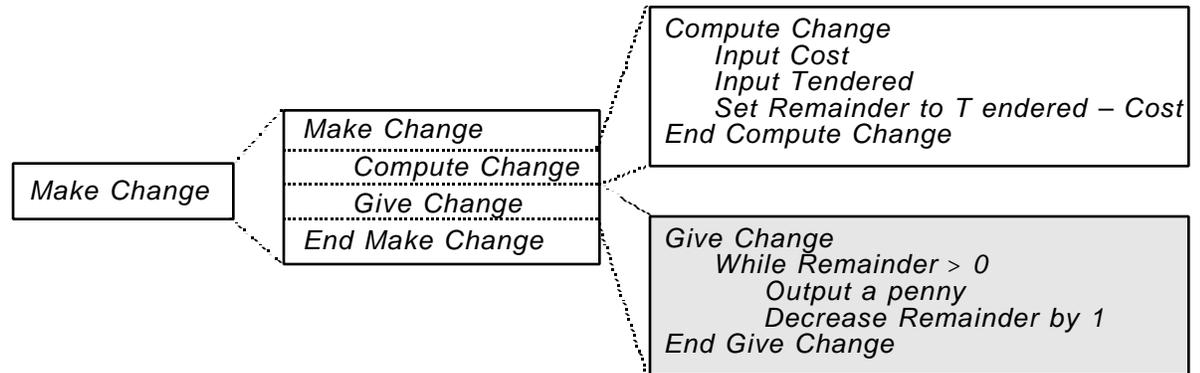
Note that an algorithm similar to Make Change was previously shown in Chapter 3 (Figures 3.32 and 3.33).

The order-taker algorithm of Figure 4.19 referred to the sub-algorithm Make Change which, as the name suggests, makes change for a customer. We will further illustrate the top-down algorithm design process, by developing this *Make Change* sub-algorithm using only pseudocode. Notice that the difference between an algorithm and a sub-algorithm is slight; a sub-algorithm is simply an algorithm that can be invoked by other algorithms.

The sub-algorithm, *Make Change*, will make change from an amount tendered for an item whose cost is given in cents. At the very top level, there is a simple action *Make Change*.

By itself, this does not help very much, but it does provide a start from which to develop the next level, which consists of a sequence of two sub-actions. The next step is to expand these two sub-actions. As indicated by their names, *Compute Change* computes the amount to be returned to the customer and *Give Change* produces the proper coins. Expanding *Compute Change* is easy and shown in Figure 4.20.

Figure 4.20 Break-out of Make Change sub-algorithm



The rest of this section will be used to develop the *Give Change* algorithm, as there are many ways to solve this problem. One way, shown in the shaded box in Figure 4.20, is to output *Remainder* all as pennies. If the cost is 1 cent and the amount tendered is a dollar, this means that 99 pennies are output! This is one solution that works, is correct, complete and short.

This solution is, however, not practical or convenient because giving a customer change in pennies creates “ill will”. This means that Pseudocode 4.8 must be replaced with a solution more beneficial to the customer.

Another common way of making change involves adding up coins from the amount *Cost* up to the amount *Tendered*. This method is often preferred by people who wish to avoid computing the remaining amount because they prefer not to subtract. This method will be treated in Chapter 5.

Yet another way of making change is to modify the first solution in Pseudocode 4.8. The modified version is shown in Pseudocode 4.5.

Pseudocode 4.5 A better way of refining Give Change

```

Give Change
    {Remainder = 0}
    Give Quarters
    Give Dimes
    Give Nickels
    Give Pennies
    {Remainder = 0}
End Give Change
    
```

Since we wish fewer coins to be output, we first consider large coins, quarters, followed by dimes, nickels and then pennies. What we do for each coin is a detail not considered at this level. The sub-actions will be “opened up” at the next lower level.

Assertions are very useful to list along with algorithms, here (Pseudocode 4.5) they are shown in braces at the right of the algorithm. For example, at the

beginning it is assumed that the amount to be returned to the customer, *Remainder*, is positive {*Remainder* > 0}. At the end, the *Remainder* is zero.

At the next level, each of the sub-actions of *Give Change* of Pseudocode 4.5 will be shown in detail. We will combine this with the *Compute Change* part to give the complete *Make Change* algorithm in Pseudocode 4.6.

Pseudocode 4.6 The complete, detailed Make Change algorithm

```

Make Change
  Input Cost
  Input Tentered
  Set Remainder to Tentered - Cost
  {Remainder > 0}
  While Remainder >= 25
    Output a quarter
    Decrease Remainder by 25
    {Remainder < 25}
  While Remainder >= 10
    Output a dime
    Decrease Remainder by 10
    {Remainder < 10}
  While Remainder >= 5
    Output a nickel
    Decrease Remainder by 5
    {Remainder < 5}
  While Remainder >= 1
    Output a penny
    Decrease Remainder by 1
    {Remainder = 0}
  End Make Change
    
```

The sub-actions from Pseudocode 4.5 are broken down in Pseudocode 4.6 as follows:

- *Give Quarters*, the first sub-action, is broken out into a Repetition form, because more than one quarter may be output. While the *Remainder* is greater than 25, a quarter is output and the remainder is decreased by 25. This continues until the *Remainder* is less than 25 as indicated by the assertion {*Remainder* < 25}.
- *Give Dimes*, the second sub-action, is broken out in a similar way to *Give Quarters*. When its actions are done, *Remainder* is less than 10 as shown in the assertion.
- *Give Nickels*, the third sub-action, is slightly different from the previous two because at most one nickel can be output. If *Remainder* is greater than 5, a nickel is output and *Remainder* is decreased by 5.
- *Give Pennies*, finally is done with a Repetition similar to the first two sub-actions, because more than one penny may be output.

Similarity is a useful property. Similar things should be treated similarly. Most of the coins in Pseudocode 4.6 were treated similarly with a Repetition form, but *Give Nickels* was done with a Selection form. This Selection form in *Give Nickels* can be converted into a Repetition form as shown in Pseudocode 4.7.

Pseudocode 4.7 Two equivalent versions of Give Nickels

```

                                {Remainder < 10}
If Remainder ≥ 5
  | Output a nickel
  | Decrease Remainder by 5
                                {Remainder < 5}
If Remainder ≥ 5
  | Output a nickel
  | Decrease Remainder by 5

```

To show the equivalence in the behavior of these two versions, assertions are also indicated. Initially the *Remainder* is less than ten. Considering the Repetition form (right), if *Remainder* is not greater than or equal to 5, then nothing is done. When you apply this same condition to the Selection form, nothing is done as well. This means that when *Remainder* is not greater than or equal to 5, then both forms are equivalent.

If *Remainder* is greater than or equal to 5 (and less than 10 as the assertion states), then the body of the Repetition is performed (a nickel output and *Remainder* decreased by 5). Once the Repetition form has been performed once, the new value of *Remainder* is now less than 5, so no further Repetition is possible. This again is equivalent in behavior to the Selection form at the left. Hence these two sub-algorithms, in the context given by the assertions, are equivalent in behavior.

As with all algorithms, modifications to the *Make Change* algorithm are possible. For example, it could be generalized by allowing the input of any amount to be tendered rather than at most a dollar—this is assumed by the fact that no bills are considered in the change making.

This algorithm could also be extended to more denominations (fifty-cent coins, dollar and two-dollar bills), and it could be made foolproof by testing that the input values are in the proper range (cost is positive, and amount tendered is greater than or equal to the cost). Notice that the block form suggests that modifications can be done by inserting or substituting blocks, which encourage “modular” design. More of these equivalent substitutions will be considered in later chapters.

A Game Example, Fifty

Sports and games often provide examples of algorithms because they involve rules that have the same properties as algorithms: generality, completeness, consistency and finiteness.

Games may involve chance (using dice, cards, pebbles, and so on) or they may involve skill (using balls, bats, arrows, targets, and so on) or both. Here we will concentrate on a simple game involving dice. Dice games have a long history; they have been found in ancient Egyptian tombs dated 1500 BC. The earliest dice were probably a cube shaped bone, the astragalus, from the ankle of a sheep.

Dice usually consist of cubes (of bone, ivory, sugar, etc.) made with 1 to 6 dots on a face, so that opposite sides add up to 7. When a die is rolled, each face has an equal chance of landing face up. When two dice are thrown, some sums are more likely than others. For example, a sum of two can be formed in only one way (1 + 1), whereas the sum of 7 can be formed in many ways (1 + 6, 2 + 5, 3 + 4, ...).

Fifty is a dice game played by two people with two dice. It is described in Figure 4.21.

Figure 4.21 Verbal description of the dice game Fifty

Fifty: a dice game for two people with two dice.

The players each take a turn in one round and the rounds continue as long as neither person's score has reached 50.

During a player's turn, the two dice are thrown once. There is no change in score unless two identical numbers come up.

If both dice are sixes, then 25 points are added to that player's score.

If both dice are threes, then that player's score is reset to zero.

If any other doubles are thrown, five points are scored.

The winner is the one whose score after a round is at least 50 and the higher of the two scores. A tie is also possible.

We will use the development of an algorithm to play Fifty as another example of the top-down development method. At the first level of detail, *Fifty* splits into three sub-actions, as shown in Pseudocode 4.8.

Pseudocode 4.8 Refining Fifty into three actions

```

Fifty
  Set up
  Play
  Evaluate
End Fifty

```

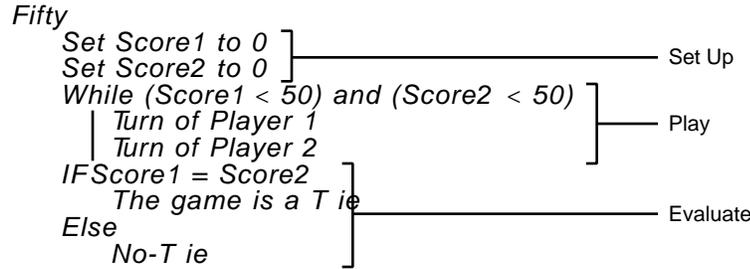
Set up initializes the scores, *Play* goes through a game and *Evaluate* decides the outcome. These three sub-actions are explained as follows:

- *Set up* simply sets the two scores *Score1* and *Score2* to zero. Notice that the order of playing is not important; the player who goes first does not have an advantage, since both players get a turn during every round. In other games, the setup is often more complex.

- *Play* loops through a round during which each person gets a turn, and this continues as long as neither player has a score of 50 or more.
- *Evaluate* will choose the winner. It first determines whether there is a tie. If there is no tie, then at the next level, the winner is determined.

At the second level of development, the algorithm is shown in Pseudocode 4.9.

Pseudocode 4.9 Refining each action in Fifty further

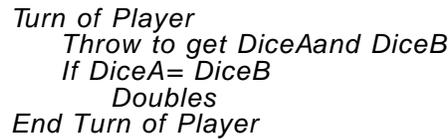


The sub-algorithm *Turn of Player* must also be broken out further into the following stages or levels:

- First, the dice are thrown to get *DiceA* and *DiceB* using the *Throw* sub-algorithm.
- Then, a Selection determines whether there are any *Doubles*. For example, if the scores *DiceA* and *DiceB* are equal, another sub-algorithm determines whether they are a *Good Double* or not.

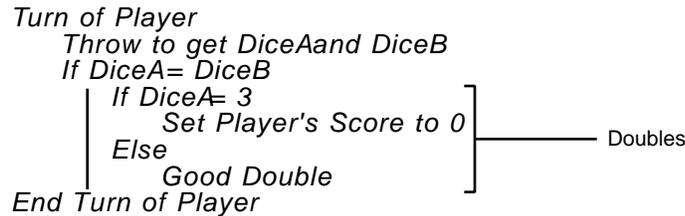
At this level, the sub-algorithm *Turn of Player* is shown in Pseudocode 4.10.

Pseudocode 4.10 Defining the sub-algorithm Turn of Player



The sub-algorithm *Doubles* may then be expanded according to the rules of the game so that the sub-algorithm *Turn of Player* becomes Pseudocode 4.11.

Pseudocode 4.11 Refining Turn of Player further



Finally, *Good Double* is expanded to determine how much should be added to the player's score. The fully-expanded *Turn of Player* sub-algorithm is shown in Pseudocode 4.12.

Pseudocode 4.12 The complete, detailed sub-algorithm Turn of Player

```

Turn of Player
  Throw to get DiceA and DiceB
  If DiceA = DiceB
    If DiceA = 3
      Set Player's Score to 0
    Else
      If DiceA = 6
        Add 25 to Player's Score
      Else
        Add 5 to Player's Score
  End Turn of Player
  
```

Doubles

Good Double

The only other sub-algorithm left to be expanded is *No-Tie* and, when this is done, *Evaluate* becomes the sub-algorithm in Pseudocode 4.13.

Pseudocode 4.13 Refining the No-Tie part of Evaluate

```

Evaluate
  If Score1 = Score2
    The game is a Tie
  Else
    If Score1 > Score2
      Player 1 wins
    Else
      Player 2 wins
  End Evaluate
  
```

No-Tie

We now have all the parts of algorithm *Fifty* and we can produce the final solution by putting them all together. The algorithm, fully expanded, is shown in Pseudocode 4.14.

Pseudocode 4.14 The complete Fifty algorithm

```

Fifty
  Set Score1 to 0
  Set Score2 to 0
  While (Score1 < 50) and (Score2 < 50)
    Turn of Player 1
    Turn of Player 2
  IF Score1 = Score2
    The game is a Tie
  Else
    If Score1 > Score2
      Player 1 wins
    Else
      Player 2 wins
  End Fifty
  
```

Since Turn of Player occurs twice, it is better to keep it as a sub-algorithm that will be invoked twice.

We have described this of the previous algorithms in elaborate detail as an illustration of the top-down development method because it is so important to the production of well structured algorithms. In practice, most of these stages would go on in the mind of the programmer without being put on paper, **but they would take place!**

These shortcuts come only with experience. If you are a beginner, you should try to design your algorithms by mimicking what we have done in this example. After creating a few algorithms on your own, you will adapt the method to your particular way of thinking, and creating algorithms will become easier.

Note: Whether you are a beginner or not, you will always need method.
--

Incidentally, the top-down method did not come into being with computers. It has been known for hundreds of years. René Descartes, the French philosopher and mathematician, wrote in 1637 in his *Discourse on Method*,

*Divide each difficulty into as many parts as possible
so that it may be overcome more easily.*

*Starting with the simplest and easiest to understand,
consider things in order, moving by degrees
to the most complex.*

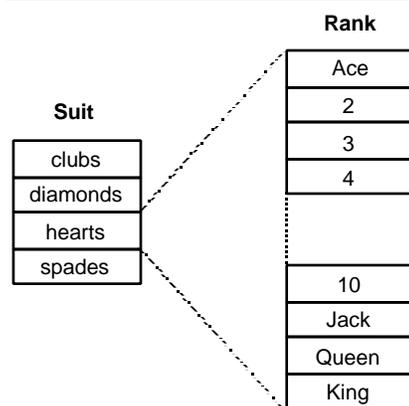
4.5 How Data and Algorithms Fit Together

In Chapter 3, we defined algorithms to be plans for performing actions on data. In this chapter, we have placed the emphasis so far on the actions, and we have hardly mentioned the data. In fact, all the data we have used in our earlier examples have been numbers that seem simpler than actions, but we will soon see that data can be much more complex. Let's try to establish a more balanced view of data and actions.

Structured Data

Emphasis so far has been on the flow of control, or sequence of actions and the structure of this flow. Our data have been simple Integers or Real Numbers, but data can also be structured. As with algorithms, structured data can be described by break-out diagrams.

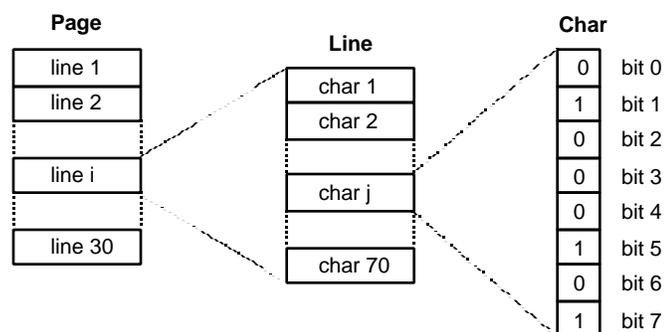
Figure 4.22 Card deck decomposition



For instance, a deck of playing cards provides an example of structured data. It consists of 52 cards, broken into four suits, each having thirteen ranks. Figure 4.22 shows a deck as a break-out diagram that is a simple, self-explanatory structure.

Similarly, Figure 4.23 shows how a page of text (stored in a computer memory) consists of lines, which further break up into characters. Inside the computer, each character is represented by eight binary digits, *bits*. This simple structure shows a page of 30 lines, each having 70 characters, with each character represented by 8 bits. So, each page uses a total of $30 \times 70 \times 8$ or 16 800 bits.

Figure 4.23 Text break-out diagram



In Figure 4.24 we show again how a year can be split up into months, which are split into days and then into hours. Since there is not a constant number of days in each month— it varies from 28 to 31— the last number in the second level is represented by an “n”.

Figure 4.24 Time break-out diagram

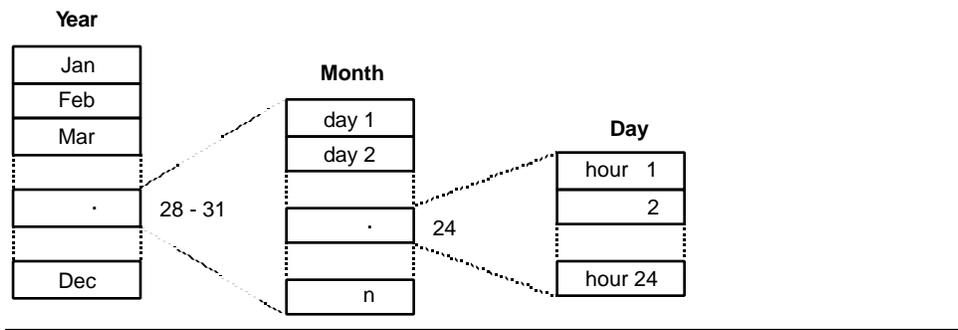
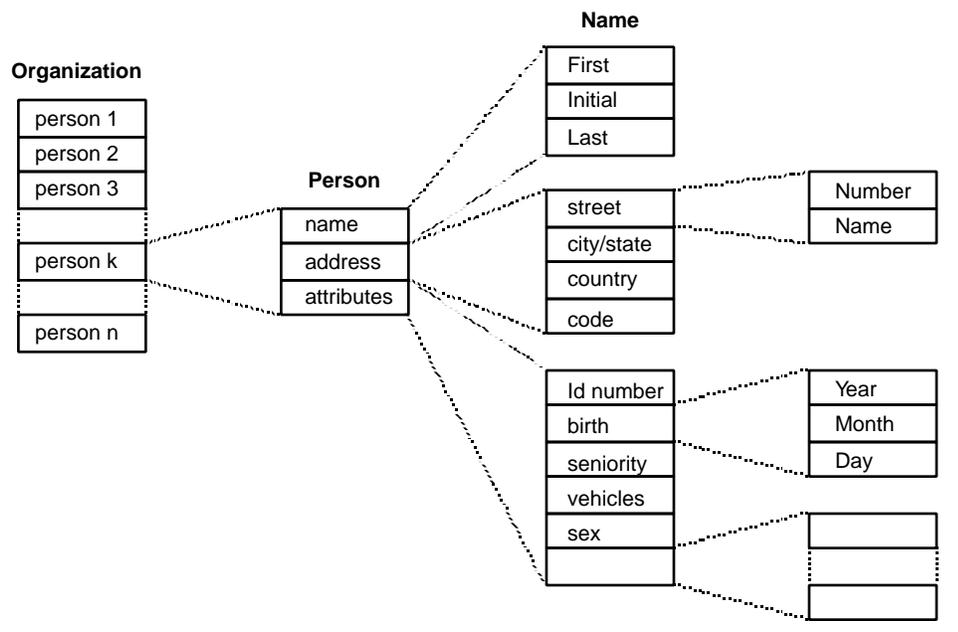


Figure 4.25 shows an organization of people, each specified by name, address and attributes. Each of these three parts is then broken out further. Notice that at the first level all the break-outs were of the same form (persons), but at the second level each break-out has a different form.

Figure 4.25 People break-out diagram



Structured data of the type in Figure 4.25 can also be represented in a linear form using indentation. Each level is indented from the preceding level, as shown in the representation of the previous example in Figure 4.26.

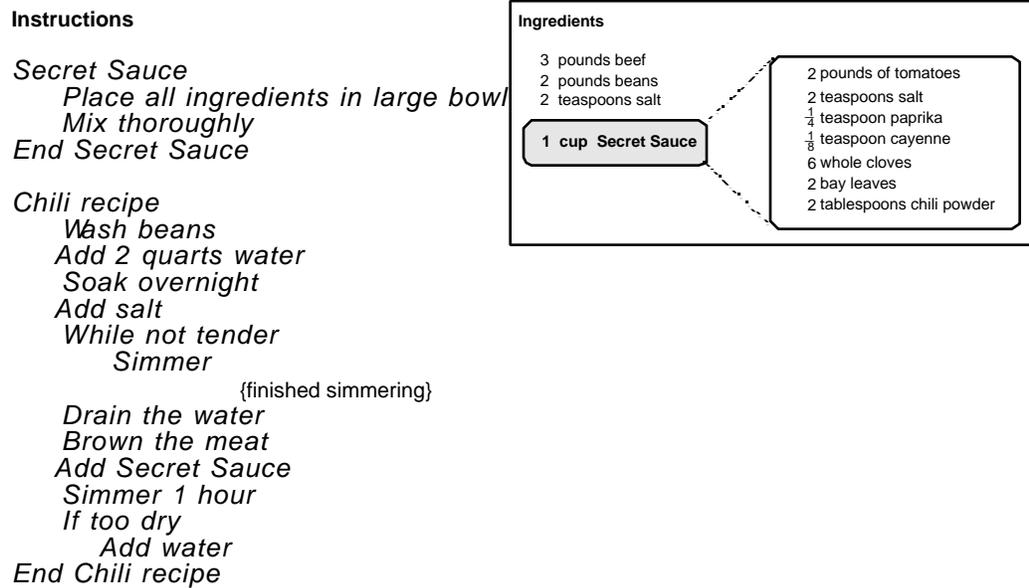
Figure 4.26 Linear form of structured data

Person 1:	
Name	
FirstName	: John
Initial	: M
LastName	: Motil
Address	
Street	: 1234 Main Street
City/State	: Northridge, CA
Country	: USA
Code	: 91330
Attributes	
IdNumber	: 123-45-6789
Birth	
Year	: 2000
Month	: Feb
Day	: 29

As you may have already guessed, structured data is very important in computing. We introduce structured data here because we want you to be aware that data is usually more complex than what you have seen so far. Structured data will not be used in the next three chapters in order to concentrate on the action parts of algorithms, only simple data values will be used. In Chapter 8, more complex data structures will be introduced and used.

Chili Recipe Example

The recipe shown in Figure 4.27 is an almost typical cooking algorithm for making Chili; however, it can also serve as a model for programs in many modern programming languages. The recipe comprises two major sections: the ingredient list (right) and the algorithm (left) which gives the sequence of operations to be performed using the ingredients.

Figure 4.27 Cooking algorithm for Chili

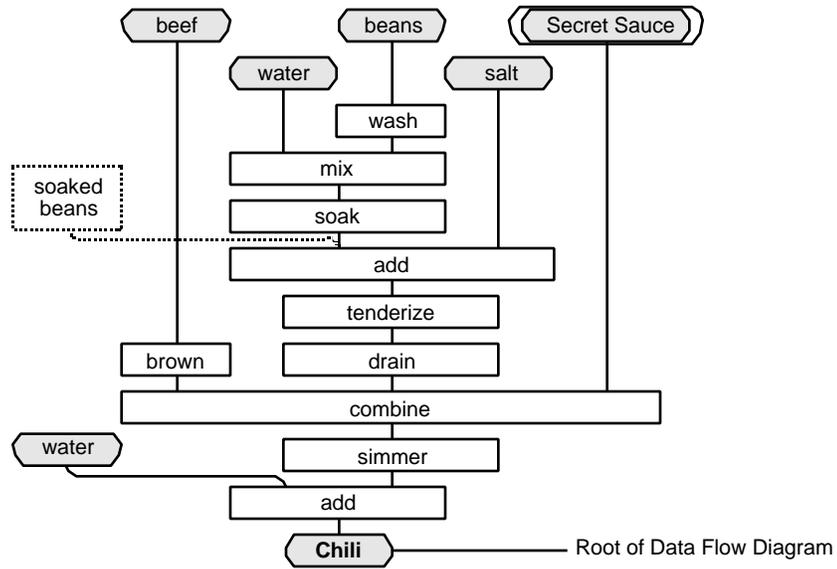
Secret Sauce is a sub-recipe, listed as a part of the ingredients at the top of Figure 4.27, and then defined as a separate recipe in the list of instructions. *Secret Sauce* is viewed as an ingredient for the main recipe, but is also viewed as a separate recipe by the cook!

In a parallel manner, an algorithm is comprised of two parts:

- the data list, and
- the list of instructions which shows the sequence of operations to be performed on the data.

The model presented by Figure 4.27 (data specifications followed by sub-algorithms followed by the main algorithm), corresponds to the structure of programs written in many modern programming languages such as Pascal or Modula-2.

The same recipe for Chili, shown as a data-flow diagram in Figure 4.28, contrasts the previous flow of actions. Notice how Figure 4.28 resembles a tree. Its leaves represent the data and each node indicates how the data “flows together” to make new data, eventually arriving at a final node (the root of the tree) which indicates the final data, Chili!

Figure 4.28 Data-flow diagram for Chili

There is a huge difference between the representations (pseudocode and data-flow diagram) used in Figures 4.26 and 4.27. As we have already mentioned in Chapter 3, pseudocode describes flow of control (the sequence of actions in time), whereas the data-flow diagram describes the flow of data without concerning itself with actions. Pseudocode will show repetitions and selections since its emphasis is on actions; in a data-flow diagram, repetitions and selections of actions have no meaning.

Assertions can be made in the two algorithm representations. We have seen that assertions were comments that indicate the situation at any given point of an algorithm. In pseudocode, between any two statements, we can make an assertion about the state of the operation of the algorithm. Similarly, in a data-flow diagram, we can put an assertion with any line joining two boxes.

The assertion in the pseudocode of Figure 4.26 is shown in braces, while the assertion in Figure 4.28 is shown in a dotted box. Notice the difference between what is being asserted in each case:

- In the pseudocode, which is action oriented, the assertion is “finished simmering”, which makes a statement about the progress of the action in relation to time.
- In contrast, the assertion in the data-flow diagram, “soaked beans”, makes a statement about the state of the data.

This does not mean that we cannot use the data flow assertions in the pseudocode, as they give indications on the data that could be the results of an action. The opposite is also true for we could probably use most pseudocode assertions in data-flow diagram. However, in practice, the two representations use more specialized assertions that are action-oriented or data-oriented.

To understand an algorithm, you might find that sometimes the flow of actions is more useful, while at some other times, the flow of data is more important.

In this example, we have overemphasized the data to remind you that the role of data in an algorithm is of equal importance as the role of actions. Our next examples will be more action oriented, but we will define their data as well.

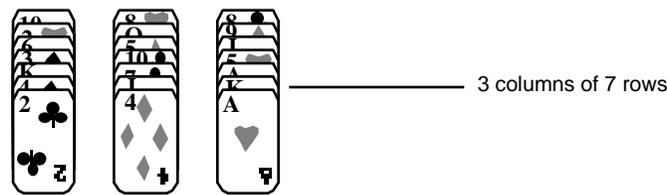
Card Trick Example

For more information on arrays, see Chapter 8.

Now, let's look at an algorithm that uses structured data. This example involves a card trick.

21 playing cards are placed face-up in 7 rows and 3 columns, as shown in Figure 4.29. Such a data structure is often called an *array*. The cards in any column are overlapped to maintain their order, and to make a column easy to pick up quickly.

Figure 4.29 Arrangement of cards for Card Trick



Once the cards have been arranged in this manner, perform the following steps:

1. Ask someone to select one of the 21 cards, without picking it up.
2. Without pointing at the card, have the person indicate the column that contains the selected card.
3. Pick up the columns in this order:
 - A non-indicated column,
 - The indicated column
 - The other non-indicated column
4. Deal out the cards in its familiar 7 rows by 3 column structure, **row by row!**
5. Repeat steps 2 through 4, twice.
6. Finally, count off 10 cards. The 11th card will be the card that the person selected.

This process is entirely described by Pseudocode 4.15. Try it. Notice that it does not take any knowledge or skill on your part, just the ability to follow directions.

Pseudocode 4.15 The Card Trick algorithm*Card Trick*

Arrange 21 cards face up row by row in 3 columns and 7 rows
 Have someone secretly choose a card and indicate the column of
 the chosen card

Repeat 2 times

Pick up the cards column by column, with the indicated column
 placed in the middle

Place the cards on the table again, row by row in
 3 columns and 7 rows

Have the person indicate the column containing the chosen card

Pick up the cards in "sandwich" order as before

Count off ten cards

Show the eleventh card: it is the chosen card

*End Card Trick***Binary Conversion Example**

Our next example (Pseudocode 4.16) is an algorithm to convert a positive integer number in base 10 to its equivalent in base 2. It uses one item of data, a number N , initially stored in a variable, whose value keeps changing during the computation. This variable is repeatedly divided by 2 and the remainder of the divisions is output until it reaches zero. The output values (always 0 or 1), when reversed, make up the required equivalent binary number.

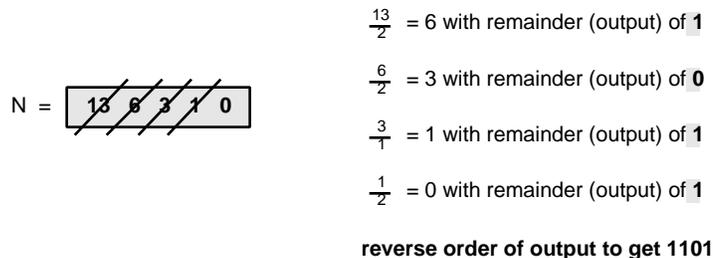
Pseudocode 4.16 Decimal to Binary algorithm*Set N to the number to convert**While $N > 0$*

Divide N by 2 yielding Quotient and Remainder

Output Remainder

Set N to Quotient

For example, let us convert the decimal number $N = 13$. Figure 4.30 contains a representation of the variable N (left) showing its successive values as the algorithm is performed (right).

Figure 4.30 Using the Decimal to Binary algorithm to convert $N=13$ 

As Figure 4.30 shows, the binary equivalent of 13 is 1101, or the value of the outputs taken in reverse order.

There are many methods to convert decimal numbers to binary. Notice that this method (Pseudocode 4.16) is very different from a previous method used in Chapter 3, which divided the number N by successively smaller powers of 2 (8, then 4, and finally 2).

Guessing Game Example

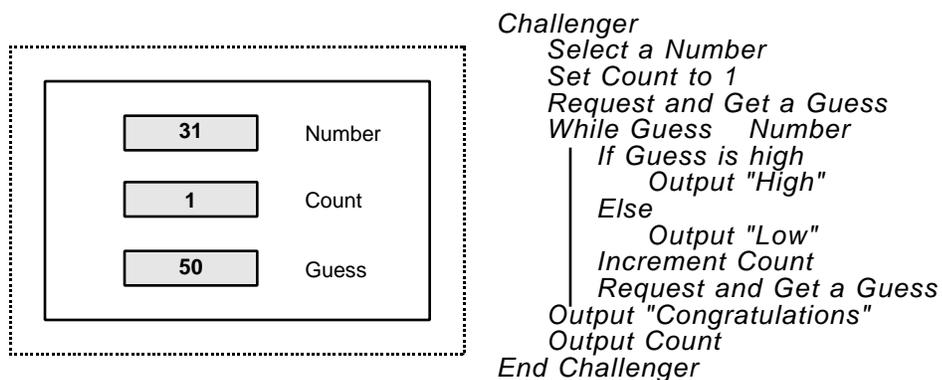
A guessing game involves two players, with either one being a computer. We have already seen one version of this game in Figure 4.7. Here we will develop a solution using a similar method, but with a different approach.

One player, the Challenger, selects a number between 0 and 100 and the other player, the Guesser, tries to guess it in the fewest number of tries. The only information given to the Guesser by the Challenger is whether a guess is higher or lower than the selected number.

Challenger, in Figure 4.31, is the algorithm that could be followed by the player who selects the number and tells the challenger if her guess is too high or too low. This algorithm accomplishes the following actions:

- It selects the mystery number,
- Rates the guess, and
- Keeps count of the number of guesses.

Figure 4.31 Challenger sample data and algorithm

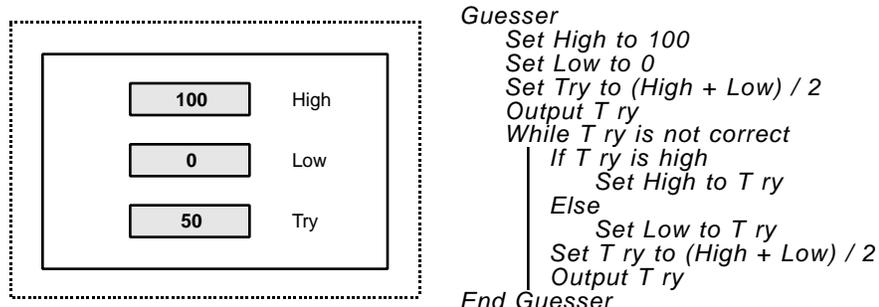


Note: This method of halving the correct range of values at each try is called **Bisection or Bracketing**. It will also be useful later in solving equations, and efficiently searching through data.

The algorithm *Guesser*, Figure 4.32, describes one systematic way of making guesses, and could be followed by the player trying to guess the Challenger's

number (between 0 and 100). It simply keeps track of the high and low values that were guessed, and chooses the middle value as the next guess. The middle value is the average of these two values. Depending on the outcome of the guess, one of the limits is changed (to this middle value).

Figure 4.32 Guesser sample data and algorithm



Notice that both algorithms have a similar structure consisting of a Selection within a Repetition. Both also involve only three numbers as data. The *Challenger* must know the following data:

- The *Number* selected,
- The *Count*, and
- The *Guess*.

The *Guesser* must know the following data:

- The *High* values, and
- The *Low* value.
- From these two limits, the *Guesser* computes the middle value which is taken to be the *Try*.

A typical run, trace, or “conversation” between the *Challenger* and *Guesser* follows. Suppose that the *Challenger*’s mystery number is 31, which is between 0 and 100. The trace in Figure 4.33 shows how the values of the two players’ data change as the game is played. Remember, each player is working according to an algorithm so that the two algorithms are performing simultaneously but with some kind of synchronism.

Figure 4.33 Trace of data between Challenger and Guesser

Challenger Trace	Guesser Trace
1. Number = 31	
2. Count = 1	
3.	High = 100
4.	Low = 0
5.	Try = (High + Low)/2 = 50
6. Guess = 50	
7. Guess Number	
8. "High"	
9. Count = 2	
10.	Try is not correct
11.	Try is high
12.	High = 50
13.	Try = (50 + 0)/2 = 25
14. Guess = 25	
15. Guess Number	
16. "Low"	
17. Count = 3	
18.	Try is not correct
19.	Try is low
20.	Low = 25
21.	Try = (50 + 25)/2 = 37
22. Guess = 37	
23. Guess Number	
24. "High"	
25. Count = 4	
26.	Try is not correct
27.	Try is high
28.	High = 37
29.	Try = (37 + 25)/2 = 31
30. Guess = 31	
31. "Congratulations"	
32. "Count = 4"	

In the above figure, the number was discovered in just 4 guesses. In general, it would have taken at most 7 guesses. This is because the right answer always lies between the values of High and Low, this is the "Range". The Range starts at 100 and is halved each time a guess is made. After seven tries, the range will be 1, which must be the right answer. Often, the right number will be tried before seven guesses. The algorithms will terminate earlier. In n tries, this bisection algorithm can select between 2^n numbers. So in ten tries, it can guess any number between 0 and 1023. Using this technique, if the *Challenger* selected a number between 0 and 1 000 000, the *Guesser* would take at most 20 tries to find it!

4.6 Review Top Ten Things to Remember

1. *Structured Programming* is a method of organizing algorithms in a simple form, using a small number of building blocks, with simple limited interconnections between them. It usually results in algorithms which are clear, orderly, understandable, efficient, modifiable and provable.
2. In Structured Programming, all algorithms are built from four standard building blocks or fundamental forms:
 - *Sequence*: actions are sequentially ordered and are performed one after the other.
 - *Selection*: a condition chooses which action will be performed.
 - *Repetition*: one or more actions will be repeatedly performed a number of times.
 - *Invocation*: defines a group of actions and names it. These actions are performed by invoking the group name.

Each of these forms has a single entry and a single exit.

3. All possible algorithms may be created using the Four Fundamental Forms, interconnected using one of the following two methods:
 - *Serial*. one form follows another.
 - *Nesting*. one form is contained within another.
4. Algorithms should be viewed as a collection of connected forms rather than a number of boxes for a one large form. Viewing algorithms in this way helps reduce their apparent complexity, thus keeping them simple. To create well-structured algorithms, use flowblock diagrams or pseudocode instead of flowcharts.
5. *Abstraction* is the process of reducing the complexity by hiding details. Actions are grouped and replaced by the group name, thus reducing the amount of detail, and giving a simpler, abstract view.
6. *Equivalent Algorithms* are identical in behavior, but may be structured differently. When two algorithms are equivalent, there is an opportunity to select one or the other, depending on your goals. For now, it is important to remember that there can be many different ways of constructing an algorithm to solve the same problem.
7. *Alternative Algorithms* are algorithms which act the same, but do not produce the same output for all possible cases. Alternative algorithms are similar in behavior and may be structured differently.
8. *Top-down Design* (whose motto is *divide and conquer*) is the process of developing an algorithm by starting at the most general view— bird's-eye view, and then proceeding stepwise by refining all parts until ending up at the details— worm's eye view.

9. Actions and Data are both considered in this chapter, but the emphasis is on the structure of actions (control flow). To keep the two in perspective, stepwise refinement can also be applied to data with break-out diagrams.
10. *Data-flow diagrams* emphasize data rather than actions and provide another approach to the design of solutions. They are a complement of the control flow diagrams (flowblocks and pseudocode).

4.7 Glossary

Abstraction: A method of conceptual simplification through the suppression of details that are irrelevant to the application of the abstraction.

Action: some small process applied to some data.

Array: A data organization by rows and columns.

Assertion: A statement that is either true or false.

Binary search: A search algorithm that repeatedly halves the area of search until it is reduced to one element, the object of the search or it is found that the item is not present in the data being searched.

Bisection search: Synonym for Binary search.

Data: Something given or measured.

Data Structure: Organization of data elements grouped for a given purpose.

Divide and Conquer: A method to break the complexity by considering small parts of a problem.

Four Fundamental Forms: The Sequence, Selection, Repetition and Invocation forms; each of which has a single entry point and a single exit point. These four forms together are sufficient to build all programs.

Hierarchical programming: Synonym for top-down design.

Invocation: Use of a sub-algorithm. One of the Four Fundamental Forms.

Method: Orderly procedure.

Modular design: Software design where the program is divided into separate sections or modules that are relatively independent so that any modifications to the program tend to be confined to only a few modules.

Nesting: The inclusion of a given form into another.

Record: An aggregate of data values of possibly different types arranged in a hierarchical manner.

Repetition: One or more actions repeatedly performed a number of times. One of the Four Fundamental Forms.

Selection: A condition or decision which chooses an action to be performed. One of the Four Fundamental Forms.

Sequence: One of the Four Fundamental Forms.

Structured programming: A technique for organizing programs by building them from a few basic blocks like the Four Fundamental Forms.

Structured Data: data made of several parts.

Top-down: A way of defining things starting at a general level and descending gradually into details.

4.8 Problems

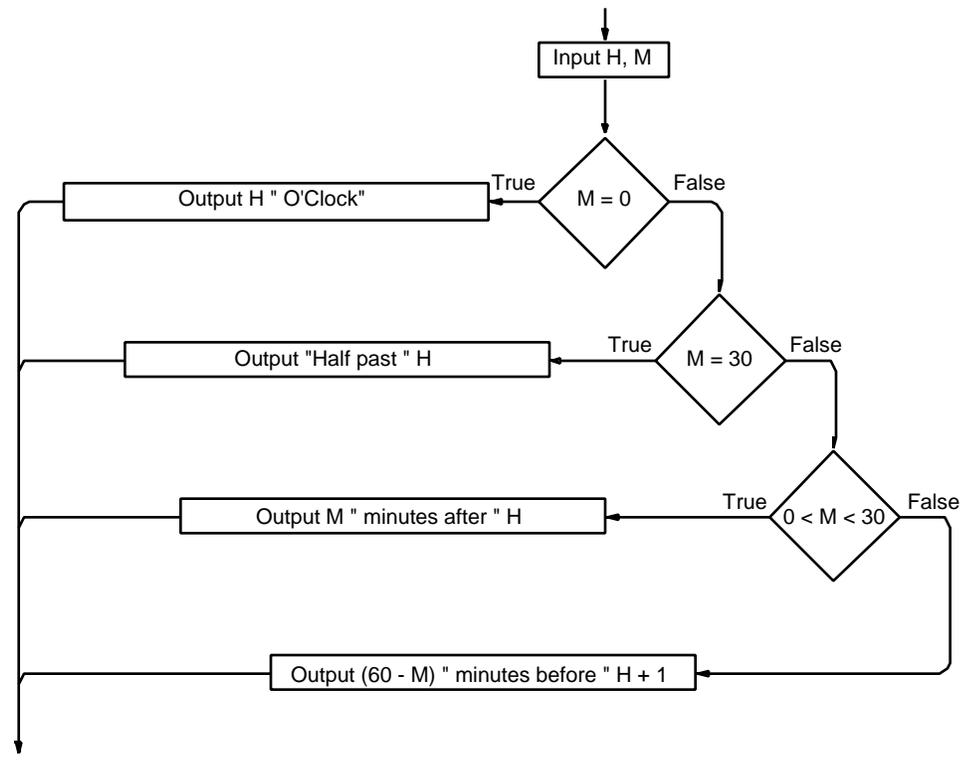
1. Change Change

The change-making program of Pseudocode 4.6 can be modified in many ways. One way is to anticipate problems in the input values and act accordingly. For example, the input cost may be negative, or it may be more than the amount tendered (one dollar in this case). Draw a flowblock diagram to take this situation into account.

2. Friendly Time-out

The following algorithm accepts input of a time in 12-hour digital form as hours H (ranging from 1 to 12) and minutes M (ranging from 0 to 59). It outputs the time in a friendly form as shown on the following flowblock diagram. However, there is an error in this algorithm; it does not work for some values. Test this algorithm and find the error.

Problem 2

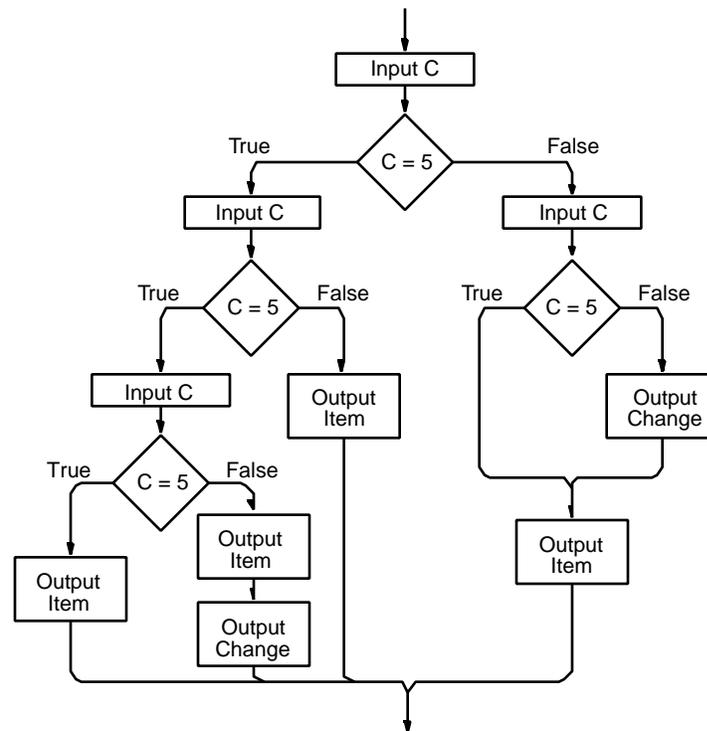


3. Dispense 15

The given flowchart describes a machine that accepts a sequence of input coins (nickels and dimes only) and outputs a 15-cent item and the appropriate change of 5 cents, or zero.

This algorithm consists of a “complex” nest of Selection forms. Draw this in the form of a flowblock diagram. Then create another simpler algorithm by using a Repetition form.

Problem 3



4. Chili Block

Convert the Chili recipe pseudocode, in Figure 4.27, into a flowblock diagram.

5. Convert

Convert the pseudocode for the *Triangle Classification 1* algorithm shown in Pseudocode 4.4 into a flowblock diagram.

6. More Compare

Extend the *Compare* algorithm of Figure 4.5 so that it indicates also whether the input values are constant, non increasing (for example 3, 2, 2), non decreasing, increasing then decreasing (like 1, 3, 2), decreasing then increasing.

7. Change Make Change

Modify the *Make Change* algorithm of Pseudocode 4.6:

- a) so that any amount T can be tendered (input).
- b) so that half-dollars can be given as change.
- c) so that two-dollar bills can be given as change.

8. More-Time

Create an algorithm to convert a given number of seconds S (say one million) into the equivalent number of days D , hours H , minutes M and seconds S . Represent this algorithm in two different ways, in pseudocode similar to *Make Change* and as a data-flow diagram.

9. Pints

Create an algorithm to convert a given number of pints into its equivalent number of gallons, quarts and pints. Note: 4 quarts = 1 gallon, 2 pints = 1 quart.

10. Romanums

Create an algorithm to convert Arabic numbers (ordinary positive integers) into their corresponding Roman numbers. Assume inputs less than 300 at first, then extend to 3000.

- a) if, at most, four consecutive occurrences of a single symbol are allowed, where 4 is `IIII`, 90 is `LXXXX` and 1984 is `MDCCCCLXXXIIII`.
- b) if, at most, three consecutive occurrences of a single symbol are allowed where 4 is `IV`, 90 is `XC` and 1984 is `MCMLXXXIV`

Note: $I = 1$, $V = 5$, $X = 10$, $L = 50$, $C = 100$, $D = 500$, $M = 1000$

Hint: See the Change Maker problem.

11. Dispense 7

Create an algorithm to dispense items costing 7 cents. The inputs are sequences of nickels or pennies only, with only one coin entered at a time. The outputs are an item and appropriate change.

12. General Problems on Top-Down

Design an algorithm, top-down, for four of the following. Show the break-out diagrams for three to four levels only.

1. Start a car.
2. Make a phone call.
3. Shave (face or leg).
4. Shampoo your hair.
5. Operate a combination lock.
6. Set an alarm clock.
7. Parallel park a car.
8. Balance checkbook.
9. Wash your clothes.
10. Shop for groceries.
11. Replace flashlight batteries.
12. Clean a fish.
13. Paint a wall.
14. Change a light bulb.
15. Make a bed.
16. Sharpen a pencil.
17. Tie a knot (bow).
18. Find a name in a telephone directory.
19. Fold a newspapers to create a hat.
20. Decide which four of the above to do.

13. Create Your Own

Create an algorithm describing something of your own choice. Develop it in a structured top-down manner.

Some examples are:

- operating a machine, camera, camp stove, projector, computer,
- going through a process, repairing something, developing film,
- playing or scoring a game, bowling, tic-tac-toe, hide and seek.

Dice Problems

Create algorithms (top-down and structured, of course) describing the following dice games. The problems involving archery and darts are really disguised dice problems. The dice problems could also be translated into similar games of skill rather than games of chance.

14. Rotation Dice

The game of Rotation is played with two dice and two people. The players each take a turn for a round. There are 11 rounds in a game, one for each of the combinations: 2, 3, 4, up to 12. During the first round, the goal is to throw a 2, during the second round it is to throw a 3 — and so on up to 12. Each time a player is successful, that number of points is added to that player's score; otherwise nothing is added. The winner is the player with the higher score after the 11 rounds.

15. Pig Dice

The game of Pig is played with one die and two people. The players each take a turn during a round and the rounds continue while all scores are less than 100. During a player's turn, the die is repeatedly thrown and the score accumulated until either a 1 comes up or the player chooses to stop. If the 1 comes up, the sum is lost; otherwise, it is added to the player's score. The winner is the one whose score is the highest at the end of the game.

16. Dice Climb

Create an algorithm describing the following game involving two players and one die. The players try to roll a 1, then a 2, a 3 and so on, up to 6, in that order, but not necessarily one number immediately following the other. First, they roll a die to determine who goes first (the highest). Then, they alternate turns, stopping as soon as one player (the winner) reaches the value 6. During each turn, a player rolls the die once and keeps rolling only if the desired numbers are obtained.

17. Dice 21

The game of Dice 21, or dice blackjack, is played with one die and any number of players. Each player in turn rolls the die until the accumulated sum is 16 or over. If the sum is over 21, the player "goes bust" and is eliminated from the game. The winner is the player (or players) whose score is closest to 21 after all players have had a turn.

18. Archery Scram

Create an algorithm describing the following game of skill involving two players shooting arrows at a target. The target consists of nested circles labeled with values from 1 to 6; value 1 is farthest out and value 6 is in the center, or Bull's eye.

The players shoot one arrow each. The one who comes closest to the center becomes the “stopper”, and the other becomes the “scorer”. They then take turns, each shooting three arrows per turn. The stopper aims to close a sector by hitting it. The scorer aims to get as many points as possible, before all sectors are closed. When all sectors are closed, the players swap roles. The winner is the player who scores the highest.

19. Darts

Create an algorithm describing the following simple dart game. It involves two people throwing darts at a circular target that is divided into different scoring areas. The first turn goes to the player who throws a dart closest to the center of the target. Each player throws three darts in a turn, starting with a given score (say 101) and attempts to reduce the score exactly to zero by subtracting points corresponding to where the darts land. If a player's score would take the player past zero, the score does not change. The first player to reach zero wins.

Chapter 5 Algorithm Behavior

In this chapter, we finally begin to consider programs, which are algorithms that are intended for computers. All the algorithm concepts we have seen so far apply to programs, including their properties, their various representations and the Four Fundamental Forms.

Chapter Overview

5.1	Preview.....	176
5.2	Programs.....	176
	Data for Programs.....	176
	Actions In Programs.....	179
5.3	Sequence Forms.....	182
	More Programs Using the Sequence Form.....	183
5.4	Selection Forms.....	185
	Simple Selection Forms.....	185
	Proofs of Equivalence for Simple Selection Forms.....	187
	Larger Selection Forms.....	189
	Proofs of Equivalence for Larger Selection Forms.....	192
	Nested Selections.....	194
	Logical Conditions.....	198
	Using Logical Conditions to Simplify Selections.....	201
5.5	Repetition Forms.....	203
	The Repeat-Until Form.....	203
	The Disadvantages of Using the Repeat-Until Form..	206
	The While Loop Form.....	209
	Getting Insights Using Traces and Invariants.....	212
5.6	Invocation Forms.....	219
	Seconds Example.....	221
	De-militarize Time Example.....	222
5.7	Improving Programs.....	224
	Nested Loops and Selections.....	224
	Using Invariants.....	229
5.8	Review Top Ten Things to Remember.....	234
5.9	Glossary.....	235
5.10	Problems.....	236

5.1 Preview

In order to concentrate on the algorithm actions, we will now view *data* uniformly as boxes containing the various kinds of values, like Integers, Real Numbers, Logical values, and so on. Actions, including arithmetic operations, input and output operations, and assignments, will be limited to manipulations of these simple data values.

The *Four Fundamental Forms* (Sequence, Selection, Repetition and Invocation) will be considered here in more detail, from a behavioral point of view. This means that we will concentrate on the dynamic aspect of an algorithm. In other words, what happens in the computer as it carries out or executes the algorithm. This dynamic view of an algorithm is in contrast to the static view, which corresponds to the algorithm on paper or on the computer screen.

As we study the behavior of an algorithm, we will be concerned with all possible paths that can be followed through its actions. We will show and prove the equivalence of behavior of certain algorithms.

Although very simple, the *Sequence form* is extremely important, and will be considered again briefly.

Selection Forms, which we have already introduced, will be reviewed in this chapter and the equivalence of several different selections will be studied and verified using symbolic logic truth tables.

The behavior of *Repetition Forms* is considerably more complex than other forms. In fact, Repetition Forms are the most complex form. We will describe them using two-dimensional traces, so as to yield insight into their dynamic behavior. Certain assertions about the state of a program are not changed by executing the body of a loop; these are known as *loop invariants* and are briefly introduced. Later in this chapter, we will use loop invariants to improve programs.

We can view *Invocation Forms* as data-flow diagrams or black boxes, in order to provide an easy and early introduction to an otherwise complex mechanism.

Here, we limit our consideration of *nested forms* to simple nests. In the subsequent chapters we will consider larger programs and their design.

5.2 Programs

Data for Programs

The algorithms considered until now have been quite general, involving data that are as diverse as people, dice, cards, resistors, pebbles, triangles, recipes and money.

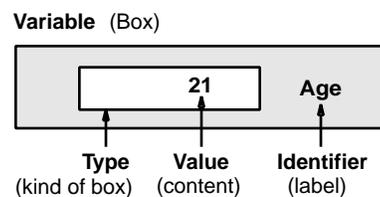
Programs are algorithms expressed in a form that can be carried out, or executed, by computers. Any program has a limited set of data to manipulate. Such program data can be represented by labeled boxes, called *variables*, that contain various types of data values. The contents of these boxes may be examined, copied, or replaced, by the program as it executes. The physical realization of the boxes may be electronic, magnetic, chemical or other, but this aspect is relatively unimportant for programming.

A variable has the following three components:

- an identifier,
- a data type, and
- a value

Figure 5.1 gives a graphical representation of these three components.

Figure 5.1 Components of a variable in a computer program



An *identifier* is a symbolic name that serves to label a programming variable. In actual programs, it is a good idea to use descriptive variable names such as *MinimumAge* or *CountOfSheep* in order to make the programs easier to understand. Here, in a few short sample programs, we sometimes use brief names such as *X*, *Y*, or *Z* as in algebra, for convenience.

A *data type* describes an item of data. It indicates a range or set of values that the data item may have, along with the operations that may be applied to these values. We could view the data type as the size or shape of the box containing a data value. In this chapter, we will consider three main data types: Integers, Real Numbers and Characters, some examples of which are shown in the following figures.

- When counting, the *values* are whole or integral numbers (such as 0, 1, 2001, -7, etc.); they are of *Integer* type. Integers could describe populations, dice throws, days in a month or the age of people. Actually, computers have a limit as to how large an integer can be. This limit may be as small as 32 768 or (2^{15}) , but is usually much larger. Integers will be the most common data type used in this chapter.
- When measuring, we obtain *values* that may be portions of a unit and usually have a decimal point; they are *Real Numbers*. For example, the radius of a circle could be a Real Number. The constant (3.1415926535...) is also a Real Number.

There are also other data types such as the *Logical* type (also called *Boolean* type) that comes from decision-making, which comprises only two values, *True* and *False*. Logical expressions are usually found as conditions in either the Selection or Repetition Forms.

Figure 5.5 Examples of other types

Logical	False	Tall
String	'Once upon a time ... lived happily ever after.'	Story

Other data types often used include the *String* type, consisting of a sequence of characters, as shown in Figure 5.5.

Figure 5.6 Examples of two aggregate data types

An array of 21 cards
3 columns of 7 rows

Person 1:

Name

 FirstName : John

 Initial : M

 LastName : Motil

Address

 Street : 1234 Main Street

 City : Northridge, CA

 Country : USA

 Code : 91330

Attributes

 IdNumber : 123-45-6789

 Birth

 Year : 2001

 Month : Feb

 Day : 29

Structured data may also consist of a collection or aggregate of individual values. For example, an array of cards from the card trick example and the employee record from a personnel file first seen in Chapter 4 are shown in Figure 5.6. Aggregate data types will be considered in Chapter 8.

Actions In Programs

The kind of operations or actions that can be performed on data depends on the type of the data, and this is why the operations are an implicit part of a data type. For example, the logical operators AND, OR, and NOT only apply to Logical type values—they cannot apply to Real Numbers, Integers, or Characters. On the other hand, Real Numbers and Integers may be added, subtracted, multiplied and divided, yielding results of the same type. Such arithmetic operations cannot act on variables of Logical or Character types—dividing two characters would be meaningless.

One of the most important operations in a program is the *assignment*, which operates on all data types. The assignment is the process of giving a value to a

variable, the act of putting some content into a box. The assignment operation is represented by the phrase “Set *Variable* to *Value*”, as in the following assignment:

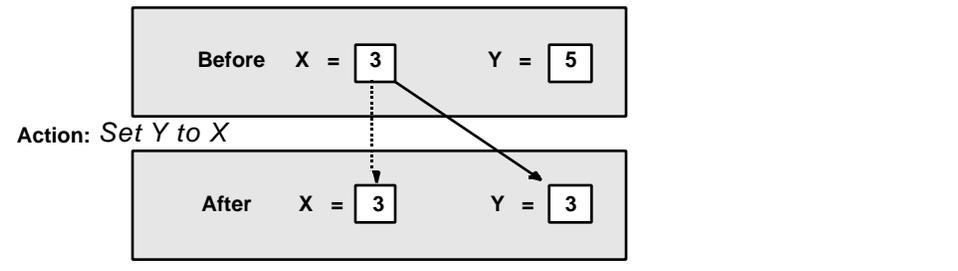
Set *X* to 3

or alternately,

$X \leftarrow 3$

This puts the value 3 into the variable box labeled *X*. If box *X* contained a value before this assignment, the old value is replaced by the new value 3. A more general assignment notation involves two variables as illustrated in Figure 5.7.

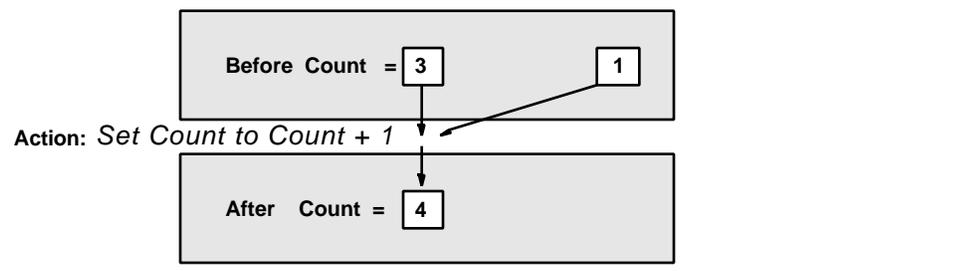
Figure 5.7 An Assignment example



The statement, “Set *Y* to *X*”, can be also read as “*Y* gets *X*” or “*Y* becomes *X*” or simply “*Y* is assigned *X*.” It specifies that the value is to be copied from box *X*, and put into box *Y*, destroying the previous value of *Y*. Assigning a value is a process of *copying*, not of *moving* this operation does not change the value contained in *X*.

Figure 5.7 shows a snapshot of the values of variables *X* and *Y*, both before and after the assignment. If *X* had not previously been given a value, then this assignment would be meaningless. If the value to be assigned is an expression (a mathematical formula), then that expression is evaluated, and its resulting value is assigned to the variable.

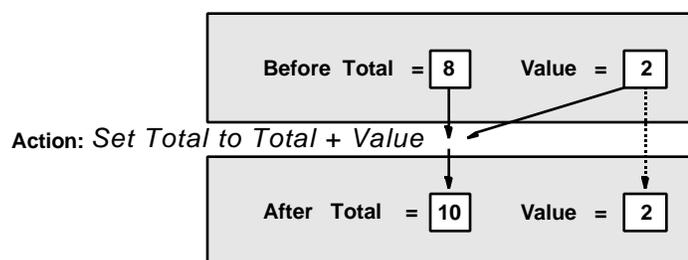
Figure 5.8 The Increment operation



The increment operation of Figure 5.8 shows how the value of variable *Count* is increased by a constant. This action is often used to increment a counter and it

can also be denoted by *Increment Count by 1*. It should be viewed as one action, and not as a sequence of smaller actions: get the value in *Count*, add one to it, put the result back into *Count*.

Figure 5.9 Accumulating a Total



The operation of Figure 5.9, Sum or Accumulate, is a convenient operation where *Value* is added to the value in *Total*, thus accumulating a sum; it is equivalent to *Increment Total by Value*.

Other useful actions that do not directly use the assignment operator are Input/Output operations. An Input operation allows an external value to be read into a variable, thus destroying any previous value of that variable. Similarly, the Output operation displays the value of a given variable, without modifying that variable.

Common operations between data of the same type include the operations of comparison. Usually such operations apply to numbers, but they also have meaning for characters. Comparisons include checking relations of equality, superiority, inferiority, etc. Characters are assumed to be in alphabetical (or lexicographic) order, so character 'a' is less than character 'b'.

Other actions on variables include special functions (such as finding the square root, or the trigonometric sine), which will be considered when they are encountered.

5.3 Sequence Forms

The simplest computer algorithms involve a sequence of actions executed one after the other, and the simplest Sequence programs involve actions that are all similar.

Let's take an example and suppose that the four variables *North*, *South*, *East* and *West* represent four traffic densities: the number of cars that travel *North*, *South*, *East* or *West* through some intersection in one minute.

Let's design an algorithm to compute the average traffic density, *Mean*, of the four values *North*, *South*, *East*, and *West*. This *Mean* value is a measure of the activity of the intersection that will be computed and printed out every minute to show how the traffic density through the intersection changes with time.

The value of *Mean* is obtained by summing the four values and dividing their sum by 4.

The *Average* algorithm, which computes this *Mean*, is shown at left of Figure 5.10 with a trace of its execution at the right. Suppose that the values of *North*, *South*, *East* and *West* were 20, 10, 40, and 30 respectively. First, variable *Sum* is set to zero. Then, the value in *North* is added to it, replacing the zero in *Sum* by 20. At the third step, the value of *South* is added into *Sum* bringing it to 30. Similarly in step 4, the value of *East* is accumulated bringing *Sum* to 70 and finally the value of *West* is added, ultimately yielding a *Sum* of 100. Finally, at step 6, this value of 100 is divided by 4 to yield the value of *Mean*, 25.

Figure 5.10 The Average algorithm

<i>Average</i>	Step	Sum	Mean	North	South	East	West
<i>Set Sum to 0</i>	1	0	?	20	10	40	30
<i>Set Sum to Sum + North</i>	2	20	?	20	10	40	30
<i>Set Sum to Sum + South</i>	3	30	?	20	10	40	30
<i>Set Sum to Sum + East</i>	4	70	?	20	10	40	30
<i>Set Sum to Sum + West</i>	5	100	?	20	10	40	30
<i>Set Mean to Sum / 4</i>	6	100	25	20	10	40	30
<i>End Average</i>							unchanged values

The *trace* of an algorithm is a series of “snapshots” of the data values as the various operations of the algorithm are executed. A trace of the execution of the *Average* algorithm is shown in Figure 5.10, with the data values shown to the right of each step of the algorithm. Since the values of *North*, *South*, *East* and *West* are not changed, they need not be repeated. The query mark, “?”, shown for the values of *Mean*, indicates that, at that point in the execution of the program, the value of *Mean* is unknown. Traces will be very useful for studying the behavior of algorithms, especially the more complex ones.

Figure 5.11 Data and actions for Average

Data Data Structure		Actions Algorithm Structure	
North	20	1	<i>Set Sum to 0</i>
South	10	2	<i>Set Sum to Sum + North</i>
East	40	3	<i>Set Sum to Sum + South</i>
West	30	4	<i>Set Sum to Sum + East</i>
Sum		5	<i>Set Sum to Sum + West</i>
Mean		6	<i>Set Mean to Sum / 4</i>

Figure 5.11 shows two kinds of structures. On the left of the figure, the data structure comprises six variables or boxes, and on the right side of the figure the algorithm structure consists of six actions. To average 50 numbers with this method would require 52 data boxes (one for *Sum*, one for *Mean*, and one for each data item), and also 52 actions (one for initializing *Sum*, one for each accumulation, and one for dividing). For large amounts of data, this sequential method is long and tedious. Later, we will find a better way to accomplish this using the Repetition form. For a few numbers though, this Sequence form of averaging is acceptable.

Although the Sequence Form in Figure 5.16 is correct, our simple traffic average could also be computed by using a single action:

Set Mean to (North + South + East + West)/4

The simple *Average* program of Figure 5.11 can be generalized by replacing the constant value of 4 by a variable *Count*. This variable can be set to the actual number of values to average and used in the division. We will see more on this in Chapter 6.

More Programs Using the Sequence Form

Pseudocode 5.1 shows a simple algorithm consisting of a series of assignments. It describes the calculation of the full selling price, including sales tax, for some quantity of a single item.

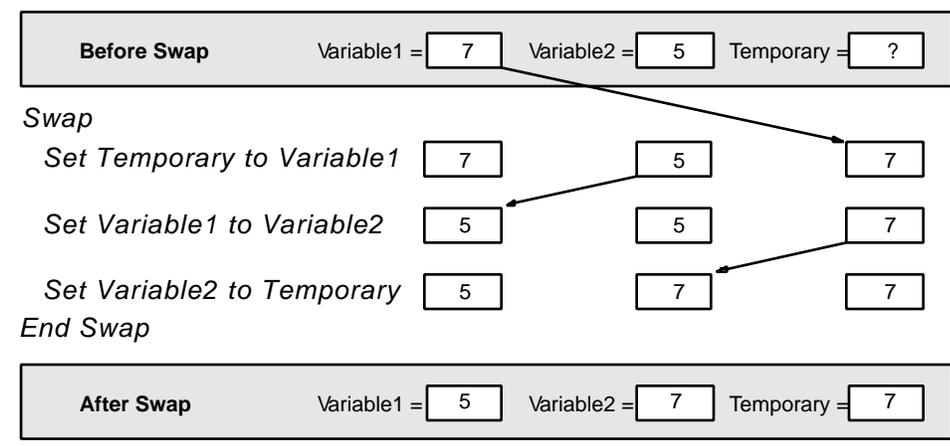
Pseudocode 5.1 A sales program with comments

<i>Sales</i>	
<i>Input ItemPrice</i>	Enter the price of an item
<i>Input Quantity</i>	Enter the number of items
<i>Set SalePrice to ItemPrice × Quantity</i>	Calculate selling price
<i>Set TaxRate to 0.065</i>	Set sales tax rate
<i>Set TaxPaid to TaxRate × SalePrice</i>	Calculate tax on selling price
<i>Set FullPrice to SalePrice + TaxPaid</i>	Calculate full price with tax
<i>Output FullPrice</i>	Write out the full price
<i>End Sales</i>	

Another example, shown on Figure 5.12, is an algorithm that interchanges the values of two variables. To do this, it uses a *Temporary* variable. The purpose of the *Temporary* variable is to save the value of *Variable 1*, which is replaced in step 2.

A trace of this *Swap* algorithm is shown at the right of the figure. As we have seen before, the trace consists of a series of snapshots of the values of the variables showing their change from “Before state” to the “After state”. This trace could be made more general by replacing the constant values 5 and 7 by markers like *x* and *y* that can represent any value.

Figure 5.12 Swap Algorithm



The *Swap* algorithm swaps the values of *Variable1* and *Variable2* using following three steps:

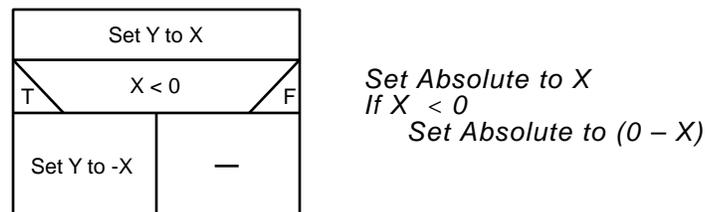
1. The value of *Variable1* (7) is put into *Temporary*.
2. The value of *Variable2* (5) is put into *Variable1* (replacing the 7 already in *Variable1*).
3. Finally the value of *Temporary* (7) is put into *Variable2* (replacing the 5 already in *Variable2*).

5.4 Selection Forms

Simple Selection Forms

Our first example to illustrate Selection Forms will be based on finding an absolute value. The absolute value of a number (positive or negative) is the positive value of the number without its sign. A flowblock diagram of an algorithm, which produces the absolute value of a variable *X* and puts it into the variable *Absolute*, is shown at the left of Figure 5.13. Its corresponding pseudocode is given at the right of the same figure.

Figure 5.13 The Absolute Value algorithm

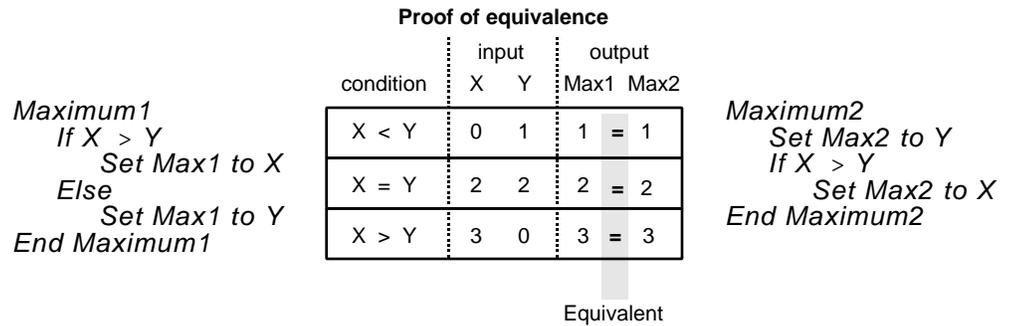


If the value is negative, it is subtracted from 0 and the positive result assigned to Absolute. If the value is positive, it is left untouched. Alternatively, the algorithm could have multiplied any negative value by -1, again resulting in a positive value. Since multiplication on a computer is more complex and slower than subtraction, we will use the algorithm that uses subtraction.

Note that instead of subtracting X from 0 for negative numbers, we could have simply written *Set Absolute to -X* where the minus operator is put in front of the X.

The next examples that illustrate Selection Forms show two *Maximum* algorithms at the left and right sides of Figure 5.14. They show two different ways of finding the maximum value of two variables, X and Y.

Figure 5.14 Maximum value



See Chapter 3, Figure 3.44 for a previous version of Maximum1.

- We have seen *Maximum1* before. It states that if X is greater than Y, then *Max1* (the maximum value) is assigned the value X. Otherwise, *Max1* is assigned Y.
- *Maximum2* starts by assigning Y to be the maximum value, *Max2*. Then, it tests to see if Y was, in fact, the greater of the two values. If this was not the case and X is the actual maximum, the algorithm assigns the value of X to *Max2*. This example illustrates a commonly used method or paradigm of computing: first make a guess, then correct it, if necessary.

To get a better understanding of what equivalence is, see Chapter 4, Section 4.3.

The center of Figure 5.14 demonstrates the equivalence of *Maximum1* and *Maximum2*. It consists of a table showing all possible conditions (the three combinations of values of X and Y shown in the leftmost column). For each of the three conditions, typical values for X and Y are chosen and the results, *Max1* and *Max2* of each algorithm, are evaluated. If these results are the same in all three possible cases, then the algorithms *Maximum1* and *Maximum2* are equivalent.

These *Maximum* examples illustrate that algorithms having different structures can still have the same behavior. In this example, one structure is not particularly preferable over the other.

Let's extend somewhat the *Maximum* algorithms we have just considered, into the three Big-Small algorithms, shown in Pseudocode 5.2. These will return two results: the maximum value *Big*, and the smallest value *Small*, of the two values *X* and *Y*.

Pseudocode 5.2 The Big-Small algorithms

<pre> BigSmall1 If X > Y Set Big to X Set Small to Y Else Set Big to Y Set Small to X End BigSmall1 </pre>	<pre> BigSmall2 Set Big to Y Set Small to X If Small > Big Set Big to X Set Small to Y End BigSmall2 </pre>	<pre> BigSmall3 Set Big to Y Set Small to X If Small > Big Swap Big, Small End BigSmall3 </pre>
---	--	--

- The algorithm *BigSmall1* is very similar to the algorithm *Maximum1*, but with the result including the minimum value as well as the maximum value.
- The algorithm *BigSmall2* is structured like the algorithm *Maximum2*, but extended to also find the minimum value.
- The algorithm *BigSmall3* begins much like *BigSmall2* by making an initial and arbitrary assignment to *Big* and *Small*. Then, it tests to see if the assignment was correct and, if not, the values of *Big* and *Small* are swapped. In order to swap the values, the final algorithm uses the sub-algorithm *Swap*, which we defined previously in Figure 5.12.

All of these algorithms are more or less equivalent, but we must make sure by proving that they are equivalent without assuming it! Let's look at the algorithm equivalence in more detail.

Proofs of Equivalence for Simple Selection Forms

Algorithms involving Selection Forms can be proven to be equivalent by testing them for all possible combinations of values, as we did for the *Maximum* algorithms in the previous section. This process is similar to the "truth table" proofs of symbolic logic. Let's look at two simple algorithms involving a Selection of individuals based on age and sex.

Figure 5.15 Two algorithms involving Selection Forms

		Conditions		Actions	
		Old	Male	Left	Right
<i>If old</i> <i> Aged</i> <i>Else</i> <i> Child</i> <i> If male</i> <i> Boy</i>	F	F	Child	=	Child
	F	T	Child,Boy	=	Child,Boy
	T	F	Aged	=	Aged
	T	T	Aged	=	Aged

		Conditions		Actions	
		Old	Male	Left	Right
<i>If old</i> <i> Aged</i> <i>Else</i> <i> Child</i> <i> If young male</i> <i> Boy</i>	F	F	Child	=	Child
	F	T	Child,Boy	=	Child,Boy
	T	F	Aged	=	Aged
	T	T	Aged	=	Aged

Equivalent

At the left and right of Figure 5.15 are two algorithms involving two Selections based on age and sex. There are three actions *Aged*, *Boy*, and *Child* corresponding to three categories with the same names. These algorithms could be used within a larger algorithm where *Aged*, *Boy*, and *Child* are actions that update counters. For example, every girl could cause a *Children* variable to be incremented, and every boy could cause both the *Boys* and *Children* variables to be incremented.

These two algorithms can be proved to be equivalent in behavior by creating a table of all possible combinations of conditions, and testing if the resulting actions are the same for both algorithms.

The truth table in the center of Figure 5.15 shows the table of combinations corresponding to the two algorithms. In all four cases the two algorithms behave identically, and are thus equivalent. Either one of these algorithms could be substituted for the other.

You may have noticed the difference between the two algorithms: the first one uses nested Selections while the second one uses a sequence of two Selections. Which algorithm should we prefer? The answer is not always clear. Sometimes sequentially connected Selections are preferred over nested Selections. Figure 5.16 shows another example where nested Selections are compared with sequentially connected Selections.

Figure 5.16 Nested versus sequentially connected Selections

		Conditions		Value	Actions	
		Age<12	Age>21	Age	Left	Right
<i>If Age < 12</i> <i> Increment Low</i> <i>Else</i> <i> If Age > 21</i> <i> Increment High</i>	F	F	18	none	none	
	F	T	25	High	High	
	T	F	10	Low	Low	
	T	T	??	???	???	

		Conditions		Value	Actions	
		Age<12	Age>21	Age	Left	Right
<i>If Age < 12</i> <i> Increment Low</i> <i>If Age > 21</i> <i> Increment High</i>	F	F	18	none	none	
	F	T	25	High	High	
	T	F	10	Low	Low	
	T	T	??	???	???	

Equivalent

The table at center of Figure 5.16 shows all four combinations of the possible conditions. The first combination includes all ages from 12 to 21 inclusive, while the second includes all ages greater than 21, and the third includes all ages less than 12. The last combination, however, (Age < 12 and Age > 21) can have no age that satisfies it; this logical combination is not physically possible and cannot happen, so it need not be considered. All of the other conceivable combinations are possible, and produce identical results, so the algorithms are equivalent.

The test values shown are 18, 25, and 10, and are sufficient to trace all three paths through these two algorithms (*Increment Low, Increment High*, do nothing) to prove their equivalence. Other test values could also be used. For example, the test value of 18 corresponds to the condition where (Age < 12) is False and (Age > 21) is also False, or to the equivalent condition where (Age < 12) and (Age > 21) is True. So, any age between 12 and 21 inclusive can replace the test age of 18. Similarly, the test value of 25 can be replaced by any value larger than 21 and the test value of 10 can be replaced by any value smaller than 12 (but not by a negative age!).

Figure 5.17 Another proof involving Selections

		Conditions		Actions	
		Cond1	Cond2	Left	Right
<i>If Cond1</i> <i>Action1</i> <i>Else</i> <i>If Cond2</i> <i>Action2</i>	F	F	none	none	<i>If Cond1</i> <i>Action1</i> <i>If Cond2</i> <i>Action2</i>
	F	T	Action1	Action1	
	T	F	Action2	Action2	
	T	T	Action1	Action1,Action2	

Not Equivalent

Figure 5.17 shows yet another set of examples that have a form similar to the examples in Figure 5.16. Nested Selections are compared with two Selections connected sequentially. The two pairs differ, however, in their conditions. The table of combinations, at the center of Figure 5.17, shows one combination (the last) where the behaviors differ. This one case is sufficient to prove that they are not equivalent.

All the above examples involved only two Selections, so at most four combinations needed to be tested. If an example involved three such conditions, then there would be eight combinations to test. We will consider such an example next.

Larger Selection Forms

When people vote in meetings, they usually vote for or against some proposal, and the votes are counted so that a majority may be determined. Majority voting can be applied to a collection of two-valued data with values 0 and 1,

True and False, or Yes and No. Let's define an algorithm to act on three binary variables A, B, and C (each having the value 0 or 1) and to output their majority value. A table describing the *Majority* algorithm is shown in Figure 5.18.

Figure 5.18 Specification of Majority Voting

I	A	B	C	M
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

We first encountered this algorithm in Chapter 3, Figure 3.16

In the above table, the combination of values (1, 0, 1) corresponds to the third line from the bottom of this table, which has an index number of 5. There are at least six possible algorithms that can produce the results specified by the table. These algorithms are shown below in Pseudocode 5.3 through 5.8, *Majority Method 1* through *Majority Method 6*.

Pseudocode 5.3 Algorithm Majority Method 1

```

Majority Method 1
  If A = 0
    If B = 0
      If C = 0
        Set Majority to 0           {Combination 0}
      Else
        Set Majority to 0           {Combination 1}
    Else
      If C = 0
        Set Majority to 0           {Combination 2}
      Else
        Set Majority to 1           {Combination 3}
  Else
    If B = 0
      If C = 0
        Set Majority to 0           {Combination 4}
      Else
        Set Majority to 1           {Combination 5}
    Else
      If C = 0
        Set Majority to 1           {Combination 6}
      Else
        Set Majority to 1           {Combination 7}
  End Majority Method 1
    
```

Majority Method 1 is based on a systematic and exhaustive enumeration of all possible values. It involves a total of seven Selections (count them) to evaluate the 8 possible combinations. The numbers 0 to 7 in the assertions in braces correspond to the index in the table of combinations in Figure 5.18. You might have noticed that this index corresponds to the binary number represented by the combination (for example, the 1, 1, 0 combination is the binary representation of number 6).

Pseudocode 5.4 Algorithm Majority Method 2

```
Majority Method 2
  Set Sum to 0
  If A = 1
    Set Sum to Sum + 1
  If B = 1
    Set Sum to Sum + 1
  If C = 1
    Set Sum to Sum + 1
  If Sum >= 2
    Set Majority to 1
  Else
    Set Majority to 0
End Majority Method 2
```

This second version simply accumulates in variable *Sum* the number of times the value of 1 appears in the three cases. Then, if the value of *Sum* is greater than or equal to 2, the value of *Majority* is 1, otherwise it is 0.

Pseudocode 5.5 Algorithm Majority Method 3

```
Majority Method 3
  If (A + B + C) >= 2
    Set Majority to 1
  Else
    Set Majority to 0
End Majority Method 3
```

This third algorithm is shorter and simpler. It has a single but complex Selection, which asks if the sum ($A+B+C$) is greater than or equal to 2 and if so, the *Majority* is set to 1, otherwise it is set to 0.

Pseudocode 5.6 Algorithm Majority Method 4

```

Majority Method 4
  If A < B
    If A < C
      Set Majority to 1      {Combination 3}
    Else
      Set Majority to 0      (Combination 2)
  Else
    If B < C
      Set Majority to A      {Combination 1, 5}
    Else
      Set Majority to B      {Combination 0, 4, 6, 7}
  End Majority Method 4

```

Note that the value assigned to Majority is dependent on the values of A or B.

Most likely path taken: half of the combinations take it.

The fourth method (Pseudocode 5.6) involves three Selections. The numbers in the assertions show the indices of the 8 combinations indicating the paths taken. Take note that all paths are not equally likely; half of the combinations take the path that includes the last pseudocode statement.

Pseudocode 5.7 Algorithm Majority Method 5

```

Majority Method 5
  If A = B
    Set Majority to A
  Else
    If B = C
      Set Majority to B
    Else
      Set Majority to C      {C = A}
  End Majority Method 5

```

The fifth method involves nested Selections, all testing for the equality of variables. Essentially, if any two of the three values are equal, then the value of *Majority* is that value.

Pseudocode 5.8 Algorithm Majority Method 6

```

Majority Method 6
  If A = B
    Set Majority to A
  Else
    Set Majority to C
  End Majority Method 6

```

This last algorithm, *Majority Method 6*, is the simplest. It first asks if $A=B$, and if so, the value of *Majority* is A (or B). Otherwise A and B “cancel one another” leaving C to determine the *Majority*.

It is interesting to notice the different conditions in each of the six methods.

- Some methods (1 and 2) compare a variable to a constant.

- One method (4) compares two variables for size.
- Some methods (5 and 6) compare two variables for equality.
- One method (3) uses complex conditions to make the comparison.

Finally, let us compare *Majority Method 3* and *Majority Method 6*, which are simple and seem similar. *Majority Method 3* depends on the fact that the values are numeric and can be added. *Majority Method 6* makes no such assumption (the values to be compared can be characters such as T and F) which makes this method more general.

Extending the majority to five variables would be interesting. Some of the six methods extend more easily than others. Try to extend them all.

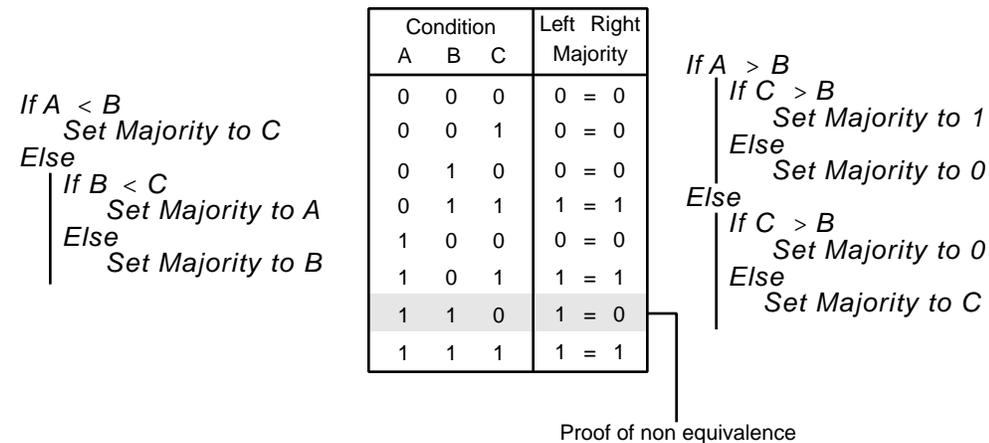
We will continue with a couple of majority algorithms and take a look at proofs, or verification of the correctness of algorithms.

Proofs of Equivalence for Larger Selection Forms

The more variables algorithms have, the more data combinations are possible. For example, consider the two algorithms shown below, which claim to find the *Majority*. The three variables *A*, *B*, and *C* each have one of two values, 0 or 1. We have already called such variables “binary variables”. Three binary variables yield a total of 2^3 or 8 possible combinations of values, as enumerated in the table shown in Figure 5.19.

As we have already noted, the values of *A*, *B* and *C* could represent the digits of a binary number so that each combination corresponds to a unique number. Thus, the combinations may be listed systematically as binary numbers zero through seven.

Figure 5.19 A right and a wrong way of computing Majority



The table in Figure 5.19 contains a column labeled Left, which shows the values of *Majority* evaluated by the algorithm on the left. Similarly, a column labeled Right shows the values assigned by the algorithm at right. If these two “output” columns had the same value for all combinations, then the two algorithms would be equivalent.

However, they behave differently in one case (shown shaded), and that is sufficient enough to disprove the equivalence of these two algorithms. In fact, only one of the two algorithms of that figure (the one on the left) is actually a *Majority* algorithm. Notice that it is a modification of *Majority Method 4* from Pseudocode 5.6.

The algorithm in Figure 5.19 involves three binary variables and requires 2^3 rows in the table. An algorithm which involves n binary variables, requires 2^n rows. So for 10 binary variables, we would need 1024 rows and for 20 variables, over a million rows. When the number of variables becomes large, this exhaustive method is exhausting!

Figure 5.20 Two ways of finding the maximum of three values

	Condition (X, Y, Z)		Left Max	Right Max
	<pre> If X < Y If Y < Z Set Max to Z Else Set Max to Y Else If X < Z Set Max to Z Else Set Max to X </pre>	<pre> X < Y < Z X < Z < Y Y < X < Z Y < Z < X Z < X < Y Z < Y < X </pre>	<pre> Z = Z Y = Y Z = Z X = X Y = Y X = X </pre>	<pre> Set Max to X If Max < Y Set Max to Y If Max < Z Set Max to Z </pre>

Figure 5.20 shows two algorithms for computing the *Maximum* of three values X , Y , and Z . Let's check if the two are equivalent. If the values of X , Y , and Z are limited to binary values, then eight combinations would be required for the proof of equivalence. However, if X , Y , and Z are integers, it seems as though we would need to test for all possible combinations of three integer values—an infinite number—to show equivalence. Luckily, since we are only interested in relative values, we can show equivalence by taking all possible orderings for X , Y , and Z . Assuming that the three values are all different, there are only six possible conditions, as shown in the table in Figure 5.20. Evaluating both algorithms for all of these combinations yields an identical behavior.

Another, less abstract, proof of equivalence can also be given using six sets of test values. We have chosen 1, 2, 3 as test values. Any other set of test values would have been just as good.

For example, the combination 10, 20, 300 gives the same results as 1, 2, 3, as does any other increasing sequence of three values. A proof of equivalence is given in the following table:

X	Y	Z	Condition (X, Y, Z)	Lift Max	Right Max
1	2	3	$X < Y < Z$	3	= 3
1	3	2	$X < Z < Y$	3	= 3
2	1	3	$Y < X < Z$	3	= 3
3	1	2	$Y < Z < X$	3	= 3
2	3	1	$Z < X < Y$	3	= 3
3	2	1	$Z < Y < X$	3	= 3

If the problem specification allowed for some of the values to be equal, then more sets of combinations should be tested (such as 2, 1, 1 or 1, 2, 1, and so on). Can you guess how many?

Nested Selections

In the previous sections, we saw a few examples of nested Selections. Let's look closer to more examples of nested Selections in order to understand them better and to see if it's possible to simplify them.

The following *Grades* algorithm, which assigns grades to students based on their percentage scores, might not be the best way to assign grades, but is a good example to illustrate many nesting concepts. The specification for the *Grades* algorithm is as follows:

Grades:
A score of 90 to 100 gets a grade of 'A'
A score of 80 to 89 gets a grade of 'B'
A score of 60 to 79 gets a grade of 'C'
A score of 50 to 59 gets a grade of 'D'
A score of less than 50 gets a grade of 'F'

The following algorithms, (Pseudocode 5.9 to 5.12) solve this problem in four different manners (*Grades 1* to *Grades 4*).

The first method, *Grades 1*, tests the largest percentage range first, and tests each range in descending order until the proper range is found. Once the proper range is found, the corresponding grade is output.

Pseudocode 5.9 The Grades 1 algorithm

```
Grades 1
  Input Percent
  If Percent >= 90
    Output "A"
  Else
    If Percent >= 80
      Output "B"
    Else
      If Percent >= 60
        Output "C"
      Else
        If Percent >= 50
          Output "D"
        Else
          Output "F"
  End Grades 1
```

The second method (*Grades 2* in Pseudocode 5.10) is similar to method 1, but it starts from the smallest percentage range and tests each range in ascending order.

Pseudocode 5.10 The Grades 2 algorithm

```
Grades 2
  Input Percent
  If Percent < 50
    Output "F"
  Else
    If Percent < 60
      Output "D"
    Else
      If Percent < 80
        Output "C"
      Else
        If Percent < 90
          Output "B"
        Else
          Output "A"
  End Grades 2
```

The third method of assigning grades (*Grades 3* in Pseudocode 5.11) compares *TestValue* to zero at each stage. This way of doing things might be useful for machine level programming since machines can compare values to zero very easily.

Pseudocode 5.11 The Grades 3 algorithm

```

Grades 3
  Input Percent
  Set TestValue to Percent - 50
  If TestValue < 0
    Output "F"
  Else
    Set TestValue to TestValue - 10
    If TestValue < 0
      Output "D"
    Else
      Set TestValue to TestValue - 20
      If TestValue < 0
        Output "C"
      Else
        Set TestValue to TestValue - 30
        If TestValue < 0
          Output "B"
        Else
          Output "A"
  End Grades 3

```

Try out some values for this Selection. Are the outputs correct?

Unlike the previous three methods, which involve nested choices, the fourth method (*Grades 4* in Pseudocode 5.12) involves no nesting but a series of choices. This method is often simpler to program in older programming languages which have a limited IF structure. Notice that this method requires the assignment of character values, while none of the other methods uses assignments.

Pseudocode 5.12 The Grades 4 algorithm

```

Grades 4
  Input Percent
  Set Grade to "A"
  If Percent < 90
    Set Grade to "B"
  If Percent < 80
    Set Grade to "C"
  If Percent < 60
    Set Grade to "D"
  If Percent < 50
    Set Grade to "F"
  Output Grade
End Grades 4

```

The four methods given here are not all the possible methods for the Grades algorithms; at least two other different structures are also possible.

Whenever we define an algorithm, we should test it. This is in fact step 4 and part of step 5 of our problem solving method. Testing algorithms is extremely important as errors can creep up even in simple algorithms like our Grades algorithms, and in fact, there is an error in one of them.

Our testing strategy is always to include a test value in each of the possible ranges as well as "limit" values. Here, the ranges are defined by the five

For more details on the problem solving method, see Chapter 2, Section 2.2.

grades and suitable test values would be 95, 85, 70, 50, and 25. Examples of limit values are 89, 90, and 91.

A good testing strategy will also include extreme or out of range values, like a negative grade, or a grade that is greater than 100. Using these values, you will find that the algorithm *Grades 3* (Pseudocode 5.11) contains an error for grades higher than B! Constant 30 in the last subtraction should be 10!

These Grades algorithms could be improved so that the grade boundaries (50, 60, 80, 90) may be more easily altered. The constant values representing these boundaries could be replaced by other symbolic values (LowD, LowC, LowB, and LowA), to which are assigned the constant values at the beginning of the algorithm. Then, if the grading “boundaries” change, they could be easily modified in one place, at the beginning of the algorithm, rather than at various places throughout the algorithm.

Note: Pseudocode 5.12 contains an error. It does not calculate grades higher than B. The constant 30, in the last subtraction, should be changed to 10. By using a good testing strategy, we can discover errors like these in our own algorithms.

The *efficiency* of algorithms describes the speed of their operation. This speed, or time to execute an algorithm, could be determined by counting the number of operations (such as Selections). For example, the last algorithm outlined in *Grades 4* always requires four Selections; every path through this algorithm must go through every Selection. The other three algorithms require four Selections only in the worst case. In general they involve fewer Selections. Therefore, the fourth algorithm is less efficient than the others.

An analysis of efficiency also depends on the input values provided. For example, given an input value of 95, *Grades 1* requires only one Selection, whereas *Grades 2* requires four. Such an analysis should not be done for just one value, but for a collection of values. For example, if the grade distribution is high, then *Grades 1* would be faster on the average. On the other hand, if most grades were around 65, a better algorithm for such a distribution would test first for C grades.

For more information on Logical conditions, see Logical Data-Flow Diagrams in Section 3.4. De Morgan's First and Second Laws are also mentioned there.

Logical Conditions

The conditions we have used in our Selections are usually called *Logical conditions* because they can only be true or false. The conditions we have used so far were simple, but we can define more complex conditions by using the three main connectives AND, OR and NOT, which we already encountered in Chapter 3, Figures 3.27 and 3.29. Their behavior is specified by the truth tables of Figure 5.21.

Figure 5.21 Truth tables

Conjunction AND			Disjunction OR			Negation NOT	
P	Q	C	P	Q	D	P	N
F	F	F	F	F	F	F	T
F	T	F	F	T	T	T	F
T	F	F	T	F	T		
T	T	T	T	T	T		

Conjunction is a logical operation between two logical values *Condition1* and *Condition2*, denoted as *Condition1 AND Condition2*. For example, the following condition is true when I, J, and K contain values that are in increasing order:

$$(I < J) \text{ AND } (J < K)$$

Using a compound condition with the AND operator can considerably simplify the structure of an algorithm since it reduces the number of Selections used. Pseudocode 5.13 illustrates such an example by showing two different pieces of pseudocode that are equivalent.

Pseudocode 5.13 Nested Selections and the AND connective

<pre> If Condition1 If Condition2 Set Cond to True Else Set Cond to False Else Set Cond to False </pre>	<pre> If Condition1 AND Condition2 Set Cond to True Else Set Cond to False </pre>
---	---

In higher level programming languages, this kind of tradeoff is usually possible. However, in low level programming languages such as assembly languages, conditions often cannot be compounded and so, only simple conditions can be used. The equivalence of two logical formulas can be proved by drawing the corresponding truth tables.

Logical operations OR and NOT, respectively called Disjunction and Negation, are also defined by the truth tables of Figure 5.21. Using these three logical operations, we can write any logical formula or expression. As an example, let's look again at De Morgan's rules that we have already introduced in Chapter 3.

Figure 5.22 DeMorgan's rules

Rule 1	$\text{NOT}(\text{P OR Q}) = \text{NOT}(\text{P}) \text{ AND } \text{NOT}(\text{Q})$
Rule 2	$\text{NOT}(\text{P AND Q}) = \text{NOT}(\text{P}) \text{ OR } \text{NOT}(\text{Q})$

To verify the first rule, consider this statement:

It is not true that either the pig is blue or the horse is green.

It is equivalent to:

The pig is not blue and the horse is not green.

Similarly, the second rule can be verified by considering this statement:

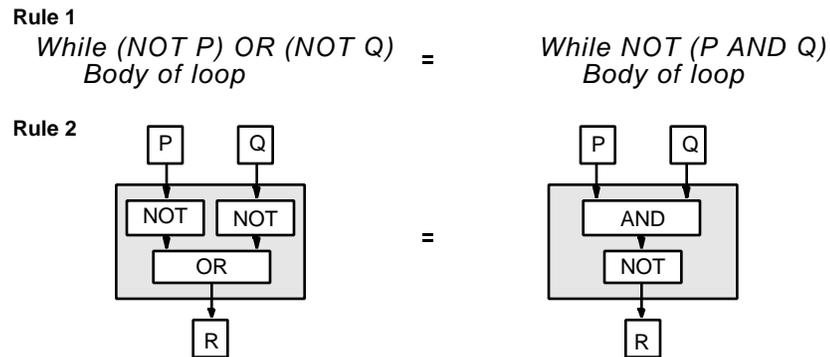
It is not true that both the pig is blue and the horse is green.

It is equivalent to:

Either the pig is not blue or the horse is not green.

This equivalence can be used in various ways, for example a complex condition controlling a loop could be replaced by a simpler one as shown in the pseudocode at the top of Figure 5.23. Two NOTs and one AND are replaced by one NOT and one OR. The bottom part of the same figure shows the second De Morgan's rule in the form of data-flow diagrams

Figure 5.23 DeMorgan's rules



DeMorgan's Rules can also be used to negate a formula. To negate the AND of two variables, you simply OR the two negated variables. You can see this at the bottom of Figure 5.23. For example, the condition for remaining in a loop is the opposite of that for leaving a loop. In the simple Dice game of Chapter 3 (Figure 3.41), this was the condition for looping (and throwing again):

(Total = 7) AND (Total = Point)

The assertion after leaving the loop was just the opposite:

(Total ≠ 7) OR (Total ≠ Point)

This results directly from DeMorgan's second rule as the condition for leaving the loop is the opposite (negation) of the condition for looping:

$$\text{NOT} ((\text{Total } 7) \text{ AND } (\text{Total Point})) = (\text{Total}=7) \text{ OR } (\text{Total}=\text{Point})$$

The proof of DeMorgan's first rule was shown in Chapter 3 without any detail (see Figure 3.29). To refresh your memory, Figure 5.24 shows the detailed proof of the first rule.

Figure 5.24 Truth table proof of DeMorgan's first rule

	1	2	3	4	5
P Q	NOT P	NOT Q	NOT P AND NOT Q	P OR Q	NOT (P OR Q)
F F	T	T	T	F	T
F T	T	F	F	T	F
T F	F	T	F	T	F
T T	F	F	F	T	F

Equivalent

In this figure, the four rows correspond to the combinations of the possible truth values of P and Q. The demonstration proceeds column by column (in the numbered order). Finally columns 3 and 5, corresponding to the left and right sides of DeMorgan's first rule, are seen to be identical.

Logical conditions which involve n logical variables can be proven equivalent by truth tables which contain 2^n rows. For example, let us consider the following logical distributive law:

$$A \text{ AND } (B \text{ OR } C) = (A \text{ AND } B) \text{ OR } (A \text{ AND } C)$$

As these formulas involve three logical variables, there are 2^3 or 8 possible combinations. The left side and right side of the expression are evaluated, and found identical in all cases as shown in Figure 5.25.

Figure 5.25 Truth table for distributive law

3 logical variables

	A	B	C	Left	Right
	F	F	F	F	= F
	F	F	T	F	= F
	F	T	F	F	= F
	F	T	T	F	= F
	T	F	F	F	= F
	T	F	T	T	= T
	T	T	F	T	= T
	T	T	T	T	= T

8 possible combinations

Notice that this distributive law for logical operations is equivalent to the distributive law of algebra:

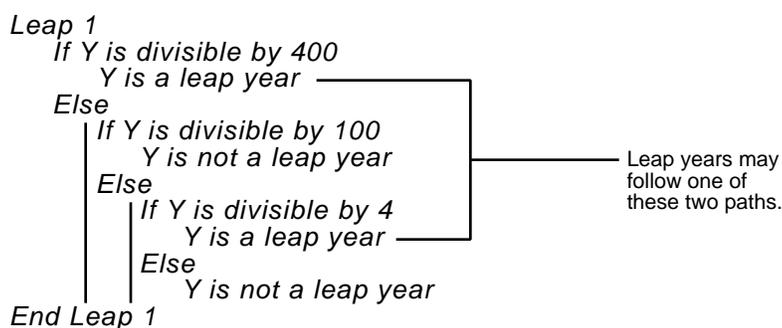
$$a(b + c) = ab + ac$$

Using Logical Conditions to Simplify Selections

We have seen in the last sections that algorithms could be simplified by making their conditions more complex, as already illustrated in Pseudocode 5.13. This actually means that we can reduce the number of Selections in an algorithm provided we compound the various conditions. This trade-off can be further illustrated by the following *Leap* algorithms.

The first *Leap* algorithm (to determine whether a given year Y is a leap year) was introduced in Chapter 3 (Figure 3.2), and three different pseudocode versions were given in Chapter 4 (Pseudocode 4.1 to 4.3). We repeat them here.

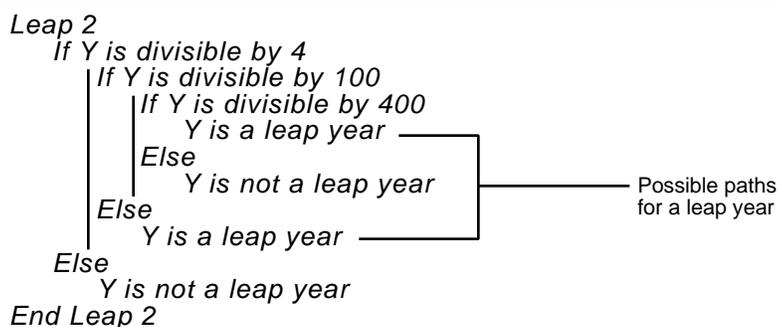
Pseudocode 5.14 Algorithm Leap 1 from Chapter 4



There are two possible paths through this algorithm leading to a leap year: when Y is divisible by 400, or when Y is not divisible by 100 and is divisible by 4. Putting this symbolically leads to one larger condition we will call $C1$ where the condition “ $Y \text{ D } 4$ ” means that Y is divisible by 4.

$$C1 = (Y \text{ D } 400) \text{ OR } (\text{NOT}(Y \text{ D } 100) \text{ AND } (Y \text{ D } 4))$$

Pseudocode 5.15 Algorithm Leap 2 from Chapter 4



Again, the logic expressed by the pseudocode of *Leap 2* can be reduced to a single logical expression $C2$:

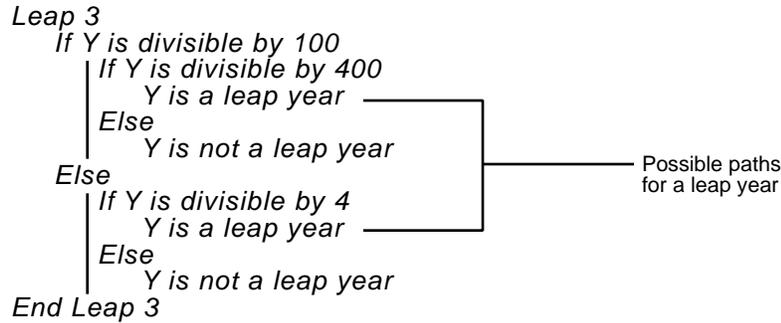
$$C2 = (Y \ D \ 4) \ \text{AND} \ ((Y \ D \ 100) \ \text{AND} \ (Y \ D \ 400)) \ \text{OR} \\ \text{NOT} \ (Y \ D \ 100))$$

If we note that $(Y \ D \ 100) \ \text{AND} \ (Y \ D \ 400)$ can be reduced to $(Y \ D \ 400)$ because a value divisible by 400 is certainly divisible by 100, we can reduce C2 to the following:

$$C2 = (Y \ D \ 4) \ \text{AND} \ ((Y \ D \ 400) \ \text{OR} \ \text{NOT} \ (Y \ D \ 100))$$

Convince yourself that this is correct.

Pseudocode 5.16 Algorithm Leap 3 from Chapter 4;



Here in *Leap 3*, the equivalent logical expression is C3:

$$C3 = ((Y \ D \ 100) \ \text{AND} \ (Y \ D \ 400)) \ \text{OR} \\ (\text{NOT} \ (Y \ D \ 100) \ \text{AND} \ (Y \ D \ 4))$$

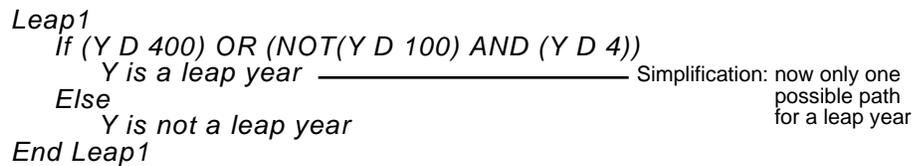
It can be also simplified:

$$C3 = (Y \ D \ 400) \ \text{OR} \ (\text{NOT} \ (Y \ D \ 100) \ \text{AND} \ (Y \ D \ 4))$$

You should verify C3 for yourself (looks like C1?).

All three of these Leap algorithms involve three separate Selections. If the conditions were allowed to be more complex, like C1, C2 or C3, the resulting algorithm could be as simple as the following for *Leap 1*:

Pseudocode 5.17 New Algorithm, Leap 4



In fact, all of the *Leap* algorithms can be similarly simplified.

5.5 Repetition Forms

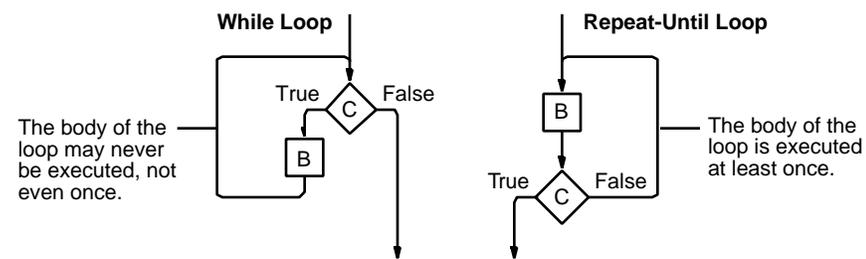
The Repeat-Until Form

Since computers are extremely good at repetitive tasks, algorithms based on Repetition Forms are important in computing. Repetition Forms in algorithms are very often called “loops”, as we loop through the algorithm while executing it. Note that a loop is more visible on a flowchart because an arrow has to loop back to an upper part of the diagram (look back at Section 3.5 of Chapter 3 as well as at Figure 4.3 of Chapter 4)

Although there exist several forms for the loops, thus far we have used only the While form of the loop. In this form, a test of the condition is made and, if the condition is met, the body of the loop is executed. This process is repeated until the condition is found to be False when tested.

An alternative loop form, the Repeat-Until form first executes the body of the loop and then continues repeating—iterating—until the condition on which it is based is found to be True when tested. Figure 5.26 shows a comparison between the flowcharts of these two looping forms.

Figure 5.26 Comparison of flowcharts for While and Repeat-Until loop forms



Notice that the body of the Repeat-Until loop is always executed at least once, and that the same is not true for the While loop. Another difference between the two forms is that the condition for termination of the Repeat-Until loop is the negation of the condition for termination of the While loop. The reason for that difference is simple: the While loop condition is a condition for looping, and the Repeat-Until condition is a condition for terminating the loop

To show the closeness of these two forms, we could build a Repeat-Until loop from the body of the loop and a While loop connected sequentially, as in shown in Pseudocode 5.18.

Pseudocode 5.18 Equivalence of Repetition Forms

Repeat
 B
 Until Cond is exactly equivalent to B
 While NOT Cond
 B

The Repeat-Until loop always performs the actions in the body at least once, this makes it well adapted to specific situations. However, the While form makes it possible for the body of the loop not to be executed if need be and this added control is usually preferred.

The behavior of a While loop is dynamic or moving as the actions of the loop body are executed repeatedly, whereas its structure is static. We will illustrate this difference in Figure 5.27. The pseudocode demonstrates the static form of an algorithm to calculate the remainder *Rem* of the integer division of a numerator *Num* by a divisor *Den*. To the right of the pseudocode, the behavior of the While loop is shown as a series of actions forming a trace. The numbers 14 and 4 were arbitrarily assigned to *Num* and *Den* to demonstrate this trace.

The loop body *Set Rem to Rem - Den* is repeated while the condition *Rem Den* is True and the repetition ends when this condition becomes False. This algorithm is generally known as the modulus operation.

Figure 5.27 Structure and trace of the Modulus algorithm

Structure	Trace			
<i>Modulus</i> <i>Input Num</i> <i>Input Den</i> <i>Set Rem to Num</i>	Num = 14 Div = 4 Rem = 14			
<i>While Rem Den</i>	14 4 is True	10 4 is True	6 4 is True	2 4 is False
<i>Set Rem to Rem - Den</i>	Rem = 10	Rem = 6	Rem = 2	
<i>Output Rem</i> <i>End Modulus</i>				Output 2
	Iteration 1	Iteration 2	Iteration 3	

We have already seen that a trace is a series of snapshots showing the behavior of an algorithm. In the trace of Figure 5.27 the result of each step is written directly to the right of that step in the algorithm. This creates a series of iterations, each iteration being an execution of the loop body represented by a column in the trace.

This trace shows the *Modulus* algorithm being executed to find the remainder of the division of 14 by 4. The algorithm proceeds by repeatedly subtracting 4 from 14 until the result is less than 4 (leaving a remainder of 2 in this example).

We will refine our traces to more convenient two-dimensional trace tables that will prove to be extremely useful.

Figure 5.28 Trace of the Divide algorithm

Structure	Trace			
<i>Divide</i> Input Num Input Den Set Rem to Num Set Quot to 0	Num = 14 Div = 4 Rem = 14 Quot = 0			
While Rem Den loop	14 4	10 4	6 4	2 4
Set Rem to Rem - Den	T Rem = 10	T Rem = 6	T Rem = 2	F Rem = 2
Set Rem to Rem - Den	Quot = 1	Quot = 2	Quot = 3	Quot = 3
Output Quot Output Rem End Divide				Output 3 Output 2
	Iteration 1	Iteration 2	Iteration 3	

Figure 5.28 shows the *Divide* algorithm introduced in Figure 3.45, which divides a numerator *Num* by a divisor *Den*, yielding a quotient *Quot* and a remainder *Rem*. The trace shows how 14 divided by 4 yields a quotient of 3 and a remainder of 2.

This algorithm first inputs the values for *Num* and *Den* and then initializes *Rem* with the value of *Num* and *Quot* to zero. The loop subtracts *Den* from *Rem*, counting the number of times it does this in *Quot* until *Rem* is less than *Den*. Finally, *Quot* and *Rem* are output.

Trace tables, as shown in Figures 5.27 and 5.28, are very useful ways to study the dynamic behavior of algorithms. A trace table is a two-dimensional arrangement of boxes set up at the side of an algorithm in either pseudocode or in flowblock form. For each iteration, the loop creates a stack of boxes showing the effect of the actions performed. This two-dimensional trace can then be viewed both horizontally and vertically, as will be shown in the following sections.

The *Divide* algorithm described here differs from the division operation available in most programming languages. First, the division of two Real Numbers (written *X/Y*) yields another Real Number expressed with a decimal point. Second, the division of two Integer values usually yields only the quotient, while the remainder can be obtained from the Modulus operation just described.

The Disadvantages of Using the Repeat-Until Form

Let's consider creating an algorithm to find the *Product* of two non-negative integers *X* and *Y*. The product is to be computed by successively adding the value

X a total of Y times. This algorithm for computing a product is not necessarily very useful because computers can multiply in a much more efficient manner. However, it helps illustrate many ways of inadvertently doing things wrong, and of what happens when we try to correct them.

Pseudocode 5.19 An erroneous algorithm infinite loop

<pre> Product A Set Product to 0 Repeat Add X to Product Decrement Y Until Y = 0 End Product A </pre>		Problem: if Y is initially 0, we get an infinite loop.
---	--	--

This algorithm, Pseudocode 5.19, does not use the fundamental While loop form but instead uses the Repeat-Until form, where the test for termination is made after executing the loop body. Now, let's see the unfortunate consequences of delaying the test until after the body has been executed!

This algorithm works well for most values, but not if Y is initially 0. In that case, Y is first decreased to -1, then Y is tested to see if it reached 0. But since Y has already passed 0, the algorithm keeps looping, and Y keeps decreasing forever! This is called an *infinite loop* and is a very common programming error.

Pseudocode 5.20 An erroneous algorithm incorrect product

<pre> Product B Set Product to 0 Repeat Add X to Product Decrement Y Until Y = 0 End Product B </pre>		Problem: if Y is initially 0, we get an incorrect product.
---	--	--

The revised version of *Product A* “fixes up” the infinite loop by leaving the loop when Y is less than or equal to zero (Pseudocode 5.20). This way, it does not loop forever, but it still does not compute the correct result for the case where $Y=0$. For that case, the algorithm first sets *Product* to zero and then immediately increases it by X and halts. So using this algorithm to multiply any value by zero does not yield a zero!

Pseudocode 5.21 An erroneous algorithm Y value destroyed

<pre> Product C Set Product to 0 If Y = 0 Repeat Add X to Product Decrement Y Until Y = 0 End Product C </pre>		Problem: using Y as a counter destroys its initial value, making this algorithm useless if more than one product must be calculated.
--	--	--

In Pseudocode 5.21, *Product B* has been patched-up to handle the case where *Y* is zero by first testing for it, and immediately circumventing the loop. But *Product C* still has a problem! By using variable *Y* as a counter, it destroys its original value. This *side effect* may not always be harmful. But if *Y* is to be used again later in the program that uses *Product C*, its value will have been changed to 0.

For example, suppose *Y* represents a constant rate of pay— say \$10 per hour— that is to be multiplied by various numbers of hours *X* on successive invocations of *Product C*. The first execution of *Product C* will yield a correct result but the value of *Y* will be set to 0. Thus, all subsequent products of any *X* with *Y* will yield zero.

Pseudocode 5.22 The final, corrected algorithm Product D

```

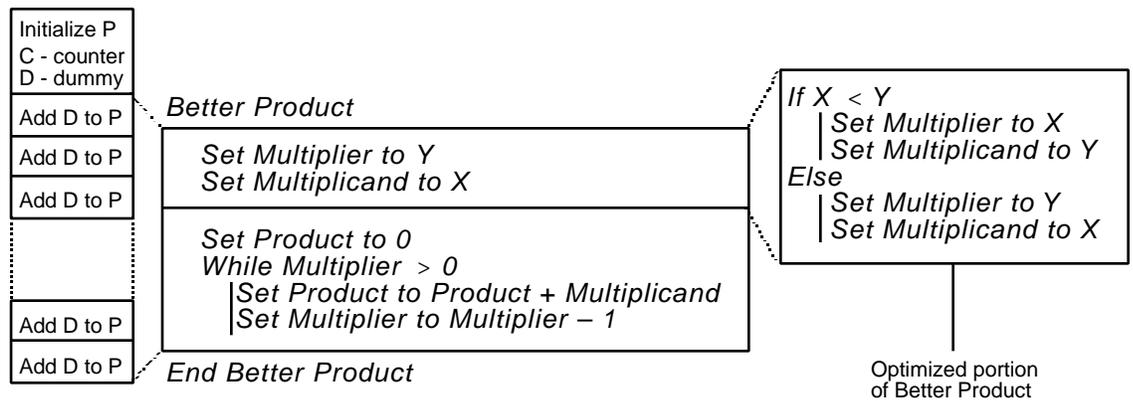
Product D
  Set Product to 0
  Set Count to Y
  If Count = 0
    Repeat
      Add X to Product
      Decrement Count
    Until Count = 0
  End Product D

```

Pseudocode 5.22 finally fixes this side effect in *Product C* by first placing a copy of *Y* into the variable *Count*, and using *Count*, instead of *Y*, as a counter. This algorithm finally works, but it is more complex and messy than it should be. It is called a “kludge”¹—something that works for the wrong reasons. There is certainly no elegance here!

This process of trial and error followed by many fix-ups and more errors is entirely unnecessary. Some prior planning and refinement, using the Four Fundamental Forms, could yield a better algorithm as shown in Figure 5.29.

¹ Pronounced “klooj” and said to be from Yiddish *klug* “smart”.

Figure 5.29 Better Product Algorithm

The development of the algorithm, in Figure 5.29, starts out at a very primitive level by planning how the algorithm is to execute—essentially by writing a trace of the execution of the algorithm. This “planning trace” starts by setting up not only a counter *C* equal to the multiplier (the number of additions to perform), but also a “dummy” variable *D* equal to the multiplicand (the value to add), and a product *P* that is initially zero. Then the successive additions are shown as a series of accumulations.

A simple refinement of this is done by replacing the sequence of additions in the plan with a loop. This is shown in the pseudocode at the center of Figure 5.29. Note that some more descriptive names have been given to the variables, *Multiplier* and *Multiplicand*. This could be the final algorithm; there is nothing more to it! By using the proper structures in the proper way we have created a better *Product*. In fact, since we avoided wasting time fixing and patching, we could take some time to optimize the finished product.

The further refinement on the right of the figure shows an optimized *Better Product*, where the counter *Multiplier* is first chosen to be the smaller of the two values *X* and *Y*. This way, the algorithm loops the fewest number of times. Time is saved for the machine, but not necessarily for the human programmer. Further optimization of this *Better Product* algorithm is possible, and we will discuss it at the end of this chapter.

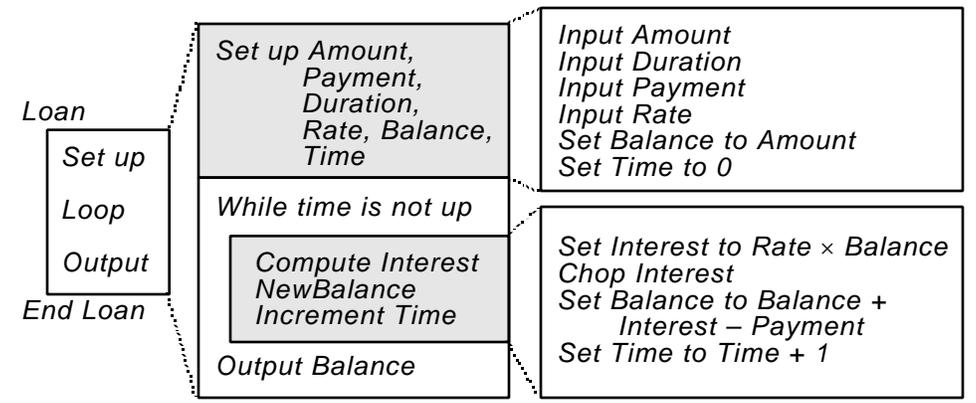
The While Loop Form

Borrowing money usually involves interest-*ing* algorithms. As an example, let us consider taking a loan of *Amount* dollars (say \$6000) for *Duration* years (say six years). Each year, *Payment* dollars (say \$1000) are repaid and the interest is calculated at *Rate* percent (say 10% annually) on the remaining *Balance* dollars. This amount of interest is chopped off to the next lower dollar (you can now be sure that this algorithm was not designed by a bank!). All these

numbers and conditions have been chosen for our convenience in making the trace. Later, when using a computer, we can be more realistic (with 72 monthly payments).

After making the payments for *Period* years, there is still an amount to repay at the end; this amount is called the *balloon payment*, because it “balloons” into a much larger amount than we expect if the monthly payment is too low. The problem is to compute the balloon payment. You may wish to guess the value before reading on.

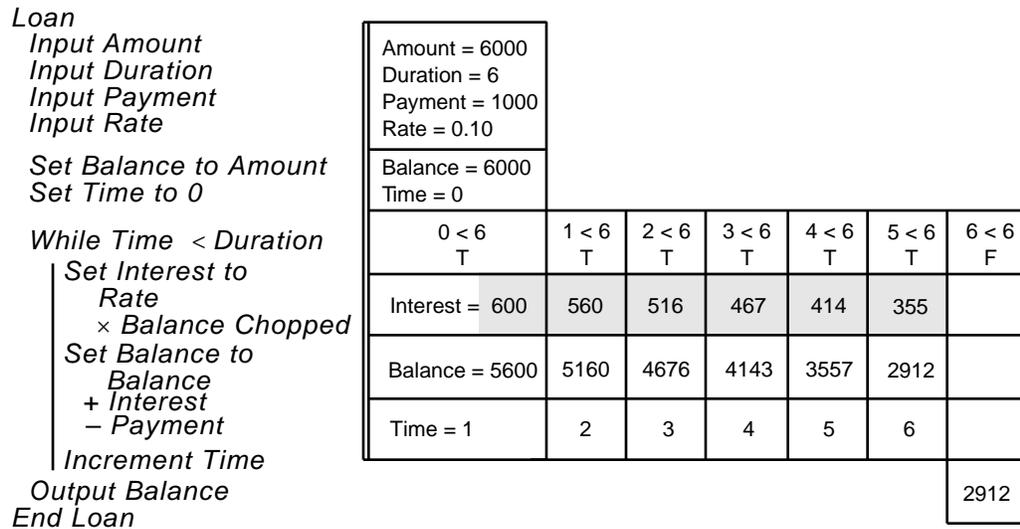
Figure 5.30 Top-down development of loan program



The development of the *Loan* algorithm is shown in Figure 5.30 as a top-down breakout of three levels.

- The first level shows a very general view. It consists of three steps: setting up, looping and then outputting the final balance (the balloon).
- The next level to the right refines each of these three blocks, but still not in detail. It shows the loop explicitly and breaks out the actions within the body of the loop.
- The final level at the right of the figure shows even more details of the setup and of the loop body, as pseudocode fragments.

Figure 5.31 Loan repayment algorithm and trace



The trace of the *Loan* algorithm is shown as a two-dimensional table form in Figure 5.31. This trace shows a solution of \$2912 as the balloon payment. It is almost half the loan amount!

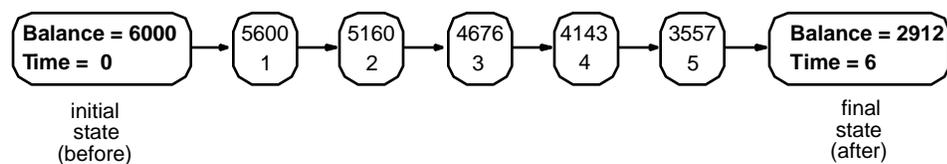
The trace also shows yet another way of computing this balloon payment and so can serve as a check on the answer. It results from realizing that, after the six payments of \$1000, the entire loan amount has been paid off and the balloon payment is due only to the interest. The total interest can be determined by summing the *Interest* computed each month:

$$600 + 560 + 516 + 467 + 414 + 355 = 2912$$

This corresponds to summing the second row of the trace (shown shaded). Be sure to note that this alternative method works only if the sum of the payments alone equals the amount borrowed.

Notice that if the yearly payment is \$600 (equal to the interest), then the balloon payment equals the original loan amount (of \$6000), regardless of the duration! This is called an “interest-only” loan.

Figure 5.32 State trace for the Loan Program



A shorter form of the algorithm trace, called a *State trace*, is illustrated by Figure 5.32. The state trace shows only the really relevant variables: *Balance*

and *Time*. These two variables are essential to the algorithm; should the process be interrupted at any stage, it is possible to continue it if only these two variables are known.

Getting Insights Using Traces and Invariants

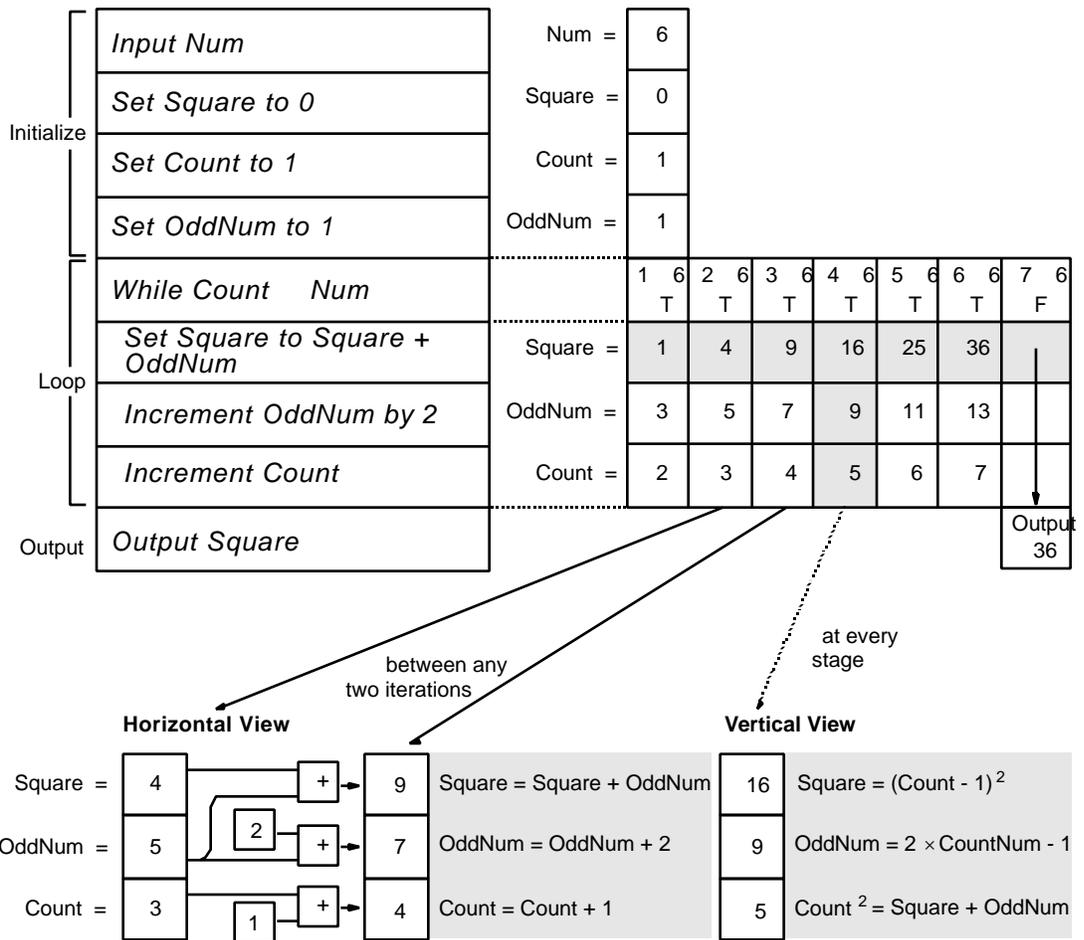
The trace of an algorithm can yield many insights. Oftentimes, a study of such a trace will show an error in an algorithm. When no errors are detected, the trace might suggest improvements or even spark an idea for an alternative algorithm. Traces also might show useful relationships among variables. We will illustrate these concepts throughout the rest of this chapter with a number of examples.

Our first example, the *Odd Square* algorithm, shown in Figure 5.33, finds the square of any integer *Num* by summing the first *Num* odd integers. The algorithm has three parts

- *Initialization*: The starting values are assigned to the variables.
- *Loop*: The value of the square is calculated by summing a sequence of odd numbers.
- *Output*: The calculated value of the square is output.

The Odd Square algorithm was introduced in Chapter 3, Figure 3.9.

Figure 5.33 The Odd Square algorithm and traces

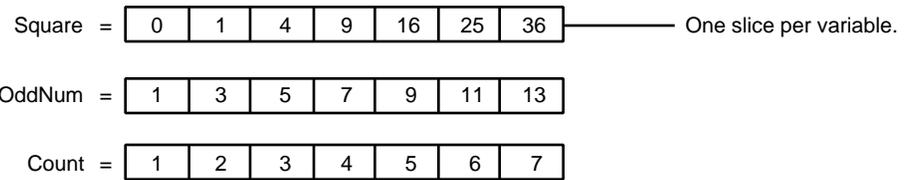


Two-dimensional tracing, in the form of a table, is shown at the top of Figure 5.33. For each run through the loop, called an *iteration*, there is a new column of values added to the right side of the table. This tracing table may be viewed both horizontally and vertically.

Horizontal views of the computation correspond to the iteration by iteration computation of each variable, as specified in the algorithm. The trace table in Figure 5.33 essentially shows horizontal views. At each stage, *Square* is incremented by *OddNum*, *OddNum* is incremented by 2 and *Count* by 1.

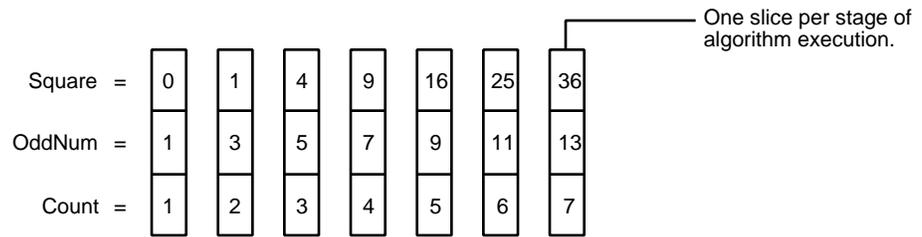
Figure 5.34 shows the horizontal slices that correspond to each variable. Notice that the *Square* of the numbers 1 through 5 are computed before the final square of 6.

Figure 5.34 Some horizontal slices



Vertical views consider each stage separately and show all the variables at each iteration, as shown in Figure 5.35.

Figure 5.35 Some vertical slices



We have said that observing a trace often reveals relationships among variables. In our *Odd Square* example, at each iteration, adding *Square* and *OddNum* yields the square of *Count*:

After iteration 1, $1 + 3 = 2 \times 2$

After iteration 2, $4 + 5 = 3 \times 3$

After iteration 3, $9 + 7 = 4 \times 4$

....and so on.

In general:

$$Square + OddNum = Count \times Count$$

Some assertions about the state of a program (such as the one above) are unaffected by the execution of the body of a loop—if the assertion is true *before* execution of the body, it will still be true *after* its execution. Such assertions are called *loop invariants*. As an example, Pseudocode 5.23 repeats the pseudocode loop of the *Odd Square* algorithm, this time including the invariant between braces.

Pseudocode 5.23 Loop from Odd Square algorithm

```

While Count Num {Square + OddNum = Count × Count}
  Set Square to Square + OddNum
  Increment OddNum by 2
  Increment Count
    
```

It is important to realize that an assertion may not be true at a point part of the way through the execution of a loop body. The reason for this is because only

some of the variables used in the assertion may have been changed. Once the body has been completely executed, the assertion is true again; the relationship among the variables is constant, or invariant.

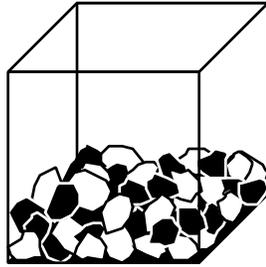
For example, in the above loop, the assertion is no longer true after the first statement in the loop body has been executed. It does not become true again until after the third statement has been executed.

Further observation of the trace of Figure 5.33 yields two more formulas that hold at every stage, giving us three invariants for the same loop:

- $Square + OddNum = Count \times Count$
- $Square = (Count - 1)^2$, and
- $OddNum = 2 \times Count - 1$.

This last relation between *OddNum* and *Count* suggests that the use of both of these variables is not necessary. In fact a simpler algorithm with only one of these variables, *Count*, is possible. Try it.

In the *Odd Square* example there were many invariants, and they were rather easy to observe. In general, loop invariants are not always easy to find and there is no foolproof technique for finding them. Loop invariants are very important because they precisely describe the action of the loop in which they appear. Sometimes, they provide the essential piece of information that allows us to understand the action of the loop. As an example of this, let's look at the Go Stone game as described in Figure 5.36.

Figure 5.36 Specifications of the Go Stone Problem**The Box****The Bag** - infinite supply of stones
(black & white)**The Go Stone Problem**

The Japanese game of Go Stone is played with black stones and white stones. A box contains some of these black and white stones, and a large bag acts as an inexhaustible supply of these black and white stones. The following algorithm is to be performed.

```

While number of stones in box > 2
  Randomly select two stones from the box
  If the two stones are of same color
    Throw out the two stones
    Put a black stone from the bag into the box
  Else
    Put the white stone back in the box
    Throw away the black stone

```

Each iteration of the loop reduces the number of stones in the box by one. The loop will terminate when there is exactly one stone left in the box. The question for you to answer is: by counting the number of black and white stones in the box before the algorithm is executed, is it possible to know the color of the final stone that will be left in the box when the loop terminates? Before reading on, try to solve this problem yourself.

Let us examine the way in which the number of stones is reduced:

- If a matching pair of black stones is removed, then one black stone is put back into the box, the number of black stones is reduced by one and the number of white stones stays constant.
- If a matching pair of white stones is removed, a black stone is put back into the box, the number of black stones is increased by one and the number of white stones is reduced by two.
- If a non-matching pair is removed, a white stone is put back into the box and the number of white stones remains unchanged while the number of black stones is reduced by one.

Thus, the number of black stones always changes by one at each iteration while the number of white stones either remains constant or is reduced by two. This means that, if we started with an even number of white stones, there will always be an even number of them; if we started with an odd number of white stones, there will always be an odd number of them.

In other words, the parity of the number of white stones is invariant; this is our loop invariant. The only way that we can finish with a single white stone—an odd number of white stones—is if we start with an odd number of white stones. If we start with an even number of white stones, the last stone will be black. The only way that we can understand the loop well enough to make this statement is through the use of the loop invariant.

Later, in Section 5.7, we will show how we can use loop invariants to improve algorithms. Finally, note that loop invariants are also used to formally prove the correctness of algorithms.

When we evaluated the trigonometric sine function in Chapter 3, we used the Factorial mathematical function. Factorials are also used in probability, statistics and counting problems. The Factorial of any non-negative integer N is written as $N!$ and defined as follows:

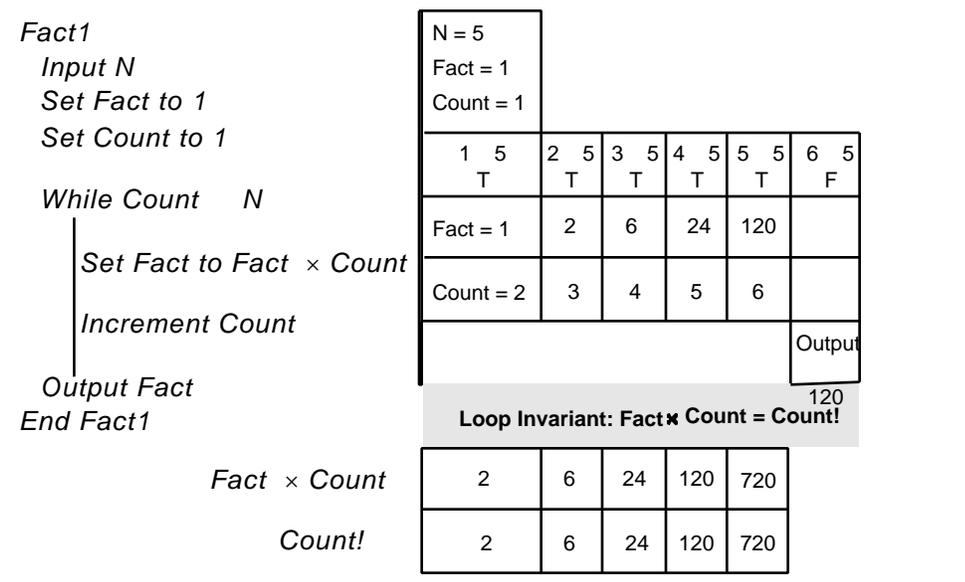
$$N! = N \times (N - 1) \times (N - 2) \times (N - 3) \times \dots \times 3 \times 2 \times 1$$

Because multiplication is commutative, the products can be done in either increasing or decreasing order as...

$$5! = 1 \times 2 \times 3 \times 4 \times 5 \quad \text{or} \quad 5! = 5 \times 4 \times 3 \times 2 \times 1$$

Figures 5.37 and 5.38 illustrate these two different algorithms. *Fact1* multiplies products in increasing order while *Fact2* does so in decreasing order.

Figure 5.37 Trace of the Fact1 algorithm for $N = 5$



Fact1, of Figure 5.37, initially sets *Fact* and *Count* to 1, and then loops as long as *Count* is less than or equal to the input value N . In the loop body, *Fact* is multiplied by the value of *Count* and *Count* is incremented. When *Count* is incremented past N , the loop is terminated and the final value of *Fact* is output.

The trace shows this computation for an input value of $N = 5$. You will quickly notice that as $Count$ increases gradually, the value of $Fact$ increases dramatically. Just doubling the value of N to 10 would produce an output value of 3 628 800. Such a computation can quickly exceed the bounds of any computer, so we must beware of computing the factorials of large numbers.

If we observe this trace and extend it, the following two loop invariants become apparent:

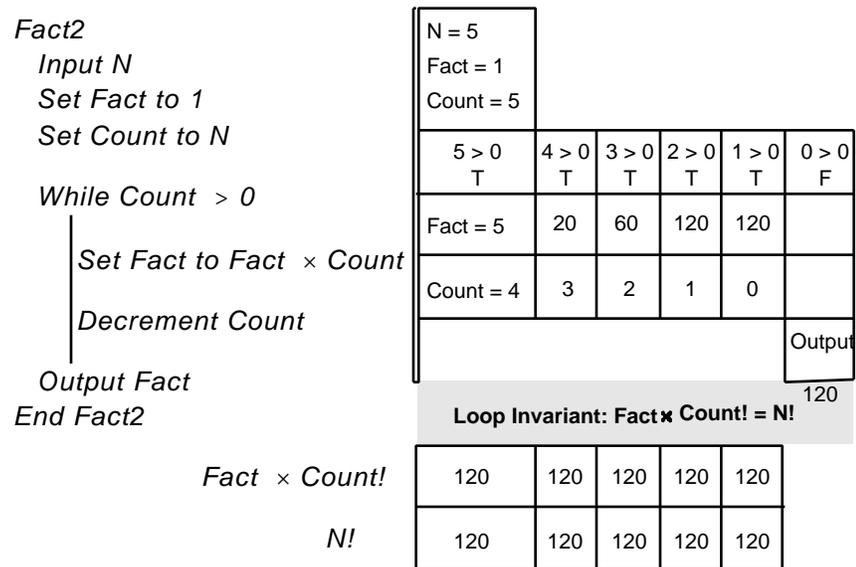
- $Count! = Fact \times Count$, and
- $Fact = (Count - 1)!$

These two relations can be combined into a third relation:

- $Count! = Count \times (Count - 1)!$

This is a definition of factorial in terms of itself. Such a definition is called a *recursive definition* and will be discussed in Chapter 7.

Figure 5.38 Trace of the Fact2 algorithm for $N = 5$



The algorithm *Fact2*, of Figure 5.38, proceeds in the opposite way from *Fact1*. It starts the $Count$ at the value of N , and loops, decreasing this value until it reaches zero. During each loop, it multiplies $Fact$ by the decreasing value of $Count$. Figure 5.38 presents a trace of this algorithm as well.

Observing this trace, we see what at each stage the following invariant holds:

$$N! = Fact \times Count!$$

Note: It is interesting that since the loop invariant holds true even when $Count = 0$, the factorial of 0 must be 1.

Notice that this loop invariant for *Fact2* is very different from the previous invariant for *Fact1*. This is normal since different ways of computing the same value may lead to different loop invariants.

If the condition $Count > 0$ were incorrectly written as $Count = 0$, the output of *Fact2* would be zero because *Fact* is finally multiplied by the lowest value of *Count*, which in that case would be zero. This is the error of one iteration too many, and is usually called a *one-off error*. Slight errors like this one can result in much larger errors in the final results. Tracing helps find these errors. The one-off error is one of the most common errors in programming.

Two-dimensional tracing is useful in many ways:

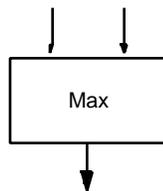
- Tracing yields insight into the dynamic behavior of algorithms.
- Tracing detects errors (such as the one-off error just mentioned).
- Tracing yields interesting relations among variables.
- Tracing suggests alternative algorithms.

5.6 Invocation Forms

Sub-algorithms (or subprograms) are simply algorithms that have been packaged into single units that may be used in other algorithms, as any other action is. Sub-algorithms hide data and actions internally and so appear simple externally. When a sub-algorithm is used as an action in another algorithm, we say that it is “called” or “invoked”. In pseudocode, this is done by giving its name and listing the data that it must use.

Figure 5.39 shows a data-flow diagram of *Max*, a sub-algorithm that finds the maximum of two values.

Figure 5.39 Data-flow diagram for the sub-algorithm *Max*

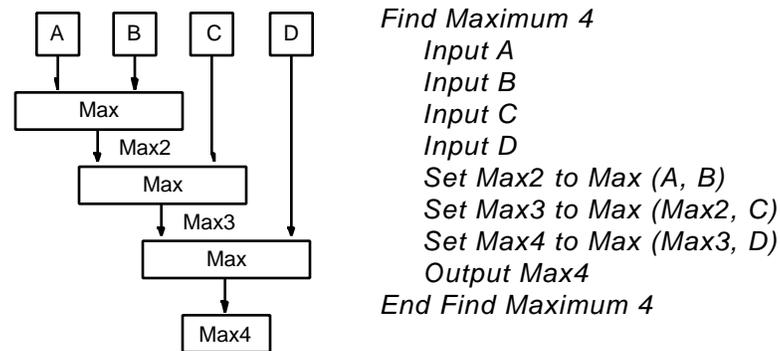


In this view, a black-box view, we show all that is needed in order to be able to use *Max*—we see that it takes in two data values and passes out the maximum of these two. In the following small piece of pseudocode (Pseudocode 5.24), the two values *X* and *Y* are input, the sub-algorithm *Max* is invoked to find the larger of *X* and *Y*, this value is assigned to *Bigger* and, finally, *Bigger* is output.

Pseudocode 5.24 Input X and Y and invoke Bigger sub-algorithm

Input X
Input Y
Set Bigger to Max X, Y
Output Bigger

Figure 5.40 shows the data-flow diagram and the pseudocode for an algorithm that makes use of sub-algorithm Max three times to find and output the maximum of four input values A, B, C, D.

Figure 5.40 Data-flow diagram and Pseudocode to find maximum of four values

This notation is similar to the following function in algebra which reads “M is the maximum of x and y.”

$$m = \text{Max}(x,y)$$

The power of the sub-algorithm comes from its ability to hide the details of how the maximum value is computed, thus simplifying the algorithm that uses it. If all the details were shown, the pseudocode in Figure 5.40 would resemble Pseudocode 5.25.

Pseudocode 5.25 Unsimplified version of Maximum 4

```
Find Maximum
  Input A
  Input B
  Input C
  Input D
  If A > B
    Set Max2 to A
  Else
    Set Max2 to B
  If Max2 > C
    Set Max3 to Max2
  Else
    Set Max3 to C
  If Max3 > D
    Set Max4 to Max3
  Else
    Set Max4 to D
  Output Max4
End Find Maximum
```

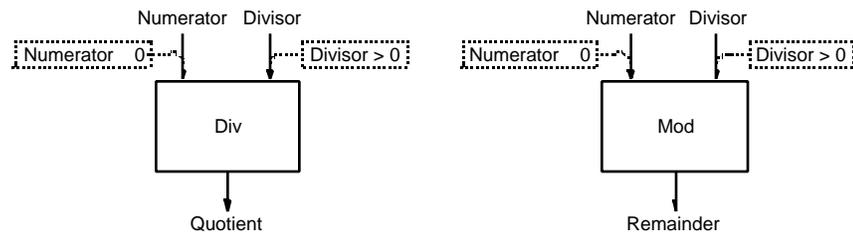
The version in Pseudocode 5.25 is much more complex. Here, the reader must examine the details carefully to see that the same method is used each time to find the maximum of two numbers. If such a degree of simplification can be achieved for such a trivial sub-algorithm as Max, imagine the improvement in clarity that can be obtained when a complex sub-algorithm is involved! This process of reducing the complexity by hiding the details is called *abstraction*.

Seconds Example

Suppose you were asked to convert a given number of seconds (say $Time = 1\,000\,000$ seconds) into days, hours, minutes and seconds. You could proceed as follows:

1. Divide $Time$ by the number of seconds in a day (86 400), yielding a quotient, the number of *Days*, and also yielding some remaining number of seconds in *Seconds*.
2. Then this remainder is divided by the number of seconds in an hour (3 600), yielding the number of *Hours* and some remaining *Seconds*.
3. Finally, the above remainder is divided by the number of seconds in a minute (60), yielding a quotient that is the number of *Minutes* and a remainder that is the number of *Seconds*.

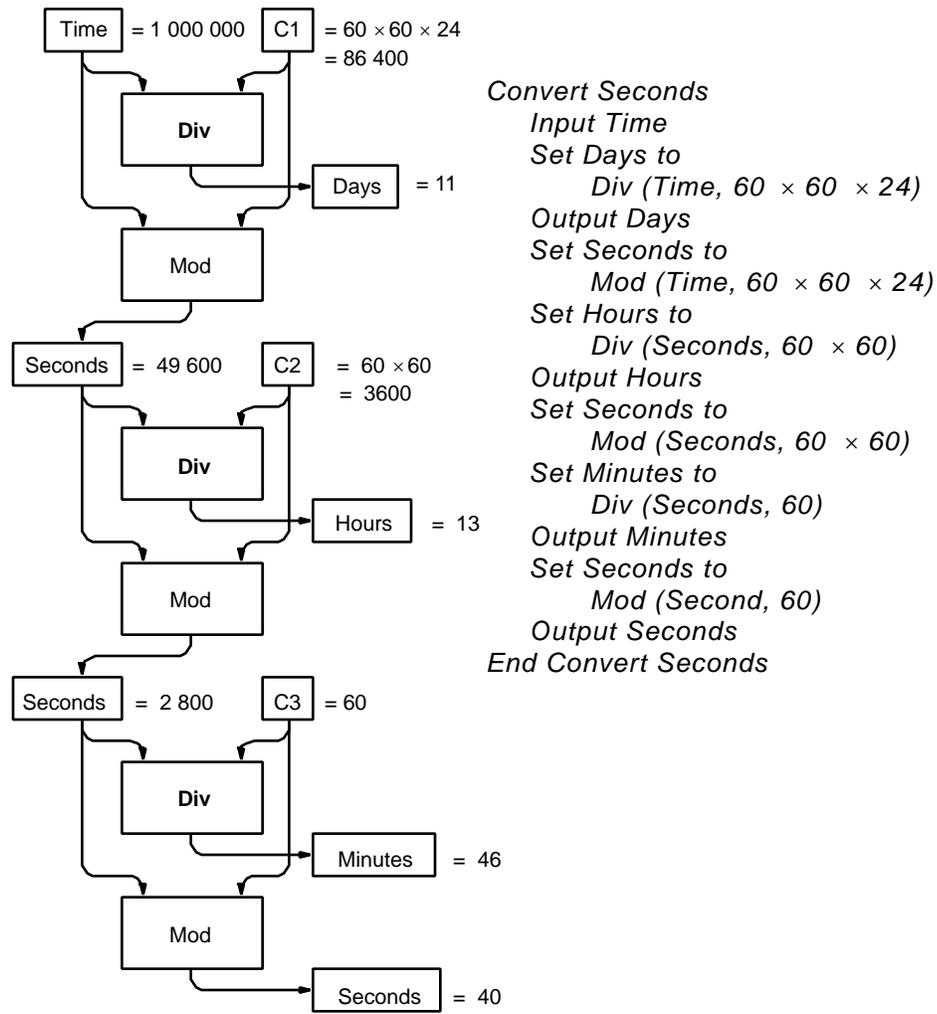
Figure 5.41 Data-flow diagrams for the *Div* and *Mod* sub-algorithms



To do this, we make use of two sub-algorithms, *Div* and *Mod*, which are defined by the data-flow diagrams shown in Figure 5.41. Note the assertions about the data values that *Div* and *Mod* work with. In both cases the Numerator and Divisor must be positive integers. In addition, the Divisor cannot be zero because of the mathematical commandment:

“Thou shalt not divide by zero!”

Figure 5.42 Algorithm Convert Seconds



The algorithm *Convert Seconds* of Figure 5.42, shows a data-flow diagram and corresponding pseudocode for the algorithm just described. Remember that products such as $60 \times 60 \times 24$ need not be evaluated by us; computers can do that.

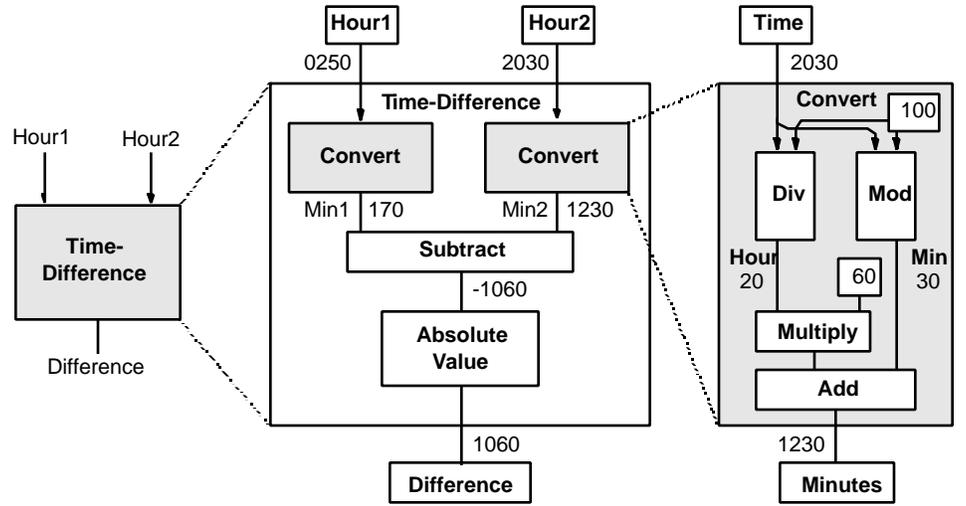
De-militarize Time Example

As another example of the invocation form, let's take a look at military time and the problem of finding a military time difference. As you know, military time is expressed as an integer between 0 and 2400. However, the difference between two times given in military form cannot be found by simply subtracting the two military times. For example, the military times 1400 and 1350 do not have a 50 minutes difference ($1400 - 1350 = 50$), but only a 10-minute difference. This is not a problem with the 24-hour form of time but with the use of a base 10

representation for a number that is not pure base 10. We would have a similar problem if we tried to represent a person's height by the integer 511 instead of 5' 11".

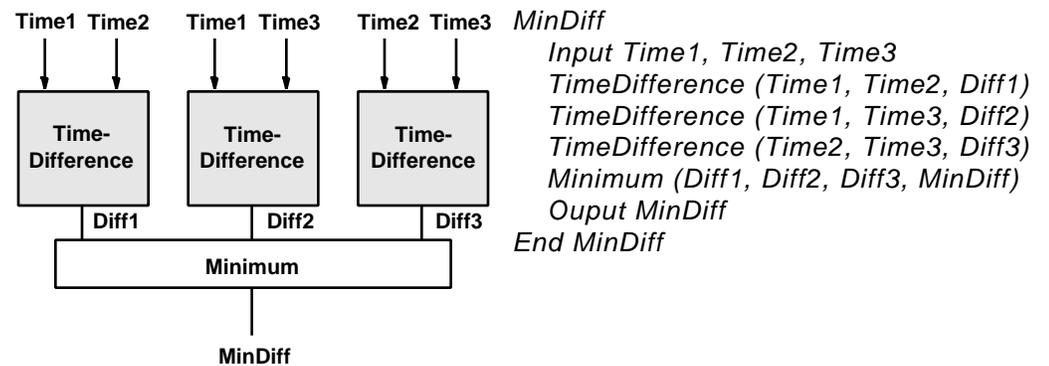
If each of the military times is converted to minutes past midnight, then these numbers of minutes can be subtracted. An algorithm to compute this military time difference is shown in Figure 5.43. It includes Invocations of a conversion sub-algorithm that is similar to the *Convert Seconds* algorithm we have just seen.

Figure 5.43 Time Difference algorithm with Invocations



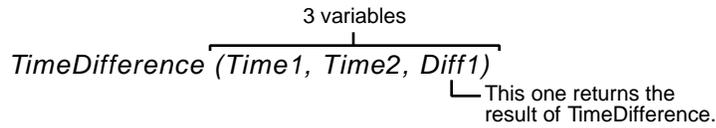
If what we really want is the minimum difference between any three military times, we can proceed as indicated on Figure 5.44. We use *TimeDifference* as a sub-algorithm that we invoke three times, after which we invoke a sub-algorithm, *MinDiff*, to find the minimum of three values.

Figure 5.44 Minimum time difference



Note that *TimeDifference* is invoked with three variables, the last one being used to return the result of the sub-algorithm execution.

Pseudocode 5.26 Invoking the TimeDifference sub-algorithm



5.7 Improving Programs

Nested Loops and Selections

Until now, the algorithms of this chapter involved only simple forms. Here we will consider some combinations of the Four Fundamental Forms, but we are mainly interested in tracing these more complex structures. Their design will only be considered in Chapters 6, 7, and 8.

Our first combination example computes the greatest common divisor of two integers, that is, the largest integer that divides both of them. For example, the greatest common divisor of 111 and 259 is 37. The Greatest Common Divisor algorithm was first created by Euclid, in about 300 BC. (algorithms did not wait for computers!) Figure 5.45 presents the Greatest Common Divisor (*GCD*) algorithm as an algorithm that illustrates the nesting of a Selection form inside a Repetition form.

Figure 5.45 The Greatest Common Divisor algorithm and trace

GCD

<i>Input P</i>	P = 111				
<i>Input Q</i>	Q = 259				
<i>Set X to P</i>	X = 111				
<i>Set Y to Q</i>	Y = 259				
<i>While X = Y</i>	111 259 T	111 148 T	111 37 T	74 37 T	37 37 F
<i>If X > Y</i>	111 > 259 F	111 > 148 F	111 > 37 T	74 > 37 T	
<i>Set X to X - Y</i>	X = 111	X = 111	X = 74	X = 37	
<i>Else</i>					
<i>Set Y to Y - X</i>	Y = 148	Y = 37	Y = 37	Y = 37	
<i>Output Y</i>					Output 37
<i>End GCD</i>					

The result and the trace of the Greatest Common Divisor algorithm are also shown in Figure 5.45. One use of the Greatest Common Divisor is for sharing or partitioning two kinds of “whole” items, such as pebbles or cans of goods, that cannot be split further. For example, if we have 111 cans of one kind of item and 259 cans of another kind, then the largest number of identical piles of cans equals the Greatest Common Divisor, which is 37. Each of the 37 piles would have 3 cans of one type ($111/37 = 3$), and 7 cans of the other type ($259/37 = 7$). There are many other uses of the Greatest Common Divisor, but they are usually more complex.

Our next example is an algorithm that converts decimal integers into their equivalent binary form. Figure 5.46 shows this conversion algorithm together with its trace. We can see, for example, that it converts the decimal number 13 into the binary number 1101.

Figure 5.46 The convert decimal to binary algorithm and trace

Convert to Binary					
Input X					
Set Power to 16					
Set Temp to X					
While Power > 0					
Set Diff to					
Temp – Power					
If Diff < 0					
Output "0"					
Else					
Output "1"					
Set Temp to					
Diff					
Divide Power by 2					
End Convert to Binary					
X = 13					
Power = 16					
Temp = 13					
16 > 0 T	8 > 0 T	4 > 0 T	2 > 0 T	1 > 0 T	0 > 0 F
Diff = -3	Diff = 5	Diff = 1	Diff = -1	Diff = 0	
-3 < 0 T	5 < 0 F	1 < 0 F	-1 < 0 T	0 < 0 F	
0	1	1	0	1	
	Temp = 5	Temp = 1		Temp = 0	
Power = 8	Power = 4	Power = 2	Power = 1	Power = 0	

Notice first the following binary “break-up” of the number:

$$13 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$$

This algorithm does the conversion by extracting various powers of 2, starting from the highest power and decreasing by a factor of 2 during each loop. Here the algorithm starts from the power of 2 just higher than the number to be converted; to convert 13, it starts with $Power = 16$.

1. There is no 16 in 13, so the output is 0.
2. There is an 8 in 13, so the output is 1 and there remains 5.
3. There is a 4 in the remaining 5, so the output is 1 and there remains 1
4. There is no 2 in the remaining 1, so the output is 0.

5. There is a 1 in the remaining 1, so the output is 1 and there remains 0.

The resulting output sequence 01101 is the binary equivalent of 13.

For larger values, this algorithm cannot start with *Power* = 16, but with a power of 2 that is just larger than the number to be converted. This could be done with a separate sub-algorithm, that we could call Initialize Power. It would start with *Power* equal to 2, and would loop, each time doubling *Power* until *Power* exceeded *X*.

Our next example is a simple multiplication algorithm that will help us illustrate the nesting of a Repetition form within a Selection form.

Figure 5.47 Signed Product algorithm and trace

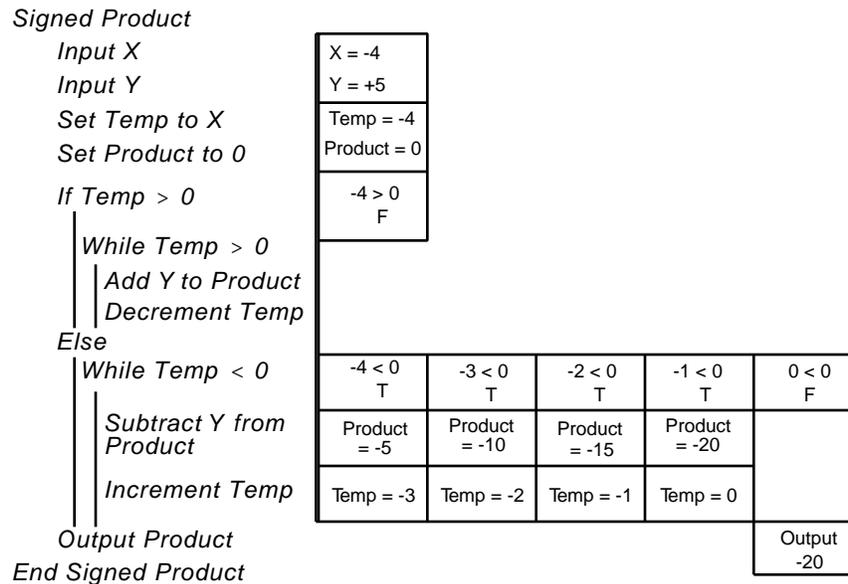


Figure 5.47 shows the multiplication algorithm for two integers that may be positive or negative. In this example we can find two *While* loops nested in the two parts of a *Selection*. The trace illustrates the dynamics of the *Else* part of the *Selection* since the first value to multiply is negative. As in the preceding conversion algorithm, we use an extra variable, *Temp*, to avoid modifying the original value of variable *X*.

Change may be made in many ways, illustrating many combinations of forms. In Chapter 4 we designed an algorithm to Make Change. Let us consider now a slightly modified version of Change Maker, which provides a count of the quarters, nickels, and pennies in exchange for a dollar when buying an item for *Cost* cents. In this version we chose to have no dimes or half-dollars involved.

See Pseudocode 4.10 for a detailed Make Change algorithm.

Pseudocode 5.27 Modified version of Change Maker

```

Change Maker 1
Input Cost
Set Change to 100 - Cost
Set Quarters to 0
While Change > 25
    Increment Quarters
    Set Change to Change - 25
Output Quarters
Set Nickels to 0
While Change > 5
    Increment Nickels
    Set Change to Change - 5
Output Nickels
Set Pennies to Change
Output Pennies
End Change Maker 1

```

Note that change can only be made from \$1.

Giving quarters

Giving nickels

Giving pennies

The algorithm in Pseudocode 5.27 uses the method of subtraction and is very similar to the previously described method of Chapter 4. It consists of a series of loops, starting at the higher denominations (quarters), and subtracting away these denominations, counting as it proceeds.

Let's look now at another version of Change Maker.

Pseudocode 5.28 Second new version of Change Maker

```

Change Maker 2
Input Cost
Set Change to 100 - Cost
Set Quarters to Change / 25
Output Quarters
Set Change to Change - Quarters * 25
Set Nickels to Change / 5
Output Nickels
Set Pennies to Change - Nickels * 5
Output Pennies
End Change Maker 2

```

Giving quarters

Giving nickels

Giving pennies

The second algorithm, Pseudocode 5.28, is still based on the method of division. *It makes use of the fact that a division of integers gives an integer quotient.* This version of Change Maker is essentially the same as the previous method. It simply recognizes that the previous method was really doing division in disguise.

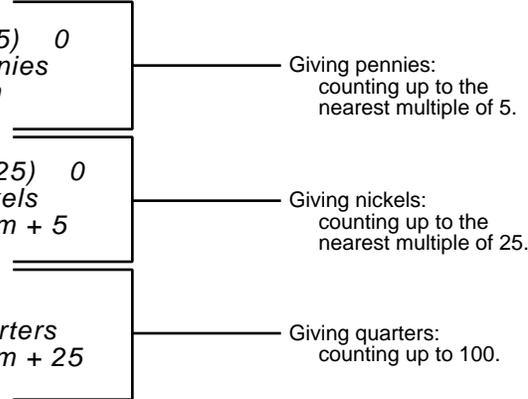
The next version we will develop will be based on addition and close to the actual method used by cashiers.

Pseudocode 5.29 Third new version of Change Maker

```

Change Maker 3
Input Cost
Set Sum to Cost
Set Pennies to 0
While Mod (Sum, 5) ≠ 0
    | Increment Pennies
    | Increment Sum
Output Pennies
Set Nickels to 0
While Mod (Sum, 25) ≠ 0
    | Increment Nickels
    | Set Sum to Sum + 5
Output Nickels
Set Quarters to 0
While Sum < 100
    | Increment Quarters
    | Set Sum to Sum + 25
Output Quarters
End Change Maker 3

```



The third version in Pseudocode 5.29 corresponds to the way people usually prefer to make change, for it involves only addition. Here, the algorithm starts with the value *Cost* and the lower denominations; pennies are first added until *Sum* is a multiple of 5. Note that the test for *Sum* to be a multiple of five makes use of the *Mod* function of Figure 5.41 which gives the remainder when *Sum* is divided by 5. When this remainder is zero, *Sum* is a multiple of 5. Then nickels are similarly added until *Sum* is a multiple of 25. Finally, quarters are added until *Sum* reaches a dollar.

Let's rewrite our Change Maker algorithm one more time using a different combination of forms; this time with Selections nested within a Repetition.

Pseudocode 5.30 Fourth Change Maker using nested forms

```

Change Maker 4
  Input Cost
  Set Change to 100 – Cost
  Set Quarters to 0
  Set Nickels to 0
  Set Pennies to 0
  While Change > 0
    If Change ≥ 25
      Increment Quarters
      Set Change to Change – 25
    Else
      If Change ≥ 5
        Increment Nickels
        Set Change to Change – 5
      Else
        Increment Pennies
        Decrement Change
  Output Quarters _____ Giving quarters
  Output Nickels _____ Giving nickels
  Output Pennies _____ Giving pennies
End Change Maker 4

```

The fourth version of Change Maker, Pseudocode 5.30, is written as a single loop, having nested Selections within it. Other than that, this method is quite similar to our first method of subtraction; it simply trades off a series of Repetitions for a nest of Selections.

Each of these four methods can very easily be extended to include more denominations such as dimes, half-dollars, two-dollar bills, and so on. They could also be extended to accept any amount tendered instead of assuming that it is \$1. But if you try these extensions, you will find that some of these methods extend much more easily than others.

These are not the only methods for making change for other methods of making change are still possible. We will see more about change-making in Chapter 7.

Using Invariants

We have seen earlier that loop invariants were any assertions (relations, conditions, equations, etc.) whose truth is unaffected by the execution of the body of the loop. We will use loop invariants here to try and improve the speed of algorithms.

Note: Loop invariants are the same *before and after* loop execution, but not necessarily during.

As an example, consider the algorithm which calculates the product of two positive numbers, shown in Figure 5.48.

Figure 5.48 The Positive Product algorithm, version 1

<i>Positive Product 1</i>						
<i>Input X</i>	X = 5					
<i>Input Y</i>	Y = 6					
<i>Set Multiplier to X</i>	Multiplier = 5					
<i>Set Multiplicand to Y</i>	Multiplicand = 6					
<i>Set Product to 0</i>	Product = 0					
<i>While Multiplier > 0</i>	5 > 0 T	4 > 0 T	3 > 0 T	2 > 0 T	1 > 0 T	0 > 0 F
<i>Add Multiplicand to Product</i>	Product = 6	Product = 12	Product = 18	Product = 24	Product = 30	
<i>Decrement Multiplier</i>	Multiplier = 4	Multiplier = 3	Multiplier = 2	Multiplier = 1	Multiplier = 0	
<i>Output Product</i>	Loop Invariant: $X \times Y = \text{Product} + \text{Multiplier} \times \text{Multiplicand}$					Output
<i>End Positive Product 1</i>						30

The loop invariant in the Positive Product algorithm of Figure 5.48 is:

$$(1) \quad X \times Y = \text{Product} + \text{Multiplier} \times \text{Multiplicand}$$

This invariant has an intuitive meaning here. The product of *X* and *Y* at any point is determined by summing the partial *Product* and the remaining portion *Multiplier* × *Multiplicand* that is yet to be added into *Product*. This breakdown into two parts (an already computed part, and a part yet to be computed) is often useful for determining invariants.

A significant aspect of loop invariants is that the actions within the body of a loop do not change the invariant. For example, these are the actions within the loop:

Add Multiplicand to Product

and...

Decrement Multiplier.

These two actions could just as easily have been written as follows:

Set Product to Product + Multiplicand

and...

Set Multiplier to Multiplier - 1

Substituting these two into the original relation produces what is shown in Pseudocode 5.31.

Pseudocode 5.31 From one iteration to the next

$$(1) X \times Y = \boxed{\text{Product}} + \boxed{\text{Multiplier}} \times \text{Multiplicand} \quad \text{original relation}$$

$$(2) X \times Y = \boxed{(\text{Product} + \text{Multiplicand})} + \boxed{(\text{Multiplier} - 1)} \times \text{Multiplicand}$$

$$= \text{Product} + \cancel{\text{Multiplicand}} + \text{Multiplier} \times \text{Multiplicand} - \cancel{\text{Multiplicand}}$$

$$= \text{Product} + \text{Multiplier} \times \text{Multiplicand}$$

This yields again the original invariant! The invariant holds at the beginning of the body of the loop, and also at the end of it. In other words, this relation remained invariant after execution of the *complete* body of the loop. The adjective “complete” is important here, since the invariant relation is not necessarily true part way through the execution of the loop body. This point is extremely important in the following discussion.

Note In this example, both assignments must be added to the loop body together and in order for the invariants to hold. You can't add one without the other.

Given the invariant relation (1), it is interesting to see whether we can find other assignments that could be added to the body of the loop and that would keep the given relation invariant. Here is one such pair of assignments:

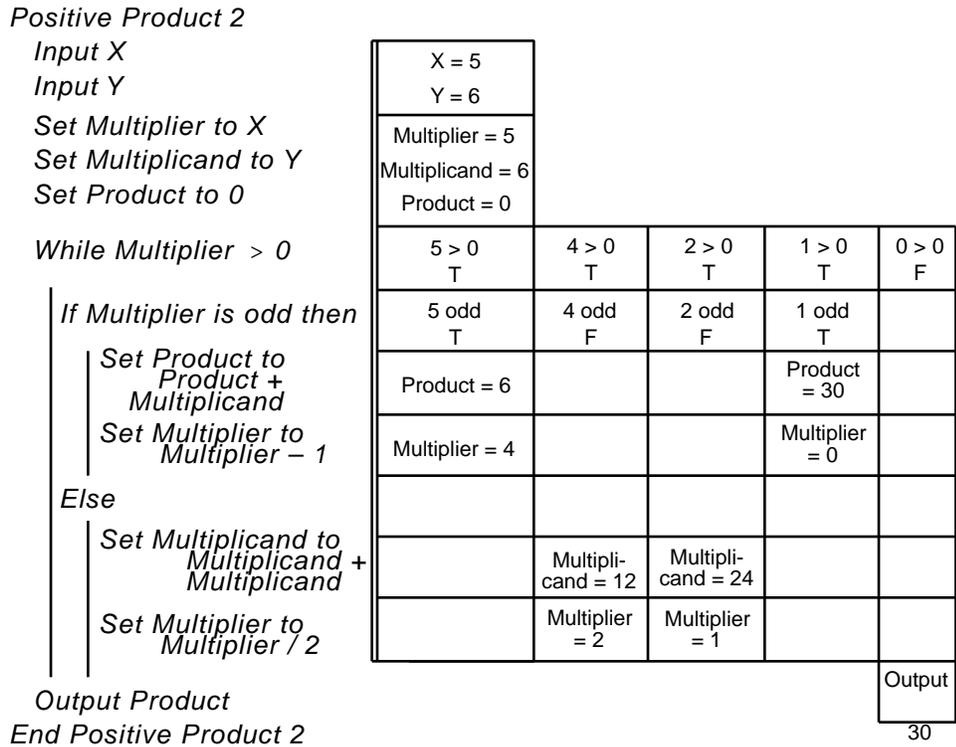
Pseudocode 5.32 Other equivalent assignments

Set *Multiplier* to *Multiplier* / 2
 Set *Multiplicand* to 2 × *Multiplicand*

In other words, if *Multiplier* is halved and *Multiplicand* is doubled, their product will remain the same. And incidentally, this halving and doubling is a faster way of finding the product than is the previous method, of adding a single *Multiplicand* and subtracting one from *Multiplier*.

Let's apply our new knowledge and develop a new *Product* algorithm. Of course, this halving can only be used if *Multiplier* is an even number so that we do not lose one turn in our Repetitions. So for odd numbers, we will still use the original pair of actions. If we then modify the body of the loop of Figure 5.48 to choose between these two pairs of assignments (both of which keep the relation invariant), we obtain a more efficient product algorithm as shown in Figure 5.49.

Figure 5.49 The Positive Product algorithm, version 2



Note that we have written the original two actions in a slightly different (but equivalent) way. Also, the doubling of *Multiplicand* was done by an addition so as to be more efficient, and above all so as to avoid using the product we are defining! A slightly more efficient algorithm is given in Figure 5.50, where we double and halve during every iteration.

Figure 5.50 The Positive Product algorithm, version 3

<i>Positive Product 3</i>				
<i>Input X</i>				
<i>Input Y</i>				
<i>Set Multiplier to X</i>				
<i>Set Multiplicand to Y</i>				
<i>Set Product to 0</i>				
<i>While Multiplier > 0</i>				
<i>If Multiplier is odd</i>				
<i>Set Product to Product + Multiplicand</i>				
<i>Set Multiplier to Multiplier - 1</i>				
<i>Set Multiplicand to Multiplicand + Multiplicand</i>				
<i>Set Multiplier to Multiplier / 2</i>				
<i>Output Product</i>				
<i>End Positive Product 3</i>				
X = 5				
Y = 6				
Multiplier = 5				
Multiplicand = 6				
Product = 0				
5 > 0 T	2 > 0 T	1 > 0 T	0 > 0 F	
5 odd T	2 odd F	1 odd T		
Product = 6		Product = 30		
Multiplier = 4		Multiplier = 0		
Multiplicand = 12	Multiplicand = 24			
Multiplier = 2	Multiplier = 1			
				Output
				30

The efficiency of these algorithms can be illustrated by considering the product of two numbers as big as a million. Version 1 would make about 1 000 000 loops, Version 2 would make about 40 loops, and Version 3 would make about 20 loops! If the time difference from 1 000 000 to 20 does not impress you, think of these numbers as money!

Notice particularly that we achieved this efficiency because of changes caused by rather simple algebraic manipulation of the loop invariant. This should convince you that the loop invariant is a very useful concept.

5.8 Review Top Ten Things to Remember

1. In this chapter, we have considered the *dynamic* behavior of algorithms, as opposed to their *static* structure as seen in Chapter 4. The algorithms which are intended for execution by computers are called *programs*.
2. Algorithms manipulate data which are stored in variables. The concept of data comprises both value and type. A variable is viewed as a labeled box containing a value of some type. In this chapter, types were mainly limited to numbers: Integer and Real Numbers.
3. Actions involving data include assignment, input, output, comparison and various arithmetic operations. These simple actions can be combined to yield complex actions and algorithms.
4. Sequence Forms are quite simple, and can replace some of the other structures. This shows in particular that different structures can have the same behavior.
5. Selection Forms are capable of more complex behavior, as they make it possible to define several alternative paths through the actions of the algorithm. But you should remember that the class of all possible paths through the algorithm is finite. It is possible to compare two Selection Forms and determine equivalent behavior by comparing all paths.
6. Repetition Forms have considerably more complex behaviors. This behavior is best described by a form of tracing that produces a two-dimensional typical path. Tracing yields insights into dynamic behavior by showing a typical trajectory, by helping detect errors, by yielding interesting relations among variables and by suggesting other equivalent algorithms.
7. *Loop invariants* are relations whose truth is unaffected by execution of the body of the loop. They are useful for providing some insights into algorithms. They can also be used to both prove and improve programs.
8. Sub-algorithm forms were introduced by data-flow diagrams describing their behavior. They emphasize higher level views: what is being done, rather than how it is done.
9. The use of sub-algorithms is helpful in simplifying algorithms by hiding the details of what happens in the sub-algorithm; this is known as *abstraction*.
10. This chapter introduced the most fundamental concepts of computing. The following five chapters are simply extensions of these principles.

5.9 Glossary

Character: A data type whose values are textual symbols such as letters, digits, punctuation marks together with other symbols used to control spatial arrangement or text, such as tabs and end of page.

Identifier: A symbolic name that is used to refer to a programming entity such as a sub-program or variable.

Infinite Loop: A repetition form whose condition is never met: the loop keeps looping.

Instruction: A single action in a program.

Iteration: A single execution of the body of a repetition form.

Kludge: [Pronounced “klooj” and said to be from Yiddish *klug* = “smart”.] An attempt to fix a programming error by modification of the symptoms of the error rather than the logical cause for the error. If the fix is successful, the program works, but for the wrong reasons. By extension, applied to a program that has been fixed in this way and is thus devoid of elegance.

Loop invariant: An assertion whose truth is not changed by the *complete* execution of the body of a loop. Thus, if the assertion is true before the loop is executed, it will still be true after the loop has terminated.

Paradigm: A model or template for a design.

Scientific notation: A method of expressing real numbers as a decimal

number x in the range 1.0×10.0 and a multiplier that is some power of 10, for example, the speed of light is 3.10×10^{10} centimeters per second; expressed in many programming languages as 3.0E10.

Side effect: A consequence, frequently unwanted and unexpected, of an action in a program that is not connected with the goal of the action. A common example is the change in the value of a variable within a sub-algorithm resulting from assignments outside the sub-algorithm.

State trace: A trace where the values shown are restricted to as few as possible to show the program’s action.

Trace: A sequence of “snapshots” of a program’s data values showing the effects of the program’s execution.

5.10 Problems

Mid: The Middle Value

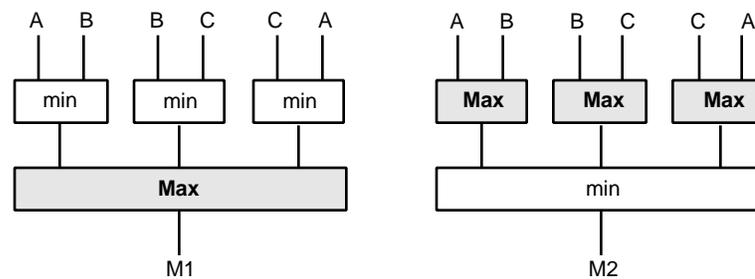
The middle value of an odd number of different values is that value which has as many values lower than it, as it has higher than it. It is not the average value. For example, the middle value of the three integer values 3, 6 and 5 is 5.

There are many ways of finding the middle value; some of these ways follow.

1. Mid Data-Flow Diagrams

Prove (or disprove) that the following data-flow diagrams compute the Mid value of the three input values A, B, C.

Problem 1

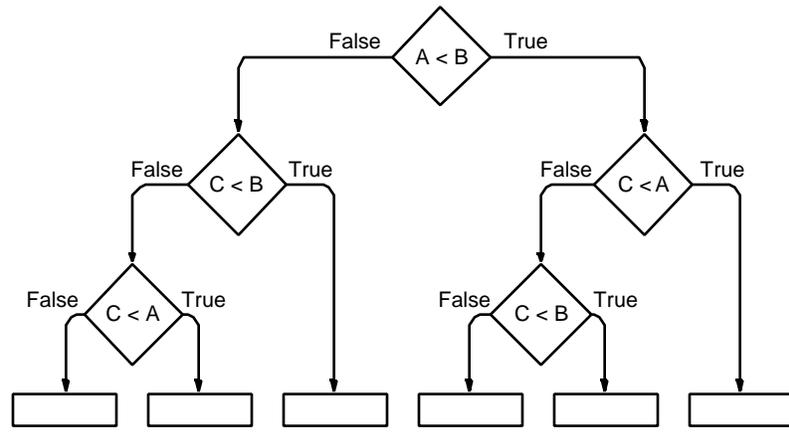


2. Mid Tree

The given tree corresponds to a nasty nest of Selection Forms. Fill out boxes at the right so that M is assigned the middle value of the three values A, B, C.

Show assertions at branches of the tree.

Problem 2



3. More Mids

Create an algorithm (flowblock diagram, pseudocode, and so on) that is simpler than the tree of Problem 2, having fewer nests (only 3 simply nested), but involving more complex conditions.

Create another Mid3 algorithm from flowblocks by using Sum and Difference blocks also.

4. Others

Redo the tree of Problem 1 to find the maximum of three values, and then redo it again to find the minimum of three values. Finally, redo it to sort three values.

How many tests are necessary to prove the equivalence of the Mid of five different values?

How many tests are necessary if some of the five values could be the same?

5. Simple Sequence Form

Indicate briefly what the following sequence of Set statements (A → B is equivalent to Set A to B) does, in general, with inputs A, B and output C. Explain what, not how.

- | | | | | | | | | |
|----|---|-------|----|---|-------|----|---|-------|
| a. | A | A + B | b. | A | A - B | c. | A | A - B |
| | B | A + B | | B | A - B | | B | B - A |
| | C | A + B | | C | A + B | | C | A × B |

6. More Sequences

Describe briefly what the following algorithms do (but not how they do it).

- | | | |
|-----------------|----------------|---------------------|
| a. $A \quad 2$ | b. $E \quad A$ | c. $Z \quad A$ |
| $B \quad A + A$ | $A \quad B$ | $Z \quad B + Z + Z$ |
| $C \quad B + B$ | $B \quad C$ | $Z \quad C + Z + Z$ |
| $D \quad C + C$ | $C \quad D$ | $Z \quad D + Z + Z$ |
| $E \quad D + D$ | $D \quad E$ | $Z \quad E + Z + Z$ |

7. Charge Again

Create an algorithm to compute a charge C , given by the formula:

$$C = 4 \times K + 6 \times A$$

for A adults and K kids, without using multiplication and without any looping. Do this in two ways, using a different number of additions each time. Hint: The charges are Integers.

8. Verification of Sequence

Prove (or disprove) that the following two Sequence algorithms are equivalent in producing the same output E . Draw the data-flow diagrams.

- | | |
|-------------------------|----------------------|
| a. $C \quad A \times A$ | b. $C \quad A + B$ |
| $D \quad B \times B$ | $D \quad A - B$ |
| $E \quad C - D$ | $E \quad C \times D$ |

9. Logical

If the two logical values are represented by the numbers 0 and 1 ($F=0$, $T=1$), then prove the following equivalencies, which are in terms of arithmetic operations:

- a. $\text{NOT}(P) = 1 - P$
- b. $P \text{ AND } Q = P \times Q$
- c. $P \text{ OR } Q = P + Q - P \times Q$

10. Logical Proof

Prove (or disprove) the following logical expressions.

- $\text{NOT}(P \text{ AND } Q) = (\text{NOT } P) \text{ OR } (\text{NOT } Q)$
- $P \text{ AND } (Q \text{ OR } R) = (P \text{ AND } Q) \text{ OR } (P \text{ AND } R)$
- $P \text{ OR } (\text{NOT } P \text{ AND } Q) = P \text{ OR } Q$

Problems on Selection Forms and Verifications**11. Equality of Selection**

Compare the following two algorithms and try (intuitively) to determine if they are equivalent. Then prove, or disprove, this equivalence.

Problem 11

<pre> If P If Q X Else Y Else If R Y Else Z </pre>	<pre> If P and Q X Else If R Y Else Z </pre>
--	--

12. Bigger Equivalence

Prove whether or not the given algorithms are equivalent.

Problem 12

<pre> If P If Q If R W Else X Else X Else If Q If R W Else Y Else Z </pre>	<pre> If Q If R W Else If P X Else Y Else If P X Else Z </pre>
--	--

13. Trade-off Structure for Condition

Create algorithms equivalent to the following by combining conditions with logical operations.

Problem 13

```

If P
  |
  | If Q
  |   X
  |   Else
  |     Y
Else
  |
  | If Q
  |   Y
  |   Else
  |     Z
    
```

```

If P
  X
Else
  |
  | If Q
  |   |
  |   | If R
  |   |   X
  |   |   Else
  |   |     Y
  |   Else
  |     Y
    
```

14. Choices with Integers

Prove (or disprove) equivalence of the following algorithms, assuming non-equal values (i.e. $A \neq B$, and so on...).

Problem 14

```

If B < C
  |
  | If C < A
  |   X
  |   Else
  |     Z
Else
  |
  | If A < B
  |   Y
  |   Else
  |     Y
    
```

```

If C < A
  |
  | If B < C
  |   X
  |   Else
  |     Y
Else
  |
  | If A < B
  |   Y
  |   Else
  |     Z
    
```

15. Many Ways to Grade

Create a Grades algorithm equivalent to the ones previously outlined in Section 5.4, but starting with the first condition being (Percent ≥ 80).

Create another algorithm with the first condition being (Percent ≥ 60).

Indicate a set of values necessary to test this Grading algorithm.

16. Complements

Prove (or disprove) that the given two conditions are opposite (complementary) in behavior.

$(X \geq 30) \text{ AND } ((X > 20) \text{ OR } (X \leq 10))$
 $(X > 10) \text{ AND } ((X > 30) \text{ OR } (X \leq 20))$

17. Test Equivalence

Prove whether or not the following algorithms are equivalent in behavior.

Problem 17

<pre> If A If B X Else Y Else If C Y Else Z </pre>	<pre> If A and B X Else If A or C Y Else Z </pre>	<pre> If A or C If A and B X Else Y Else Z </pre>
--	---	---

Loop Tracing Problems**18. Convert**

Trace the following algorithm for $X = 13$, and describe its general behavior briefly. Assume Integer values.

Problem 18

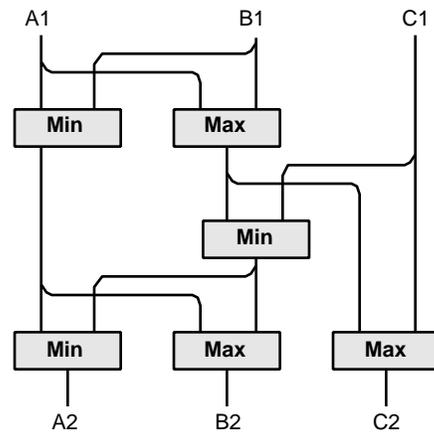
```

Input X
Set Y to Chop X / 2
While X > 0
  |
  | If (Y + Y) = X
  |   Output "0"
  | Else
  |   Output "1"
  | Set X to Y
  | Set Y Chop X / 2
        
```

19. Test Sort

Prove whether or not the following data-flow algorithm sorts variables having unequal values.

Problem 19



20. Pow!

Trace the following algorithm for $X = 13$ and $Y = 2$. What does this algorithm do in general? Choose a better set of names for the variables.

Problem 20

```

Input X
Input Y
Set C to X
Set D to Y
Set P to 1
While C > 0
  If C is odd
    Set P to P × D
    Set C to C - 1
  Else
    Set D to D × D
    Set C to C / 2
Output P
    
```

21. Pi Square

Trace the following algorithm, which computes the square of π , and indicate what formula it computes (but do not work out the arithmetic). Also describe briefly how it works.

Problem 21

```

Set N to 7
Set F to 0
Set P to 0
Set Index to 1
While Index < N
  If F = 0
    Set P to P + 1 / (Index × Index)
    Set F to 1
  Else
    Set P to P - 1 / (Index × Index)
    Set F to 0
  Set Index to Index + 1
Set S to 12 × P

```

22. Refine Prime

A prime number is defined to be any positive integer which is divisible only by 1 and itself.

The following algorithm indicates whether any value X is prime or not. Modify this algorithm in at least two ways, so that it does less looping.

Problem 22

```

Input X
Set P to 0
Set Index to 2
While Index < X
  If Mod(X, Index) = 0
    Set P to 1
  Set Index to Index + 1
If P = 1
  Output "Not Prime"
Else
  Output "Prime"

```

Problems on Creating Loops**23. Square Again**

The square of an integer N can be computed by summing all the integers from 1 to N and then back down to 1. For example:

$$4^2 = 1 + 2 + 3 + 4 + 3 + 2 + 1 = 16$$

Create an algorithm to compute the square of any Integer N using such a method.

24. Power

Create an algorithm to find the Nth power of any number X, where N is a *non-negative Integer*. Then modify this algorithm to work for any integer N.

25. Fibonacci

One simple model of population growth (of rabbits) is given by the series:

1 1 2 3 5 8 13 21 34 55 89 ...

where the new population P at one month is determined by summing the previous two months' populations (called latest L and second latest S). Create an algorithm to determine the population at month M, where $M > 2$.

26. Divide and Conquer

Create an algorithm to divide any integer A by another integer B, yielding a quotient Q and a remainder R. Do this by using multiplication. Do it also another way.

27. Another Signed Product

Create yet another algorithm for the Signed Product algorithm described in Figure 5.48.

28. Extend Change

Extend the Change Maker algorithms (1 through 4 of Section 5.7) to output the count of dimes also.

29. Expo

Create an algorithm to compute the exponential function from the first N terms of the following series:

$$\text{Expo}(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

30. Log

Create an algorithm to compute the natural logarithm function (base e) by the following series approximation (for $0 < x < 2$):

$$\ln(x) = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \frac{(x-1)^4}{4} + \dots$$

31. Sine

Create an algorithm to compute the trigonometric sine from the first N terms of the following series (where angle X is in radians). Recall that the sine is positive in the first two quadrants.

$$\text{Sine}(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \dots$$

Problems on Sub-algorithm Forms

32. Score

In some sporting events, a number of judges each gives a score. The overall score is determined by dropping the highest and lowest scores and averaging the remaining scores. Create a data-flow diagram to determine the score for an event, having five judges, giving 5 grades (each having values ranging from 0 to 10).

33. More Majority

Draw a data-flow diagram for the formula:

$$M = A \times B + A \times C + B \times C - (2 \times A) \times (B \times C)$$

first using only binary functions (having two inputs), then using functions of any number of inputs (i.e., using a four-input product). Prove that this formula behaves as a majority.

34. Big Majority from Little Majs Grows?

The given algorithm below shows four majority units $Maj3$ connected in an attempt to create the majority U of five variables P , Q , R , S , and T . Indicate whether the given algorithm does find the majority of five variables. If it does not do it, then disprove it. Otherwise show how you would prove it.

$$\text{Set } U \text{ to } Maj3(Maj3(P, Q, R), Maj3(R, S, T), Maj3(P, R, T))$$

35. ISBN Flow

Draw a data-flow diagram describing the ISBN algorithm of Chapter 3. The inputs are the nine digits D_1, D_2, \dots, D_9 and the output is the check symbol C .

36. What Is It?

Draw a data-flow diagram for the following formula, and indicate briefly its behavior for all integers.

$$M = (\text{ABS}(A + B) - \text{ABS}(A - B)) / 2$$

37. One More Day

The following formula is supposed to compute the number of days D in any month M . Draw the data-flow diagram of it and show what it does in a tabular form.

$$D = 30 + \text{Mod} (\text{Mod Abs}(2 \times M - 15), 4), 3$$

Loops and Invariance**38. Invariance of Division**

Find the loop invariant of the Divide algorithm, Figure 5.29.

39. Invariance of Square

Create an algorithm to compute the square of any nonnegative integer N by successively adding N for a total of N times. Find the loop invariant.

40. Invariance of Power

Create an algorithm to compute the power of any number X raised to some integer value N , by looping N times and multiplying. Find the loop invariant, and use it to improve this algorithm.

41. Invariance of Pow

Find the loop invariant of the Pow algorithm shown in Problem 20.

42. Invariance of a Cube

The given algorithm computes the cube of any positive integer N . Trace it for $N = 5$. Then indicate which of the following is the loop invariant.

- $C^3 = A + B + C + 6$
- $C = B + D^3$
- $C = B + (A/6)^3$
- $D^3 = A^3 + B^2 + C + 6$
- $6 = A + B + C + D^3$

43. Invariance of Loan

Find the loop invariant of the Loan algorithm of Figure 5.32. You need to introduce more variables, representing some sums.

44. More Invariance

Trace the following algorithms *Cube* and *First*, and find their loop invariants. Find two invariants for *Second*.

Problem 44

<i>Cube</i>	<i>First</i>	<i>Second</i>
<i>Input N</i>	<i>Set B to 50</i>	<i>Set N to 7</i>
<i>Set A to 0</i>	<i>Set J to 1</i>	<i>Set C to 0</i>
<i>Set B to 1</i>	<i>While B < 300</i>	<i>Set S to 1</i>
<i>Set C to 1</i>	<i>Set B to B + B / J</i>	<i>While C < N</i>
<i>Set D to 1</i>	<i>Set J to J + 1</i>	<i>Set C to C + 1</i>
<i>While D < N</i>	<i>End First</i>	<i>Set S to S + C + N</i>
<i>Set A to A + 6</i>		<i>Set N to N - 1</i>
<i>Set B to A + B</i>		<i>Output S</i>
<i>Set C to B + C</i>		<i>End Second</i>
<i>Set D to D + 1</i>		
<i>Output C</i>		
<i>End Cube</i>		

Applications (to business, engineering, and so on...)**45. Loan Again**

Modify the Loan algorithm of Figure 5.38

- so that the interest rate increases by 1% each year,

- b. so that the payment is either 20% of the unpaid balance, or \$25, whichever is greater,
- c. to determine the time to pay off the entire loan and compute also the total interest paid.

46. Checks and Balances

An amount of \$1,000 is deposited every year in a bank account for 5 years, at an interest rate of 10% per year (computed annually), with the interest computed on the present (increasing) balance, but chopped to the next lower dollar amount. Create an algorithm which indicates the balance in the account after 5 years, and trace this algorithm to compute this balance. Check this final balance by using another method.

47. Saving and Withdrawing

An amount of \$10,000 is deposited in a bank account for 5 years, at an interest rate of 10% per year (computed annually) with the interest computed on the present (increasing) balance, but chopped to the next lower dollar amount. Each year \$500 is withdrawn from this account. Create an algorithm which indicates the balance in the account after 5 years, and trace this algorithm to compute this balance. Check this final balance by using another method.

48. Growth (of population, money, and so on...)

The growth of various quantities (money, population, disease) is often a fixed portion of the present quantity. For example, the yearly interest gained on an amount of invested money is given by a fixed rate R of interest, which is multiplied by the present amount A (or balance) each year.

Create an algorithm to determine the number of years required for the money to double. Modify the algorithm to determine the number of years to reach a certain final amount F .

49. Reliability (of systems)

The reliability R of a system of N independent series components, each having Probability P is determined by:

$$R = P \times P \times P \times P \times \dots \times P \times P \text{ (where there are } N \text{ Ps)}$$

For example, consider a chain made of links each having a probability P of .99 of successfully withstanding a certain load. If 70 of these are

connected together ($N = 70$), the probability of all the links successfully withstanding the load is:

$$R = 0.99 \times 0.99 \times 0.99 \times \dots \times 0.99 = 0.50$$

Create an algorithm to determine the number of components N (each having given probability P) to reach a given reliability R .

Chapter 6 Bigger Blocks

In this chapter, we will extend the concepts seen in the previous chapters so that we may obtain more convenient tools. As we mentioned in the review of Chapter 5, there will be fewer fundamental concepts but more details and techniques.

Chapter Overview

6.1	Preview.....	252
6.2	Using External Data and Files.....	252
	End-Of-File Markers.....	255
6.3	More Building Blocks.....	259
	The Select Form.....	259
	The For Form.....	262
6.4	Using the For Forms in Nests.....	265
	Nesting Fors in Fors.....	265
	Nesting Selections in Fors.....	269
	Creating Plots with For Nests.....	273
6.5	More Data Types.....	275
	The Character Type.....	275
	The Logical Type.....	279
6.6	Some General Problem-Solving Algorithms.....	281
	Bisection Finding Values.....	281
	Maximum Power Optimizing Power Output.....	284
	Solver Solving Equations.....	287
6.7	Review Top Ten Things to Remember.....	290
6.8	Glossary.....	291
6.9	Problems.....	292

6.1 Preview

So far, we have limited the data we used in our examples to a few fixed number of values. All of these values were part of the algorithms and stored inside the computer. Here, we will extend the data we will use to include any number of values, as well as data stored outside the computer.

We have already seen several examples of mixed forms, especially involving both repetitions and selections, and in this chapter we will see more of these including slightly more complex algorithms and some useful paradigms and applications.

We will also extend the fundamental Selection form to include a more general Select (or Case) form, where any number of choices can be made. The Repetition form will also be extended to include a For loop (or loop-and-count).

In accordance with our problem-solving method, we will stress, which makes it possible to develop in a number of stages. Each stage is simple because it usually involves only one repetition or selection form. The various stages may lead to deeper nesting of loops and selections.

But the bigger structures introduced in this chapter are not necessarily always better! For example, while the For and Select Forms may sometimes be more convenient than the simpler While and Selection forms, they are not as general and have limitations.

This chapter will help you synthesize a little better what you have learned in the previous chapters, with some emphasis on top-down design. This chapter comprises again a good number of examples that illustrate the various ideas discussed.

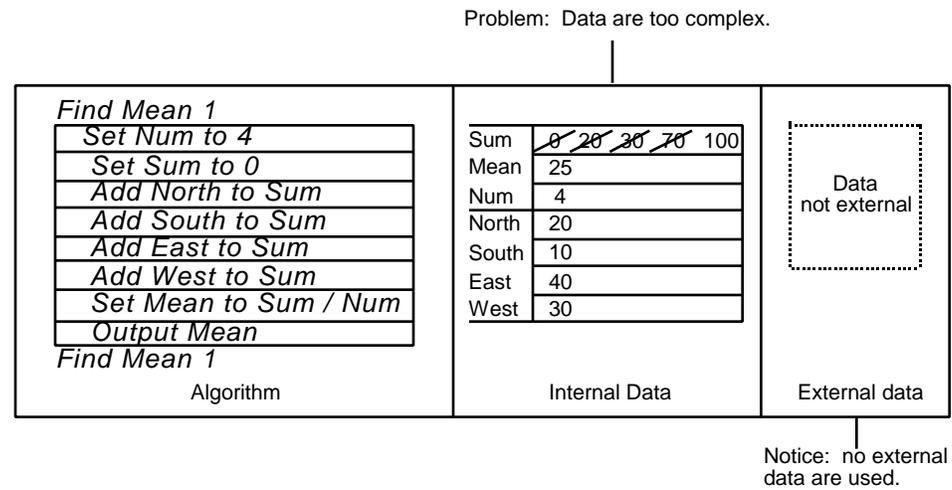
6.2 Using External Data and Files

So far, the data used have been simple values that could be viewed as values in boxes. Also, all of our data manipulations have involved simple variables that were set up by some actions in our algorithms. Occasionally we have input some data from outside the program, but in all our examples, there were only very few values of data.

In real problems, on the other hand, it is often the case that there are large quantities of data. The data are usually stored outside the computer (on a diskette for instance) and brought in only when necessary, one value at a time.

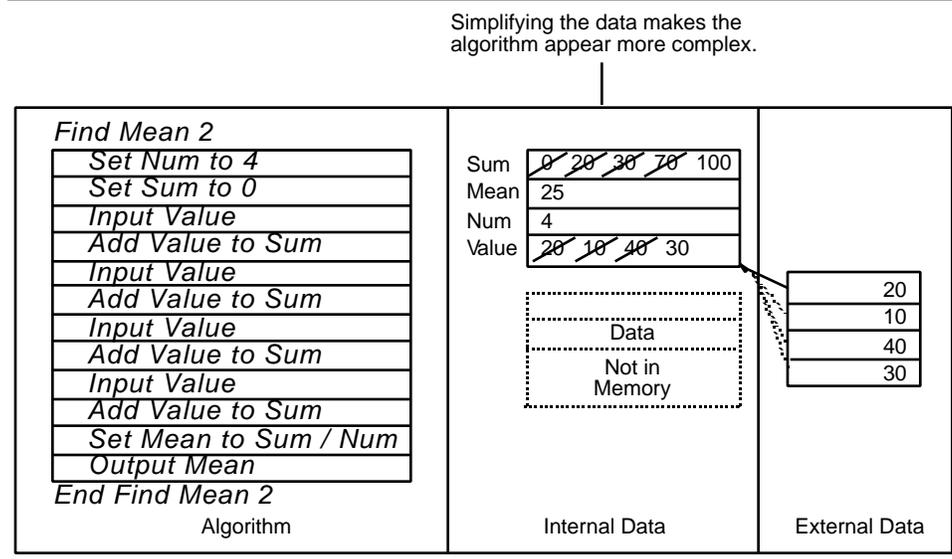
For example, consider the problem of computing some statistics, such as the average (or mean) value of a set of numbers. In Chapter 5, we calculated the mean for four data values *North*, *South*, *East*, and *West* (see Figures 5.10 and 5.11), all internal to the program, as shown again as *Find Mean 1* in Figure 6.1.

Figure 6.1 An Average program Find Mean 1



First, the number of values to be averaged, *Num*, is set to a value of 4 and the variable *Sum* is initialized to a value of zero. Then each value is accumulated into *Sum*. Finally, *Sum* is divided by the value of *Num*, to determine the *Mean* value.

Figure 6.2 Extended Average Find Mean 2

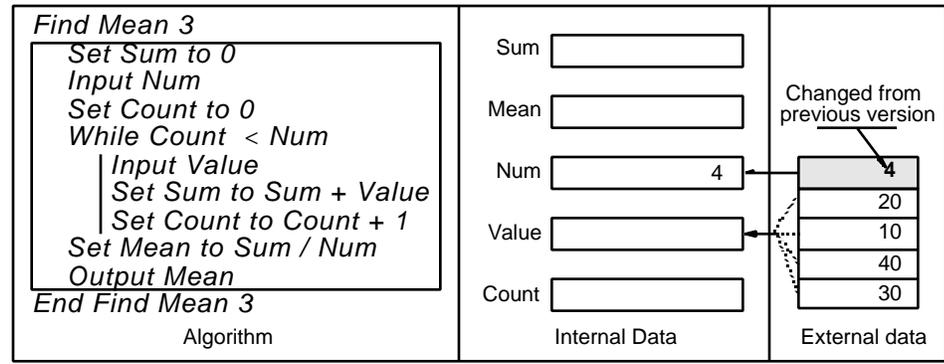


This method can be modified by keeping the data values external, and reading in each of these external values, one at a time, into a single internal variable *Value*, which is added into *Sum*. This method simplifies the internal data structure, but makes the algorithm longer, as shown by *Find Mean 2* in Figure 6.2. Here we are faced with the ever present dilemma in computing; if we

simplify the data, then the algorithm becomes more complex, if we simplify the algorithm then the data becomes more complex.

If we are dealing with only a few values, either of the preceding methods (*Find Mean 1* or *Find Mean 2*) could be used. But if there are many values, it is crucial that we find the best method. The preceding program may be further generalized to average any number of values by first reading in the number of values to be averaged, and then by replacing the series of input-accumulate actions with a loop, as shown by *Find Mean 3* in Figure 6.3.

Figure 6.3 General Find Mean algorithm Find Mean 3



The number of values to average could be stored with the external data as the first item and input into *Num*. The algorithm then includes a variable, *Count*, which starts at 0 and is increased once for each input value, until it reaches the value in *Num* and stops the looping. We will further refine the *Find Mean 3* algorithm in the next section.

You may wish to try to refine the algorithm in Figure 6.3 first before reading any further, for there are a number of ways to refine it.

The external data can be input in many ways. In the early days of computers, the values were punched into cards that were read by card readers. Nowadays, the values may be input from a keyboard, one value at a time. Yet another common method, described below, uses files.

Files are a means of storing data external to the computer on physical devices such as tapes and disks. Many of these devices are based on magnetic phenomena, but these physical aspects are not important to know at this time. In fact, files can be viewed as collections of data. The data values that are stored in a file may easily be retrieved, input to a program, changed, stored and output again to form a modified or updated file. So files can provide another source of input to programs. They can also be used to pass data from one program to another.

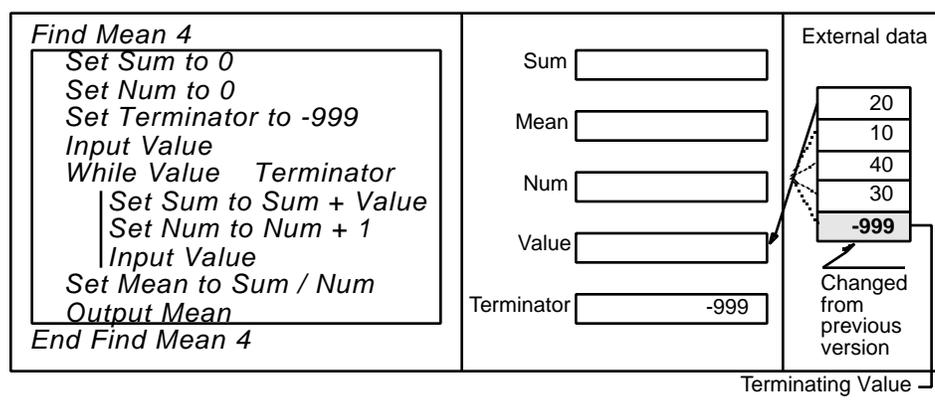
All of this makes it possible to separate the data from the program, thus making the program more general, and applicable to many sets of data.

End-Of-File Markers

Our *Find Mean 3* algorithm can be refined in a number of different ways. We will present and discuss two such refinements that involve different ways of indicating when all the data has been processed.

The problem with the general method of Figure 6.3 is that the value *Num* must be determined externally before the computation begins. If the number of data values is large and a human is to count them, there could easily be an error. Furthermore, humans should not need to do the mundane job of counting when a computer can do the job so much better.

Figure 6.4 Find Mean 3 with a terminating value



The first way of refining *Find Mean 3* is to use the algorithm to count the number of values as they are input, as shown on Figure 6.4. This is done by placing a special “terminating” value after the last input value. The terminating value (or end-of-file marker or sentinel value) is a value that is different from all possible data values. When it is input, it causes the looping to stop. Of course, this terminating value is not counted or included in the average.

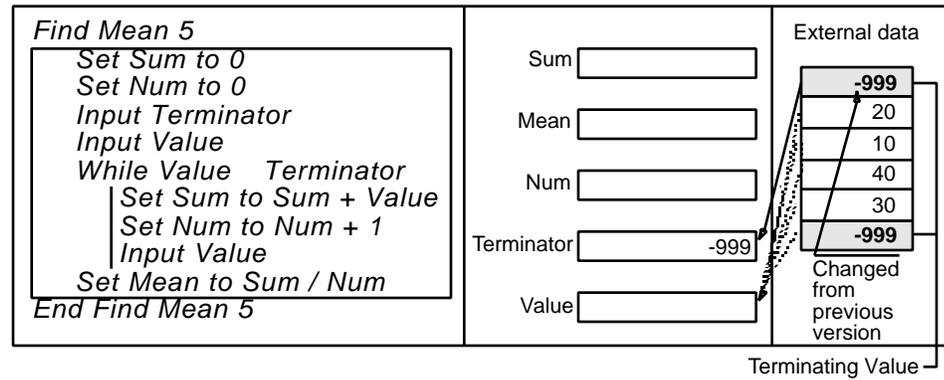
The value used to mark the end-of-file depends on the data because it must be distinct. For example, if all values were percentages (ranging from 0 to 100), then a terminating value could be any number outside of this range (say 101 or 500). Similarly, if all values were positive numbers, then any negative number could be used as a terminating value. Sometimes a computer supplies its own end-of-file marker, often called EOF.

Note: It is often useful to choose an unusual terminating value that is simple to remember, and easy to spot visually, such as -999.

A second refinement of *Find Mean 3* is a generalization of our first refinement. The first method used an end-of-file marker whose value was a constant written in the program. This program was not general, since it did not work for all the possible values.

If the nature of the data changed so that the value -999 were now a legitimate data value, another terminator would have to be chosen and the program in Figure 6.4 would have to be modified. To avoid modifying the program, the second refinement, shown in Figure 6.5, provides the terminator as the first piece of data and tests for its occurrence as the last piece.

Figure 6.5 Find Mean 3 with sandwich



The two terminator values in Figure 6.5 “sandwich” the data. Now the *Find Mean 5* program averages any kind of numerical values except for the value entered as the terminator. terminating “sandwich”

Note: Putting the terminating value at both the beginning and the end of the external data list makes the program easier to maintain. When this value is changed, only the data list must be changed.

Program reusability is the main reason for selecting the last refinement. It allows us to view the program as a black box. When there are some changes to the data ranges, we only need to make changes to the data and not the program; we can use our program as is on the new data.

As we have seen so far, many algorithms can produce the same results. However, some of these algorithms can be more easily extended than others, which makes them more interesting to use. For example, there are two ways to find the maximum value of two variables A and B, as shown in Pseudocode 6.1.

Pseudocode 6.1 Two ways to find the maximum of two values

The two algorithms Maximum 1 and Maximum 2 were introduced in Figure 5.14.

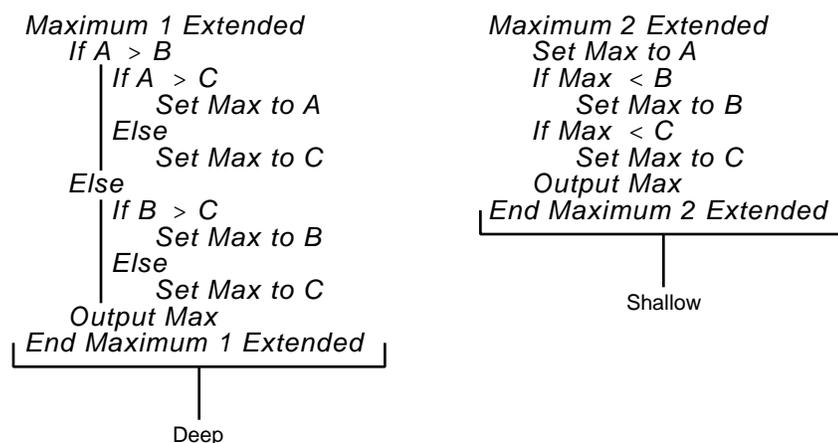
<pre> Maximum 1 If A > B Set Max to A Else Set Max to B Output Max End Maximum 1 </pre>	<pre> Maximum 2 Set Max to A If A < B Set Max to B Output Max End Maximum 2 </pre>
--	---

The two algorithms in Pseudocode 6.1 are explained as follows:

- *Maximum 1* compares the two values directly and assigns the maximum value to *Max* immediately.
- *Maximum 2* assigns one of the values to be the maximum, then checks this choice, and if the decision was wrong, it changes the maximum to be the other value. Although this second method may seem more complex and unnatural, it will turn out be easier to modify and extend.

To demonstrate this, let’s extend both algorithms shown above (*Maximum 1* and *Maximum 2*) such that they now find the maximum of three values: A, B, and C. In both cases, this additional value C means that more selections must be added to the algorithms.

Pseudocode 6.2 Two ways to find the maximum of three values

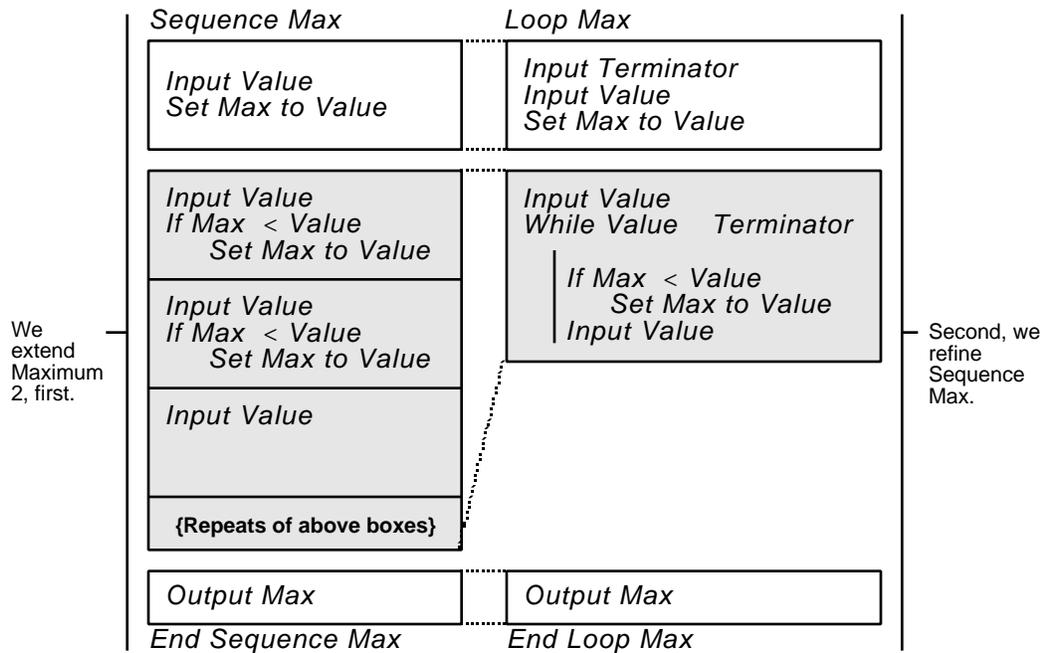


The extended algorithms in Pseudocode 6.2 are explained as follows:

- To extend *Maximum 1*, we must add nested selections. The more values we wish to find the maximum of, the deeper this nesting of selections will be.
- To extend *Maximum 2*, the new selections are added in series with the others. Similarly, the more values we wish to find the maximum of, the more selections there will be added one after the other. Hence, *Maximum 2 Extended* could be referred to as the “shallow” one, since no nesting was involved. By the same token, *Maximum 1 Extended* could be called the “deep” one because of its deep nesting. You will certainly agree that the shallow algorithm of *Maximum 2* was the easier to extend.

We could easily extend *Maximum 2 Extended* even further so that it finds the maximum of any number of values. To do this, the necessary number of selections would be added in series; this is shown at the left of Figure 6.6. This long series of selections and inputs can easily be replaced by a loop, giving the *Loop Max* algorithm at the right of Figure 6.6.

Figure 6.6 A long Sequence Max and a Loop Max



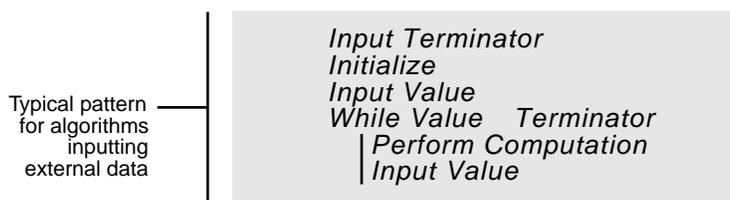
The first thing *Loop Max* does is read in the terminating value. Then, the first of the numbers to be compared is input and is immediately assigned to be the maximum, *Max*. Once this is done, *Loop Max* successively inputs values and compares each value to the maximum, *Max*, updating it if necessary, as long as the input value is not the terminating value. Finally the maximum is output.

Notice that this algorithm assumes that at least one value is given between the terminating values. If it is possible that no values be given (other than the two terminating values), then the algorithm must be modified by testing before the first assignment to *Max*.

This *Loop Max* algorithm could be extended in many ways. For example, it could find not only the largest but also the second largest value. Try it! It could also be extended to find the minimum value, to count the number of values, to sum all the values, to compute averages, to give running averages, variances, ranges, the number of values greater than a given value, as well as perform many other computations.

Each of these computations usually involves an initialization of values, then some extension to the body and finally some output at the end. You may recognize this pattern for it was used in both *Loop Max* and *Find Mean 5*. This pattern is illustrated in Pseudocode 6.3.

Pseudocode 6.3 Input pattern used in many algorithms



These computations may also be done using arrays, as we will see in Chapter 8.

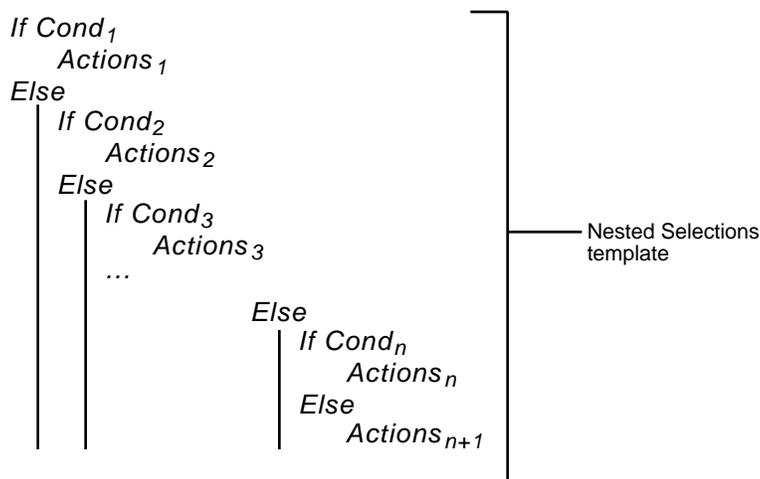
6.3 More Building Blocks

The Select Form

We have already pointed out that the Four Fundamental Forms (Sequence, Selection, Repetition and Invocation) are sufficient to create all algorithms. However, other forms may also be used, like the Select Form, which is a convenient way of expressing multiple choice selections.

The Select Form (sometimes called the Case form) is a natural extension of the Selection form, where, instead of selecting from only two alternatives, there may be any number of choices nicely nested within one another as shown in Pseudocode 6.4.

Pseudocode 6.4 Cumbersome nested Selection Forms



The n conditions shown are tested in order ($Cond_1$ then $Cond_2$, and so on) until the first condition that holds is found, and the corresponding set of actions is performed. If none of the conditions from $Cond_1$ to $Cond_n$ are True, then all the actions represented by $Actions_{n+1}$ are performed.

Notice that the pseudocode to achieve this commonly required operation is cumbersome and, because of the way in which we indent the True and False parts of a selection, moves rapidly to the right of the written form.

Since the ability to select one set of actions out of many is so useful, a special form, the *Select Form*, was developed. Using the notation of this form, Pseudocode 6.5 shows how Pseudocode 6.4 would appear.

Pseudocode 6.5 The Select Form

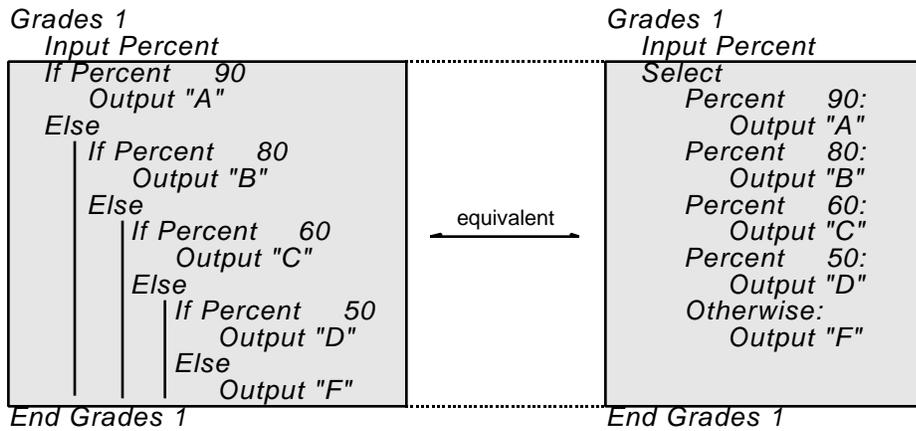
```

Select
  Cond1  Actions1
  Cond2  Actions2
  Cond3  Actions3
  ...
  Condn  Actionsn
  Otherwise  Actionsn+1
    
```

The Grades 1 algorithm was first introduced in Chapter 5, Pseudocode 5.9.

The form in Pseudocode 6.5 is much simpler to understand than the Pseudocode 6.4 which uses only Selection Forms. To fully illustrate the difference between these two forms, Figure 6.7 shows the *Grades 1* algorithm expressed as a nested set of Selection forms (left) and as a single Select Form (right). This figure also shows how the value of *Percent* selects the grade for output.

Figure 6.7 Comparison of two representations of the Grades algorithm



Notice that, as with the nested Selection template (Pseudocode 6.4), once a condition has been satisfied, we are not limited to a single action as *Actions_n* stands for one or a group of actions. For example, if *Grades 1* were also required to count the number of As, Bs, Cs, and so on, the algorithm in Pseudocode 6.6 could be used.

Pseudocode 6.6 Select Form with many actions

```

Input Percent
Select
  Percent 90:
    | Output "A"
    | Increment CountA ] Group of actions
  Percent 80:
    | Output "B"
    | Increment CountB
  Percent 60:
    | Output "C"
    | Increment CountC
  Percent 50:
    | Output "D"
    | Increment CountD
  Otherwise:
    | Output "F"
    | Increment CountF
    
```

Let's illustrate further the Select Form with an algorithm for determining the unit price, depending upon the quantity ordered, shown in Pseudocode 6.7.

Pseudocode 6.7 Algorithm Price, showing special case of Select Form

<pre> Price Input Quantity Select Quantity = 1: Set Price to 99 Quantity = 2: Set Price to 98 Quantity = 3: Set Price to 98 Quantity = 4: Set Price to 95 Quantity = 5: Set Price to 95 Quantity = 6: Set Price to 95 Quantity = 7: Set Price to 90 Quantity = 8: Set Price to 90 Quantity = 9: Set Price to 90 Otherwise: Set Price to 85 Output Price End Price </pre>	<pre> Price Input Quantity Select Quantity 1: Set Price to 99 2, 3: Set Price to 98 4, 5, 6: Set Price to 95 7, 8, 9: Set Price to 90 Otherwise: Set Price to 85 Output Price End Price </pre>
--	--

In this algorithm (Pseudocode 6.7), the conditions are all simple comparisons of a variable to various constants. This allows us to express the algorithm in a shorter variant of the Select Form known as the Case Form, shown on the right.

In the Case Form, we name the variable being used for the comparison at the head of the form and the constant values are listed, separated by commas, as the conditions. The Case Form is a special instance of the Select form where the conditions test for equality to constants.

The For Form

The only Repetition forms considered so far have been the While and the Repeat-Until forms. Of these two, the While is the more fundamental, and, as we saw in Chapter 5, no other loop form is necessary. There is, however, one other loop form that is useful or convenient at times: the For loop.

For loop forms, sometimes called Loop-and-Count forms, are one of the most useful extensions of the Four Fundamental Forms. A For loop is used in all cases where the number of repetitions is known, and usually replaces a While loop where the condition involves a counter. The For loop specifies the initial value of that counter, the value by which it is incremented during each loop, and the limit that the counter must reach to stop the loop. Using a While loop, it is necessary to explicitly set up the counter's initial value, to test the counter, and to increment it. On the other hand, the For loop does all that implicitly.

Figure 6.8 Use of the For loop to calculate a product

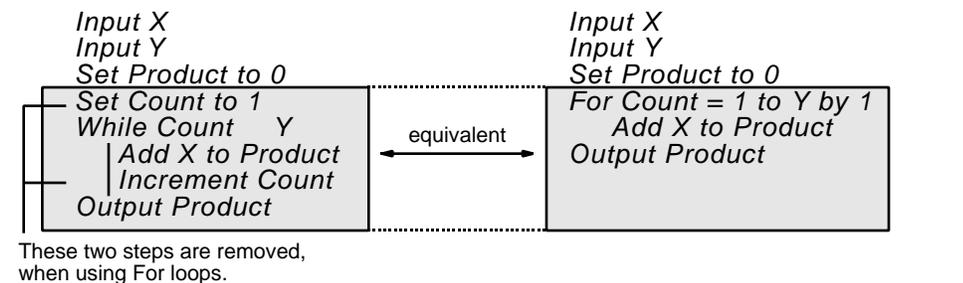


Figure 6.8 illustrates the difference between While and For loops by showing a variation of one of the algorithms for calculating a product that we studied in Chapter 5. Here, the product $X \times Y$ is calculated by summing values of X , Y times. On the left of Figure 6.8, a standard While loop is used while on the right the same algorithm is expressed using the For loop.

This new notation decreases the complexity of some algorithms since the full counting mechanism is specified in the statement at the head of the loop instead of having it stated in three separate statements as illustrated in Pseudocode 6.8.

Pseudocode 6.8 The full counting mechanism in While loops

```

Set Count to initial value
While Count ≤ final value
  ...
  Increment Count
    
```

Although the full counting mechanism is easily discernible in Pseudocode 6.8, it might not be as visible in a large example. The For form allows us to think in terms of larger blocks, so making our algorithms look smaller and more intellectually manageable.

Pseudocode 6.9 The Factorial algorithm with a While loop and with a For loop

<pre> Factorial Input N Set Factorial to 1 Set Count to 1 While Count ≤ N Set Factorial to Factorial × Count Increment Count Output Factorial End Factorial </pre>	<pre> Factorial Input N Set Factorial to 1 For Count = 1 to N by 1 Set Factorial to Factorial × Count Output Factorial End Factorial </pre>
--	---

In Chapter 5 we saw an algorithm to calculate the factorial of N , which is shown again at the left of Pseudocode 6.9 while on the right, the same algorithm is shown using the For loop. It is clear that the hiding of the counting mechanism in this version reduces its complexity.

Let's take another algorithm from Chapter 5 and transform it so that it uses a For loop. Pseudocode 6.10 shows the *Odd Square* algorithm that computes the square of an integer Num by summing the first Num odd integers. This is done by starting counter $OddNum$ at one and looping with a step size of two, adding this odd counter value to $Square$ during each repetition.

Pseudocode 6.10 The Odd Square algorithm revisited

<pre> Square Input Num Set Square to 0 Set OddNum to 1 While OddNum ≤ (Num + Num) Set Square to Square + OddNum Set OddNum to OddNum + 2 Output Square End Square </pre>	<pre> Square Input Num Set Square to 0 For OddNum = 1 to Num + Num by 2 Set Square to Square + OddNum Output Square End Square </pre>
--	---

The For form is so powerful that it is used very often, but it is often also misused for it requires a counter and such a counter may not be necessary or natural for some applications!

Finally, the following two pseudocode fragments compare the pseudocode forms for the While loop and the For loop (Pseudocode 6.11).

Pseudocode 6.11 Comparing While and For loops

While	For
<pre>Set Count to First While Count <= Last Body Set Count to Count + Step</pre>	<pre>For Count = First to Last by Step Body</pre>

There are many limitations of the For form, which may make this form unsuitable in some cases. For example, in most programming languages, the initial value and the final value of the counter, which could both be given as expressions, are evaluated only once on entry to this form, and thus should not be changed inside the loop. Also, the counter should not be modified in the body of the loop, since this would change the number of repetitions. Additionally, in some programming languages, the initial, final and increment values may not be zero or negative or of Real Number type.

Finally, after the loop terminates, the value of the loop counter (or loop control variable) may be undefined! The While loop has no such restrictions, so you may always use it in cases where these restrictions would cause problems to a For loop.

Note the While loop is more general than the For loop. When in doubt, you can always use the While loop.

6.4 Using the For Forms in Nests

Nesting Fors in Fors

The nesting of Repetitions (loops nested within loops) is very common and useful. However, if too much of the looping mechanism is visible, this nesting may seem very confusing. To see this, we will look at an example first expressed with nested While loops and then with nested For loops.

See Figures 3.6 and 3.15 for the verbal Charge algorithm along with its table of charges.

Pseudocode 6.12 shows two forms of an algorithm with doubly nested counting loops. It is a version of an algorithm we have discussed in various forms in Chapter 3, and that computes the total admission *Charge* if *Adults* pay three dollars each and *Kids* pay two dollars each. The algorithm creates a table of charges for one to two adults and zero to three kids. On the top of the figure, is a version based on While loops. Because it is written using nested While forms with all the counting mechanisms visible, the algorithm appears complex.

Pseudocode 6.12 Nested Loops

Using While Loops

```

Set Adults to 1
While Adults = 2
  Set Minors to 0
  While Minors = 3
    Set Charge to  $3 \times \text{Adults} + 2 \times \text{Minors}$ 
    Output Charge
    Increment Minors
  Increment Adults

```

Using For Loops

```

For Adults = 1 to 2 by 1
  For Minors = 0 to 3 by 1
    Set Charge to  $3 \times \text{Adults} + 2 \times \text{Minors}$ 
    Output Charge
  
```

} Notice how the same algorithm is greatly simplified by using For loops.

On the bottom of Pseudocode 6.12, the same algorithm has been rewritten using For loops. Notice that this notation hides most of the details. This version, consisting of only *three parts*, two loop headers and a body, is much simpler than the original one that has *seven parts* (two initializations, two tests, two increments, and a body). We will prove that with the traces of the two algorithms.

Figure 6.9 Detailed trace of doubly-nested While loops

Adults =	1	1	1	1	1	2	2	2	2	2	3
Adults = 2	T	T	T	T	T	T	T	T	T	T	F
Kids =	0	1	2	3	4	0	1	2	3	4	
Kids = 3	T	T	T	T	F	T	T	T	T	F	
Charge =	3	5	7	9		6	8	10	12		
	3	5	7	9		6	8	10	12		
Kids' =	1	2	3	4		1	2	3	4		
Adults' =					2						3

```

Set Adults to 1
While Adults = 2
  Set Kids to 0
  While Kids = 3
    Set Charge to  $3 \times \text{Adults} + 2 \times \text{Kids}$ 
    Output Charge
    Increment Kids
  Increment Adults

```

The first trace, shown on Figure 6.9, illustrates the execution of the algorithm at the top of Pseudocode 6.12. The large amount of detail increases the complexity of the trace, and this can be avoided, as we will see in the next figure. In a trace all of the detail is needed for understanding how the

algorithm works; it is also important for tracing extremely complex nested algorithms. However, most nesting of loops is quite simple, even with three or four levels of nesting—provided it can be viewed properly.

Figure 6.10 Simple trace of doubly-nested For loops

Adults =	1	1	1	1		2	2	2	2	
Kids =	0	1	2	3		0	1	2	3	
Charge =	3	5	7	9		6	8	10	12	
	3	5	7	9		6	8	10	12	

Outer loop control variable

Inner loop control variable

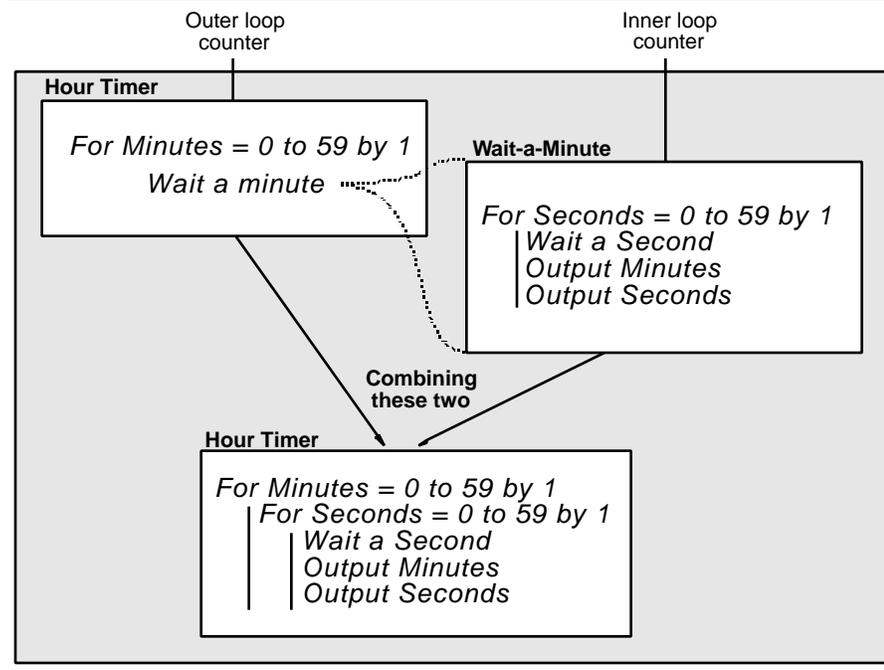
```

For Adults = 1 to 2 by 1
  For Kids = 0 to 3 by 1
    Set Charge to 3 × Adults + 2 × Kids
    Output Charge
  
```

In Figure 6.10 we can see the trace of the algorithm at the bottom of Pseudocode 6.12. The two traces in Figures 6.9 and 6.10 are traces of equivalent algorithms. However, we can see that the second trace is much simpler than the first one. It simply lists all combinations of the loop control variables. The inner loop control variable, *Kids*, changes more rapidly compared to the outer loop control variable, *Adults*. The resulting *Charge* is also listed completing the entire table of charges. Because the details are all hidden, the trace is much simpler!

Many algorithms involve loops directly nested within others, like the Charge algorithm. Such algorithms have a similar behavior; they cycle through all combinations, like a big counter, as illustrated by the following examples. This cycling could be compared to that of a digital clock or a mileage odometer.

Let's develop a timer, using a top-down approach, as shown on Figure 6.11. The minutes counter loops from 0 to 59, and for each minute, the seconds counter also loops from 0 to 59. The body of the seconds loop consists of a delay of one second and a display of the minutes counter and of the seconds counter. Notice that the inner loop counter (seconds) changes faster than the outer loop counter (minutes). At the bottom of the figure, the nested loops are combined and shown in a single piece of pseudocode.

Figure 6.11 An hour timer

A decimal counter can be implemented by a similar piece of pseudocode as shown in Pseudocode 6.13.

Pseudocode 6.13 Nest For forms for a decimal counter

```

For Tens = 0 to 9 by 1
  For Units = 0 to 9 by 1
    Output Tens
    Output Units
  
```

Pseudocode 6.13 outputs the following sequence of values:

00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12, ... 89, 90, ... 99.

Pseudocode 6.14 shows another similar piece of pseudocode.

Pseudocode 6.14 Nested For forms for die combinations

```

For Dice1 = 1 to 6 by 1
  For Dice2 = 1 to 6 by 1
    Output Dice1
    Output Dice2
  
```

Pseudocode 6.14 lists all 36 possible dice combinations in order:

11, 12, 13, 14, 15, 16, 21, 22, 23, 24, 25, 26,

31, 32, 33, 34, 35, 36, ..., 46, ..., 56, ... 66.

Pseudocode 6.15 shows the algorithm for a Binary Counter.

Pseudocode 6.15 Binary counter

```

For Bit1 = 0 to 1 by 1
  For Bit2 = 0 to 1 by 1
    For Bit3 = 0 to 1 by 1
      Output Bit1
      Output Bit2
      Output Bit3
  
```

Pseudocode 6.15 shows how binary numbers can be generated in increasing order. Its output is as follows:

000, 001, 010, 011, 100, 101, 110, 111.

This Binary Counter algorithm could be used to generate all of the possible combinations to test the Majority algorithms seen in Chapters 3 and 5.

Going one level deeper, Pseudocode 6.16 illustrates the For forms for a mileage odometer.

Pseudocode 6.16 Nested For forms for a mileage odometer

```

For Hundreds = 0 to 9 by 1
  For Tens = 0 to 9 by 1
    For Units = 0 to 9 by 1
      For Tenths = 0 to 9 by 1
        Output Hundreds
        Output Tens
        Output Units
        Output '.'
        Output Tenths
      
```

The outermost counter changes most slowly.

The innermost counter changes most quickly.

Pseudocode 6.16 shows how four nested loops generate all decimal values from 000.0 to 999.9. Notice the order in which the values of the variables change. *Hundreds*, the control variable of the outermost loop, changes the slowest while *Tenths*, the control variable of the innermost loop changes the most rapidly, just as they do on the odometer of an actual car.

Finally, Pseudocode 6.17 shows the Clock algorithm.

Pseudocode 6.17 The Clock algorithm

```

While True
  For Hours = 0 to 23 by 1
    For Minutes = 0 to 59 by 1
      For Seconds = 0 to 59 by 1
        Wait a second
        Output Hours
        Output Minutes
        Output Seconds
      
```

Note that this While loop condition is always True.

Pseudocode 6.17 simulates a 24-hour clock that works perpetually, using a condition for the While loop that is always true. This is one of the few occasions where an infinite loop happens by design instead of by error.

Those combinations could then be tabulated, as seen in Figure 5.24.

The nesting of many such loops is convenient and can be elegant, but not all nestings are so well structured. In the following section we will look at more examples of nested forms.

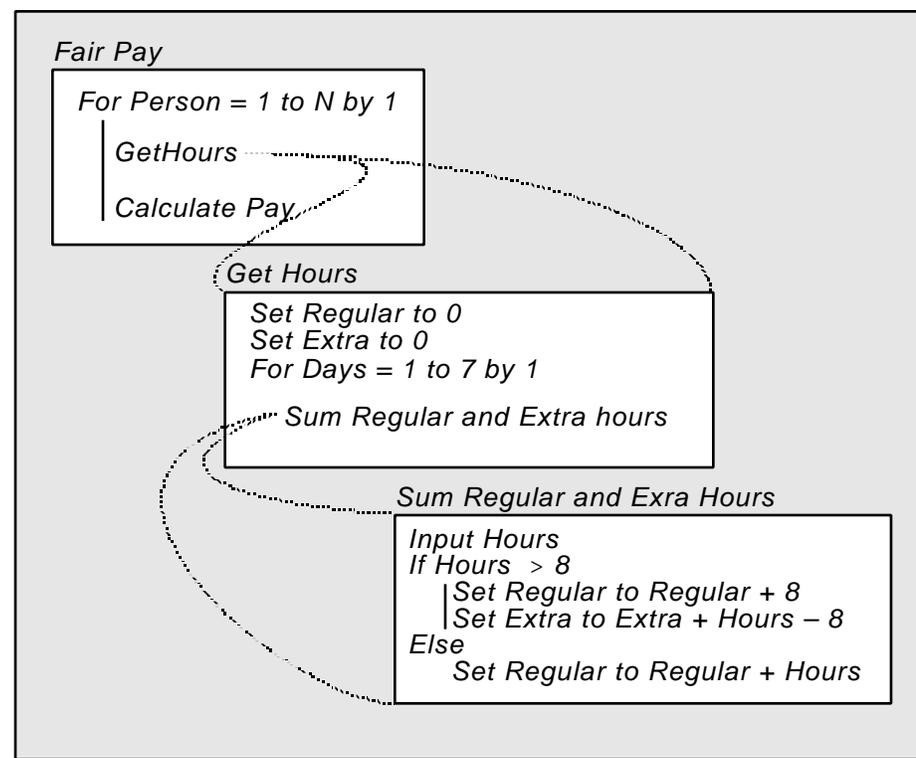
Nesting Selections in Fors

Most programs involving nested loops are not as simple as the previous examples of nested loops. We will consider now more general nestings of loops with the idea that the creation of complex loop structures is easiest when done top-down in stages.

This approach is in accordance with steps 2 and 3 of our problem-solving method, so that each stage only shows one loop.

See Chapter 2, Section 2.2 for more details on this problem-solving method.

Figure 6.12 Development of the Fair Pay algorithm



The development of the Fair Pay algorithm, in Figure 6.12, shows the nesting of two Loops and a Selection, done in three stages. It describes the solution to a pay problem where overtime is paid for all hours over eight worked in a day. The first stage (or top level) consists of a single loop repeated for each *Person*; first *Get Hours* is invoked and then the pay is calculated.

The second level, *Get Hours*, is then considered by itself, looping over the seven days of the week to accumulate the number of hours (both regular hours and

extra hours). The accumulation of these two sums is then broken out at yet another level, as a Selection form.

Similarly *Calculate Pay* could be broken down further. If all of these stages were combined into one large stage, then the nesting might appear complex. However, viewing them as stages, as done in Figure 6.12, makes the whole algorithm seem simpler.

Figure 6.13 Production data

1990	dept 1	person 1	30
		person 2	40
	dept 2	person 1	40
		... 2	20
	dept 3	... 3	30
		person 1	40
1991	dept 1	... 2	40
		person 1	60
	dept 2	person 1	10
		... 2	30
	dept 3	... 3	20
		... 3	20
		person 4	40

Our next example is an algorithm that analyzes the manufacturing of items made by the employees of a company over some years. The production data (count of items) is shown in Figure 6.13, where two years and three departments with varying numbers of employees are involved. The main goal of the Production algorithm is to find the department with the maximum total production for each year.

Figure 6.14 A typical input/output for the Production algorithm

```

FOR YEAR 1990
  DEPT 1
    HOW MANY PEOPLE? 2
    ENTER PRODUCTION 30 40
  DEPT 2
    HOW MANY PEOPLE? 3
    ENTER PRODUCTION 40 20 30
  DEPT 3
    HOW MANY PEOPLE? 2
    ENTER PRODUCTION 40 40
FOR YEAR 1990...
DEPT 2 HAS MAX PRODUCTION OF 90

```

Notice that this example (Figure 6.13) involves four very different entities: years, departments, employees and production. When four such diverse entities are mixed in an algorithm, the result could be great confusion. Creating the program in stages will help us minimize this confusion.

Figure 6.14 shows a trace of the input of the production data for a given year, and of the corresponding result. Notice its structured form (with indentation),

which reflects the structure of the data. Actually such large amounts of data would not usually be input in such a “conversational” mode from where the computer prompts you, but from a data file stored on tapes or disks.

Figure 6.15 Top-down development of Production algorithm

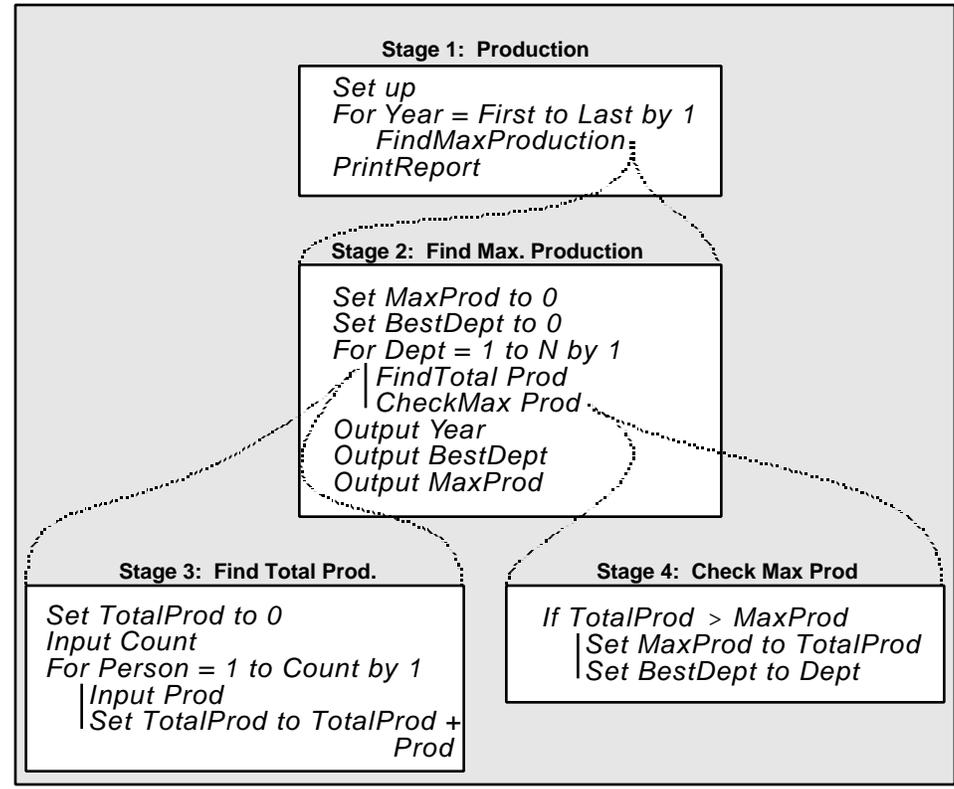


Figure 6.15 shows a three-stage top-down development of our Production algorithm that analyzes the data just described.

- The first stage shows only a setup, a looping from the *First* year to the *Last* year, followed by a report of results. Within this first loop, the sub-algorithm *FindMaxProduction* is invoked.
- This sub-algorithm is further refined in Stage 2, where we show how the maximum total production, *MaxProd*, is computed by looping over all departments from 1 to N. Within this loop, we invoke *FindTotal* to find the total production, *TotalProd*, of each department. Then, invoking *CheckMax.*, we compare it to *MaxProd* (replacing the values of *MaxProd* and *BestDept* when necessary). After this, the maximum production and the best department of that year are output.
- Stage 3 refines *FindTotal* by looping and accumulating the production, *Prod*, of each *Person* of the *Count* employees in the department. Stage 3 also refines *CheckMax*, using a simple selection.

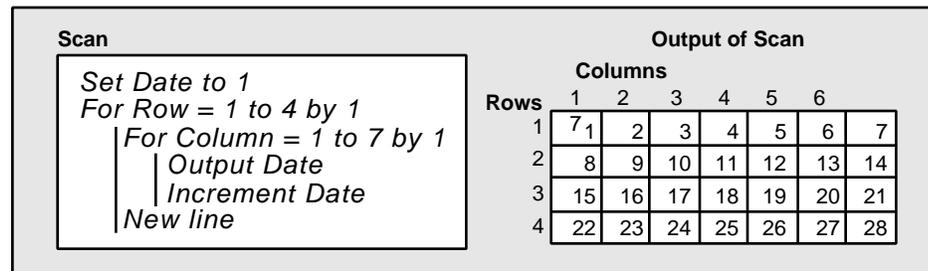
These three stages could be “pushed together” into a single big algorithm, but this would hide its basic simplicity. Also, when an algorithm is kept in the form of sub-algorithms, it is simpler to modify. For example, we could modify *Production* to find the maximum departmental average production per person by simply comparing the ratio *TotalProd/Count* in the Selection form of *CheckMax*. Similarly, we could count the total number of people, accumulate the total production and compute the best average production per person over all the years (not just for each year). Such modifications are often made after a program is written, so it helps to write the program keeping possible modifications in mind!

Creating Plots with For Nests

Doubly nested loops are very convenient for two-dimensional output of tables, grids, calendars, graphs and plots.

For example, the algorithm shown in Figure 6.16 moves and prints in the pattern that many people use for reading or scanning: left to right along a row and proceeding downwards. This *Scan* algorithm prints consecutive numbers as it scans this grid pattern for four rows and seven columns. Its output could be viewed as a simple calendar model.

Figure 6.16 Scan algorithm and its output

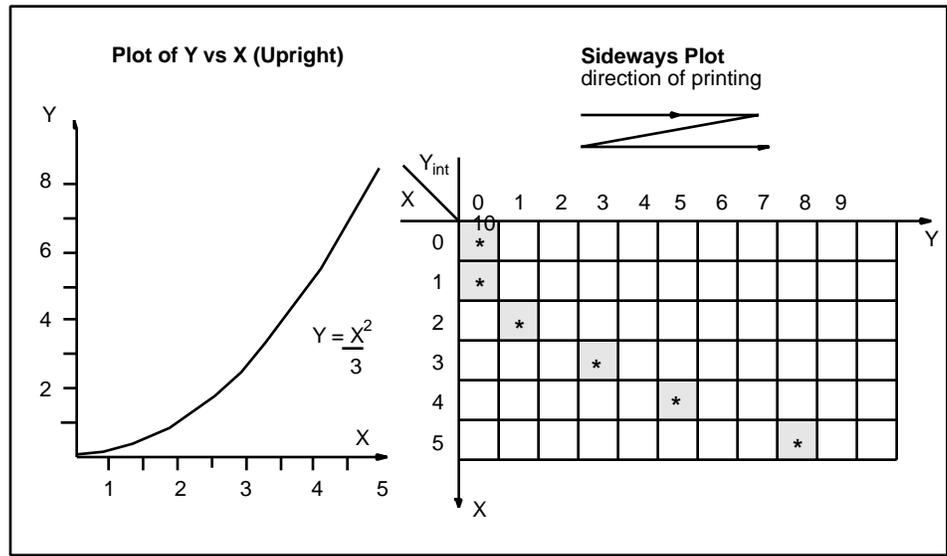


In a similar fashion, two-dimensional plots of a function such as

$$y = \frac{x^2}{3}$$

can be created by printing marks on a grid as shown at the left of Figure 6.17. In practice, using screens or printers, it is more convenient to plot this on its side as shown at the right of the figure.

Figure 6.17 Plot of Y vs. X



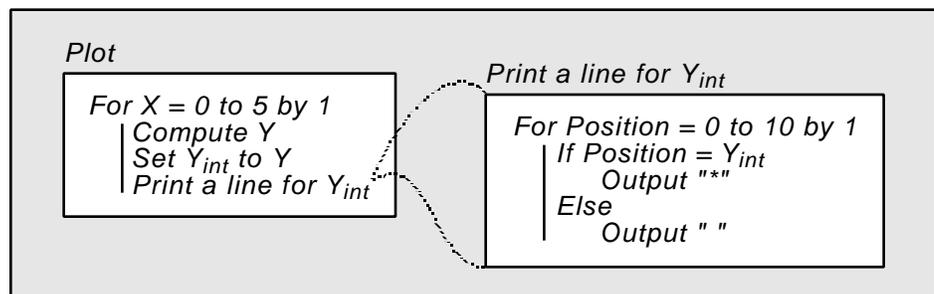
In this sideways plot, each row contains only one mark (an asterisk * in this case). We will view the printer head (or cursor) as “advancing” row by row from top to bottom, and in each row from left to right. You may note that the asterisks are placed into a grid of X versus Y_{int} (and not Y), where Y_{int} represents the integer portion of Y.

Figure 6.18 Sample values

X	1	2	3	4	5
Y	$\frac{1}{3}$	$\frac{4}{3}$	3	$\frac{16}{3}$	$\frac{25}{3}$
Y_{int}	0	1	3	5	8

The table of Figure 6.18 shows how the values of the output Y are rounded into integers Y_{int} .

Figure 6.19 Algorithm for plotting a graph

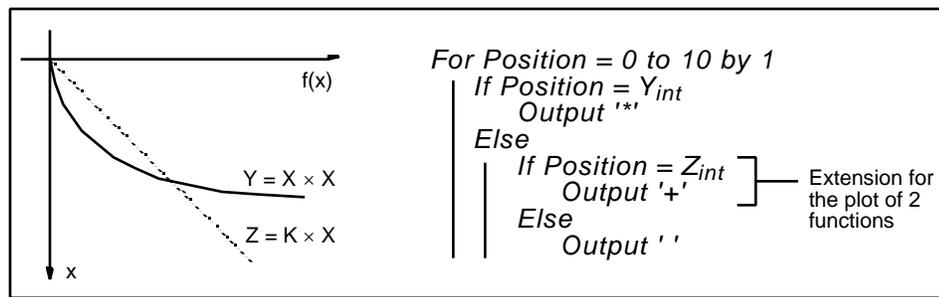


On the left of Figure 6.19 is the top view of the graph-plotting algorithm. It loops through the various values of X , first computing Y , then rounding it to Y_{int} and then printing a line corresponding to that value.

The sub-algorithm that prints a line is then refined and shown at the right of the figure. The printing head advances, stepping $Position$ from 0 to 10. If $Position$ equals Y_{int} , then a mark is output, otherwise a blank is output.

Such a plot may be extended by displaying headers, marking the axes, and even “scaling” any given values to fit onto a page or a screen. It can also be extended to plot two functions as shown in Figure 6.20. Here, the Plot algorithm is modified to compute a second value Z_{int} , and the *Print a Line* sub-algorithm is modified to output either an asterisk for the first function, a plus for the second function or a blank. Further extensions to plot three or more functions are done in a similar manner.

Figure 6.20 An extended plot of two graphs



6.5 More Data Types

The Character Type

Until now, our algorithms have manipulated only numeric values that are of type Integer or Real Number. We have also already mentioned other common data types including Character, Logical and String. Here, we will explain in more detail the Character and Logical data types.

The Character data type includes values that represent all the symbols on a computer keyboard. This includes all letters (upper and lower case), digits, punctuation marks, brackets and other miscellaneous symbols (% , \$, and so on). As we have seen earlier, the Character values are put within *single* quotation marks to distinguish them from variables.

Note In most programming languages, the variables used in a program must be declared as being of one type, and this declaration is usually given at the beginning of the program.

Here are a few samples of Character variables and their values:

- Grade : 'A', 'B', 'C', 'D', 'F'
- Reply : 'Y', 'N', 'y', 'n' (for Yes or No)
- Operator : '+', '-', '*', '/'
- Digit : '0', '1', '2', '3', '4'... '9'
- Bracket : '(' ')', '[']'

Operations on Characters include assignment, comparison, input and output. The assignment of values to character variables is done in the usual manner as illustrated by Pseudocode 6.18.

Pseudocode 6.18 Assigning values to character variables

```
Set Fail to 'F'  
Set Hyphen to '-'  
Set Grade to Fail
```

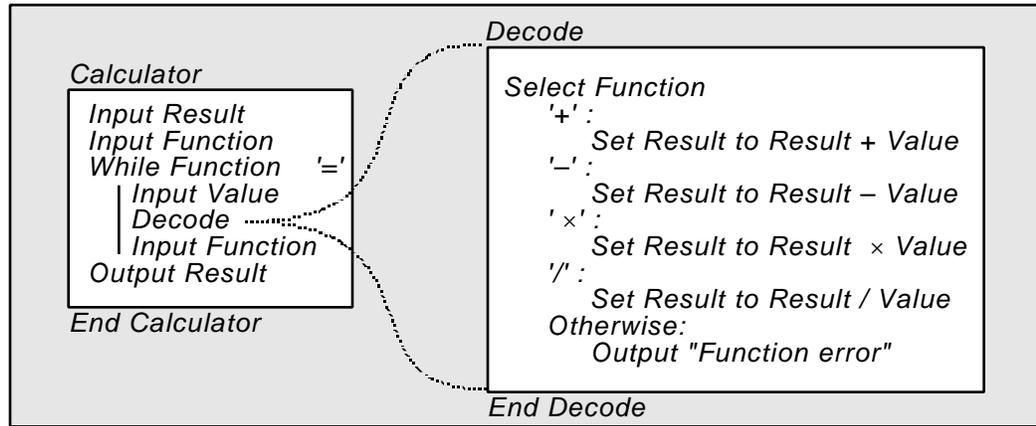
The comparison of Characters is based on the character codes used by the computer. Fortunately these codes are such that alphabetical characters are considered ordered in the usual way: 'A' is considered to be less than 'B'. For an example, consider the statement in Pseudocode 6.19.

Pseudocode 6.19 Character variables used in Selection forms

```
If Grade < 'C'  
Set Counter to Counter + 1
```

This statement will increment *Counter* if the grade is the character 'A' or 'B'. Notice that character 'A' is less than character 'B', which is not the conventional view of grades! If you compare punctuation signs you will need the character codes to know how they are ordered! The most common standard character code is called ASCII.

Figure 6.21 Algorithm for four-function calculator

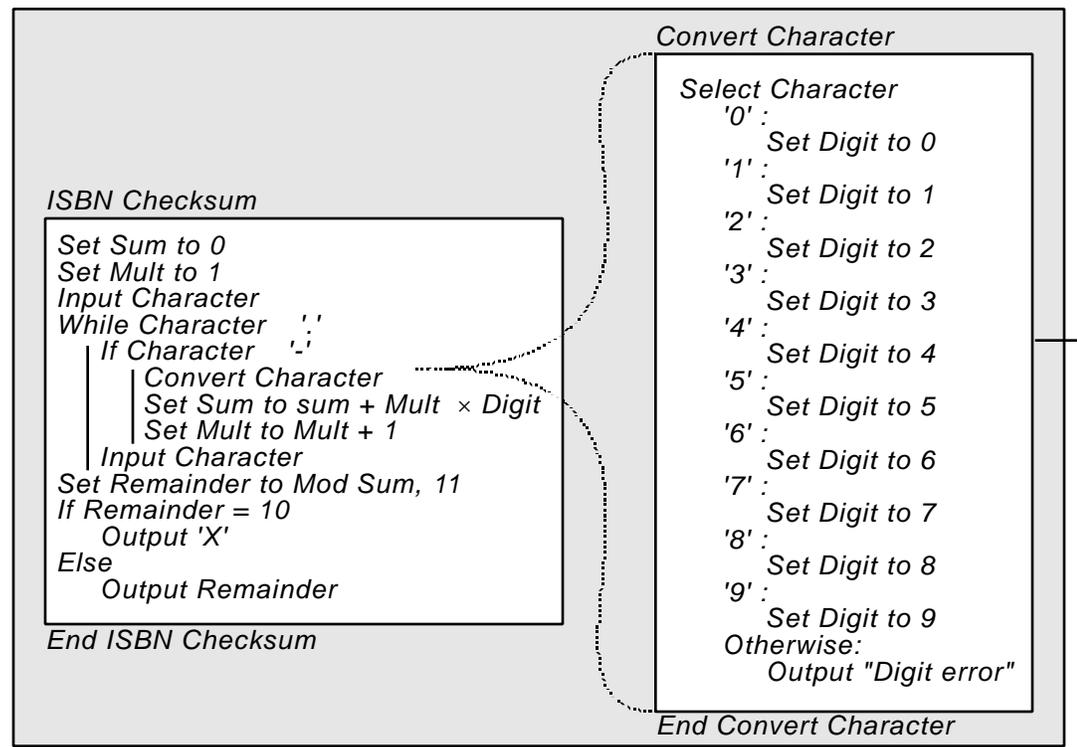


To illustrate the processing of Character data, we will develop an algorithm to implement a simple four-function calculator. Our algorithm is shown in Figure 6.21 and behaves as an unusual four-function calculator that accepts sequences of alternating numbers and operations.

The Decode algorithm, shown on the right of Figure 6.21, recognizes the operations (an asterisk denotes multiplication) and applies them to the numeric values until it encounters an equal sign, as in the following examples:

- 1 + 2 * 3 = (which returns 3 × 3 or 9)
- 4 * 5 - 6 / 7 = (which returns 14 / 7 or 2)
- 9 / 5 * 100 + 32 = (which returns 212)

Note: The operations in the above example are applied from left to right. This is not the operator precedence which we are used to!

Figure 6.22 Algorithm for ISBN checksum computation

Notice here that the numeric Characters input must be converted first of all into numbers, before any operations can be performed on them. Other converting methods are possible.

Notice that the Decode algorithm (Figure 6.21) uses the Case Form (a variation of the Select Form) that compares a single value with a set of constants and performs the appropriate action. If the *Function* entered is not one of the recognized operations, the message “*Function error*” is output.

As another example of Character data handling, let’s look at the algorithm to compute the checksum for the International Standard Book Number (ISBN), which was already described in Chapter 3. Figure 6.22 shows the two-stage development of the *ISBN Checksum* algorithm.

You should first note that a Character such as ‘2’ is not equal to the Integer number 2. This means that when numeric Characters are input, they must be converted before they can be treated as numbers. The ISBN Checksum algorithm accepts an input sequence of mixed Characters (numeric characters and hyphens), ending with a period. It ignores the hyphens and invokes sub-algorithm *Convert Character* to convert the numeric Characters into their corresponding Integer values. It then multiplies the digits by their rank (look back to Chapter 3 for the algorithm description), and accumulates them in

For more details on how to compute the checksum for ISBNs, see Figure 3.5.

Sum. When the input is terminated, the algorithm uses the *Mod* function to find the *Remainder* of the division of *Sum* by 11. If that *Remainder* is 10, then the check symbol is 'X'. Otherwise, the check symbol is the *Remainder* itself.

The Logical Type

As mentioned in Chapter 5, Section 5.4, the Logical (or Boolean) type includes only two values, True and False (sometimes labeled T and F, or 1 and 0). Logical variables are often used in conditions within Selection and Repetition forms. A few examples of Logical variables are shown in Pseudocode 6.20.

Pseudocode 6.20 Examples of Logical variables

Female, Done, Over21, Increasing, Equilateral — All of these variable names imply a Yes/No answer.

For the truth table for the AND, OR, and NOT operators, consult Figure 3.27.

Operations on Logical type variables include the assignment operation, the conjunction operation (AND), the disjunction operation (OR), and the negation operation (NOT). Pseudocode 6.21 illustrates a few examples of logical assignments.

Pseudocode 6.21 Examples of Logical assignments

Set Male to True
Set Over21 to (Age > 21)
Set Danger to (Divisor = 0)
Set Done to (Counter = Last)
Set Triangle to ((Small + Mid) > Large)
Set Right to ((X × X + Y × Y) = H × H)
Set Close to (Abs(X - Y) < 0.001)

The conjunction operation is called AND. The conjunction of two logical variables P and Q is written “P AND Q” and is true when both P is True and Q is True. Some examples of the use of AND are shown in Pseudocode 6.22.

Pseudocode 6.22 Examples using AND

Set Increasing to (X < Y) AND (Y < Z)
Set Equilateral to (A = B) AND (A = C)
Set Eligible to Over21 AND Employed

The disjunction operation is called OR. The disjunction of two logical variables P and Q is written “P OR Q” and is true when either P is True, or Q is True (or both are True). Some examples of the use of this “inclusive” OR are shown in Pseudocode 6.23.

Pseudocode 6.23 Examples using OR

Set Win to (Sum = 7) OR (Sum = 11)
Set Error to (Age < 0) OR (Age > 120)
Set Isosceles to (A = B) OR (B = C) OR (C = A)

The negation operation is also called NOT, and simply reverses the truth values, changing True values to False and vice versa. Pseudocode 6.24 shows some examples of its use.

Pseudocode 6.24 Examples using NOT

Set Female to NOT Male
While NOT Done . . .
If NOT Increasing . . .

When converting directly from English, it is easy to produce improper forms of Logical expressions. Figure 6.23 gives some examples of improperly formed expressions, along with their corresponding correct versions.

Figure 6.23 Improper expression and their corrections

Improper		Proper
$A \text{ AND } B = 5$	should read	$(A = 5) \text{ AND } (B = 5)$
$T = 7 \text{ OR } 11$	should read	$(T = 7) \text{ OR } (T = 11)$
$A < B < C$	should read	$(A < B) \text{ AND } (B < C)$
$X \text{ OR } Y > Z$	should read	$(X > Z) \text{ OR } (Y > Z)$
$I = J \text{ AND } K$	should read	$(I = J) \text{ AND } (I = K)$
$S \text{ NOT } 7 \text{ OR } 11$	should read	$\text{NOT}((S = 7) \text{ OR } (S = 11))$

See Figure 4.13 for more details on the different types of triangles possible.

Algorithms involving complex combinations of Selection forms can often be done in a simpler manner by using Logical expressions. For example, triangles can be classified by the following series of Logical assignments (assuming angles A B C). Pseudocode 6.25 gives some examples of using Logical expression in place of Selection forms.

Pseudocode 6.25 Examples of Logical expressions

Set Triangle to $((A + B + C) = 180)$
Set Isosceles to $(A = B) \text{ OR } (B = C)$
Set Acute to $(A < 90) \text{ AND } (B < 90) \text{ AND } (C < 90)$
Set Obtuse to $(C > 90)$
Set Right to $(C = 90)$
Set Equilateral to $(A = C)$

It is possible to compare truth values, if the True value is assumed to be greater than the False value. The table below (Figure 6.24) introduces four truth tables describing some of these comparison operations which correspond to known functions of symbolic logic.

Note: Remember that True=1 and False=0. Hence, since 1>0, then True>False.

Figure 6.24 Truth table for some comparison operations

P	Q	P Q	P=Q	P Q	P>Q
F	F	T	T	F	F
F	T	T	F	T	F
T	F	F	F	T	T
T	T	T	T	F	F
column		1	2	3	4

The table in Figure 6.24 is described column by column as follows:

- Column 1 shows the operation “P Q”, where the relation “less than or equal to” is applied to Logical values. In symbolic logic, this operation is the conditional connective called implication, usually denoted by the symbol “P Q”, which is read “If P then Q”. This conditional connective is often used in logical deduction, where it is sometimes noted as “P Q” or “P Q”.
- Column 2 shows the operation “P = Q” (the Biconditional connective). It is read as “P if and only if Q” and is also noted in logic as “P Q” or “P Q”.
- Column 3 shows the operation “P Q” (the “exclusive or”) which is True when one and only one of the values is True. It is sometimes noted as “P <> Q”.
- Column 4 shows the operation “P > Q” (sometimes called the inhibit-and) where the value of P is inhibited by Q.

Other Logical operations exist, for instance $P < Q$ and $P \neq Q$.

6.6 Some General Problem-Solving Algorithms

Bisection Finding Values

The Bisection algorithm is very useful for solving many types of problems and is often also referred to as Bracketing, Divide and Conquer, or the Half-Interval Method. It proceeds by taking two limiting values and adjusting them successively to bracket the required result. This is actually the method we used in the *Guesser* algorithm of Chapter 4 (see Figure 4.7), which is reproduced and renamed at the left of Pseudocode 6.26.

Pseudocode 6.26 The bisection technique and its application to finding a square root

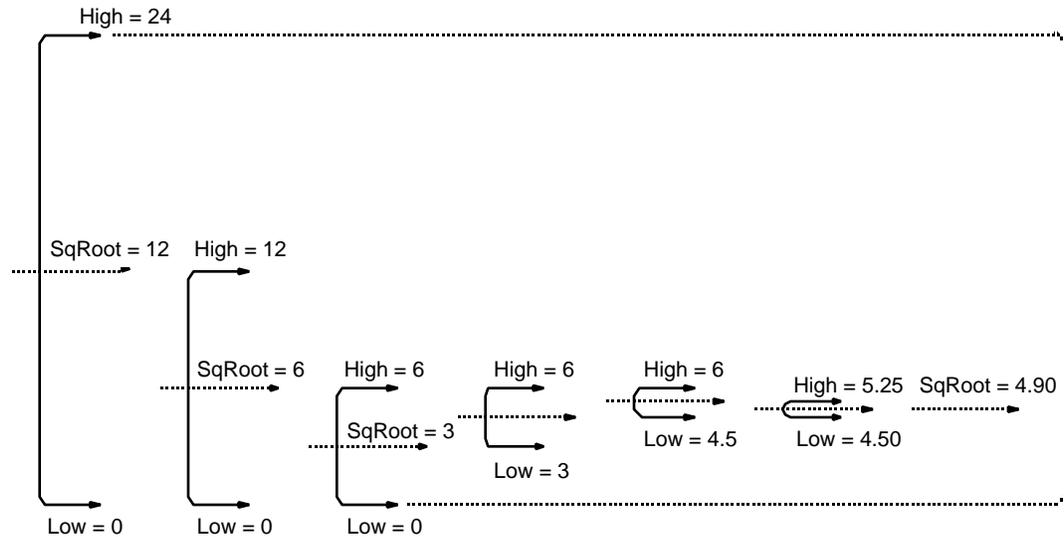
<pre> Bisection Input X Set High limit Set Low limit Set Guess to midpoint While Guess is not correct If Guess is too high Lower High limit Else Raise Low limit Set Guess to new midpoint Output Guess End Bisection </pre>	<pre> SquareRoot Input X Set High to X Set Low to 0 Set SqRoot to (High + Low) / 2 While (SqRoot × SqRoot) ≠ X If (SqRoot × SqRoot) > X Set High to SqRoot Else Set Low to SqRoot Set SqRoot to (High + Low) / 2 Output SqRoot End SquareRoot </pre>
--	---

By refining this guessing algorithm for the particular problem to solve, this bisection technique can be applied to many problems, such as calculating the square root of some number, X . This particular refinement is shown on the right of Pseudocode 6.26. The behavior of this algorithm will be better understood by looking at Figure 6.25, where the algorithm is used to find the square root of 24.

First, a *High* limit of 24 and a *Low* limit of 0 are set. The mid-point between *High* and *Low*, 12, is chosen as a first guess at the square root of 24 and is set in *SqRoot*. Since this guess is too high ($12 \times 12 > 24$), the higher limit is lowered to the middle value, *SqRoot*.

If the guess was too low, the lower limit would have been raised to the same middle value. This same process is now repeated for the new limits. This repetition continues until the two limits narrow down (or bracket) the solution to the perfect match $(SqRoot \times SqRoot) = X$, which corresponds to *SqRoot*, *High* and *Low* having the same value.

Figure 6.25 The trace of ranges for the square root of 24



This perfect match might prove elusive, and algorithms dealing with Real Number results usually introduce the concept of *precision* to determine whether a result is acceptable or not.

For example in our *SquareRoot* algorithm, this would mean that the looping continues as long as the square of the guessed root of X differs (in absolute value) from X by more than some very small constant, the desired *Precision*. This constant of *Precision* is initially chosen to be some small value, such as 0.1 or 0.00001. The smaller the value, the more looping is required to attain that *Precision*.

Symbolically, the loop condition could use the absolute function value of the difference and be written as

$$|SqRoot \times SqRoot - X| > Precision$$

instead of

$$(SqRoot \times SqRoot) \neq X.$$

Let's modify our *SquareRoot* algorithm along these lines, using function *Abs* to obtain the absolute value.

Figure 6.26 shows a trace of the new *SquareRoot* algorithm computing the square root of 24. Notice that this method of bisection is not limited to just square roots! It can equally compute the cube root by simply changing the loop terminating condition to compare the cube of the current approximation with the value X . The Bisection Method can also be used to find the roots of equations.

However, this Bisection method might sometimes have limitations. For example, the given *SquareRoot* algorithm does not work properly if the input value of X is between 0 and 1. Try a trace to see why.

Figure 6.26 New *SquareRoot* algorithm and trace for $X=24$

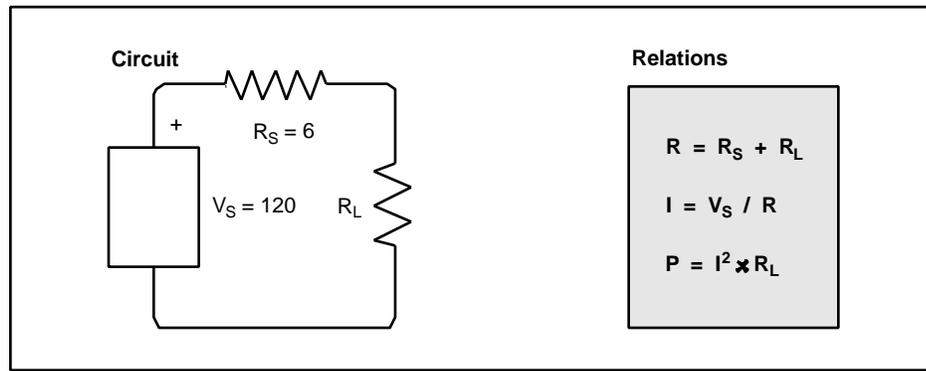
<i>Input X</i>	X = 24		
<i>Set High to X</i>	High = 24		
<i>Set Low to 0</i>	Low = 0		
<i>Set Precision to 0.1</i>	Precision = 0.1		
<i>Set SqRoot to (High + Low) / 2</i>	SqRoot = 12		
<i>While Abs(SqRoot² - X) > Precision</i>	120 > 0.1 T	12 > 0.1 T	15 > 0.1 T
<i>If SqRoot² > X</i>	144 > 24 T	36 > 24 T	9 > 24 F
<i>Set High to SqRoot</i>			
<i>Else</i>			
<i>Set Low to SqRoot</i>			
<i>Set SqRoot to (High + Low) / 2</i>			
<i>High =</i>	12	6	6
<i>Low =</i>	0	0	3
<i>SqRoot =</i>	6	3	4.5
<i>Output Balance</i>			

Bisection is a very general method for finding values, that will be useful later in many problems. For example, we will use it again in Chapter 8, to search quickly through a sorted list.

Maximum Power Optimizing Power Output

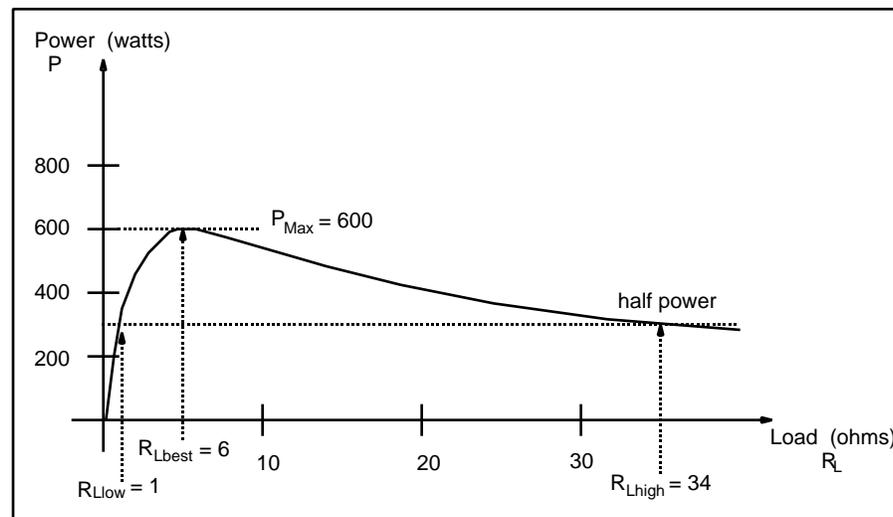
Let's turn now to an engineering application. Oftentimes in engineering we want to determine optimal values for the variables of a system. To do this, it is necessary to analyze the effect that the various variables of the system have on each other.

The diagram in Figure 6.27 is an electrical network, consisting of a voltage source of V_S volts, providing power to a load resistor R_L through a series resistor R_S . We wish to determine the value of the load that would provide the maximum power P to the load.

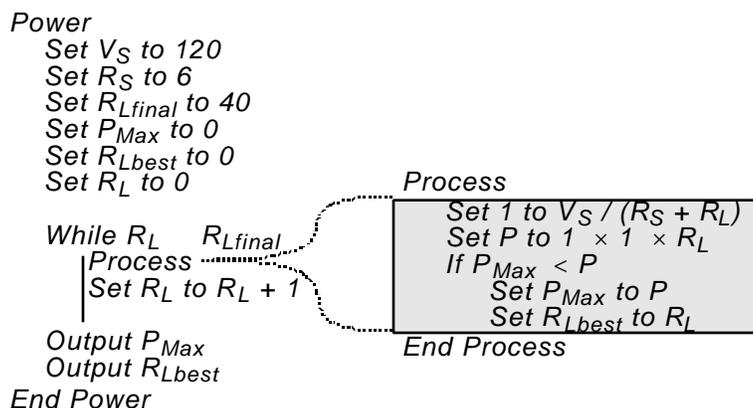
Figure 6.27 A simple circuit diagram

Relations among the variables are given by the formulas on the right in Figure 6.27 (determined from knowledge of Ohm's law and Kirchoff's laws).

The behavior of such a system, given in Figure 6.28, shows how the power P (in watts) varies, depending on the resistance R_L of the load (in ohms). From this graph, we see that the best value R_{Lbest} of the load resistor R_L is 6 ohms (same value as the series resistor R_S), resulting in a maximum power P_{Max} of 600 watts.

Figure 6.28 Power plot for circuit in Figure 6.27

The algorithm *Power*, shown in Figure 6.29, is an algorithm which analyzes this system. It is a simple loop that varies the load resistance R_L from zero to some final value R_{Lfinal} , computing the corresponding value of power P . At each iteration, the sub-algorithm *Process* is invoked. Such a sub-algorithm could define any kind of process. It could, for instance, simply output the R_L, P combinations. Or, it could also compute the maximum power P_{Max} and the corresponding best resistor R_{Lbest} , as shown in Figure 6.29.

Figure 6.29 Algorithm for finding Maximum Power

The *Process* sub-algorithm could also find the half power points; those two values of R_L (say, R_{Llow} and R_{Lhigh}) at which half power is sent to the load. Those two values are shown on the plot of Figure 6.28 as being 1 ohm and 34 ohms. Any load value between these two would result in more than half power being delivered. Notice that the half power points are not equally distant from the maximum power point R_{Lbest} . It is also possible to create an algorithm that plots the power P versus the load resistor R_L . Such a plotting algorithm would be similar to the ones developed in Section 6.4 (Creating Plots with For Nests).

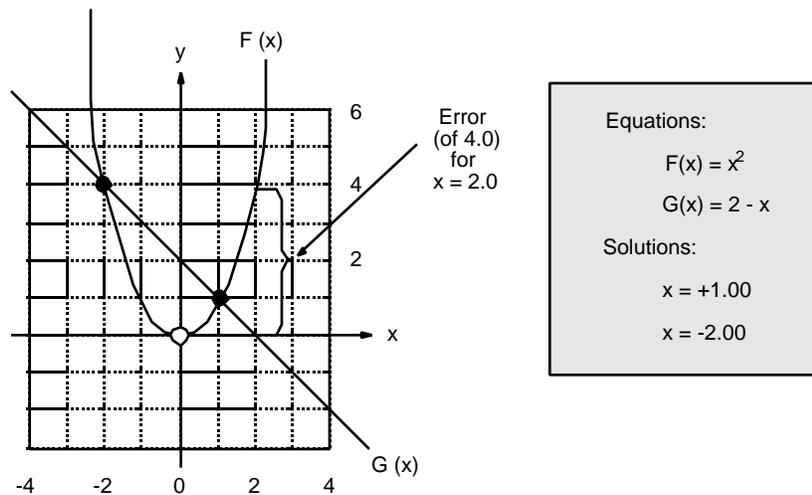
In this case, the values of the load resistor R_L were systematically selected in increasing order, from 0 to some positive final value (which is 40 in this example).

This example application gives us a model that we can apply to analyze other engineering systems. For example, we may wish to find the optimal angle to shoot some object so that it goes the farthest distance. Or we may wish to compute the best combinations of selections to make optimal profits.

Sometimes, with some extra knowledge, we can avoid having to write an algorithm to analyze a system. For instance, in the case of our example above, we could have proven, by using calculus, that the maximum power occurs when the load resistor R_L equals the series resistor R_S . However, in cases where the systems are more complex (say nonlinear), then computer methods may be better than analytical methods.

Solver Solving Equations

Now we will develop a general equation solving algorithm, *Solver*, that solves any *two* equations that are functions of a single variable, say x . The equations may be linear or highly nonlinear. *Solver* will find any number of values of x that satisfy both equations, be it zero, one, two or more.

Figure 6.30 Graphical solution of two equations

For example, consider the following functions $F(x)$ and $G(x)$ shown in Figure 6.30:

$$F(x) = x^2 \text{ and } G(x) = 2 - x$$

The figure shows that the two graphs intersect at two points, providing two solutions:

$$x = +1.00 \text{ and } x = -2.00$$

There are many ways of using a computer to solve two such equations. Here, we will use a method based on trying random values. This method is simple. Try some values of x randomly and see which is the best. The best value of x would be the value for which the difference of the two function values is the smallest. For each random value of x , the program finds the values of each function, y_1 and y_2 :

$$y_1 = F(x) \text{ and } y_2 = G(x)$$

These two values are then subtracted, yielding the error for that particular value of x .

$$\text{Error} = \text{Abs}(y_1 - y_2)$$

- If the error is zero, the functions yield the same values, and the graphs intersect for that value of x .
- Otherwise, this process of randomly trying many values of x over some range is repeated a number of times, in search of the minimum error. The final result is the value of x that produced the minimum error.

For example, if the random value chosen for x is 2.0, as shown in Figure 6.30, then the resulting error is 4.0 as shown by the big bracket—the difference of the two functions at that point.

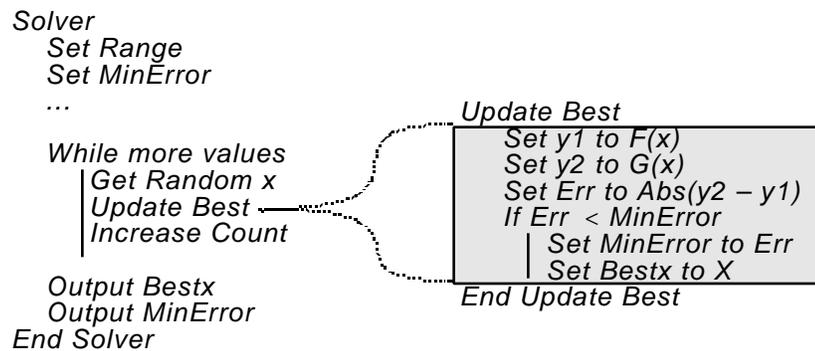
Similarly, if the random value chosen for x is 0.5 then

$$\begin{aligned} y1 &= F(0.5) = (0.5) \times (0.5) = 0.25 \\ y2 &= G(0.5) = 2.0 - 0.5 = 1.50 \\ \text{Err} &= \text{Abs}(0.25 - 1.5) = 1.25 \end{aligned}$$

If the random value chosen for x is $+1.0$, then $y1 = y2$ and the error is zero. For other values of x , there could be a larger error. We could reduce the errors by using the *Solver* again, but within a range that is much closer to the actual value of a solution.

In the case of our example, other tries of *Solver* would also yield the other solution (which is -2.00). In fact, it would be useful to run *Solver* a number of times, say 10, so that other solutions (if they exist) may be seen.

Figure 6.31 Solver algorithm for solving pair of equations $F(x)$ and $G(x)$



The algorithm to apply this method, *Solver*, is shown in Figure 6.31. It consists of a large loop that tries many values of x (say 200). Within this loop, for each value of x , it invokes sub-algorithm *Update Best* that computes both functions, finds their difference and keeps track of the smallest error and the corresponding best x . Here are sample outputs for various executions of this algorithm:

Value of x	Minimum Error
-1.98	0.006
+0.50	1.250
+0.99	0.030
+1.04	0.122
-2.02	0.060

An alternative way to determine when to terminate the While loop is not to count the number of values of x chosen, but to stop looping whenever *MinError* reaches a given lower limit. Note that the *Solver* algorithm looks for one

solution at a time. In particular, if it finds several solutions with a zero error, it will only keep the first one. If we want all solutions, we must either

- Modify the algorithm so that it keeps all solutions, or
- Run Solver a number of times in various ranges, as we have done above, and use the results for determining new ranges to try.

As we have seen with other algorithms, we can use Solver as a model to be modified to find other results. These results could include finding the roots of an equation, finding the maximum value of a function, determining if an equation has no root, and so on.

6.7 Review Top Ten Things to Remember

1. This chapter introduced *bigger blocks* of many kinds, but it is important to keep in mind that bigger is not always better.
2. We have seen that *bigger data* were in the form of external data stored outside the computer in files which are a means of storing voluminous data on physical devices such as tapes and disks.
3. To aid in reading in data from files, the concept of a terminating value to mark the end of external data was introduced, as well as the terminating “sandwich”, which marks the beginning and the end of the data with the same terminator. A terminating value, or terminator must be distinct from the data being read.
4. *Bigger algorithms* are simply extensions of the original algorithms. The obvious ways of obtaining bigger blocks was to use *bigger mixtures of forms*, especially of Repetitions and Selections.
5. We also introduced *bigger Selection forms*, such as an extension of the Selection Form: the *Select Form*. Instead of selecting from only two alternatives, the Select Form allows for any number of choices.
6. The *Case Form*, a variation of the Select Form, was also introduced and is used to compare one variable against many constants.
7. The *bigger Repetition form* that was introduced was the For loop, convenient when counting is involved, but otherwise somewhat restrictive.
8. It is easy to create *bigger nests of loops* that are also potentially difficult to understand. However, when they are viewed one at a time in a proper top-down break-out diagram, they seem simpler.
9. The *choice of data types* was also made bigger by adding the Character data type and the Logical data type. As with other data types, operations on Characters include assignment, comparison, input and output. Operations on Logical type variables include the assignment, conjunction (AND), disjunction (OR), and negation (NOT). Logical variables may also be compared.
10. The *Bisection* algorithm was also introduced and is very useful for solving many types of problems. It proceeds by taking two limiting values and adjusting them successively until they bracket the required result.

6.8 Glossary

Case Form: A variation of the Select Form.

Control Variable: See Loop Control Variable.

EOF: Abbreviation for End of File, a special value that indicates the end of a file.

File: A major unit of data storage and retrieval, generally stored outside the computer on an external storage device, for example, tape or disk.

Loop Control Variable: The variable that contains the value of the counter in a For Loop Form.

Precision: A measure of the degree of discrimination with which a quantity is stated and thus of the ability to distinguish between nearly equal values.

Pseudo-random number: An element of an ordered set of numbers that has been determined by some defined arithmetic process, but which is effectively a sequence of random values for the purpose for which it is required.

Random value: A number obtained by chance. See pseudo-random number.

Select Form: An extension of the Selection form where, instead of selecting from only two alternatives, there may be any number of choices.

Sentinel value: A marker supplied by the user rather than by the computer

system software, to indicate the end of a set of values.

Terminating value: See Sentinel value.

6.9 Problems**1. Monthly Calendar**

Create an algorithm top-down in pseudocode form to output a monthly calendar given the number of days N in a month and the first day F (where $F=1$ on Sunday, $F=2$ on Monday, etc.).

An example of a calendar follows for a month with $N = 31$ days having a first day occur on a Saturday, so $F = 7$. Notice that such a month requires 6 weeks!

Problem 1

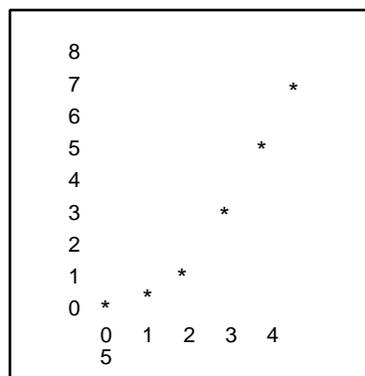
```

                                     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

```

2. Plot up

Create an algorithm top-down in pseudocode form to plot some function $F(x)$ vs. x , with the y -axis vertical and the x -axis horizontal. Use as an example the previous plot of $y = x^2 / 3$ for x varying from 0 to 5 and y varying from 0 to 8 producing an output as shown below.

Problem 2

3. Sine Function

The trigonometric sine of an angle x given in radians can be determined from the first N terms of this series:

$$\text{Sine}(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \dots$$

Create an algorithm top-down to compute this sine function. Attempt to improve upon this algorithm.

4. A Case of Max, Mid, and Maj

Create a Select structure to compute M , which is the following:

- the maximum value of three variables A , B , and C .
- the minimum value of three variables, A , B , and C .
- the majority value of three binary variables A , B , and C , each having values 0 or 1.

5. Second Max

Modify the Big Max algorithm in Figure 6.6 to find the second highest value S , assuming that all values are different.

6. Many Max

Modify the Big Max algorithm to find the number of values N which are maximum (when some values may be repeated).

7. MaxMin

Modify the Mean program (with sandwich in Figure 6.5) to compute both the maximum and minimum values.

8. Quadrant

Create an algorithm that accepts the coordinates X and Y of some point and indicates which quadrant (1, 2, 3 or 4) the point falls into. If the point falls on an axis, the quadrant should be indicated as value zero.

9. Gas

Create an algorithm that inputs sequences of two values Miles and Gals representing the odometer mileage and the gallons of gas at a succession of refills of gas tank. The algorithm is to compute and output the

immediate average miles-per-gallon (labeled Short for short range), and also the overall average mpg (since the beginning of the data), which is labeled Long for long range. A typical input-output sequence follows (and should end with negative mileage as a loop terminator).

INPUTS		OUTPUTS	
Miles	Gals	Short	Long
1000	20		
1200	10	20	20
1500	20	15	16.67
...

10. GPA

The grade point average GPA of a student is computed from all the course grades G and units U . Corresponding to each grade is a numeric value (where A has 4 points, B has 3 points, etc.) The products of each grade point and its number of units are then summed. This sum is divided by the total number of units, to yield the grade point average. Create an algorithm to compute the grade point average for a sequence of pairs of values G, U (ending with negative values to terminate).

11. Speed

Create an algorithm to analyze the speed during a trip of N stops. At each stop, the distance D and time T from the previous stop are recorded. These pairs of values are then input to a program which computes each velocity ($V = D/T$) and outputs it. It also ultimately indicates the maximum speed on the trip, and the overall average (total distance divided by total time).

SAMPLE RUN (N = 5)

D	T	V	
45	1	45	
100	2	50	
55	1	55	
120	2	60	Avg = $380/8 = 47.5$
60	2	30	Max = 60

12. Unbiased Mean

In some sports, a number of judges each ranks performance on a scale from 1 to 10. To adjust for biases, both the highest and lowest values are eliminated before computing the average. Create an algorithm to compute such an average for M judges on N performances.

Problems on Loops and Nests**13. Once More**

What action is performed by the following algorithm?

Problem 13

```
Input N
Set S to 0
For I = 1 to N by 1
  | For J = 1 to N by 1
  |   Set S to S + 1
Output S
```

14. Disproof

Show that the following two pieces of algorithm are not equivalent.

Problem 14

<pre>While A While B C</pre>	<pre>While A AND B C</pre>
--	------------------------------

15. Expo

The exponential function $Expo(x)$ can be computed from the first N terms of this series:

$$Expo(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

- Create an algorithm to compute this, assuming that Fact and Power are available as sub-algorithms.
- Create another algorithm that does not call any other sub-algorithms, and also does not keep recomputing the factorial or power (but uses previously computed results, such as $5! = 5 \times 4!$).

16. Down Timer

Create a timer to count down from a given number of hours, minutes and seconds to zero. At intervals of five seconds, it outputs the time (remaining to zero).

17. Pythagorean Triplets

Construct an algorithm to produce all integers x , y , and z that satisfy the Pythagorean theorem (relating the sides of a right triangle):

$$x^2 + y^2 = z^2 \quad (\text{where } x < y < z)$$

Let x , y , and z be positive integers, all less than some fixed input value M (say 100).

18. Thanks

Create a general algorithm which outputs “THANK YOU” for a total of N times (where N is input). This greeting is printed three times per line (possibly less on the last line). There is a blank line after every group of twelve greetings.

Problems on Types: Character and Logical**19. Logical Swap?**

Prove (or disprove) the fact that the following two algorithms swap the values of the logical variables, P and Q .

- | | | | |
|----|------------------------------|----|------------------------|
| a. | Set P to NOT(P OR Q) | b. | Set P to ($P = Q$) |
| | Set Q to NOT(P OR Q) | | Set Q to ($P = Q$) |
| | Set P to NOT(P OR Q) | | Set P to ($P = Q$) |

20. Logical Less

If False is defined as less than True, draw a truth table for the operation $P < Q$. Draw also a table for $P \leq Q$.

21. Binconvert

Create an algorithm to convert a sequence of binary input characters (not integers) into their corresponding decimal values. For example, 1101 is the decimal 13.

- a. Write the algorithm, if the input is read from left to right (ending with a period).

- b. Write the algorithm, if the input is read from right to left (ending with a blank).

22. When In Rome...

One method for converting an Arabic number into a Roman number is to separately convert each digit (the units, tens, hundreds, and thousands positions) as shown:

1	9	8	4
M	CM	LXXX	IV

Write an algorithm that accepts as inputs any numeric values up to 3999, and outputs the corresponding Roman numbers.

- a. Do this, if the number is entered digit by digit (least significant digits first, like 4 8 9 1).
- b. Do this, if the number is entered digit by digit (most significant digits first, like 1 9 8 4).
- c. Do this, if the number is entered all at once, as an integer, like 1984.

23. XOR-cise

Prove (or disprove) the following cancellation property for the usual OR and the exclusive-or, XOR.

- a. $If (A \text{ OR } B) = (A \text{ OR } C)$
 $B = C$
- b. $If (A \text{ XOR } B) = (A \text{ XOR } C)$
 $B = C$

24. Translate

Convert the following conditions into Logical statements (using ANDs, ORs and NOTs).

- a. Neither A nor B.
- b. Either A or else B.
- c. Exactly two of the three variables A, B, C are True.
- d. An odd number of the three variables A, B, C is False.

Chapter 7 Better Blocks

We have seen in earlier chapters how to decompose a solution into its components. We will continue here in a more formal manner. This chapter considers the very important problem of breaking up a program into smaller pieces called subprograms. As was the case for the algorithm examples we have seen, there are many ways to break up a program. Some are better than others, and this chapter shows how to choose the better ways.

Chapter Outline

7.1	Preview.....	300
7.2	Subprograms.....	300
	How to Simplify Large Programs.....	300
	What are Parameters?.....	305
	Data Space Diagrams.....	309
7.3	Parameter Passing.....	312
	Passing Parameters In and Out.....	312
	Special Cases.....	316
	Some Examples.....	321
7.4	Bigger, Leaner, and Meaner Programs.....	329
	Using Structure Charts.....	329
	Contour Diagrams.....	333
	Parameter Crossing A Common Mistake.....	336
	Minimizing Coupling, Maximizing Cohesion.....	338
	Deeply Nested Subprograms.....	340
	Dates Example.....	341
7.5	More Types of Subprograms.....	343
	Recursion Self-Referencing Subprograms.....	343
	Functions.....	345
	Modules.....	346
7.6	Review Top Ten Things to Remember.....	348
7.7	Glossary.....	350
7.8	Problems.....	351

7.1 Preview

We have already used some simple subprograms or sub-algorithms, glossing over the manner in which data are passed to them and how results are obtained from them. Data transfers between program and subprograms is the subject of this chapter.

We will first consider *subprograms*, *functions* and *block structure* from three viewpoints:

- Control flow,
- Data flow,
- Data space (introduced for the first time here).

To pass data into and out of subprograms, a special kind of variable called a *parameter* is used. Parameters are considered in great detail in this chapter, in particular the way in which they are used to pass data. There are two methods by which data are passed through parameters:

- by value, and
- by reference.

Recursion, the ability for a subprogram to call itself, is also briefly introduced here as another control structure.

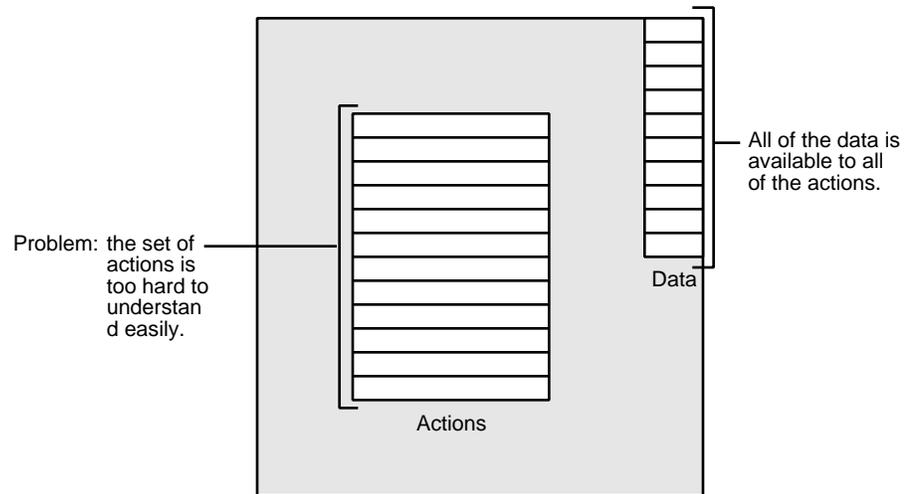
A number of examples of some small and simple blocks are given, along with some larger examples: a payroll program and a change maker program. Although the full power of subprograms is mostly felt with large programs, the smaller programs of this chapter make it possible to illustrate them in a convincing manner.

The chapter ends with the idea of modules, or pieces of a program that are written separately, compiled separately, and that are later linked together to form the complete program.

7.2 Subprograms

How to Simplify Large Programs

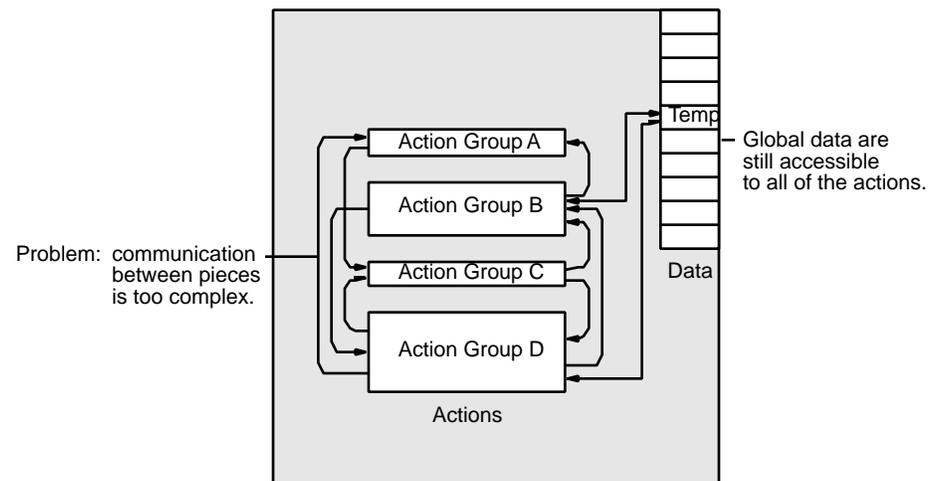
There are a number of ways to simplify a large program. The following figures illustrate these ways, going from simple program forms to more modular forms. These figures will serve as models on which the examples used later in this chapter can be based.

Figure 7.1 No Structure

Program *No Structure*, shown in Figure 7.1, is a simple program consisting of some data and some actions. Here, the set of actions form a single group or subprogram. All of the data are accessible to all of the actions. Such data are said to be *global*—known and accessible from every part of the program.

As programs become larger, they quickly become more complex and hence, more difficult to understand, create correctly and modify. For example, a program that is a thousand lines long is already very difficult to understand.

You have probably found that if you keep your papers all mixed together in a single pile, you become confused and waste a lot of time looking for what you want. As a result, you likely organize your papers into folders, divided according to subject. So it is with programs. To handle this increased complexity, the actions are often broken out into smaller pieces, each of which can be understood separately and more easily. Sometimes, one individual or a team is associated with each piece. This structuring of a program can be done in many ways, and we will now look at four different ways in which this organization of a large program can be done.

Figure 7.2 Split Structure

The program *Split Structure*, shown in Figure 7.2, has been split into groups shown as Action groups A, B, C and D. In practice, there would be many such groups, possibly hundreds, in a large program. There is considerable

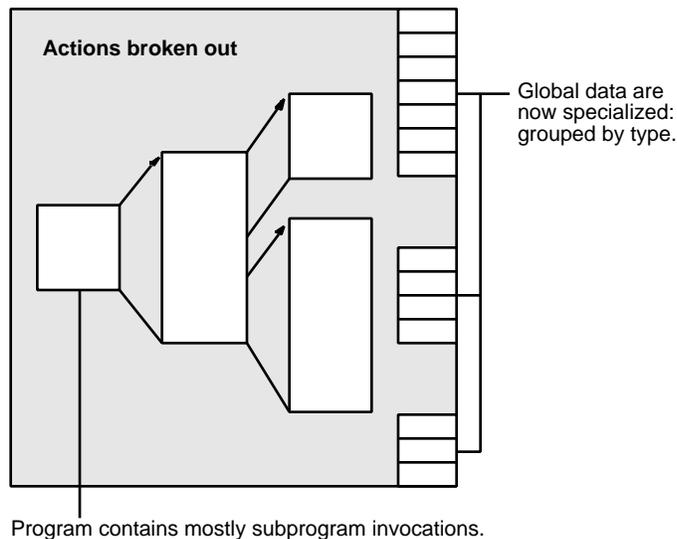
communication between the action groups; not only are they linked by flow of control but they also have access to the same set of global data, which may have thousands of separate items.

This means that the teams responsible for the different groups need to know what the other teams are doing—the communication between the pieces of a program is reflected in the communication between the teams. Also, any piece of program may access any data, so the sharing of data, a form of communication, becomes complex.

For example, two different teams may be using the same variable, Count, in two different ways and incrementing it for different reasons at different, possibly incompatible time. They might also be using the same “temporary” variable Temp and have to coordinate their uses to avoid clashes.

Clearly, this semi-structured way of breaking down a program is not ideal. On a larger scale, most programs that are structured in that way will fail because of the complexity of communication between the teams working on the various pieces of the programs.

Figure 7.3 Hierarchical structure



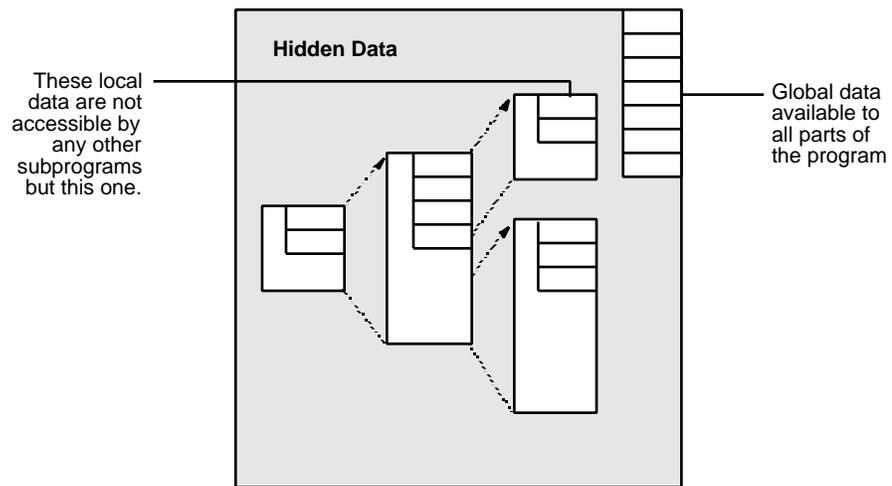
The organization of the program in Figure 7.3 is hierarchical and corresponds to the top-down development of algorithms seen in earlier chapters. The actions at the top level (at left) are mainly invocations of subprograms. This makes the actions of each of the program parts more independent and the flow of control between the parts easier to understand. However, the data are still shared by all the subprograms. The data tend to become grouped in a way that resembles the hierarchical organization of the actions—a sort of “specialization” of the data. This occurs because top-level subprograms generally refer to different kinds of data than those at lower levels.

An analogy to this may be seen in the way in which data are used in a corporation. For example, the top level people of a corporation are interested in the number of employees and the total money paid out to them. They do not need to know directly the number of hours worked by any individual. At the lower levels of the corporation hierarchy, the hours and rate of pay of an employee must be known, but knowledge of the total number of employees is not necessary.

We can improve the previous hierarchical structure by organizing a program with localized block structures as shown in Figure 7.4. Here, some data are distributed among the subprograms, and are only accessible to one subprogram.

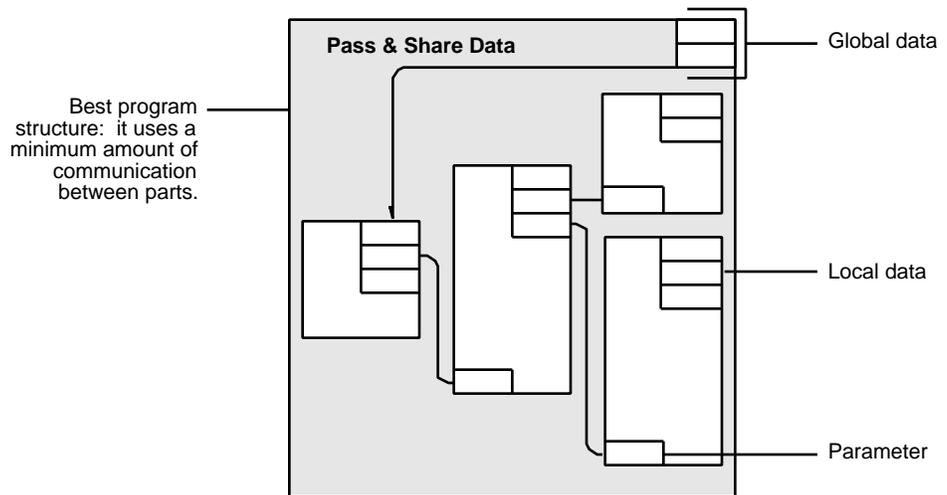
These data are *local* to a subprogram and are hidden from the other subprograms, thus being protected against being changed by them.

Figure 7.4 Localized block structure



In Figure 7.4, there still remains some global data accessible to all subprograms, but fewer of them. These data are mainly used for communication between the subprograms. This kind of structuring allows programming teams to be more independent because they have more control over the data that they alone need to know about.

Figure 7.5 Parameterized block structure



One final improvement shows the program organized as a parameterized block structure in Figure 7.5. We have added here to the previous localized block structure the interaction between the various blocks of actions. This interaction involves the passing of data between individual subprograms, thus communicating only what data are necessary and hiding the rest.

The organization in Figure 7.5 formalizes the communication between subprograms. When data must be shared between subprograms, sharing is done as a "private arrangement" between them instead of using the bulletin board approach of global variables. This serves to further reduce the amount of global data and the amount of communication between the programming teams.

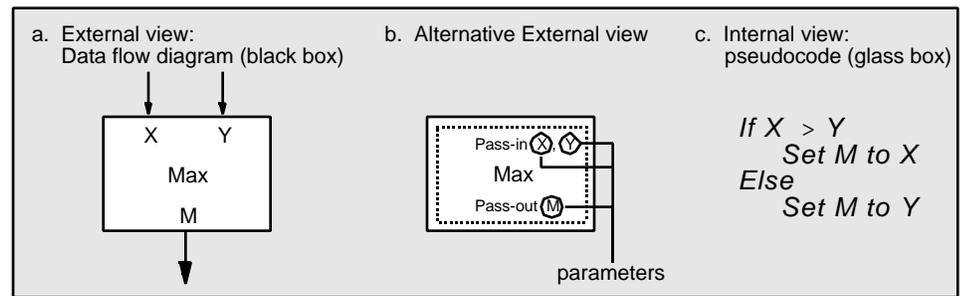
The design challenge to split up a program properly is a hard one. Sufficient data sharing must be allowed as well as adequate data hiding. In the rest of this chapter we will look more closely at the mechanisms required to do this.

What are Parameters?

For more details on black box and glass box views, see Chapter 3, Section 3.4.

We can describe subprograms the same way we described programs: from a bird's eye view or from a worm's eye view. You may remember that in Chapter 3, the bird's eye view was called the "black box" or "external" approach. Its main focus was on *what* data was passed in and out. The worm's eye view was called the "glass box" or "internal" approach. Its main focus was on *how* the data were manipulated.

Figure 7.6 Definition and representation of *Max* subprogram



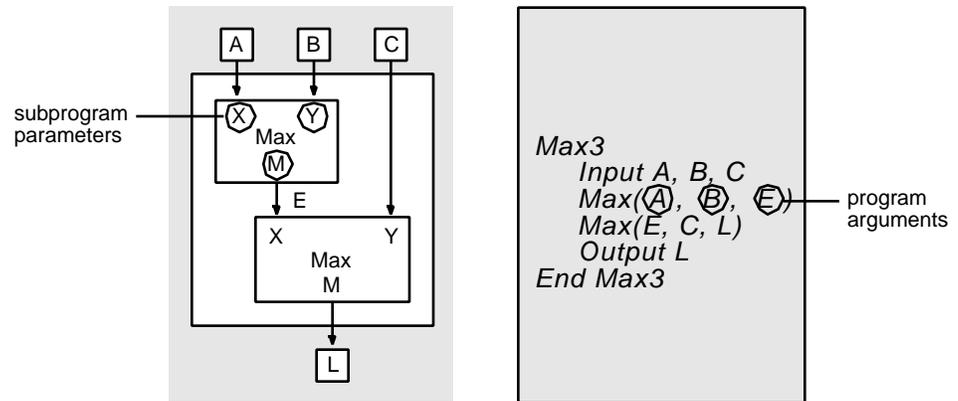
The data-flow diagram shown on the left of Figure 7.6 gives an external view of the subprogram *Max* which we already encountered in Chapter 3. In this view, *Max* is seen as a black box with two inputs *X* and *Y* and one output *M*, the maximum of the two values *X* and *Y*.

Another external view of *Max* is shown in the middle of the same figure by another black box called *Max*, having three variables *X*, *Y* and *M*. These three variables are *parameters* and are used for communicating data with the programs that use *Max*. This view, like the one on the left, shows nothing of the way in which *M* is derived from *X* and *Y*.

Finally, at the right of the Figure 7.6, the subprogram *Max* is defined completely with all its detail as a glass box. Here we can see exactly how the value of *M* is obtained from *X* and *Y*.

Although *X*, *Y* and *M* are all parameters of the subprogram *Max*, *X* and *Y* serve a different function from *M*. *X* and *Y* transfer information from the caller into the subprogram and are called *input parameters*. The parameter *M* works in the opposite way, it transfers the result of the computation out of *Max*, back to the caller. It is called an *output parameter*. The definition of the number of parameters, their data type and whether they are input or output, form the subprogram's *interface specification*.

Parameters are subprogram variables used to communicate data between the subprogram and either the main program or another subprogram.

Figure 7.7 Data-flow diagram for Max3

To see how such a subprogram is used, let's look at a program to find the maximum of three variables. At the left of Figure 7.7, the data-flow diagram of the program *Max3* shows how we can obtain the maximum of the three variables by interconnecting two of the *Max* subprograms. The right side of Figure 7.7 shows the inner workings of *Max3*. As we could have guessed from the data-flow diagram, the subprogram *Max* is *invoked* (or called) in the following two ways:

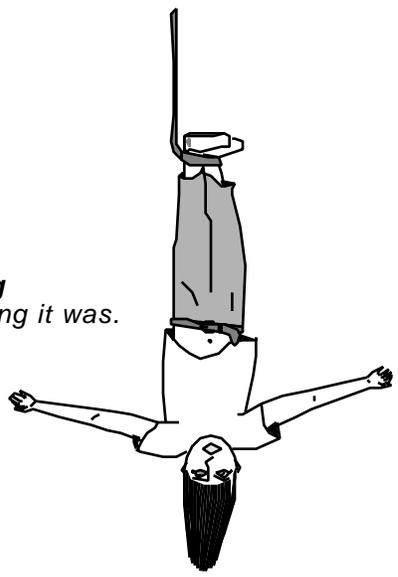
- In the first call, *Max* (*A*, *B*, *E*), *Max3*'s variables *A*, *B* and *E* are connected to *Max*'s parameters *X*, *Y* and *M*. The values of *A* and *B* are transmitted to *X* and *Y*, the input parameters. The result of the calculation, the maximum of the two values *X* and *Y*, is transmitted through *M*, the output parameter, to *E*.
- Similarly, in the second call, *Max* (*E*, *C*, *L*); *Max3*'s variables, *E*, *C*, and *L*, are connected to *Max*'s parameters *X*, *Y* and *M*. The values of *E* and *C* are transmitted to *X* and *Y*, the input parameters. The result of the calculation, the maximum of the two values *X* and *Y*, is transmitted through *M*, the output parameter, to *L*.

In the first call, *A*, *B*, and *E*, and in the second call *E*, *C*, and *L*, are said to be arguments. Thus, in a subroutine call, a connection is established between the caller's arguments and the subprogram's parameters and the values are transmitted through this connection.

Let's look at a non-computer example from everyday life to help us understand this "connection" between arguments and parameters: *bungee-jumping*. Here is a program which, if followed, would make sure a person had an exciting day.

Pseudocode 7.1 Program for an exciting day of bungee-jumping

Program Excitement
 Get up at dawn.
 Go do some **bungee-jumping**
 Tell all your friends how exciting it was.
 Go to bed.



If you prefer, you could go sky-diving instead of bungee-jumping.

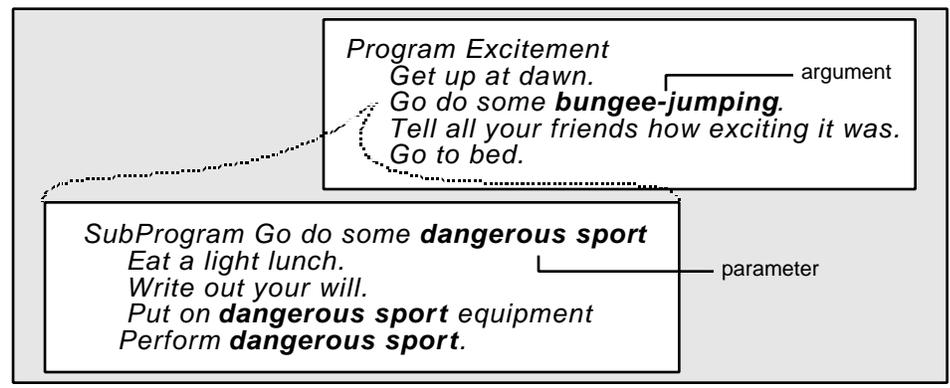
Pseudocode 7.2 Program for an exciting day of sky-diving

Program Excitement
 Get up at dawn.
 Go do some **sky-diving**.
 Tell all your friends how exciting it was.
 Go to bed.

In both cases, most of the steps in *Program Excitement* remain the same. Only the name of the sport varies: we call *bungee-jumping* or *sky-diving* in the program. If we take a closer look at the line *Go do some bungee-jumping*, we notice that it refers to more instructions.

Actually, the line *Go do some bungee-jumping* is a subprogram invocation. It calls *SubProgram Go do some dangerous sport* and performs its instructions by replacing *dangerous sport* with *bungee-jumping* each time *dangerous sport* is written. We call *dangerous sport* the subprogram's *parameter* and *bungee-jumping* the main program's *argument*. This program and subprogram are shown in Figure 7.8.

Figure 7.8 Subprogram Go do some dangerous sport



The steps to follow for *bungee-jumping* are given in Pseudocode 7.3.

Pseudocode 7.3 Steps for bungee-jumping

Get up at dawn.

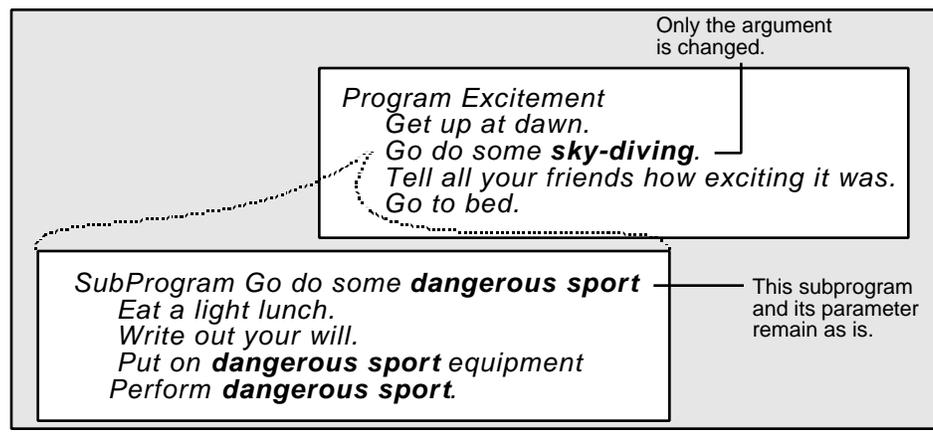
Eat a light lunch.
Write out your will.
Put on bungee-jumping equipment.
Perform bungee-jumping.

Go do some dangerous sport is invoked.

Tell all your friends how exciting it was.
Go to bed.

Notice that the subprogram is general enough so that it may be used for just about any risky sport: bungee-jumping, sky-diving, heli-skiing, and so on. Its parameter remains the same for each of them. However, as the sport practiced changes, so does the main program's argument. If you preferred to go sky-diving, as mentioned above, only the argument *bungee-jumping* would need to be changed. The rest of the program (including the subprogram) would remain as is. Figure 7.9 shows our *Excitement* program modified for sky-diving.

Figure 7.9 Excitement program using sky-diving

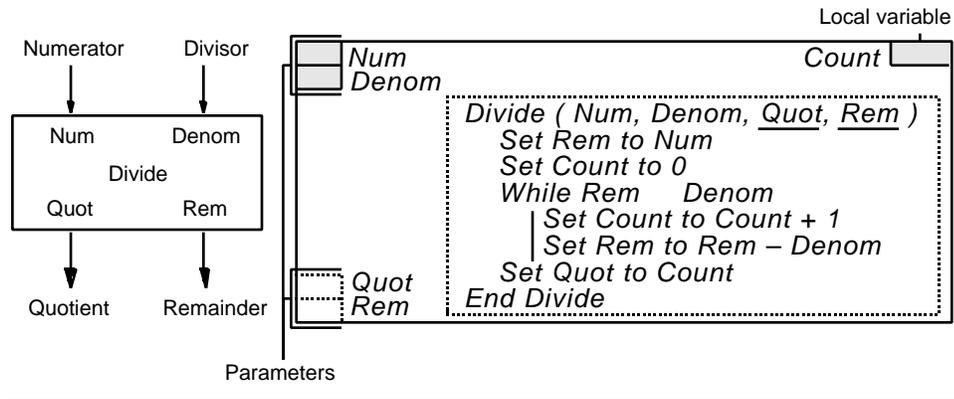


Data Space Diagrams

In previous chapters, when we considered sub-algorithms and subprograms, we concentrated on the flow of data (data-flow diagrams) and the flow of control (flowcharts, flowblocks, and pseudocode). We have just seen how during a subprogram invocation, a link is established between its call's arguments and the subprogram's parameters and that this link is used to transmit data. Now we need to be more complete. We also must concentrate on the *space* occupied by the data and on *how* the data values are communicated from one space to another.

To introduce the concept of a data space diagram, let's consider again the *Divide* subprogram. We already saw *Divide* in Chapters 3 and 5, and used it in many algorithms such as *Change Maker*, *Convert Grams*, and *Decimal to Binary*. A particular version of the *Divide* subprogram is shown in Figure 7.10.

To see these algorithms in more detail, see Figures 3.32, 3.24, 3.25, 3.26 and 3.35.

Figure 7.10 A version of the Divide subprogram

On the left of Figure 7.10, there is a data-flow diagram of *Divide*, showing that it has four parameters. Two of them, *Num* and *Denom*, are passed *in* and the other two, *Quot* and *Rem*, are passed *out*.

On the right of the same figure, the corresponding *data space diagram* shows the actual space occupied by the variables associated with the subprogram. Also shown here is the pseudocode of the subprogram, from which we see that *Divide* makes use of a temporary variable *Count* in the course of performing the division. *Count* is private, or *local*, to *Divide*. Programs that make use of *Divide* do not need or have access to *Count*.

In the pseudocode for the subprogram (Figure 7.10), the title line shows the names of the subprogram's parameters, *Num*, *Denom*, *Quot* and *Rem*. The *underlining* of the names of two of the parameters, *Quot* and *Rem*, denotes that their values will be passed *out* of the subprogram.

Note: In this chapter, only parameters that pass values out of a subprogram are underlined. The corresponding arguments are not underlined.

The following conventions are used in data space diagrams:

1. Subprogram parameters, both passed-in, like *Num* and *Denom*, and passed-out, like *Quot* and *Rem*, are drawn at the left of the diagram.
2. Passed-in parameters, like *Num* and *Denom*, are drawn as boxes at the top left. Passed-in parameters correspond to the arrows pointing into a data-flow diagram.
3. Passed-out parameters, like *Quot* and *Rem*, are shown as dotted boxes at the bottom left. Passed-out parameters correspond to the arrows pointing out of a data-flow diagram.
4. Local variables, like *Count* in our example, are accessible only from within a subprogram and are drawn at the upper right of the data space diagrams. Local variables have no meaning outside the subprogram, and are used to hide or protect any data that have no need to be accessible from the outside.
5. Solid boxes used for local variables and passed-in parameters represent actual memory locations within the subprogram, where the values are stored.
6. Dotted boxes used for the passed-out parameters do not represent actual space. They refer or point to actual memory locations outside the subprogram.

Figure 7.11 Data flow and data space diagrams

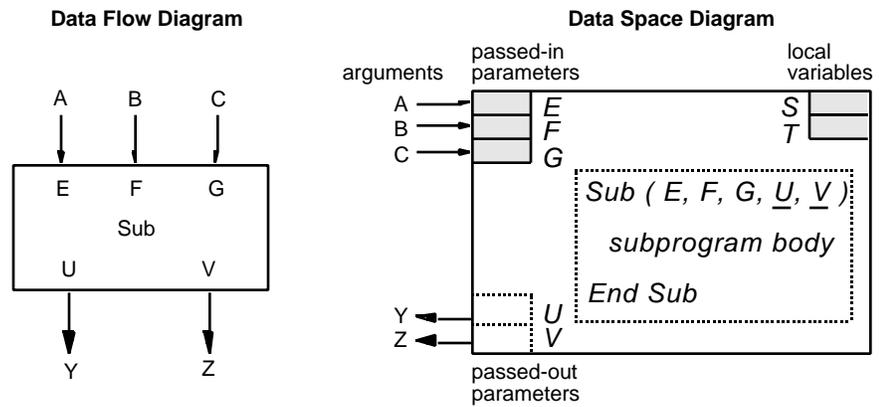
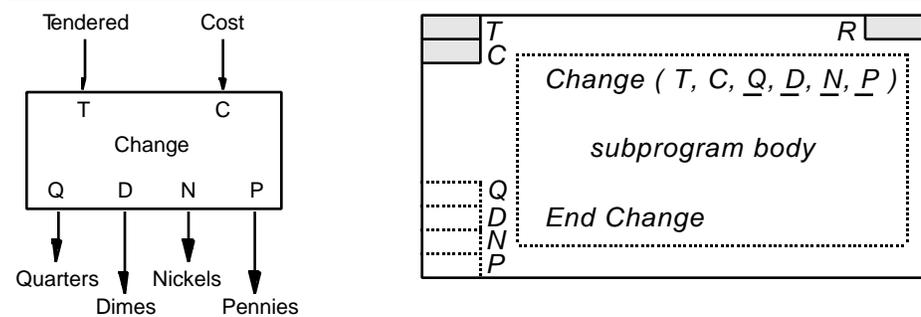


Figure 7.11 presents a general example of a data flow and data space diagram for a subprogram *Sub* with five parameters. The data space diagram at the right of the figure also shows the connections between arguments and parameters for this invocation of *Sub*:

Sub(*A*, *B*, *C*, *Y*, *Z*)

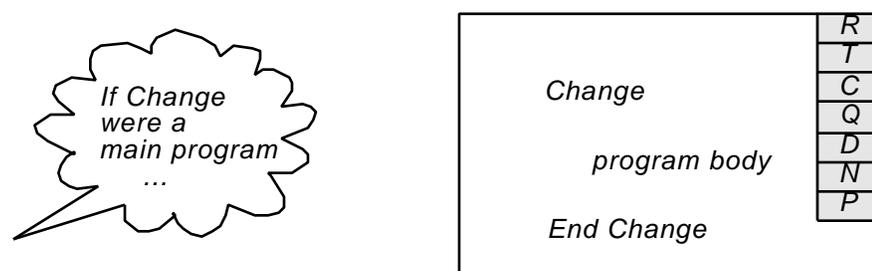
In such a call, the arguments are enclosed in parentheses and separated by comas.

Figure 7.12 The Change subprogram



The subprogram *Change*, shown in Figure 7.12, has six parameters: an amount tendered *T* and a cost *C*, which are passed-in, and the number of quarters *Q*, dimes *D*, nickels *N*, and pennies *P* that form the change, which are passed-out. There is also a local variable *R*, representing the remaining money at each stage during the computation of the change.

Figure 7.13 Change as a program



An example of this is the Change Maker programs (all 1 through 4) of Chapter 5, Section 5.7.

Let's say *Change* were a main program:

- There would be no parameters,
- All of the variables would be local to the main program—in other words they would become *global* variables. Remember that global variables are accessible to all parts of the program, including to any subprograms the program involves.

Now that we know the basics, we are ready to examine more closely just how data are input and passed in and out of subprograms.

7.3 Parameter Passing

Passing Parameters In and Out

The actual way in which the data are passed in or out of a subprogram depends on whether they correspond to input parameters (passed-in) or output parameters (passed-out):

- Input parameter: the parameter behaves as a local variable that is initialized by the value of the argument. Once this initialization has taken place, the link between the argument and parameter is broken. Thus, even though the subprogram may change the value of the parameter, this has no effect on the corresponding argument. Since only the argument's value is used, the argument can be a constant, variable or expression. This mechanism for passing data is called *passing parameters by value*.
- Output parameter: the parameter is linked to the argument in such a way that all references to the parameter in the subprogram become references to the argument, which must be a variable. Any change to the value of the parameter by the subprogram is a change to the value of the argument. Thus, the argument and the parameter become equivalent during this invocation of the subprogram. This mechanism for passing data is called *passing parameters by reference*.

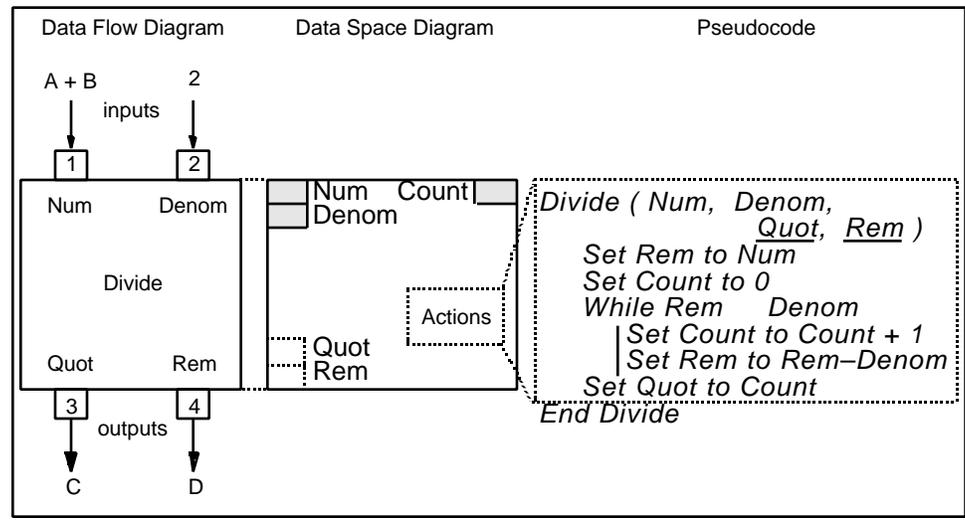
These two methods will be illustrated first by a simple program, *AverageExample*, that uses the *Divide* subprogram, and later by another program, *Change*, that also uses *Divide*.

Figure 7.14 give two different views of the subprogram *Divide*. It includes a data-flow diagram and a data space diagram with a description of the algorithm in pseudocode. The data-flow diagram shows the invocation:

$$\text{Divide}(A + B, 2, C, D)$$

where input parameters *Num* and *Denom* are initialized with the values $A + B$ and 2, and where output parameters *Quot* and *Rem* are made equivalent to C and D .

Figure 7.14 Two views of subprogram Divide



Our program *Average Example* is shown in Figure 7.15 and uses this same subprogram *Divide* to find the average of two values A and B . It also displays “Exactly” when the computed average is exact. Otherwise, it displays “Approximate”.

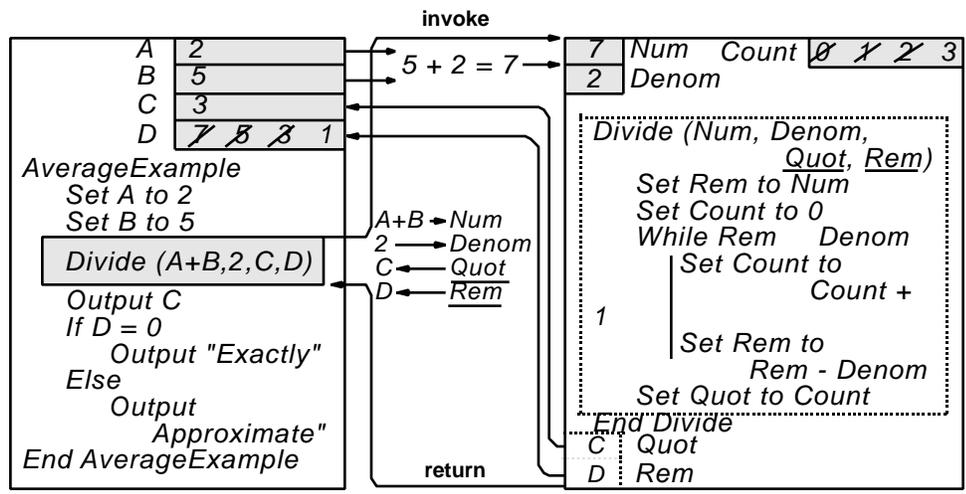
We will use our *Average Example* program to describe in detail what happens when a subprogram is invoked. The statement that invokes the *Divide* subprogram is the same as before:

$$\text{Divide}(A + B, 2, C, D)$$

Executing this invocation causes the following sequence of actions:

1. The point of return in *AverageExample* is immediately noted. The point of return is the point to which control will return after *Divide* has completed its work. Here, the point of return is the statement: *Output C*.
2. The variables for the input parameters of *Divide* and the local variable are set up. We prepare memory space to hold the values for *Num*, *Denom*, *Count*. Note that no memory space is reserved for the two output parameters *Quot* and *Rem*. Remember from the last section that only solid boxes take up memory space.
3. The links between the *Average Example* program and the *Divide* subprogram are established by setting up a correspondence between the arguments in the invoking statement $\text{Divide}(A + B, 2, C, D)$ and the parameters shown in *Divide*'s header $\text{Divide}(\text{Num}, \text{Denom}, \underline{\text{Quot}}, \underline{\text{Rem}})$
 - a. The expression $A + B$ is evaluated to 7 and that value is copied into *Divide*'s variable *Num*.
 - b. The value 2 is copied into *Divide*'s variable *Denom*.
 - c. The name of *AverageExample*'s variable C is linked to *Divide*'s parameter *Quot* so that *Quot* acts as an alias for C .
 - d. The name of *AverageExample*'s variable D is linked to *Divide*'s parameter *Rem* so that *Rem* acts as an alias for *Average Example*'s variable D .

Figure 7.15 The program Average Example invoking Divide



4. Subprogram *Divide* is executed: its actions are carried out. Each time the output parameters *Quot* and *Rem* are modified (by the *Set Rem to Rem-Denom* and *Set Quot to Count* statements), the actual values that are changed are those of *Average Example*'s variables *C* and *D*.
5. The memory space for *Divide*'s variables, *Num*, *Denom* and *Count* is released. If the subprogram *Divide* were re-invoked, a completely new memory space would be used for these three variables.
6. Control is returned to *Average Example* at the point of return, which is *Output C*.

This simple example might seem a bit complicated when we look at it with that much detail! Be sure to understand it completely as it illustrates the two methods by which data are passed between a program and an invoked subprogram:

- *By Value:* The arguments (*A + B*, *2*) corresponding to *Divide*'s parameters *Num* and *Denom* were *passed by value*. With this method, an argument (which can be an expression like *A + B* or a constant like *2*) is evaluated and copied into the temporary space allocated to the corresponding parameter in the subprogram. If this value is then changed in the subprogram, the change remains local to the subprogram and does not affect the original corresponding variable in the calling program.
- *By Reference:* The arguments (*C* and *D*) corresponding to *Divide*'s parameters *Quot* and *Rem*, which were underlined in *Divide*'s header, were *passed by reference*. With this method, the parameters of the called subprogram become *aliases* for the actual variables in the calling program. This requires that arguments be variables. Whenever the parameters of the subprogram are assigned new values (as in *Set Quot to Count*), it is the values of the corresponding arguments in the calling program that are actually changed (like *C* in this instance).

To help you remember the difference between these two methods, we can say that passing parameters between program and invoked subprogram can be done via two channels of communication:

- *By Value*: one-way communication from calling program to invoked subprogram
- *By Reference*: two-way communication.

Deciding on which way to pass parameters is usually clear from the data-flow diagram. Parameters that are input (with arrows into the box) should be passed by value: in Figure 7.14 this is the case for parameters 1 and 2. Parameters that are output should be passed by reference: in Figure 7.14 this is the case for parameters 3 and 4. Sometimes, parameters serve a dual role—both input and output. The subprogram uses the value of the argument and then modifies it. Such a parameter is passed by reference so that its modified value can be passed back to the caller. We will illustrate this with many more examples in the following sections.

Note: Arguments passed by value can be variables, expressions, or constants. Arguments passed by reference can only be variables. They cannot be expressions or constants.

Special Cases

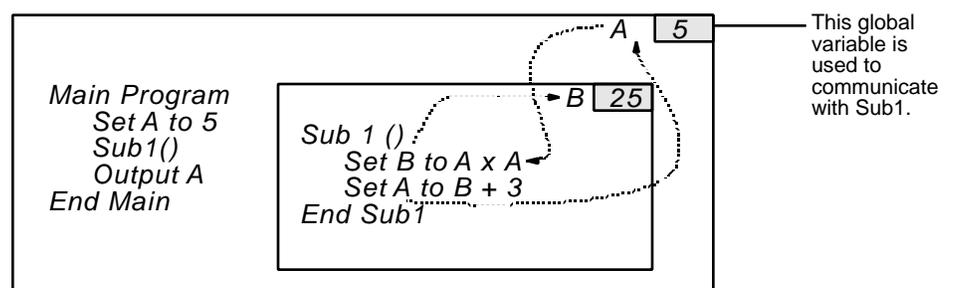
In the previous section, we were introduced to the communications between programs and subprograms via parameters. Among all possible cases for subprograms, there are two special cases:

- Subprograms with no parameters (only local variables), and
- Subprograms with only parameters (no local variables).

We will now take a look at both of these extreme cases in order to improve our understanding of communications among subprograms.

The case of a “parameterless” subprogram is illustrated by the example in Figure 7.16. There, the main program has a global variable *A* and the subprogram has one local variable *B*. The main program communicates with the subprogram through the global variable *A*.

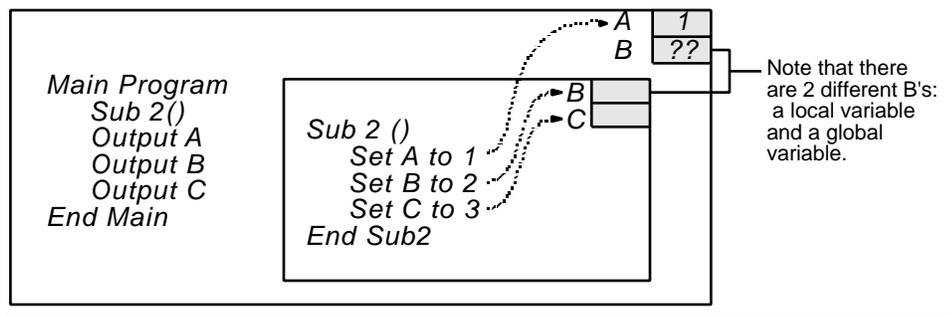
Figure 7.16 Parameterless communication



In Figure 7.16, *A* is being used not only to receive the results of the subroutine’s calculation, but also send the data to be used by the subroutine. This use of global variables is fairly common but is not recommended because it makes the program more difficult to understand. The problem is that when you read the invocation *Sub 1 ()*, there is no indication that the global variable *A* is being used as a communication channel and that its value will be changed. To discover this, you must study *Sub 1* in detail. This is not practical. Such implicit uses of global variables often lead to hard-to-find problems.

In the example in Figure 7.16, the names of the variables are different. What would happen if this were not so and the name of a local variable in the subprogram were the same as the name of a global variable?

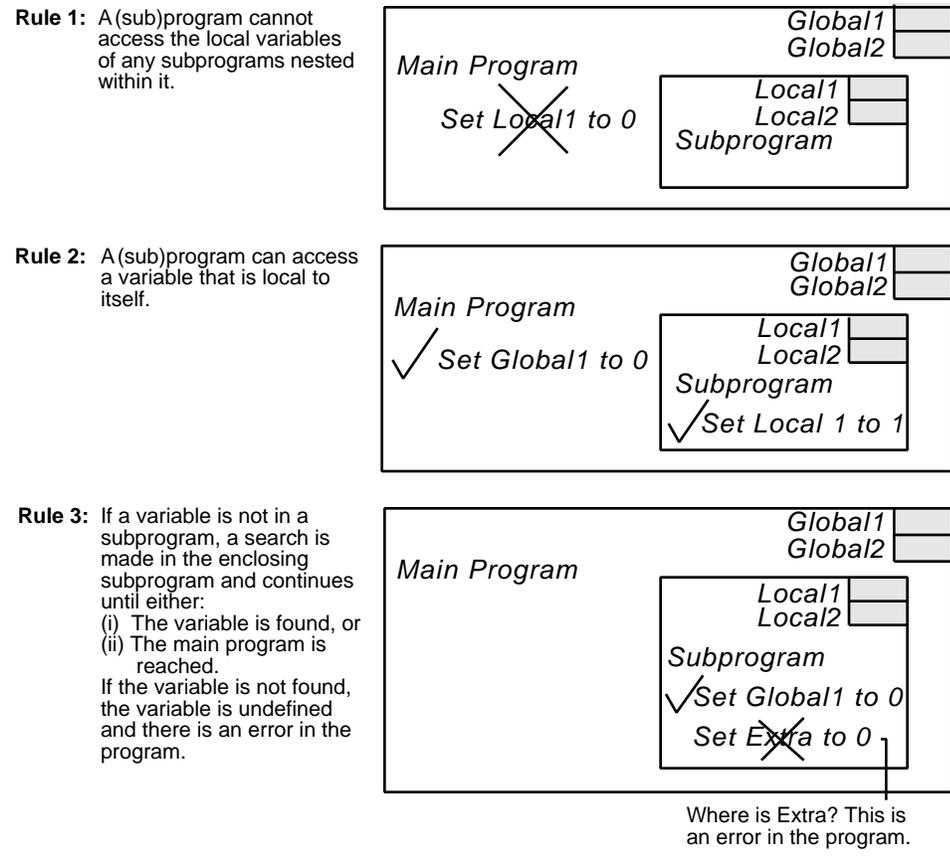
Figure 7.17 Parameterless communication with name duplication



In Figure 7.17, there is a main program with two variables *A* and *B*, and a subprogram with two local variables *B* and *C*. Notice that we have used the same name *B* to refer to two different variables, one in the main program and one in the subprogram. We have done this on purpose to illustrate the independence of name spaces. In practice, it is not recommended to use identical names in different parts of a program even though it is allowed. We will say more about this in a later section.

When we have subprograms nested in the main program or within other subprograms, access to the various variables is determined by the rules in Figure 7.18.

Figure 7.18 Rules on using variables in subprograms



Let's apply these rules to the example of Figure 7.17.

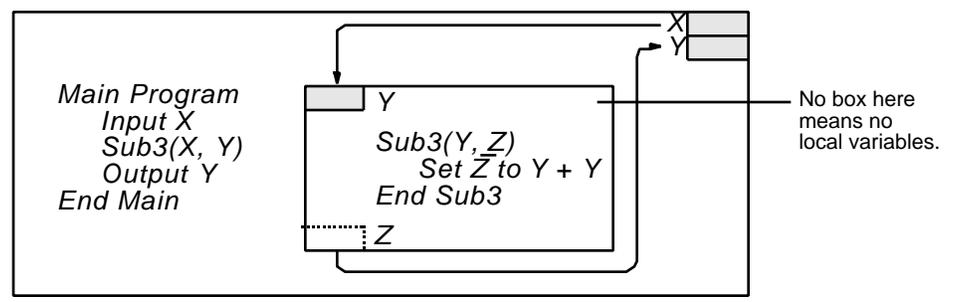
1. When the main program calls subprogram *Sub2*, it does so with the statement *Sub2()* where the argument list is empty because the subprogram has an empty parameter list.
2. Subprogram *Sub2* assigns values to the three variables *A*, *B* and *C*, beginning with *Set A to 1*. First, since there is no variable *A* within *Sub2* (Rule 3), it looks for the variable *A* in the next higher block, the main program in this case. It finds *A* in *Main Program* and sets this to the value 1. Let's call this variable A_{Main} to show where it is located.
3. The next variable *B* is found within *Sub2* (Rule 2), so that B_{Sub2} receives the value 2. The outer variable B_{Main} cannot be accessed by *Sub2* since B_{Sub2} exists and was found first (Rule 3). Its value was never set and remains undefined, denoted by "??".
4. Finally, the variable C_{Sub2} is assigned the value 3 (Rule 2).
5. After *Sub2* is invoked, the *Main Program* is to output *A*, *B* and *C*. Remember that by Rule 1, *A*, *B* and *C* can only be variables local to the *Main Program*. So *Main Program* first outputs the value of *A* (Rule 2), which is 1.
6. *Main Program* then attempts to output variable B_{Main} , but fails as B_{Main} has no value (Rule 2).
7. Finally, the attempt to output the value of *C* also causes an error because the main program cannot access C_{Sub2} (Rule 1).

It should now be clear that we can use local variables to hide some values, just as *Sub2* hid the values 2, and 3 of B_{Sub2} and *C*. This simplifies the program structure and makes the resulting program much less prone to error. However, the access to variables at higher levels—global variables—provided by Rule 3 is very dangerous and should be avoided. Why? Because, as we said before, communication through global variables is not explicit and occurs sometimes without us being aware of it.

Tip 1: Only use parameters when communicating between programs and subprograms.

Tip 2: Try to use different variable names whenever possible.

Figure 7.19 Parameter-only communication

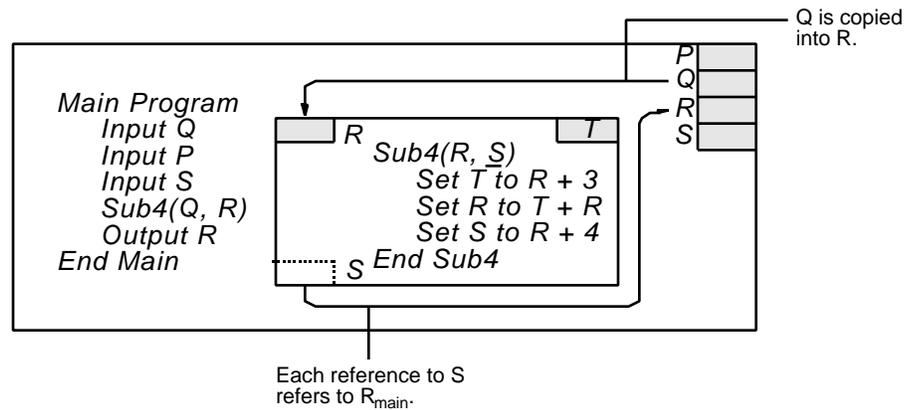


Our next example, in Figure 7.19, uses only parameters to communicate, and *Sub3* has no local variables. The main program has two global variables *X* and *Y*, and the subprogram has two parameters *Y* and *Z*. Notice that we have again used the same variable name *Y* in both the main program and the subprogram, but there are no ambiguities as there were in Figure 7.13.

The subprogram header $Sub3(Y, Z)$ indicates that the first parameter, Y , is passed by value and the second parameter, Z , is passed by reference. When the subprogram is called, the value of the first argument, X , is copied into Y_{Sub2} , is doubled and then copied into the space of the second argument, Y_{Main} .

Note: It is still preferable to use different variable names when possible.

Figure 7.20 General communication



In general, subprograms have both parameters and local variables, as shown in Figure 7.20. You will note several things here:

- Variables P and Q are global variables, which could be accessed by the subprogram $Sub4$, but should not be!
- *Main Program* has 2 more global variables: R and S . Normally they too would be accessible to $Sub4$. However, $Sub4$'s parameters are *also* called R and S . This means that any references to R and S in the subprogram are to those parameters. The global R and S of *Main Program* are therefore not accessible to $Sub4$.
- When the subprogram is invoked, the first argument is passed by value. The value of variable Q is copied into the variable R_{Sub4} and the link with Q is lost. So the value of Q cannot be modified by the subprogram even when it makes an assignment to R_{Sub4} .
- The second argument is passed by reference, and every reference to S in the subprogram refers to R_{Main} .
- Any references to variable S in the main program refer to the global S (Rules 1 and 2).
- Variable T , local to subprogram $Sub4$, is inaccessible from *Main Program* (Rule 1).

You might already have noted in the preceding examples that names are very significant. You have also seen in the last three examples that the *same names* may refer to *different data spaces* and have *different meanings* as shown in Figure 7.21. For example, the same name R is the result in the main program and is the quantity received in the subprogram. Also, S is both the sum in the main program, and the value sent by the subprogram.

Figure 7.21 Table of names and meanings

Names and Meanings of variables used in Figure 7.20		
Name	in <i>Main Program</i>	in Subprogram <i>Sub4</i>
<i>P</i>	<i>Paid</i>	
<i>Q</i>	<i>Quantity</i>	
<i>R</i>	<i>Result</i>	<i>Received</i>
<i>S</i>	<i>Sum</i>	<i>Sent</i>
<i>T</i>		<i>Temporary</i>

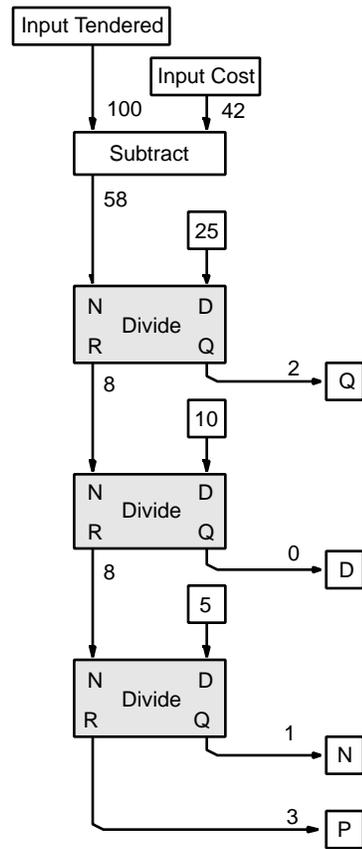
Such conflicts in naming can become very confusing for humans, but are not a problem for computers. Normally, such short and duplicated names are to be avoided, but if they happen to be chosen, they cause no problem for the computer and confuse only the reader. Obviously, in our small examples we could easily have chosen different names that could have helped the reader understand more easily. The important thing to note is that, in large programs where different people work on many parts, we do not need to have some elaborate scheme to prevent programmers from using names already used by others.

In most cases the names chosen would also be more meaningful, as they have been in the examples of the preceding chapters. However, here we have used short names to keep the lists of subprogram parameters and arguments short as well. In practice, we will always use meaningful names.

Tip: Always use meaningful names for variables, such as *Result* or *Received*. This way, your algorithms are easier to read.

Some Examples...

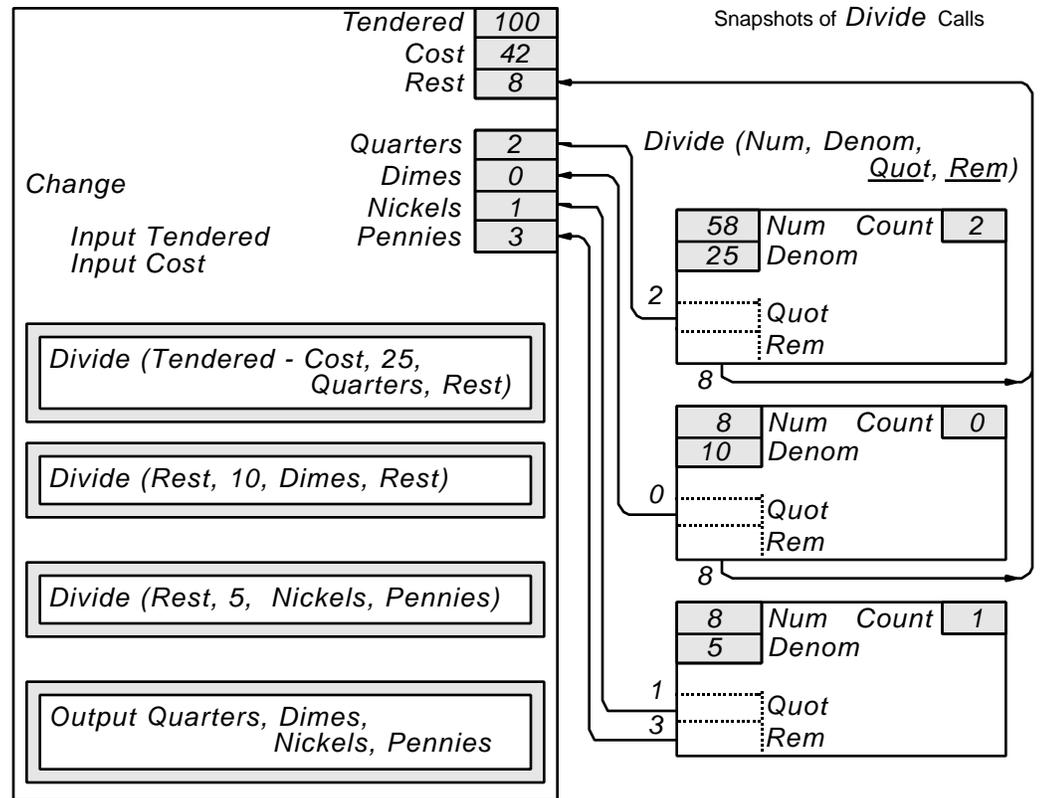
Figure 7.22 Data-flow diagram for Change program



See Figure 3.32 for the original Change Maker data-flow diagram.

Let's consider an algorithm *Change*, which calls the subprogram *Divide* three times. The *Change* algorithm inputs the amount *Tended* and the *Cost* and outputs the number of *Quarters*, *Dimes*, *Nickels*, and *Pennies*. It was introduced in Chapter 3 and is shown in the data-flow diagram of Figure 7.22.

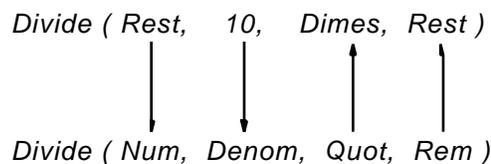
Figure 7.23 Data space diagram for Change program



The data space diagram of Figure 7.23 has been extended to show snapshots of the subprogram calls. Even though the figure shows three instances of the *Divide* subprogram, it should be noted that only one *Divide* subprogram exists at any time.

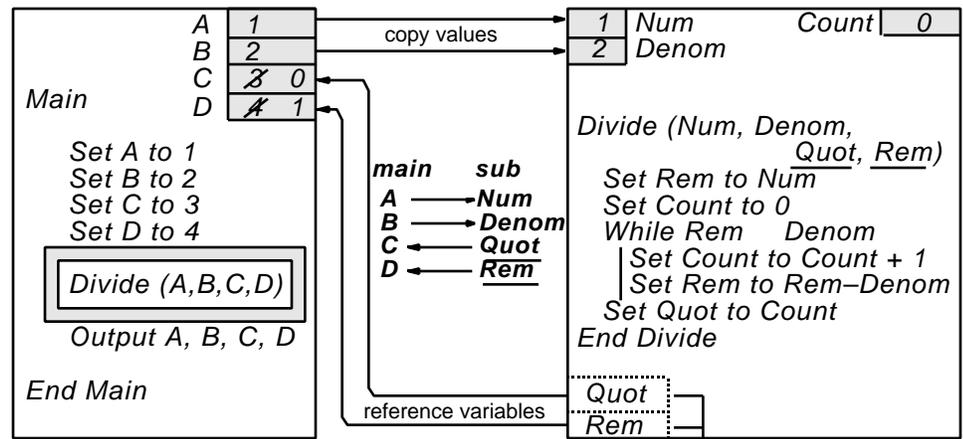
- The first call to *Divide* passes the value of the expression *Tendered - Cost* and the constant 25 as the denominator. *Divide* then returns the quotient in *Quarters* (2) and the *Rest* of the change (8).
- The second invocation of *Divide* passes the value of *Rest* obtained from the first call and the constant 10. *Divide* then returns the quotient in *Dimes* (0), and the same *Rest* (8). The correspondence between the arguments *Rest*, 10, *Dimes*, *Rest* and the parameters *Num*, *Denom*, *Quot*, *Rem* is established by their order of listing from left to right, as shown in Pseudocode 7.4. The fact that two arguments are the same variable (*Rest*) causes no problem. The value of *Rest* is used to initialize *Num* which is divided. When *Rem* is set to the remainder, this actually sets a new value in *Rest*, as illustrated in Pseudocode 7.4.
- The third and last call to *Divide* passes in the *Rest*, and the constant 5, and returns the number of *Nickels* and also of *Pennies*.

Pseudocode 7.4 Two arguments with the same variable



Normally, the examples we have just seen on how to pass parameters should be sufficient to understand everything about parameter passing. However, you will need more practice in actually passing parameters to reach complete understanding. We will look at a few more examples to help you with it. In fact, many aspects of parameter passing can be illustrated by the familiar simple *Divide* subprogram. We will look at some of these aspects here.

Figure 7.24 The subprogram invocation *Divide* (A, B, C, D)



Notice that we put the names *Quot* and *Rem* inside of the dotted boxes to serve as a reminder never to put any numbers there.

See Figure 7.15 for a closer look at Average Example.

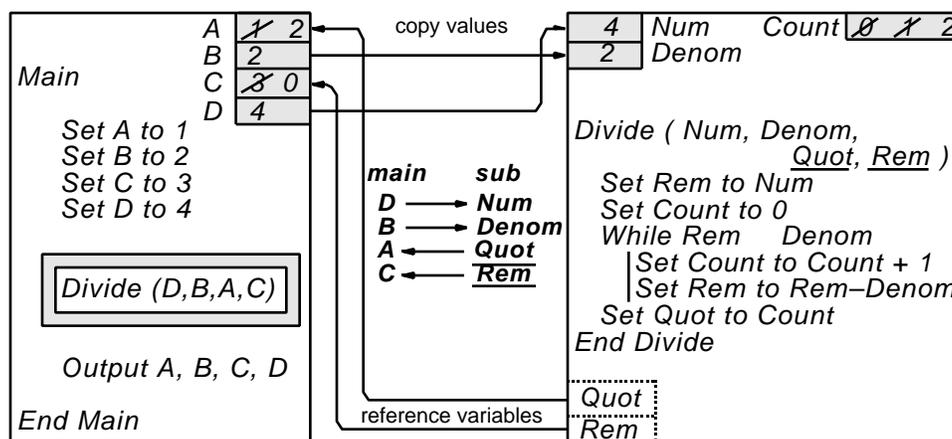
Here is a simple program (Figure 7.24) very similar to the already-seen *Average Example* program. Let's review how the subprogram *Divide* works. *Divide*(A, B, C, D) firstly copies the values of A and B into *Num* and *Denom*. The quotient *Quot* refers to C and *Rem* refers to D. The output of *Main* here are the values of A, B, C, and D: 1, 2, 0, 1.

Drawing Data space diagrams such as the one above clarifies parameter passing. You may note that *Quot* and *Rem* were written inside the dotted boxes.

Tip : Writing the output parameter names inside the dotted boxes can prevent you from making the common mistake of putting values in the boxes. Remember that passed-out parameters are only references to actual data spaces. Also, using arrows to match arguments with parameters can be very helpful.

We will use this simple *Main* program to see how changing subprogram arguments can affect output. The *Main* program contains 4 variables: A, B, C, D. These 4 variables can be passed in/out of *Divide* at least $4 \times 4 \times 4 \times 4 = 256$ different ways (such as AABC, AACB, ABCD, ABDC, ACBD, and so on). The next few examples illustrate some of these variable combinations.

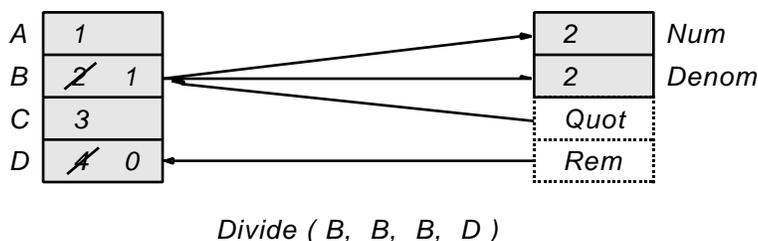
Figure 7.25 Subprogram invocation Divide (D, B, A, C)



In Figure 7.25, the subprogram invocation becomes `Divide(D, B, A, C)`. This time, the values of *Main* program variables *D* and *B* are passed into subprogram parameters *Num* and *Denom*. Quotient *Quot* references *A* whereas remainder *Rem* references *C*. Again, only *A* and *C* of the *Main* program are modified, and the output is 2, 2, 0, 4.

In our next example, the invocation becomes `Divide(B, B, B, D)`. Figure 7.26 shows the corresponding data space.

Figure 7.26 Data space for invocation Divide (B, B, B, D)



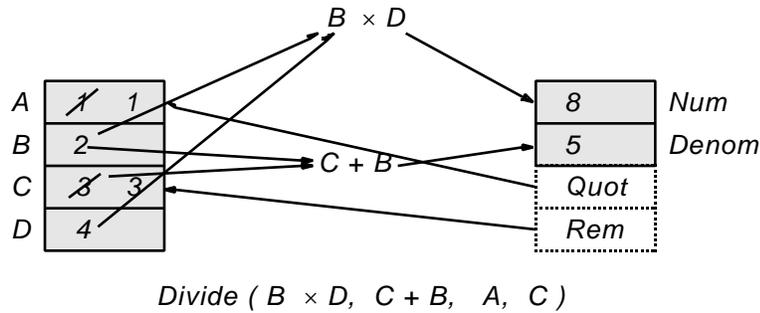
In Figure 7.26, the same value of *B*, which is 2, is copied into both subprogram parameters *Num* and *Denom*. The action of *Divide* produces a quotient of 1 and a remainder of 0, as it is dividing 2 by 2. Now *Quot* also refers to variable *B* of the main program, so that value is now changed to 1. The zero remainder is assigned to variable *D* of the *Main* program, and the output is then 1, 1, 3, 0.

In this case, the main program’s variable *B* was actually used twice to copy a value into *Num* and *Denom*, and that same variable *B* was also referenced by *Quot*, and then modified when a value was assigned to *Quot*. What would happen if we used the variable *B* for all 4 arguments? Try it and see.

Note: Using the same variable as an argument to two different output parameters, in a single invocation of a subprogram, may lead to results that are difficult to predict. *Don't do it!*

In our next example, illustrated by Figure 7.27, the invocation was changed to `Divide(B×D, C+B, A, C)`, and the data space for the invocation shows that expression values can be passed into a subprogram.

Figure 7.27 Data space for the invocation Divide (B × D, C + B, A, C)



In this case, the evaluation of the two expressions is done first: $B \times D = 2 \times 4 = 8$ and $C + B = 3 + 2 = 5$. Then, the values of 8 and 5 are copied into the subprogram parameters *Num* and *Denom* producing a quotient *Quot* of 1 and a remainder *Rem* of 3. Since parameter *Quot* refers to variable *A* the value of *A* is set to 1. Similarly, since parameter *Rem* refers to variable *C*, its value is set to 3, and the output is then 1, 2, 3, 4. The four values are exactly the same as before the subprogram call. What a complex way of doing nothing!

Figure 7.28 Data space for invocation Divide (11, 7, A, B)

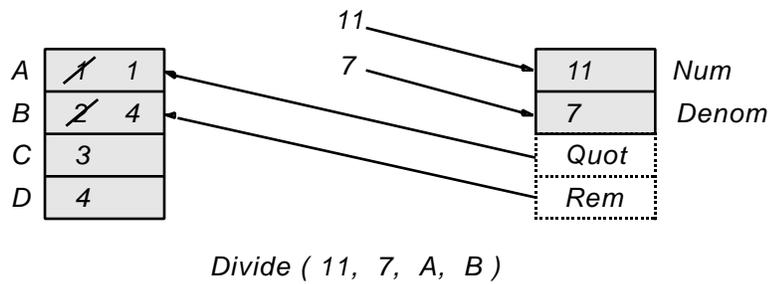
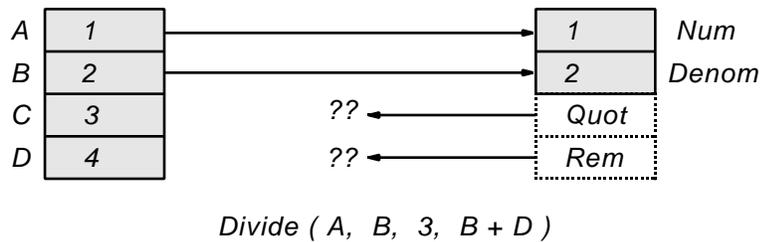


Figure 7.28 shows the data space for the next example where the invocation has become *Divide(11, 7, A, B)*. This shows that constants can be used as value arguments. Here, 11 is divided by 7 to yield a quotient of 1 and a remainder of 4. The subprogram quotient *Quot* refers to variable *A* and *Rem* refers to variable *B*, so the output is 1, 4, 3, 4.

Figure 7.29 Data space for the invalid Divide (A,B,C,B+D)



The invocation *Divide(A, B, 3, B + D)*, Figure 7.29 has two errors. The problems are:

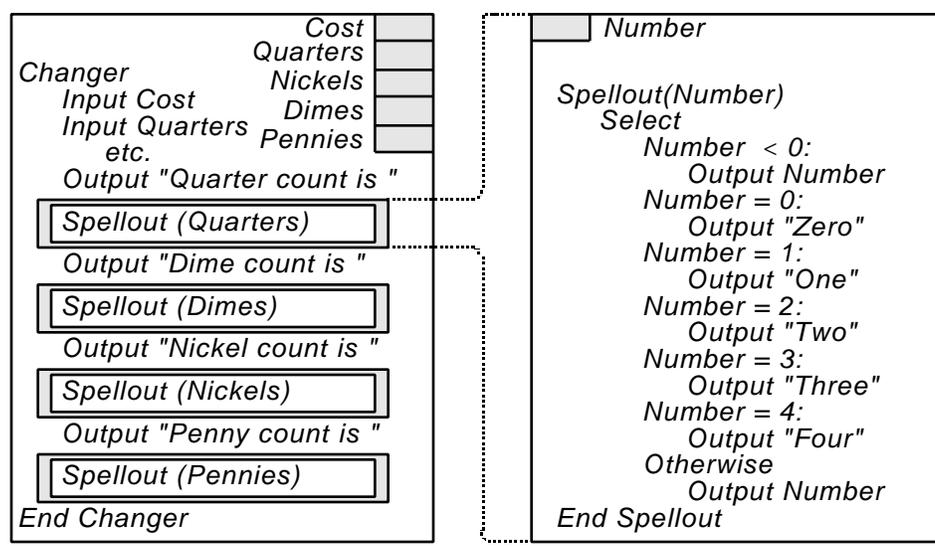
- The third argument is a constant. Arguments passed by reference must be variables.
- The fourth argument is an expression. Again, arguments corresponding to output parameters can only be variables.

So far in this chapter, we have looked at subprograms and the way they communicate with their environment through parameters. We have identified methods of passing parameters: by value and by reference, corresponding to one-way and two-way communications. Let's look at a few more examples that illustrate the following ways of passing parameters:

- Passed-in parameters only,
- Passed-out parameters only,
- Passed-in and passed-out parameters.

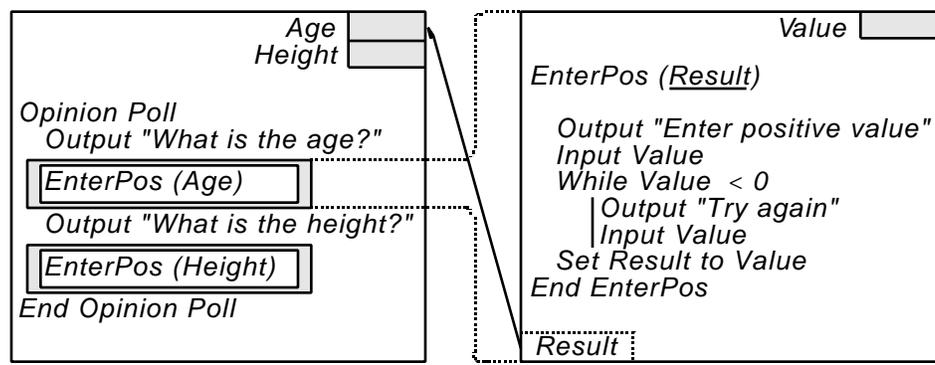
Our first example, subprogram *Spellout*, is shown in Figure 7.30. It outputs some small numerical values, *not as numbers* but spelled out as a word. The pseudocode shown in the figure only spells out integers in the range 0 to 4, but you could extend it easily. Here only one parameter, the *Number* to be spelled out, is passed into the subprogram. Nothing is passed out. However, the result of the invocation is the output of a value.

Figure 7.30 Parameter passing passing in only



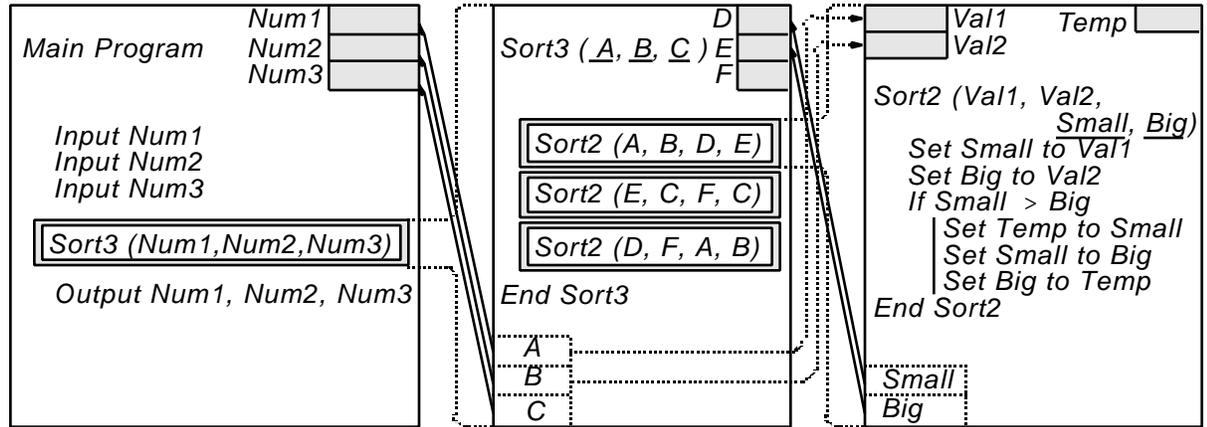
The second example in Figure 7.31 shows another subprogram, *EnterPos*, which uses an output parameter, passed by reference, to return a value to the calling program. The subprogram's pseudocode shows that *EnterPos* first prompts the user to enter a positive value, and then inputs the *Value*. As long as this *Value* is not positive, the subprogram keeps outputting the message "Try again" and inputting a new *Value*.

Figure 7.31 Parameter passing passing out only



As soon as a positive *Value* is entered, the loop terminates. The subprogram then assigns that positive *Value* to the output parameter *Result*, setting the value of *Age* and then *Height* in the main program. Using *Result* as an output parameter like this allows the subprogram *EnterPos* to be recycled. *EnterPos* can be used over and over to enter values into many different variables (such as *Age* and *Height* in the *Opinion Poll* program shown in Figure 7.31).

Figure 7.32 Parameter passing: passing “in-out” or through



Our third example shows why we can refer to parameters *passed by reference* as “passed in-out” or “passed through”. This may seem a little confusing at first, but let’s take a look at Figure 7.32. Shown is a *Main Program* which calls subprogram *Sort3*. *Sort3* sorts three values *A*, *B*, *C* into non-decreasing order with the help of another subprogram, *Sort2*.

Sort3’s parameters *A*, *B*, *C* are all passed by reference, as we can see by the underlining. Remember that parameters passed by reference are aliases of variables from the calling (sub)program. Here *A*, *B*, *C* are all passed by reference: they are all aliases of some variables. Which variables? *Num1*, *Num2*, and *Num3*. What is the calling (sub)program? *Main Program*.

Since *A*, *B*, *C* are aliases of *Num1*, *Num2*, and *Num3*, *Sort3* can perform whatever operations it wants on *Num1*, *Num2*, and *Num3*. In other words, *Sort3* has total control over those variables. When the variable *A* is called on for an operation, the value of *A* used is that of *Num1*.

So, it is as if we copied this main program value into *Sort3* and then performed the operation on it, as if we were passing values in *and* out (hence the name “in-out” or “through”). But we did not really copy it. To *actually* copy *Num1* into *Sort3*, we would have to add a parameter passed by value and *Sort3* would look like *Sort3*(*Z*, *A*, *B*, *C*). *Sort3* would have to have an extra box labeled *Z* in its top left corner. In fact, we could have made *Sort3* with 3 parameters passed in and *A*, *B*, and *C* passed out. Then we could say that parameters are passed in and passed out (not “in-out”).

The interesting thing to note here is that when *Sort3*(*A*, *B*, *C*) is executed, the original values of *Num1*, *Num2*, *Num3* are destroyed. Depending on the problem to be solved, this may be all right. However, it is usually preferable to pass the values in by value, and then out by reference, for example: *Sort3*(*Z*, *X*, *Y*, *A*, *B*, *C*). This way the original *Num1*, *Num2*, *Num3* values would be preserved.

For a closer look at *Swap*, see Chapter 5, Figure 5.18.

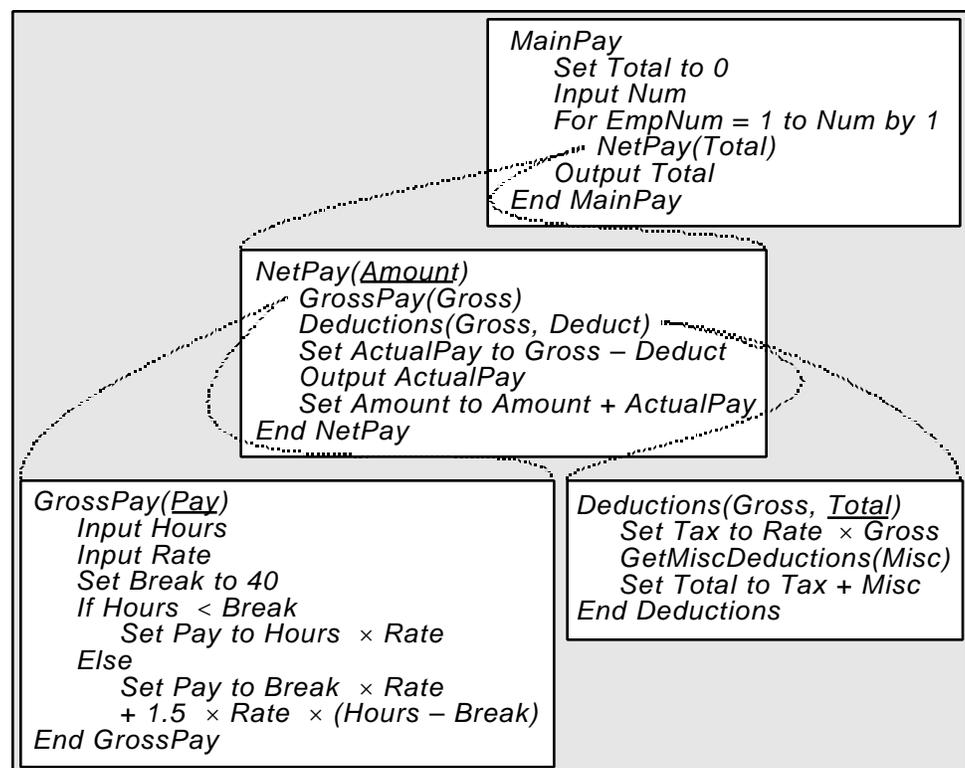
When examining Figure 7.32 closer, you may notice that subprogram *Sort2* sorts two values with the use of 4 parameters: two passed by value and two passed by reference. This means that it leaves the original values *A* and *B* (which in turn reference *Num1* and *Num2*) untouched. *Sort2* could just as well been created with only 2 parameters passed by reference (or “in-out” as we saw earlier). You may note that *Sort2* functions by swapping two values if *Small* is bigger than *Big*. The actual swapping could have been done by invoking a third subprogram, *Swap*.

7.4 Bigger, Leaner, and Meaner Programs

Using Structure Charts

So far, we have considered small programs and subprograms, with little interaction between them. We will consider now a more complex system to illustrate the inter-relations among actions, data spaces and data flows. The example we use is *MainPay*, an extended payroll problem that is sufficiently complex to suggest the power of subprograms in large systems.

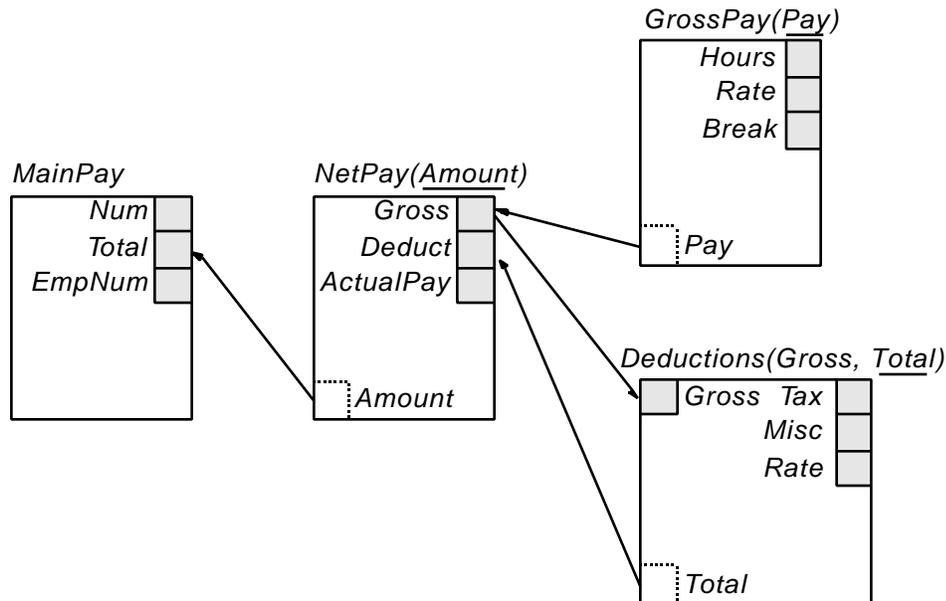
Figure 7.33 Algorithms and sub-algorithms for Main Pay



The algorithms and sub-algorithms that comprise *MainPay* are shown in Figure 7.33 as break-out diagrams of pseudocode, with the dotted lines representing the flow of control when subprograms are called. The program *MainPay* inputs the number of employees *Num*, and then executes a loop *Num* times, once for each employee. The body of the loop calls *NetPay* with *Total* as an argument, so that it can update the total pay as each employee’s pay is calculated. Finally, *MainPay* outputs the *Total* amount paid out.

Subprogram *NetPay* calls subprograms *GrossPay* and *Deductions* to be able to calculate the *Actual Pay* and output it, and *Actual Pay* is also used to update the *Total* in *MainPay*. The *GrossPay* subprogram computes the gross *Pay* in the usual way. Subprogram *Deductions* obtains the miscellaneous deductions *Misc* by calling *GetMiscDeductions*, and adds this to the *Tax*, which is a simple percentage *Rate* of the *Gross* pay. All these variables are either local or global and we will use a data space diagram to show where they belong.

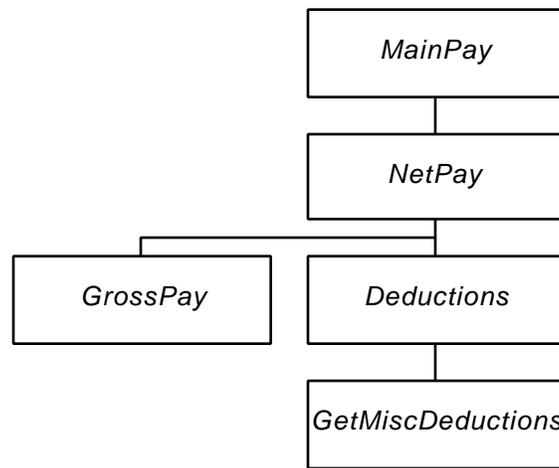
Figure 7.34 Data spaces of Main Pay and its subprograms



The data space diagram of *MainPay*, in Figure 7.34, shows how data are distributed among the subprograms. Notice first that the main program requires only three variables of its own (the count *Num*, the loop control variable *EmpNum*, and the amount *Total*), and need not have access to other variables at lower levels! Notice also that the gross pay appears three times in three different subprograms:

1. The actual gross pay is first computed in subprogram *GrossPay* and assigned to parameter *Pay* which is passed by reference.
2. Because the parameter *Pay* is an alias, this computed gross pay is in fact assigned to variable *Gross* in subprogram *NetPay* (the *GrossPay* call argument).
3. The same gross pay is also passed to subprogram *Deductions* as a value parameter where it is used to initialize *Deductions*' parameter *Gross*. Notice that, although both *NetPay* and *Deductions* have variables called *Gross*, they are quite distinct and occupy separate storage.

It is important to see that subprogram *Deductions* did not get the gross pay directly from subprogram *GrossPay*, but indirectly through subprogram *NetPay* at a higher level. This sub-dividing and hiding of data spaces is very significant in large systems. It localizes the various data values, and respects the hierarchy of subprograms defined by the structure chart for a system like Figure 7.35.

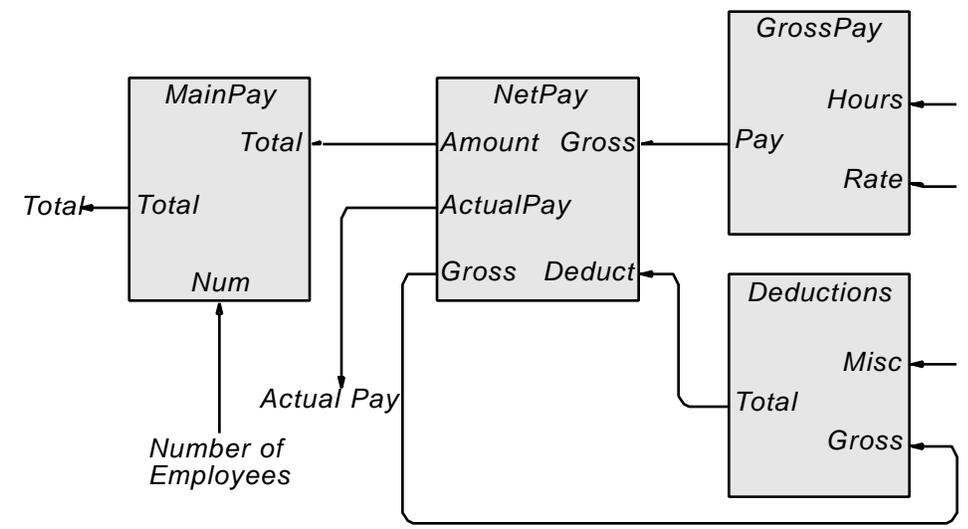
Figure 7.35 Structure Chart for Main Pay

This subdivision of the data space is extremely helpful when you are trying to find and correct an error in a large program. It reduces the amount of program that you must read and understand before you make a change. Making a local change without really understanding all its ramifications is not good enough. Even though the program may appear to work after the modifications have been made, it may not have been properly tested. The subdivision of the data space clearly defines the boundaries of possible ramifications. The big picture of the system offered by the structure charts helps to see this.

We will have yet another look at the *MainPay* example by means of a data-flow diagram, as shown in Figure 7.36. That figure shows how the external data flow in and out of the program, as well as how the data flow between the subprograms through arguments and parameters. The external inputs are the *Hours*, the pay *Rate*, the miscellaneous deductions *Misc*, and the number of employees *Num*. The external outputs are the *ActualPay* and the *Total* amount paid.

The hiding and sharing of data, when done properly, leads to a simplification of the building of the program with minimal interaction between the constituent subprograms. Each subprogram communicates only with its superior in the structure chart. Ideally, each subprogram is provided with only what it needs to perform its function:

- At lower levels, the system components need individual details of hours worked and rate of pay.
- At higher levels, the system components do not need the same details but need instead the total number of employees and the total amount paid.

Figure 7.36 Data flow of Main Pay and its subprograms

For instance, the *Deductions* subprogram needs the value of the gross pay, to use but not modify, in its computation. However, it does not need the net pay or the number of employees.

The interconnection of programs with many subprograms is often shown by trees or contour diagrams. Those are considered next.

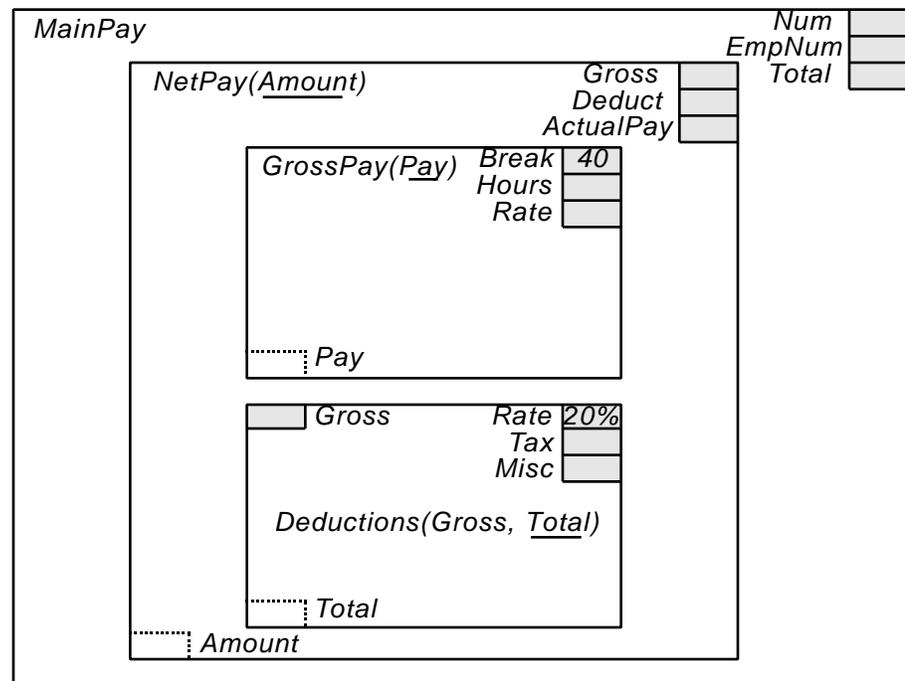
Contour Diagrams

Contour diagrams are similar to the data space diagrams we saw earlier in Section 7.2. They are made up of blocks—each representing a piece of program with its own data space. The blocks are nested one within another, showing the hierarchy of the program. A contour diagram of the Main Pay program is shown in Figure 7.37.

Notice that there are three types of information shown:

- The program or subprogram names
- Local variables
- Parameters passed in and passed out of the subprogram

As we can see, information can both be shared between blocks, and hidden from other blocks. Block-structured languages (like Algol 60, Pascal, Modula-2, C, and Ada) take advantage of this sharing/hiding of information.

Figure 7.37 Contour diagram of Main Pay

The variable access rules introduced in Section 7.3 apply directly to the blocks. Let's see how the rules apply to Figure 7.37:

- **Rule 1:** A (sub)program cannot access the local variables of any subprograms nested within it.

Here, subprogram *NetPay* cannot access (use) the local variables *Break*, *Hours*, *Rate*, *Rate*, *Tax*, *Misc*, since the subprograms *GrossPay* and *Deductions* are nested within it. We could say that these inner variables are kept protected from the outside world. (You will note here that there are 2 separate variables named *Rate* in our program, each referring to something different. The one in *GrossPay* refers to an hourly rate of pay whereas the other in *Deductions* refers to the tax rate.)

- **Rule 2:** A (sub)program can access a variable that is local to itself.

Here, subprogram *NetPay* can access *its* local variables *Gross*, *Deduct*, and *ActualPay*.

- **Rule 3:** If a variable does not appear in a subprogram, a search is made in the enclosing subprograms. This search is continued until either:
 - The variable is found, or
 - The main program is reached.

If the variable is not found in any of these nests, then the variable is undefined and there is an error in the program.

Here, subprogram *NetPay* can access the variables *Num*, *EmpNum*, and *Total* (as well as its own local variables, as we saw in Rule 2). We can see this by “looking out” of *NetPay* to the next higher level(s). In our case, there is only one higher level: the main program. This rule of “looking out” is also called the “most-closely-nested binding rule”, where the term *binding* refers to the relations between variable names and data spaces.

Another useful term to know is the *scope* of a variable. This refers to the part of a program over which a particular name is “known”, or may be referenced. In our example, the scope of *Num* is the whole program. This comes about because it is defined in *MainPay* and nowhere else—by the application of Rule 3. The same is true of *EmpNum*—they are both global variables.

However, this is not true of *Total*, the third variable defined in *MainPay*, which is also defined with a different meaning in *Deductions*. Thus, the scope of *Total* defined in *MainPay* is the whole program except for *Deductions*. This illustrates the dangers of Rule 3 and how careful we must be in our program design to avoid accessing the wrong variable.

Tip: Use different variable names whenever possible.

Figure 7.38 Table of binding for Main Pay and subprograms

Names	<i>MainPay</i>	Subprograms			Notes
		<i>NetPay</i>	<i>GrossPay</i>	<i>Deductions</i>	
<i>ActualPay</i>		Defined	Access	Access	Not accessible in <i>MainPay</i>
<i>Amount</i>		Defined	Access	Access	Not accessible in <i>MainPay</i>
<i>Break</i>			Defined		Only in <i>GrossPay</i>
<i>Deduct</i>		Defined	Access	Access	Not accessible in <i>MainPay</i>
<i>EmpNum</i>	Defined	Access	Access	Access	Global
<i>Gross</i>		Defined 1	Access 1	Defined 2	Redefined in <i>Deductions</i>
<i>Hours</i>			Defined		Only in <i>GrossPay</i>
<i>Misc</i>				Defined	Only in <i>Deductions</i>
<i>Num</i>	Defined	Access	Access	Access	Global
<i>Pay</i>			Defined		Only in <i>GrossPay</i>
<i>Rate</i>			Defined 1	Defined 2	Different meaning in <i>GrossPay</i> and <i>Deductions</i>
<i>Tax</i>				Defined	Only in <i>Deductions</i>
<i>Total</i>	Defined 1	Access 1	Access 1	Defined 2	Meaning in <i>Deductions</i> different to meaning everywhere else.

Tables of bindings show how variable names relate to the data spaces of various blocks. A table for *Main Pay* is shown in Figure 7.38. There is one row for each variable or parameter name, and one column for each (sub)program block. The table is filled as follows:

- If a given name is accessible in a block, the corresponding entry shows either “Defined” or “Access”.
- If the name is inaccessible the entry is empty.
- An entry like “Defined” indicates the name is defined in the block either as a variable or as a parameter.
- If the entry is “Access” the name is accessible.
- When there is more than one definition, the definitions and accesses are numbered to show what name is accessible.

Variables only defined in the *MainPay* block are global and can be accessed from all the subprograms. Such global variables may seem useful. For example, the number of the employee *EmpNum* could be accessed directly from the *NetPay* subprogram (without passing it as a parameter) and output onto the

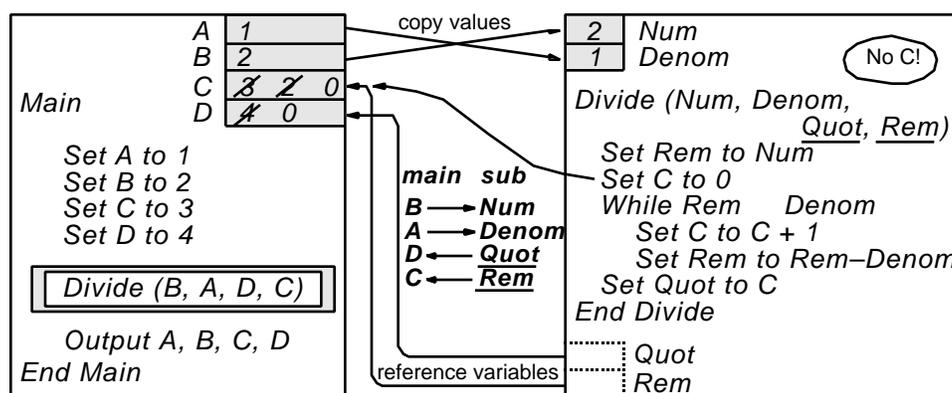
paycheck. However, this variable *EmpNum* could also be changed by accident in any of the subprograms. This kind of error could be very hard to find.

Tip: Reference only those variables that are defined in the subprogram in which you are working. Any time you think that using a global variable would be simpler, look instead to see how the program's design could be improved to avoid it. A global variable's potential for causing trouble is much greater than the simplification it can bring to the program. Access to all variables that are not local should be done only through parameters.

Parameter Crossing A Common Mistake

In order to show how destructive global variables can be, we will look here at a simple mistake that can turn out to be difficult to trace. Figure 7.39 is yet another variation of the now famous *Divide* invocation of Figure 7.24. For the sake of this example, we replaced the name of the counter “*Count*” by “*C*”.

Figure 7.39 Forgetting to declare a local variable



Suppose that in the *Divide* subprogram, we forgot to declare counter *C* as a local variable. When *C* is accessed and modified in *Divide*, since there is no local *C* defined in *Divide*, the global *C* is accessed and modified. When *Divide* increments *C*, it actually changes the value of *C* in *Main*. In our example, a Numerator of 2 is divided by a Divisor of 1. However, the result is not a quotient of 2 and a remainder of 0 (as it should be) because both *C* and *Rem* in the subprogram refer to the same value of *C* in *Main*!

Let's take a closer look at what happens. When the subprogram counter is initially set to 0, global variable *C* is set to 0. As the actions proceed, global *C* is changed every time *C* is changed in the subprogram, and also every time the remainder *Rem* is changed in the subprogram. In fact, the names *C* and *Rem* in the subprogram are both aliases of the global *C*.

Although you may already have a good idea of what is happening, let's trace the algorithm step by step (Pseudocode 7.5) to see the detail of the subprogram execution.

Pseudocode 7.5 Trace of Subprogram Divide

<i>Divide</i> (<i>Num</i> , <i>Denom</i> ,	{ <i>Num</i> = 2 and <i>Denom</i> = 1}
<i>Quot</i> , <i>Rem</i>)	
Set <i>Rem</i> to <i>Num</i>	{Since <i>Rem</i> references <i>Main</i> 's <i>C</i> , this sets <i>Main</i> 's <i>C</i> to 2.}
Set <i>C</i> to 0	{This now sets <i>Main</i> 's <i>C</i> to 0.}
While <i>Rem</i> <i>Denom</i>	{Since <i>Rem</i> references <i>Main</i> 's <i>C</i> , this test becomes 0 < 1, which is false, hence the body of loop is not executed.}
Set <i>Quot</i> to <i>C</i>	{Sets <i>Main</i> 's <i>C</i> to 0.}

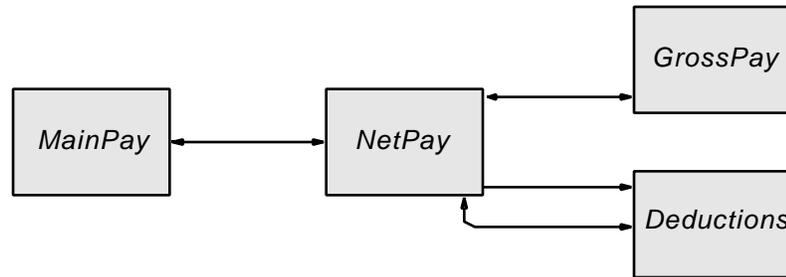
So this *Divide* subprogram produces both a quotient and a remainder of 0, which is incorrect. The values output by the program are 2, 1, 0, 0.

In general, such an accidental access of global values has very serious consequences because it produces wrong results and is very difficult to find and correct. Although they should know better, there are still some programmers who deliberately access global variables from within subprograms, inviting disastrous consequences. Accessing by parameters is the only safe way of communicating!

Minimizing Coupling, Maximizing Cohesion

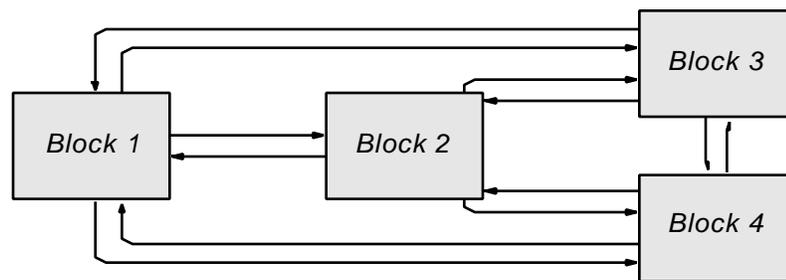
It may be useful to step back from the details of the *Main Pay* problem and return to a view similar to the data-flow diagram of Figure 7.36. Let's examine the data interactions between the various blocks. By interactions, we mean access to data, either by an argument-parameter transmission, or through global variables. *Coupling* is a measure of the degree of this interaction of the various blocks.

Figure 7.40 Loose coupling



The couplings between the four blocks of *Main Pay* are shown in Figure 7.40. There are essentially only four interactions, each shown by a line connecting the blocks. These respect the hierarchy established by the structure chart of Figure 7.35. Most interactions in the figure are through parameters passed by reference, which implies a two-way communication, and only one is through a value parameter (one-way). Since all the interconnections are through parameters, we say that these blocks are *loosely-coupled*.

Figure 7.41 Tight coupling



In contrast, Figure 7.41 shows a similar set of four blocks where every block interacts in both directions with every other block, forming 12 such interactions. These interactions are through global and local variables. This kind of system, referred to as *tightly-coupled*, would be quite complex and difficult to understand and maintain.

Maintaining a tightly-coupled system is difficult because the hierarchy of the structure chart has disappeared, and any modification is likely to affect more than one module, making it complex and error prone. In other words, if you forget just one of the changes required to keep all the blocks consistent, your program will most likely have an error.

Generalizing the interactions from this 4-block example to any number N of blocks is quite simple. In the present case, each of the 4 blocks connects to all of the remaining 3 blocks for a total of 4×3 or 12 interactions. In general, each of N blocks connects to the remaining $(N-1)$ blocks for a total of $N(N-1)$ interactions:

- For 5 blocks, there are $5 \times 4 = 20$ interactions.
- For 10 blocks, there are $10 \times 9 = 90$ interactions.
- For 20 blocks, there are $20 \times 19 = 380$ interactions.

As you can see, the maximum number of interactions grows quickly—almost as fast as the square of the number of blocks. If we assume that checking a modification implies checking all interactions, you can see why the number of interactions must be kept low! That is why loosely-coupled systems are preferred. They are usually easier to maintain than tightly-coupled ones.

Tip: Try to always make your system loosely-coupled: use parameters as much as possible when relaying information between your program parts.

A complementary measure to coupling is *cohesion*. Coupling is a measure of the interaction between blocks; cohesion (strength) is a measure of the interaction *between the elements that constitute the block*. Just as we wish to minimize the coupling, we wish to maximize the cohesion of each block. The basic intent of block cohesion is to organize the elements of a program so that closely related elements fall into a single block and unrelated elements fall into separate blocks.

To return to the *MainPay* example, subprogram *NetPay*'s sole function is to calculate the total pay *Amount*. *MainPay* has nothing to do with hours and pay rate. Those are functions of *GrossPay*, or *Deductions*. Since we have kept closely-related elements together, we can say that the blocks in *MainPay* all have cohesion.

Again, our aim is to create programs that are easy to modify. Programs that are built out of blocks with high cohesion are likely to be easy to modify since any modification is probably localized to a few blocks, which reduces the likelihood of making errors.

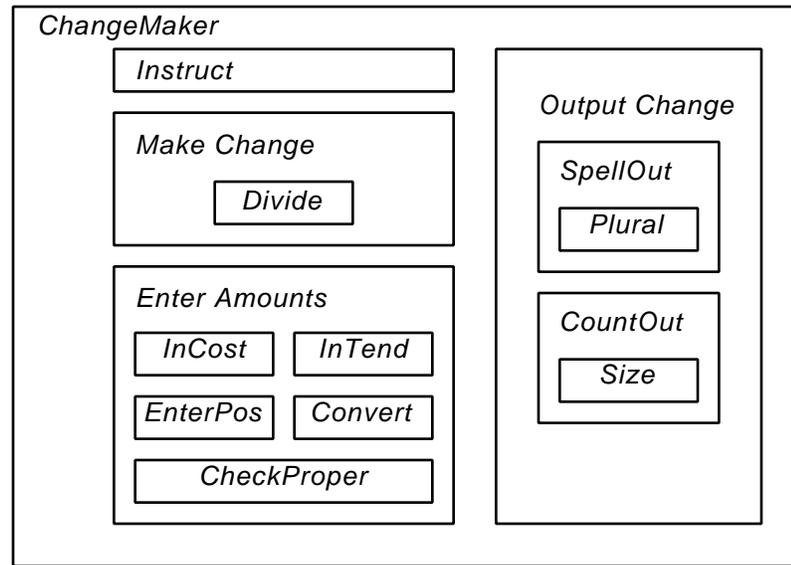
Tip: Try to maximize cohesion: keep closely-related elements together and only where needed.

What makes modification of a large program so difficult is the problem of maintaining consistency. Anything you can do during the design step to make this easier will help.

Deeply Nested Subprograms

Depending on the programming language used, subprograms can be nested many levels deep. To illustrate deep nests, we show in Figure 7.42 a complete *ChangeMaker* program that is drawn with contours and comprises 14 subprograms.

Figure 7.42 Contour diagram for *ChangeMaker*



Some parts of *ChangeMaker* have already been developed in pseudocode in Chapter 4, Section 4.4.

Notice that this contour diagram is lacking some essential things:

- The local variable boxes have not been drawn in.
- The parameters (passed in or out) have not been drawn in either.
- The main program, *ChangeMaker*, calls four subprograms: *Instruct*, *Make Change*, *Enter Amounts* and *Output Change*.
- The first subprogram, *Instruct*, asks if the user wishes instructions. If so, the subprogram produces a printed set of the user instructions.
- The second subprogram is *Make Change* that actually performs the change making computation. To do this, it calls the subprogram *Divide* a number of times.
- The third subprogram, *Enter Amounts*, contains five other subprograms to help input the various values and validate them. Subprograms *InTend* and *InCost* input and validate the amount tendered and the cost. Subprogram *EnterPos* is used to enter only positive values, and subprogram *Convert* is used to change the numerical values into monetary values. Subprogram *CheckProper* checks that the cost and the amount tendered are positive and that the cost is less than the amount tendered.
- The last subprogram, *Output Change*, contains other output subprograms and calls them to produce a nicely formatted output. Subprogram *SpellOut* spells the count corresponding to the number of coins, and calls *Plural* to append the character “s” to any written plural denomination. Subprogram *CountOut* provides a readable output of the number of coins, and calls *Size* to determine the number of digits in a number as part of formatting the output values.

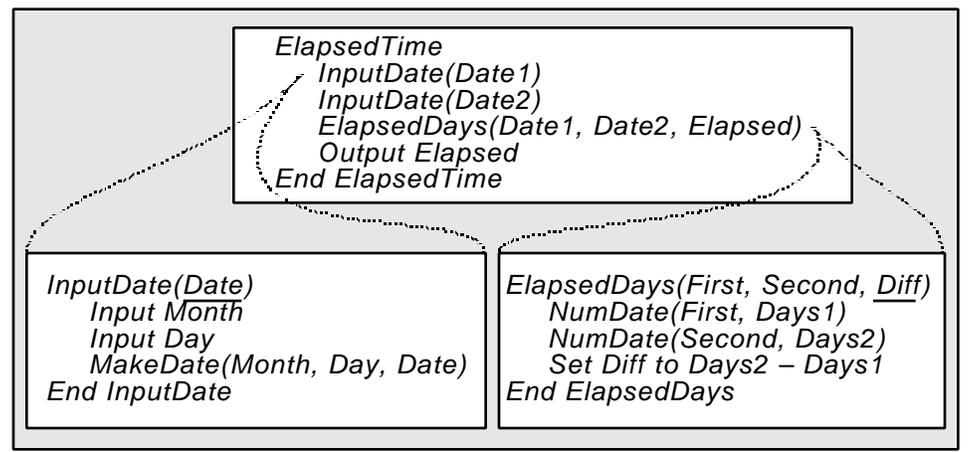
A more detailed *SpellOut* is shown in Figure 7.30.

Dates Example

So far, we have always tried to develop our various algorithms using a top-down approach, and this has usually led to better structured solutions. However, very often in programming, there is a temptation to create all algorithms from their smallest building blocks (starting everything from scratch). We have seen that it is better to create algorithms out of larger “abstract” boxes whose exact functions are yet to be defined, and later, when we are more advanced in the design, we can define these boxes in detail.

In computer applications, dates are a type of data that must often be manipulated. They usually are made of several parts, like day, month, and year. In the next example we will concentrate on date processing.

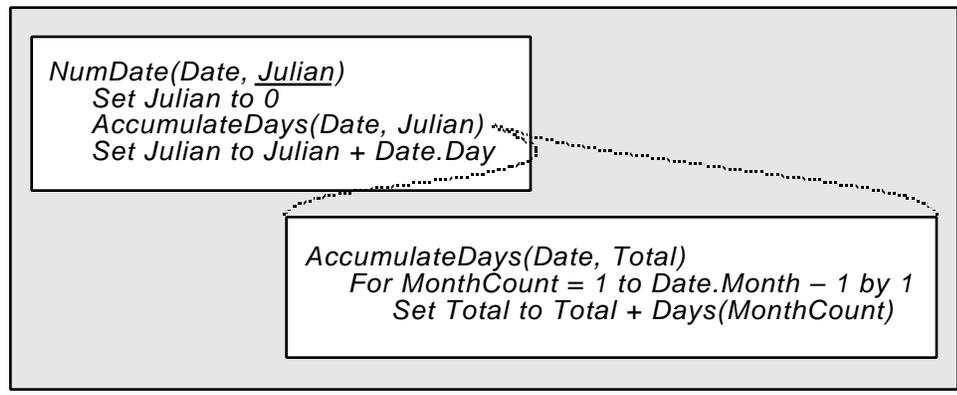
Figure 7.43 Developing the algorithm *ElapsedTime*



Our date example will be the creation of the algorithm *ElapsedTime*, which determines the number of days between any two given dates in the same year, such as two birth dates. Figure 7.43 shows the top two levels of the solution chosen for *ElapsedTime*. Proceeding top-down, the top-level program first inputs the two dates, computes the number of elapsed days between them, and then outputs that value.

Notice that, at the top level, we refer to the two dates as *Date1* and *Date2*, ignoring the fact that they are made up of two separate values: the month and the day. This abstraction makes it easier to avoid getting bogged down in details. At the next level of *InputDate*, these constituents are recognized since the *Month* and the *Day* are input. However, their combination into a single data value is left unspecified in a call to *MakeDate*, which is, as of yet, undefined.

The elapsed time between the two dates could be determined easily by *ElapsedDays* if we knew the date in its Julian form, as the number of days from the first of the year. The number of elapsed days can then be found by simple subtraction of the two Julian dates. In the solution shown in Figure 7.43, we have assumed that a subprogram, *NumDate*, to compute the number of days from the start of the year already exists.

Figure 7.44 The subprogram NumDate to find a Julian date

The subprogram, *NumDate*, converts a given *Date* into the number of days from January 1, as shown in Pseudocode 7.6. For example, March 15, 1984, has a Julian date of 75, (31 + 29 + 15). At this point in our development of the *ElapsedTime* program, we realize that in order to find the Julian date, we probably need to know how many days there are in February. For that, we need to know the year. Therefore, we must go back to our sketch of *InputDate* and add a statement to input the year. Notice that *ElapsedTime* does not have to be changed—this is an example of the advantages of designing subprograms with high cohesion.

Since we have only outlined *Input Date*, little has been changed. Each date now has three components, *Year*, *Month* and *Day*, instead of only two. This need to revise earlier subprogram outlines is typical of the program creation process and is a major reason for proceeding top down. If we had completely created *InputDate*, the change would have been much more serious and frustrating.

7.5 More Types of Subprograms

Recursion Self-Referencing Subprograms

The Solver algorithm was first introduced in Chapter 6, Section 6.6.

The problem solving method we introduced in Chapter 2 and used on various examples is based in part on a “divide and conquer” approach. A problem is divided and the smaller, easier-to-solve problems are divided in turn. This way of doing things can be used to great advantage when developing some algorithms.

For example, if an algorithm, *Solver*, is developed to solve a specific problem, in some cases it is possible to apply that same algorithm to a smaller version of the same problem; this is called *recursion*. By repeatedly dividing the problem in this manner we eventually reach the smallest case, called the *base case*, which is so simple that it can easily be solved.

In essence, the pattern of a recursive solution for *Solver* is illustrated in Pseudocode 7.6.

Pseudocode 7.6 Pseudocode for recursive problem-solving

```

Solver(problem)
  If the problem is a base case
    Solve the base case directly
  Else
    Split the problem into subproblems
    For each subproblem
      Solver(subproblem)
  End Solver

```

Thus, recursion is a process where a subprogram will call itself. Such a self-referring process may seem unusual at first, but many data structures and algorithms are more naturally described recursively. In this section, we only introduce the idea of recursion. It will be dealt with in greater depth in Chapter 8.

Recursion in its simplest form could be viewed as an alternative to the iterative form. To illustrate this, let's take as an example the computation of the square of an integer. We saw in Chapter 6 that the square of integer *Num* can be computed by summing the first *Num* odd integers. A subprogram that calculates the *Square* of a number this way is shown in Pseudocode 7.7 where we find an iterative version and a recursive version.

Pseudocode 7.7 Iterative and Recursive Square subprograms**Iterative Version**

```

Iterative Square(Num, Square)
  Set Square to 0
  For Count going from 1 to Num by 1
    Set Square to Square + 2 × Count - 1
  End Iterative Square

```

Recursive Version

```

Recursive Square(Num, Square)
  If Num = 1
    Set Square to 1
  Else
    Recursive Square(Num - 1, Square)
    Set Square to Square + 2 × Num - 1
  End Recursive Square

```

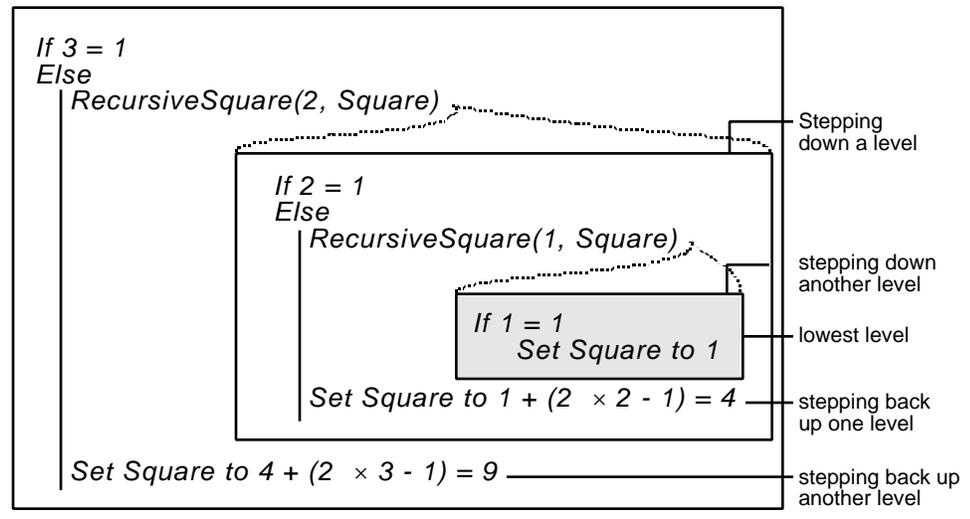
Recursive Square calls itself.

Point of return

See Pseudocode 6.10 for more details on computing the square of an integer.

To understand how subprogram *Recursive Square* works, we will trace its execution with a value of 3 for *Num*. The trace takes the form of a subprogram invoking itself twice, as shown in Figure 7.45.

Figure 7.45 Trace of Recursive Square for Num = 3



Each call of the subprogram yields a contour, resembling levels on a map (stair-trace). On entry to a subprogram, we step down a level, and then on exit we step up again. It is important to have a “lowest” level, a stopping point (or a base case, such as *Num* = 1), otherwise, the algorithm would be “bottomless” or unending. This lowest level is shown shaded on Figure 7.45.

See Section 7.3 for more details on returning control to the caller.

At this time, you do not need to know by which mechanism recursion is performed. It is only necessary to know that after a recursive call, like after any subprogram call (and subsequent detour), the control must “return to the caller”.

Remember in Figure 7.15 that the first action done when *Divide* was invoked was: “The point of return is noted.” In our case, this means that after each recursive call, the assignment of a new value to *Square* is performed, as shown in Figure 7.45. In Chapter 8, we will take a look at the actual mechanism for recursive calls.

Functions

See Figure 5.42 for more details on Convert Seconds.

There is another kind of subprogram that is useful in some circumstances: the *function*. When we first introduced subprograms in Chapter 5, we gave the example of a program, *Convert Seconds* for converting a number of seconds into days, hours, minutes and seconds.

Instead of using the *Divide* subprogram, which calculates both the quotient and remainder, we used the following two subprograms:

- *Div*, which found the quotient, and
- *Mod*, which found the remainder.

The *Convert Seconds* algorithm is shown in Pseudocode 7.9.

Pseudocode 7.8 The Convert Seconds program

```

Convert Seconds
  Input Time
  Set Days to Div(Time, 60 × 60 × 24)
  Output Days
  Set Seconds to Mod(Time, 60 × 60 × 24)
  Set Hours to Div(Seconds, 60 × 60)
  Output Hours
  Set Seconds to Mod(Seconds, 60 × 60)
  Set Minutes to Div(Seconds, 60)
  Output Minutes
  Set Seconds to Mod(Seconds, 60)
  Output Seconds
End Convert Seconds

```

Div and *Mod* are very much like the subprograms we have been using in this chapter, with one important difference. In all our examples in this chapter, we have had subprogram invocations of the form:

Divide(Time, 60×60×24, Days, Seconds)

where the third argument, *Days*, is used to return the quotient. In *Convert Seconds* we use *Div* very much like a mathematics function, like, for example the trigonometric function Sine. For this reason, the two subprograms *Div* and *Mod* are known as *functions*.

The major difference between subprograms and functions is that functions, rather than having an argument to return the result of their computation, actually produce the value in a form which can be used directly. This is very convenient in situations where the subprogram has only a single value to return. This would obviously not work with the *Divide* subprogram since it returns two results, the quotient and the remainder. The pseudocode for the *Mod* function is shown in Pseudocode 7.10.

Pseudocode 7.9 The Mod function

```

Mod(Num, Den) function
  Set Rem to Num
  While Rem > Den
    Set Rem to Rem - Den
  Return Rem
End Mod

```

There are two things to notice here:

- The header finishes with the word “*function*”, and
- The last statement is *Return Rem*, which sends the result back to the caller.

There is an important distinction between a math function and a subprogram function. A math function only does one thing: it produces a single value for each invocation. On the other hand, a subprogram function may have other effects as well as producing a value. For example, a subprogram function could alter the value of a global variable or cause values to be written on an output device. Such effects, called *side effects*, make a program that uses such functions much more difficult to understand and to change. Therefore, side effects should always be avoided in programming.

Modules

Modules are “black boxes”. Think of them as walls that surround a part of a program. These walls enclose (or hide) data, and clearly separate the inside of the module from the outside. Communication between the inside and the outside is totally under the control of the programmer.

The major use of modules is to provide a method for breaking up a large program into semi-independent pieces related by well-defined and simple interfaces. The resulting modular structure is easier to document, analyze, modify and maintain. It is also less prone to errors.

An important use of modules is for the creation of libraries of related subprograms. In most systems, there are typical libraries of mathematical functions, input/output operations, and others. Libraries can be viewed as building blocks to be used without knowing the details within them. These library modules can also make use of other modules.

Generally, the information that is provided to the programmer is a brief description of what the module does, not how it does it. Programmers do not need to know how the library module performs its function in order to be able to use it. We have come back full circle to the idea introduced in Chapters 2 and 3, that of structuring a solution with black boxes.

7.6 Review Top Ten Things to Remember

1. Many of the concepts on *program structure* and block structure have been introduced and used earlier. At this stage, we are able to understand them more fully, and to use them more creatively. In particular, the concepts of sharing and hiding information are now extremely significant.
2. The *hiding and sharing* of data can be achieved in a number of ways, that have been described and compared. Local variables are useful for hiding data, but global variables are dangerous for sharing. Sharing is better done by using parameters that are used to communicate data between the subprogram and either the main program or another subprogram.
3. During a subprogram invocation, we learned that a link is established between its call's arguments and the subprogram's parameters. This link is used to transmit data. In previous chapters, when we considered sub-algorithms and subprograms, we concentrated on the flow of data and flow of control. Now we need to be more complete by concentrating on the space occupied by the data and on *how* the data values are communicated from one space to another. Data space diagrams are useful for illustrating these concerns.
4. *Parameter passing* between program and subprograms, or between subprograms, is done in two manners: by value, one-way communication from calling program to invoked subprogram, and by reference, two-way communication.
5. Passing a parameter by value copies the argument's value into a subprogram parameter which appears as a local variable. This value is used in the subprogram but is not changed by the subprogram.
6. Passing a parameter by reference gives the subprogram access to the variable argument within the calling program. The value of this variable is changed by the subprogram and any previous value of the argument is destroyed. This previous value of the argument may or may not be used by the subprogram, depending on the subprogram's interface specification.
7. Three rules must be followed when using variables in subprograms:
 - **Rule 1:** A (sub)program cannot access the local variables of any subprograms nested within it.
 - **Rule 2:** A (sub)program can access a variable that is local to itself.
 - **Rule 3:** If a variable does not appear in a subprogram, a search is made in the enclosing subprograms. This search is continued until either the variable is found, or the main program is reached. If the variable is not found in any of these nests, then the variable is undefined and there is an error in the program.
8. A program's blocks should always be *loosely-coupled*. This means that their interactions are through parameters. *Tightly-coupled* blocks, which interact through global and local variables, are more complex and difficult to understand and maintain.
9. As an alternative control mechanism, *recursion* is often useful as it simplifies the programming effort. Recursive subprograms were introduced here, and will be used in the coming chapters.

10. The concept of *module* introduced yet another method of structuring a computer solution, that makes it possible to organize better the various parts of a solution, and to help control coupling and cohesion. Modules can be used to create program libraries which make it possible to re-use programming components.

7.7 Glossary

Alias: An identifier that refers to a variable that is also known by another name. For example, the name of a parameter that is passed by reference; since it refers to a variable in the calling program, it is an alias for that variable.

Argument: A variable, expression or constant that is passed to a subprogram in the invocation statement.

Base case: The special case in a recursive subprogram that causes the recursion to terminate.

Binding: The association of a name with a data value.

Block structure: The hierarchical arrangement of the subprograms that comprise a program.

Cohesion: The quality of a program that refers to the degree that all statements in the subprogram are concerned with the same objective.

Coupling: The quality of a program that refers to the degree of interconnection of the constituent subprograms.

Function: A subprogram that returns a single data value through the execution of a return statement.

Global data: Data that are known throughout the program.

Julian date: The ordinal of the given data in the year.

Loosely coupled: A program in which all the subprograms have the minimum number of interconnections.

Parameter: A name that is local to a subprogram and is used to refer to the arguments in the subprogram's invocation.

Pass by reference: A method of parameter passing whereby the parameter is the alias of the corresponding argument.

Pass by value: A method of parameter passing whereby the parameter is a local variable that is assigned the value of the corresponding argument.

Recursion: A situation in which a single subprogram invokes itself.

Scope: The part of a program where a particular identifier is known.

Side effect: A change in the value of a variable that does not appear in the argument list of a subprogram invocation.

Strength: Synonym for cohesion.

Subprogram: A program that can be called by an invocation statement.

7.8 Problems

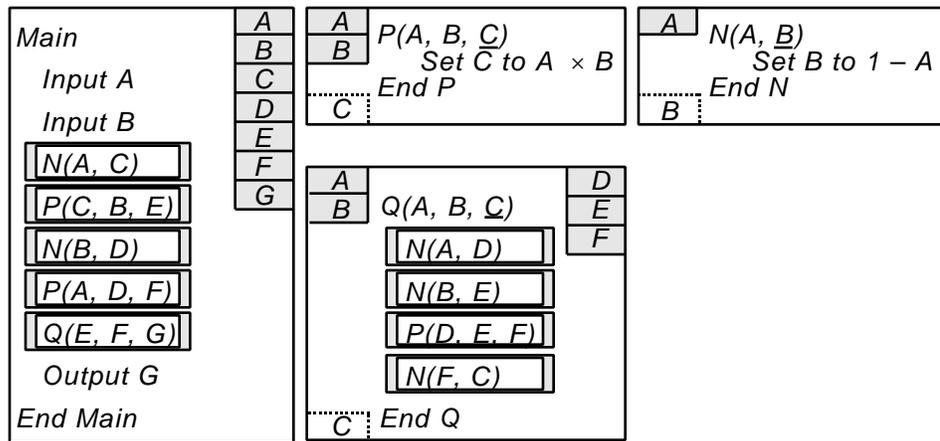
1. Divide Again

Create an algorithm to Divide using a subprogram which multiplies. It may not be efficient, but it shows another way of doing something.

2. Trace Subs

Draw a data-flow diagram corresponding to the given interconnection of the subprograms below. Trace this program for $A = 0$ and $B = 1$. Draw a tree showing the calling of subprograms and indicate the order that the subprograms are called in.

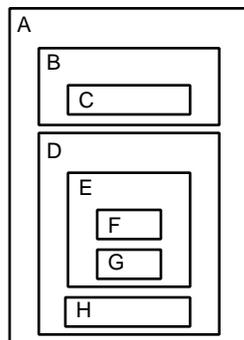
Problem 2



3. Deep Nests

Given the nested blocks shown in the following diagram, indicate which blocks may be called by each block. If a variable X is declared in E only, which blocks can access it? If a variable Y is declared in B, D and F, but is referenced in all of the modules, which variables (if any) are accessed or “seen” in each module?

Problem 3



Problems on Subprograms

4. Divide-And-Conquer

Use the previously defined Divide subprogram to:

- convert pints to gallons, quarts and pints.
- convert a decimal number to binary.
- determine whether a year is a leap year.
- convert 24-hour (military) time to civil time, indicating AM or PM.

5. Data of Easter

The algorithm to determine the data of Easter for any given year could involve a number of Div and Mod actions. Create a subprogram to determine the data of Easter according to the following algorithm for an input parameter Y , whose value is the year:

- The “golden number”, G is $(Y \text{ Mod } 19) + 1$.
- The century number C is $(Y \text{ Div } 100) + 1$.
- The number of years X in which leap year was dropped, e.g., 1900, so as to keep in step with the sun is $(3C \text{ Div } 4) - 12$
- A correction Z to synchronize Easter with the moon’s orbit is $(8C + 5) \text{ Div } 25$.
- If $D = (5Y \text{ Div } 4) - X - 10$ then March $((-D) \text{ Mod } 7)$ is a Sunday—if $(-D) \text{ Mod } 7 = 0$ then March 7 is a Sunday.
- The “Epect” E specifies when a full moon occurs. $E = (11G + 20 + Z - X) \text{ Mod } 30$. If $E = 25$ and G is greater than 11, or if $E = 24$ then E is increased by 1.
- Easter is on the “first Sunday following the first full moon that occurs on or after March 21”. The “calendar moon” used for finding Easter is defined as the N th of March where $N = 44 - E$. If $N < 21$ then set N to $N + 30$.
- To advance N to a Sunday, set $N = N + 7 - ((D + N) \text{ Mod } 7)$.
- If $N > 31$ then the data of Easter is the $(N - 31)$ April; otherwise, the date is N March.

6. Double All

What is output by the following program when it passes parameters by reference?

Problem 6

```
Main Program
  Set X to 2
  Double(X, X, X)
  Output X
```

```
Double(A, B, C)
  Set A to A + A
  Set B to B + B
  Set C to C + C
```

7. Dates

Use some of the previous algorithms (Leap, Days) and create:

- a. *Valid*, an algorithm to test whether a given date *Year, Month, Day* is a valid date.
- b. *UnDate*, an algorithm to convert a Julian date *Julian, Year* back into the Gregorian form *Day, Month, Year*.
- c. *DaysLived*, an algorithm to determine the number of days a person has lived, from the birth date to the present date.
- d. *Age*, an algorithm to determine the (integer) age of a person.
- e. *WeekDate*, an algorithm to determine the weekday a given date falls on, when given that the first day of the year falls on the *W*th day (where $W = 0$ for Sunday, $W = 1$ for Monday, ... and $W = 6$ for Saturday).
- f. *FirstDate*, an algorithm to determine the weekday of New Year's Day, given the year *Y*. Use the fact that January 1, 1901 was a Tuesday ($W = 2$). Notice that a year of 365 days has exactly 52 weeks plus one day (i.e. $52 \times 7 = 364$).

8. Bind Dates

Create a binding table and contour diagram for any of the above Dates programs.

9. Range

Create an algorithm that reads in three values *A, B, C* and outputs the range *R*, which is the difference between the largest and smallest values. This must be done using the following subprograms. Provide the maximum hiding of variables and subs possible.

The main program *Range* is to call a subprogram *BigSmall3(I, J, K, L, S)*, which finds the largest *L* and smallest *S* of the parameters *I, J, K*. This subprogram in turn calls two functions *Big(P, Q, R)* and *Small(P, Q, R)*, each of which must call a subprogram *Sort2(G, H)*, which takes *G* and *H*, and arranges them so that *G* is largest and *H* is smallest.

Problems on Passing Parameters

10. Divide Again

For the previously defined Divide subprogram and the following Main Program,

Problem 10

```

Main Program
  Set A to 1
  Set B to 2
  Set C to 3
  Set D to 4
  Divide( )
  Output A
  Output B
  Output C
  Output D
    
```

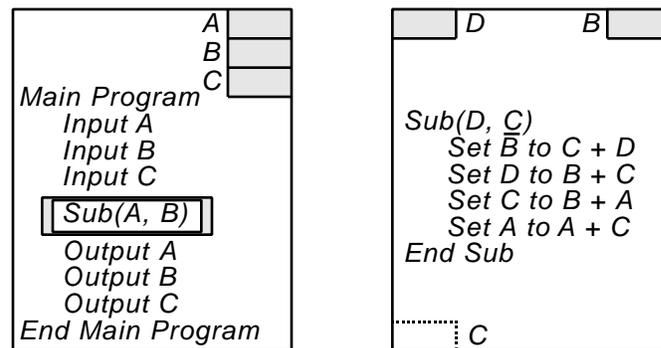
indicate the output of the following subprogram calls:

- a. *Divide(D, C, B, A)*
- b. *Divide(A, B, A, B)*
- c. *Divide(A, A, B, B)*
- d. *Divide(B × C, A - D, A, D)*
- e. *Divide(A, B - 2, C, D)*
- f. *Divide(O, 1, 2, 3)*
- g. *Divide(O, 1, B, B)*

11. Non-Divide Passing

Indicate the output of the following program if the input values are 1, 2, 3 in that order.

Problem 11



Indicate the output if the subprogram call (in the main program) were changed in each of the following ways:

(Note: The input values are still 1,2,3 in that order. Hint: Use the contour diagram!)

- a. *SUB(A, C)* (Answer: 6, 2, 5)
- b. *SUB(B, A)* (Answer: 8, 2, 3)
- c. *SUB(C, B)* (Answer: 7, 6, 3)
- d. *SUB(B, C)*
- e. *SUB(C, A)*
- f. *SUB(B, B)*

Problems Involving Recursion

12. Recursive Power

Create a recursive subprogram to compute the Nth (positive) power of X.

13. Rem (or Mod)

Create a recursive program for $Rem(A, B)$, which computes the remainder after A divides B (use successive subtraction).

14. Roll Your Own

Create a recursive program to compute something you are familiar with (another square, number conversion, craps rules).

Chapter 8 Data Structures

In this chapter we introduce the basic data structures that are found in most programming languages. This chapter also introduces dynamic variables as well as the concept of abstract data types.

Chapter Outline

8.1	Preview.....	358
8.2	Arrays.....	358
	What are Data Structures Anyway?.....	358
	One-Dimensional Arrays.....	360
	Performing Operations on One-Dimensional Arrays....	361
	Using One-Dimensional Arrays in Algorithms.....	364
	Two-Dimensional Arrays.....	367
	Performing Operations on Two-Dimensional Arrays...370	
	Matrix Multiplication.....	373
	N-Dimensional Arrays.....	375
8.3	Records.....	377
	What are Records?.....	377
	Accessing Record Components the Dot Notation.....	380
	Combining Arrays and Records.....	382
8.4	Sets.....	384
	What are Sets?.....	384
	A Difficult Sets Example (optional).....	386
8.5	Data Structure Building Tools.....	392
	Dynamic Variables.....	392
	Pointers.....	392
8.6	Abstract Data Types.....	397
	Strings.....	399
	Stacks.....	401
	Queues.....	405
	Trees.....	407
8.7	Review Top Ten Things to Remember.....	408
8.8	Glossary.....	409
8.9	Problems.....	410

8.1 Preview

In this chapter, we consider the following three basic data structures. These data structures are created by grouping together smaller items to form larger items. These three data structures in turn are used to create larger ones in the next chapter.

- *Arrays* (also called vectors, tables, matrices, n-dimensional lists, and subscripted variables) are *homogeneous* groupings of items— all items are of the same kind. The items within the arrays are accessed by way of indices. *Tables*, or arrays having two indices, are common, and will be considered here. Arrays of three and more indices are less common, so they will be considered only briefly.
- *Records* are very important data structures. They differ from arrays in that they are groupings of possibly *heterogeneous* elements (items of different kinds), whereas arrays are groupings of *homogeneous* elements. Also, values are not accessed by using indices, but by using names for the elements and a dot notation.
- *Sets* are homogeneous collections of distinct elements where the only relation between elements is that they are either in a set or not in it. Sets are very useful in a number of applications.

Although arrays, records and sets are very useful in many applications, they all have one major restriction: their size is static or set at some fixed maximum value. Because of this, they cannot represent data whose detailed structure or size are unknown when the program is written. Data structures to represent such “unknown” data can be built during program execution through *dynamic variables* and *pointers*. An example with linear lists of undefined length is used to show how these dynamic variables and pointers can be used.

The chapter also introduces the concept of an *abstract data type*, which is defined through its values and operations. Other data structures important in computer science, like linear lists and trees, are also briefly introduced.

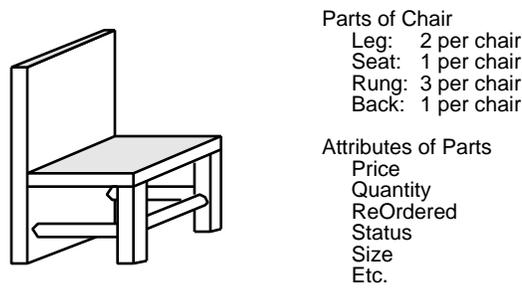
8.2 Arrays

What are Data Structures Anyway?

A *data structure* is a collection of related items organized in a certain fashion. It enables us to consider all of the related items as one entity. For example, a data structure could consist of apples, bananas, and oranges. Notice that these three items are related and organized alphabetically. They could have been organized some other way. The important thing to remember is that, because we are talking about a data structure, we can refer to the whole of these items, in our case, fruit. Arrays

Data structures range from simple (like the fruit above) to very complex. They can be viewed in many ways, as shown below.

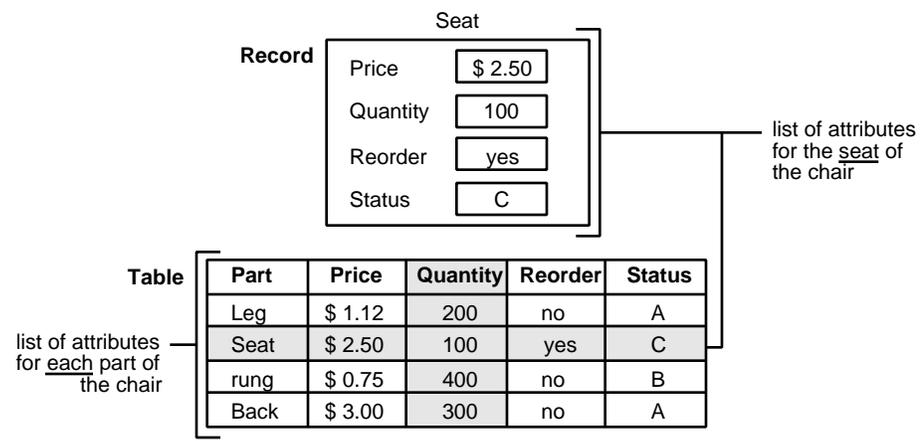
Figure 8.1 Parts of a chair



Let's use a chair as an example. A chair may be viewed physically as a collection of parts. The simple chair drawn in Figure 8.1 has some legs, some rungs, a seat and a back. If we were manufacturing this chair, we would be also interested in knowing some more of their attributes, like how much does each piece cost, how many pieces are available, and how big are the pieces. If you look at the list of attributes in Figure 8.1, you will notice that each attribute may be expressed differently from the next. Price can be expressed in dollars, quantity in numbers, reordered with a yes or a no, etc.

Let's look at the attributes of one part: the seat of the chair, in Figure 8.2.

Figure 8.2 Chairs inventory



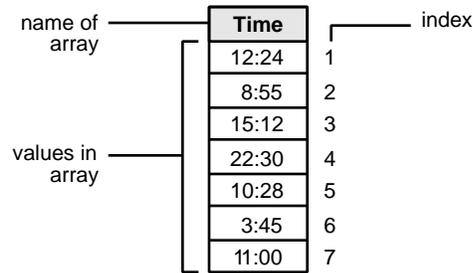
We can make a list of attributes for each part of the chair, as in the lower part of Figure 8.2. Here, the parts are listed vertically in the left column. The various attributes (Price, Quantity, etc.) are shown as columns. Each part corresponds to a row. This representation is called a *table*. Each row of this table corresponding to a part is called a *record*. Each column corresponding to an attribute is called a *field*. In our example, Seat is a record and Quantity is a field.

Now that you have seen what data structures are, you are ready to discover the different kinds of data structures used in computer science. In this chapter, three basic types of data structures are introduced: arrays, records, and sets.

One-Dimensional Arrays

The first type of data structure introduced is an array. An *array* is a *homogeneous* collection of components—all the components are of the same kind. The simplest arrays are linear: they have only one dimension and are called *vectors*, *n-tuples*, *single subscripted variables*, or more simply, *one-dimensional arrays*. An example of such a one-dimensional array is shown in Figure 8.3.

Figure 8.3 A one-dimensional Time array



This Time array could represent various times (using the 24-hour representation) at which a given event will happen over a week—a period of seven days. An array is also sometimes called an *indexed variable*, for the following two reasons:

- (i) Like a variable, it can contain values, and
- (ii) To access a value, it is necessary to use an index.

You may have seen indices used in mathematics in the form of subscripts like this:

Time₁, Time₂, Time₃, Time₄, Time₅, Time₆, Time₇

In most programming languages the index is written within square brackets as:

Time[1], Time[2], Time[3], Time[4], Time[5], Time[6],
Time[7]

In Figure 8.3, *Time [1]* refers to 12:24. Generally, the names chosen for indices can be meaningful in the context of the application being programmed. For example, we could replace *Time [1]* by *Time[Monday]*.

Figure 8.4 illustrates another one-dimensional array, giving the body temperature of a patient, recorded every hour of the day. *Temperature[3]* represents the temperature recorded at Hour 3.

Figure 8.4 Temperature vector

Hour	Temperature
1	98.6
2	100.0
3	101.1
⋮	⋮
23	99.0
24	99.6

Figure 8.5 shows yet another example of a one-dimensional array; this one stores the grades of all the students in a class. *Grades[StudentID]* represents the grade earned by the student whose ID is given by the value of StudentID.

Figure 8.5 One-dimensional Grades array

Student ID	Grades
1	A
2	C
3	B
⋮	⋮
34	F
35	D

Pseudocode 8.3 Inputting times into the Time array

```

Input Array 1
  Input Time[1]
  Input Time[2]
  Input Time[3]
  Input Time[4]
  Input Time[5]
  Input Time[6]
  Input Time[7]
End Input Array 1
    
```

A more efficient way to do this is to use a loop:

Pseudocode 8.4 Using a loop to input times into the Time array

```

Input Array 2
  For Index = 1 to 7 by 1
    Input Time[Index]
  End Input Array 2
    
```

Such input methods are not general, since we have to know how many entries we need to make ahead of time.

A more general method is shown below. You do not need to know the number of entries ahead of time anymore. We accomplish this by using a terminating value (or end-of-data marker), *Sentinel*, to detect the end of the entries. *Input Array* assigns the input values to consecutive entries of the array, counting the number of entries made. After the loop terminates, the final value of *Count* is assigned to the variable *Size*, so that now we know how many entries were made.

For more details on using a terminating value, see Chapter 6, Figures 6.4 and 6.5.

Pseudocode 8.5 Subprogram that inputs values into Vector array

```

Input Array (Vector, Size)
  Input Sentinel
  Set Count to zero
  Input Value
  While Value < Sentinel
    Increment Count
    Set Vector[Count] to Value
    Input Value
  Set Size to Count
End Input Array
    
```

Obviously this bit of pseudocode represents a subprogram which passes out both the whole array Vector and the number of entries, Size.

See Chapter 7 to brush up on subprograms.

Clearly this method is more suited to large arrays or arrays whose size varies. Once the data have been put into the array, the *Size* is known. We can write the pseudocode to output this array with one a simple loop:

Pseudocode 8.6 Subprogram that outputs Vector array

```

Output Array(Vector, Size)
  For Index = 1 to Size by 1
    Output Vector[Index]
  End Output Array
    
```

In the rare case where this *Size* is not known, we would have to make use of a more complex loop with an end-of-data marker.

Using the *Temperature* array of Figure 8.4, let's develop an algorithm to find the maximum temperature in the day by inspecting the values stored in the vector. This algorithm must also indicate the position (*Hour*) of the first

maximum value encountered (since there could be several). We want to develop the algorithm in a top-down manner so we first define a rough outline.

Pseudocode 8.7 Algorithm to find maximum temperature

Note that *TableSize* should be equal to 24, the size of the array shown in Figure 8.4.

```

Find Maximum Temperature
Input Array(Temperature, TableSize) ——— Here we use the Input Array
Set Maximum and Hour to initial values subprogram already written.
Find Maximum Temperature
Output Maximum and Hour
End Find Maximum Temperature

```

This outline can now be refined to the final solution. The first value of the array is taken as the temporary *Maximum*. One by one, the other values are compared to the *Maximum* and the first largest value encountered is kept.

Pseudocode 8.8 Final refined version of Pseudocode 8.7

```

Find Maximum Temperature
Input Array(Temperature, TableSize)
Set Maximum to Temperature[1]
Set Hour to 1
For Index = 2 to TableSize by 1
    Set Value to Temperature[Index]
    If Maximum < Value
        Set Maximum to Value
        Set Hour to Index
Output Maximum, Hour
End Find Maximum Temperature

```

Using One-Dimensional Arrays in Algorithms

The change making algorithm seen earlier in Chapters 4 and 7 can also be implemented very conveniently with an array. The previous *Make Change* pseudocode from Chapter 4 involved many repetitions, with four similar loops. It is repeated below and has been renamed *Change Maker 1*.

Pseudocode 8.9 Change Maker algorithm from Chapter 4

```

Change Maker 1
Input Tendered
Input Cost
Set Change to Tendered - Cost
While Change >= 25
    Output a quarter
    Decrement Change by 25
While Change >= 10
    Output a dime
    Decrement Change by 10
While Change >= 5
    Output a nickel
    Decrement Change by 5
While Change >= 1
    Output a penny
    Decrement Change by 1
End Change Maker 1

```

In Chapter 7, Figure 7.23, repetition was avoided by using the subprogram *Divide*. Here, we will avoid the sequence of Repetitions in another way—with an array (Pseudocode 8.10).

Pseudocode 8.10 Change Maker algorithm using Coins array

```

Change Maker 2
  Input Tendered
  Input Cost
  Set Change to Tendered - Cost
  Set Coins[1] to 25
  Set Coins[2] to 10
  Set Coins[3] to 5
  Set Coins[4] to 1
  For Index = 1 to 4 by 1
    Set Value to Coins[Index]
    While Change >= Value
      Output Value
      Decrement Change by Value
  End Change Maker 2
    
```

] replaces sequence of While loops

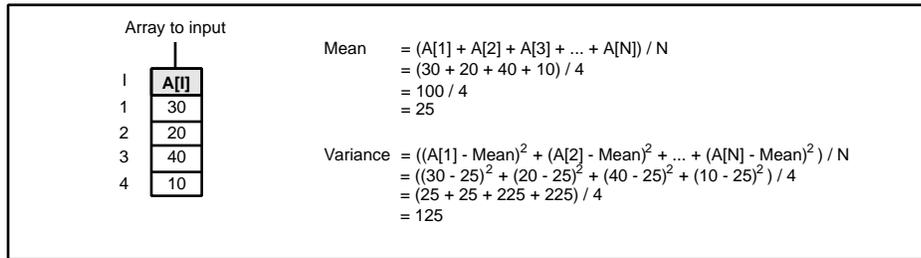
The new *Change Maker 2* algorithm, outlined above, uses an array, *Coins*, which contains the denominations 25, 10, 5 and 1 in order. As the for loop's counter, *Index*, goes from 1 to 4, the variable *Value* is assigned the corresponding array values (first 25, then 10, 5, and 1), and the change corresponding to this *Value* is computed and output.

Actually the output of *Change Maker 2* is not exactly the same as the output of *Change Maker 1*. *Change Maker 1* outputs quarters, dimes, nickels and pennies, while *Change Maker 2* will output numerical values. It is not difficult to modify this second version so that its output is identical to the output of the first version. We will let you do it.

Change Maker 2 is not only shorter than the previous versions of *Change Maker*, but most important, it can be modified more easily. Extending it to apply to more denominations (such as 5 dollar bills, 10 dollar bills, 20 dollar bills, and even 2 dollar bills) requires only a slight change in the array values (as well as the size of the *Coins* array). This algorithm could also be easily modified to make change in any foreign currency.

Another very common use of arrays is to store numerical data so that the values are available for repeated access. For example, Figure 8.7 shows the computation of the variance of a set of numbers.

Figure 8.7 Variance a first approach



Input Array (A, N)	N = 4										
Set Sum to 0	Sum = 0										
For I = 1 to N by 1	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">1 4</td> <td style="text-align: center;">2 4</td> <td style="text-align: center;">3 4</td> <td style="text-align: center;">4 4</td> <td style="text-align: center;">5 4</td> </tr> <tr> <td style="text-align: center;">T</td> <td style="text-align: center;">T</td> <td style="text-align: center;">T</td> <td style="text-align: center;">T</td> <td style="text-align: center;">F</td> </tr> </table>	1 4	2 4	3 4	4 4	5 4	T	T	T	T	F
1 4	2 4	3 4	4 4	5 4							
T	T	T	T	F							
Set Sum to Sum + A[i]	Sum = 30 50 90 100										
Set Mean to Sum / N	Mean = 25										
Set Total to 0	Total = 0										
For J = 1 to N by 1	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">1 4</td> <td style="text-align: center;">2 4</td> <td style="text-align: center;">3 4</td> <td style="text-align: center;">4 4</td> <td style="text-align: center;">5 4</td> </tr> <tr> <td style="text-align: center;">T</td> <td style="text-align: center;">T</td> <td style="text-align: center;">T</td> <td style="text-align: center;">T</td> <td style="text-align: center;">F</td> </tr> </table>	1 4	2 4	3 4	4 4	5 4	T	T	T	T	F
1 4	2 4	3 4	4 4	5 4							
T	T	T	T	F							
Set Total to Total + (A[J] - Mean) ²	Total = 25 50 275 500										
Set Variance to Total / N	Variance = 125										

The algorithm of Figure 8.7 requires the calculation of the mean of the numbers before it can compute their variance. This forces it to make two “passes” over the array components, first to compute the mean and then to compute the variance.

There is another method for computing the variance. This method, shown in Figure 8.8, does not need to compute the mean first, so only one pass over the array components is necessary. It is always useful to approach a problem two different ways, as we have done here. Doing so can help us choose more efficient solutions like the one in Figure 8.8.

Figure 8.8 Variance another way

Note how short this second way is compared to Figure 8.7.

$$\begin{aligned} \text{Variance} &= (A[1]^2 + A[2]^2 + A[3]^2 + A[4]^2) / N - \text{Mean}^2 \\ &= (900 + 400 + 1600 + 100) / 4 - 25^2 \\ &= 3000 / 4 - 625 \\ &= 125 \end{aligned}$$

Input Array (A, N)	N = 4				
Set Sum to 0	Sum = 0				
Set Total to 0	Total = 0				
For K = 1 to N by 1		1 4	2 4	3 4	4 4
Set Sum to Sum + A[K]		T	T	T	T
Set Total to Total + A[K] ²					5 4
Set Mean to Sum / N		Sum = 30	50	90	100
Set Variance to Total / N - Mean ²		Total = 900	1300	2900	3000
					Mean = 25 Variance = 125

Two-Dimensional Arrays

Two-dimensional arrays are also called *tables*, or *matrices*. A table may be viewed as a vector whose components are themselves vectors. As is the case for the one-dimensional arrays, all values in a two-dimensional array must be of the *same kind*, since arrays are homogeneous.

Figure 8.9 shows a general two-dimensional array *A* with *M* rows and *N* columns but no values inside of them. The value of a component is found by using two indices, one to identify the row (first dimension) and the other to identify the column (second dimension). The element *A*[*I*, *J*] is found by moving horizontally along row *I* and vertically down column *J* until the row and the column meet: the value of the element is at the intersection.

Figure 8.9 A general array

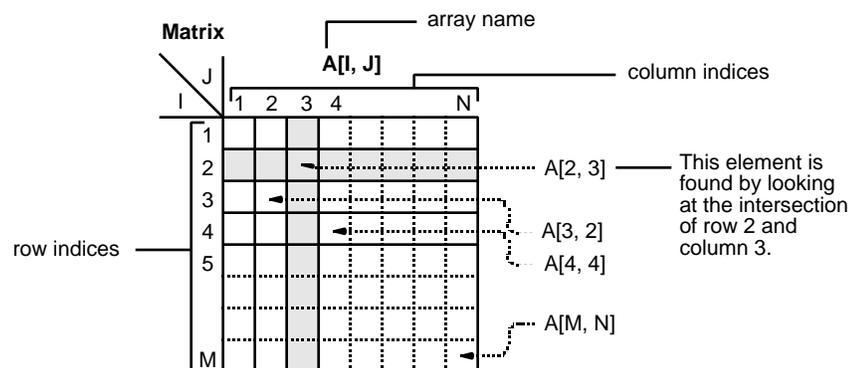


Figure 8.10 shows a more specific table which, this time, contains actual Integer values. This table represents the grades 4 different people received on 3 different quizzes. The grade of person P on quiz Q is denoted $Grades[P, Q]$. In this example, person 2 made a grade of 80% on quiz 3. This value of $Grades[2, 3]$ is not to be confused with $Grades[3, 2]$, which is the grade of person 3 on exam 2 (of 100%). The order of the indices is important! This small table has only four rows and three columns, but could be expanded easily to accommodate more people or more quizzes.

Figure 8.10 A 4 × 3 array for Grades

		Grades[P, Q]		
		1	2	3
P	1	40	60	80
	2	70	30	80
	3	100	100	100
	4	80	60	40

M = 4

N = 3

Grades[2, 3]

Grades[3, 2]

We could represent a variety of games, such as chess, checkers, or Tic-Tac-Toe using tables. A representation of Tic-Tac-Toe is shown on the left-hand side of Figure 8.11. Each table entry (i.e. $Tic-Tac-Toe[Row, Column]$) can have one of three character values: an 'X', an 'O' or a blank ' '.

Figure 8.11 Two-dimensional tables

		Tic-Tac-Toe[Row, Column]		
		1	2	3
Row	1			
	2		X	
	3	O		

		Charge[Adults, Kids]			
		0	1	2	3
Adults	1	3.00	5.00	7.00	9.00
	2	6.00	8.00	10.00	12.00
	3	9.00	11.00	13.00	15.00

See Figure 3.1 in Chapter 3 for the Charge algorithm.

Another example of a two-dimensional array is the table of admission charges Charge Table for different combinations of Adults and Kids, also shown in Figure 8.11. There are two important things to notice in this table.

- (i) The column indices begin from 0, instead of 1; and
- (ii) The row and column indices are more than simple reference points: they represent the actual number of adults and kids necessary for the corresponding charge.

Notice that the values within each of our example arrays are all of the same kind: the grades are all percentages, the game positions are all characters, the admission charges are all money values.

We saw in the last section how to scan values into a one-dimensional array. Let's look at how it is done for a two-dimensional one. The following algorithm shows how the items of Figure 8.12 can be scanned *row by row*.

Figure 8.12 Scanning an array row by row

Table [First Index, Second Index]

Col. \ Row	1	2	3
1	1	2	3
2	4	5	6
3	7	8	9
4	10	11	12

Row Traversal
 For First Index = 1 to 4 by 1
 | For Second Index = 1 to 3 by 1
 | Output Table[First Index, Second Index]
 End Row Traversal

The algorithm loops through the table elements in the order shown by the arrows: *Table*[1, 1], *Table*[1, 2], *Table*[1, 3], *Table*[2, 1], and so on, row by row. Let's take a look at the trace of the indices of *Row Traversal*:

Figure 8.13 Trace of indices from Figure 8.12

First Index =	1	1	1	2	2	2	3	3	3	4	4	4
Second Index =	1	2	3	1	2	3	1	2	3	1	2	3

Notice how the First Index changes more slowly than the Second Index.

We could just as easily have scanned the table column by column. Try it out.

The following *Input Matrix* algorithm inputs a sequence of values into a matrix, row by row.

Pseudocode 8.11 The Input Matrix algorithm

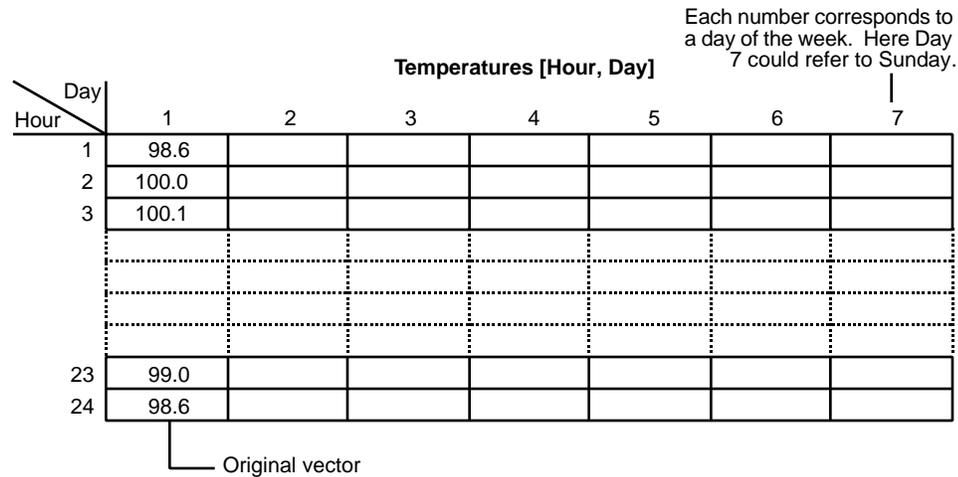
```

Input Matrix
  Input Rows
  Input Columns
  For First Index = 1 to Rows by 1
    | For Second Index = 1 to Columns by 1
    | | Input Value
    | | Set Table[First Index, Second Index] to Value
  End Input Matrix
    
```

Note that we need to know the number of rows and columns at the beginning. This might pose a problem if we use large tables. It's always better to use terminating values (as we did for inputting vectors) whenever possible.

A two-dimensional array can also simply be an extension of a one-dimensional array. For example, let's extend the *Temperature* array of Figure 8.4 by keeping the hourly temperatures for a full week (or 7 days) instead of only for one day. To do this, we need the equivalent of 7 vectors. Figure 8.14 shows the resulting matrix.

Figure 8.14 A week of Temperatures



Performing Operations on Two-Dimensional Arrays

We have already seen in this section how to perform operations on one-dimensional arrays. Let's look at what happens when you add a dimension to the array.

We will start with a calculation involving the Temperatures Table we recently constructed (see Figure 8.14). We can use the data entered in this table to calculate a number of things. For example, we could find the average temperature of a patient over the week, by summing all of the temperatures in the array and dividing this *Sum* by the total hours (7×24), as shown in the following algorithm (Pseudocode 8.12).

Pseudocode 8.12 Algorithm to find the average of Temperatures

```

Weekly Average
  Set Sum to 0
  For Day = 1 to 7 by 1
    For Hour = 1 to 24 by 1
      Set Sum to Sum + Temperatures[Hour, Day]
    Set Average to Sum / (24 × 7)
  End Weekly Average
    
```

Here we sum the hours one day at a time.

Using the Grades array of Figure 8.10, which gives the grades various students got on quizzes, we can also perform various kinds of operations. For example, it may be of interest to find the average grade for each quiz, the average grade of each student, the average weighted grade, or the average grades when the poorest values are dropped or "forgiven".

Figure 8.15 Average of columns

Grades	P \ Q	1	2	3
	1	40	60	80
2	70	30	80	
3	100	100	100	
4	80	60	40	
Averages	72.5	62.5	75.0	

2 different arrays

Let's start by developing an *Average Quiz* algorithm to compute the average grade for each quiz. We will begin with a rough outline of the steps involved.

Pseudocode 8.13 Rough outline of Average Quiz Algorithm

```

Average Quiz
  For all columns
    Set Sum to zero
    Sum Grade in column
    Set Average[Column], to (Sum/Number)
  End Average Quiz
  Averages are stored in another array.

```

This first draft consists of a For loop that selects a column, and for this column, accumulates the sum of all the grades before computing the column average.

Let's refine this algorithm by defining how the accumulation of the grades is done.

Pseudocode 8.14 Refined version of Average Quiz Algorithm

```

Average Quiz
  For Column = 1 to 3 by 1
    Set Sum to zero
    For Row = 1 to 4 by 1
      Increment Sum by Grades[Row, Column]
    Set Average[Column] to Sum / 4
  End Average Quiz

```

Summing each column, to find its average.

The resulting averages are shown at the bottom of Figure 8.15.

Going a little further, let's compute the final grade from the existing grades by computing a weighted average. Each quiz has a weight associated with it, as shown at the top of Figure 8.16 in the array *Weights*. This means that the first grade must be multiplied by 0.2 (i.e. it is worth 20% of the final mark), the second by 0.3, and the third by 0.5. For instance, the final grade of the first person (first row) is computed as:

$$\text{Final Grade} = 0.2 \times 40 + 0.3 \times 60 + 0.5 \times 80 = 66$$

Figure 8.16 The weighted average of rows

initial arrays		Weights			Final Grades — resulting array	
		0.2	0.3	0.5		
Grades	P \ Q	1	2	3		
	1	40	60	80		66
	2	70	30	80		63
	3	100	100	100		100
	4	80	60	40		54

We could develop in a top-down manner an algorithm to do this. However, this is so similar to the previous algorithm that we can simply modify it slightly, as shown below (Pseudocode 8.15). The resulting final grades are shown at the right of Figure 8.16.

Pseudocode 8.15 Weighted Averages Algorithm

```

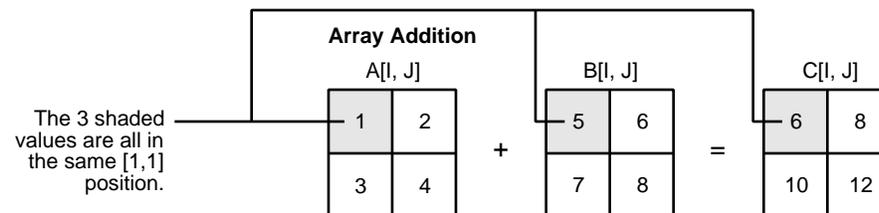
Weighted Averages
  For Row = 1 to 4 by 1
    Set Sum to zero
    For Column = 1 to 3 by 1
      Increment Sum by Grades[Row, Column] × Weights[Column]
    Set Final Grades[Row] to Sum
  End Weighted Averages
  
```

Here we sum by row, instead of by column.

Now that we have seen how to add and multiply *parts* of arrays, let's look at how to add and multiply *whole* arrays together. Adding arrays together is simple: all you have to do is add each pair of corresponding values together and place the result in the corresponding spot.

For example, in Figure 8.17, we have added the 2 tables A[I,J] and B[I,J] together to produce C[I,J]. If you look at the shading, the 1 in position [1,1] is added to the 5 in position [1,1] to produce a 6 in position [1,1].

Figure 8.17 Array addition



The algorithm to accomplish this addition is simple and is shown in Pseudocode 8.16.

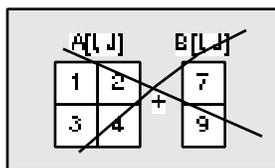
Pseudocode 8.16 Algorithm for Array Addition

```

Array Addition
  For Row = 1 to Number of rows by 1
    For Column = 1 to Number of columns by 1
      Set C[Row, Column] to A[Row, Column] + B[Row, Column]
    End Array Addition
  
```

It is important to note that only arrays of the same dimensions can be added together.

Figure 8.18 Invalid array addition

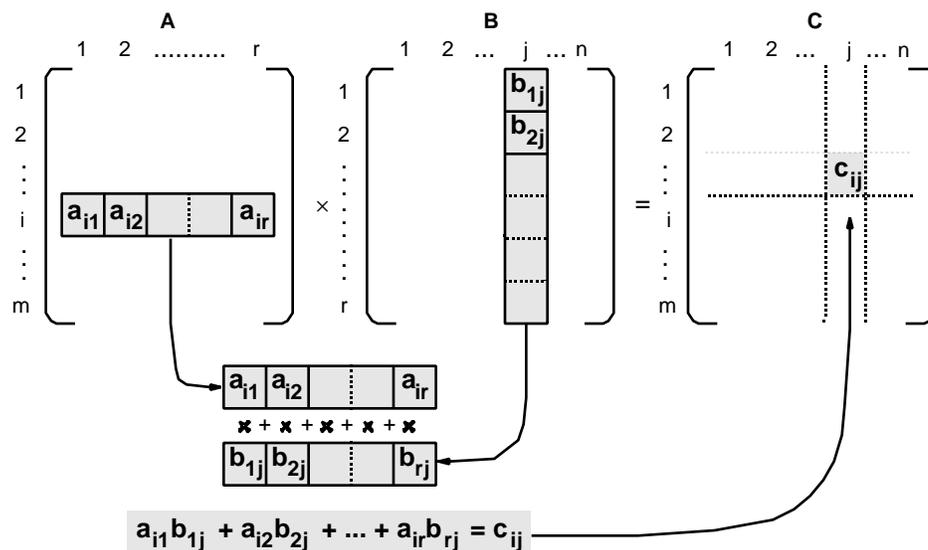


Matrix Multiplication

Multiplying two arrays is a lot trickier. Array multiplication is used in Mathematics, Science and Engineering. A complex series of operations between the rows of one matrix and the columns of another is involved, as shown in Figure 8.19. It resembles a dance between the items in a row of array A and the items in a column of array B.

Figure 8.19 Mechanism of matrix multiplication

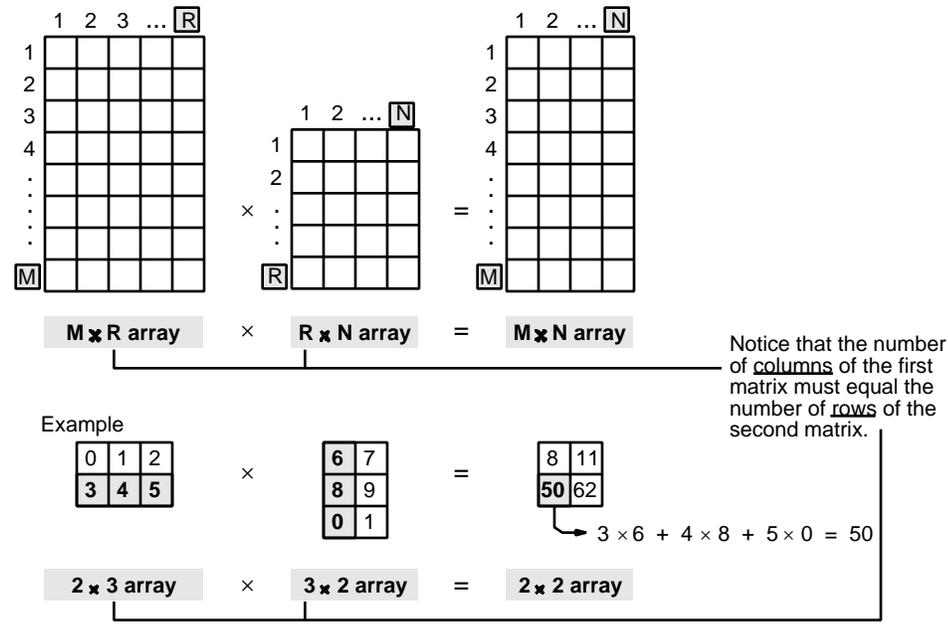
Notice that here the two arrays need not be of the same dimensions. The rule of dimensions is shown in Figure 8.20.



Note We have represented the elements differently here: we use C_{ij} instead of our usual $C[I,J]$. Both conventions are correct. Choose the one you prefer.

Each element C_{ij} is determined by taking the i th row of matrix A and the j th column of matrix B, multiplying the corresponding elements, and summing these products. Figure 8.20 illustrates further the computations involved in a matrix multiplication.

Figure 8.20 Matrix multiplication



An algorithm for matrix multiplication is defined below (Pseudocode 8.17). Note how simple it is, even though it involves a great many computations.

Pseudocode 8.17 Algorithm for Matrix Multiplication

```

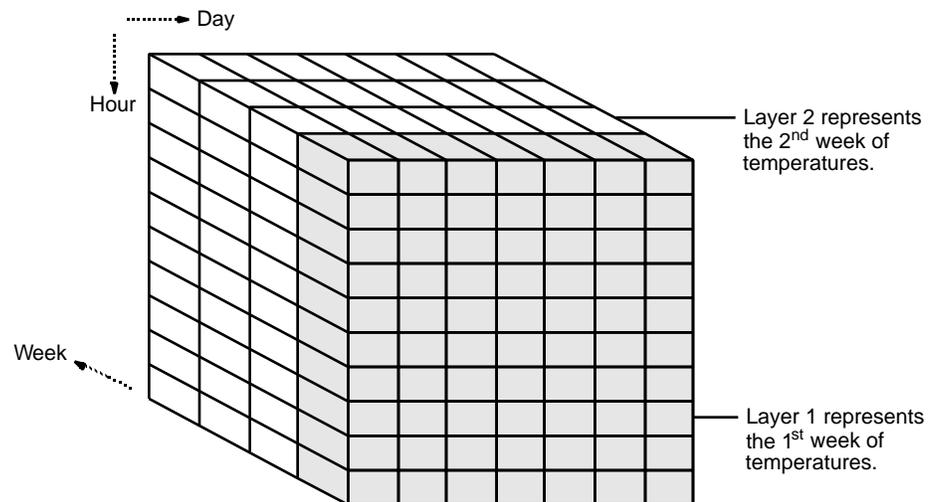
Matrix Multiplication
  For Row = 1 to Matrix 1 Rows by 1
    For Column = 1 to Matrix 2 Columns by 1
      Set Sum to zero
      For Index = 1 to Matrix 1 Columns by 1
        Increment Sum by A[Row, Index] × B[Index, Column]
      Set C[Row, Column] to Sum
    End Matrix Multiplication
  
```

We take it one row at a time.

N-Dimensional Arrays

Let's complicate things a bit and add one more dimension to our tables to make them three-dimensional arrays. This way, we can store even more information in them. Take, for example, our patient temperatures from Figure 8.14. So far, we can only store the temperatures for a patient hour by hour over one week. It would certainly be more useful for a doctor to have a record of the temperatures taken *for a whole month*. Now, we know that the number of days in a month can go from 28 to 31. To simplify matters, let's assume that our month is 28 days long: exactly 4 weeks.

So, to store temperatures for a full "month", we need 4 of the tables shown in Figure 8.14. But we do not want to deal with 4 arrays—we want all of the temperatures in one spot: in one array. To accomplish this, we need to build a three-dimensional array by putting the 4 tables together layer by layer (we will have 4 layers), forming a cube. Figure 8.21 shows this cube.

Figure 8.21 Three-dimensional temperature array

We can also extend our previous *weekly* temperature average computation to a *monthly* one. If you take a look at our previous weekly average pseudocode, you will see that all we have to do is extend the weekly temperature summations. We do this below by adding a loop for the weeks, adjusting the indices in *Temperatures* to 3, and changing the averaging divisor to reflect that we have 4 times more hours.

Pseudocode 8.18 Algorithm for Monthly Average

```

Monthly Average
  Set Sum to 0
  For Week = 1 to 4 by 1 _____ added loop
    For Day = 1 to 7 by 1
      For Hour = 1 to 24 by 1
        Set Sum to Sum + Temperatures[Hour, Day, Week]
      Set Average to Sum / (24 × 7 × 4)
    End Monthly Average

```

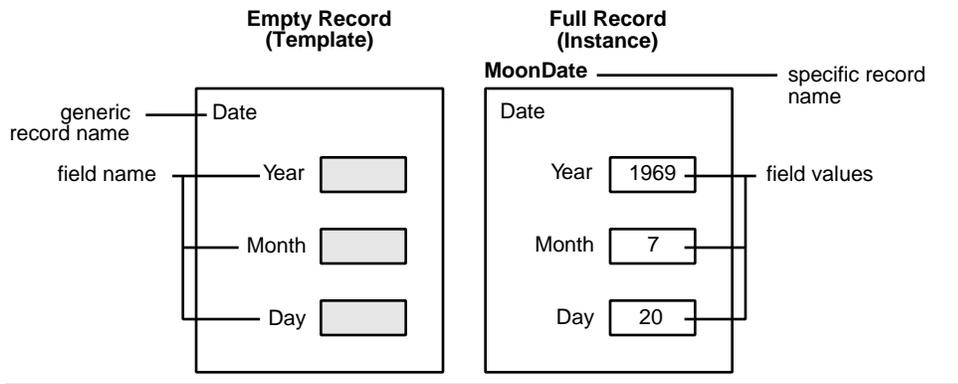
We could go even further and extend this example to a fourth dimension: we could keep the temperatures over a year (for long care patients). The four-dimensional array will be equivalent to 13 three-dimensional arrays like the one in Figure 8.21 (Why 13? Remember our “months” are “lunar”!). An individual temperature in that array would be denoted by *Temperatures[Hour, Day, Week, Month]*.

8.3 Records

What are Records?

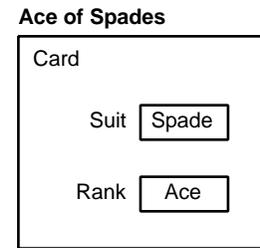
Remember that an array is a *homogeneous* collection of components; all components of the same kind. A *record* is a *heterogeneous* collection of components. In other words, the components may be of different kinds. A record is a compound data structure consisting of a number of components, usually called *fields*. Think of a record as a template that outlines each of the record’s fields.

Figure 8.22 An empty and a full Date record



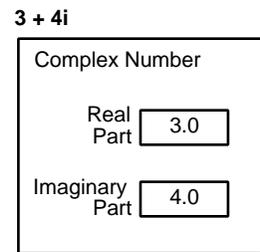
Two different Date records are shown in Figure 8.22. The one on the left is empty and the one on the right is full. Take a look at the empty one; it is a template or a skeleton. We can see that Date is a collection of 3 fields: Year, Month, and Day. On the right, MoonDate represents a specific date: the day humans first set foot on the moon. You will note that there are now values inside of the fields.

Figure 8.23 The Ace of Spades card record

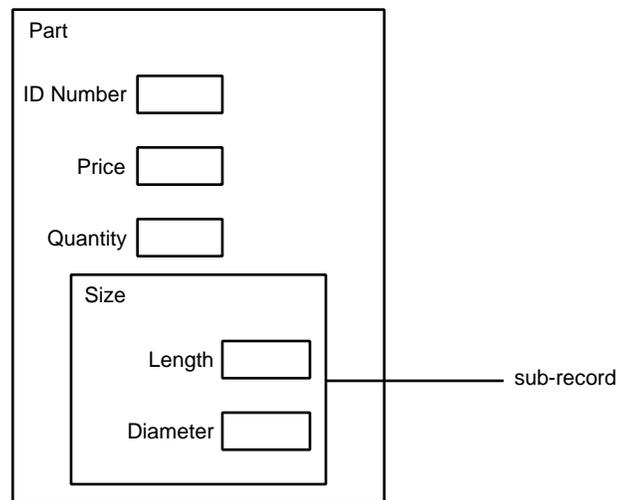


The Card record of Figure 8.23 describes a playing card in terms of its two main features, Suit and Rank. This particular diagram shows the specific Ace of Spades record. Since card decks usually have 52 cards with four possible suits and 13 possible ranks, we could use the Card record to describe any of the 52 cards.

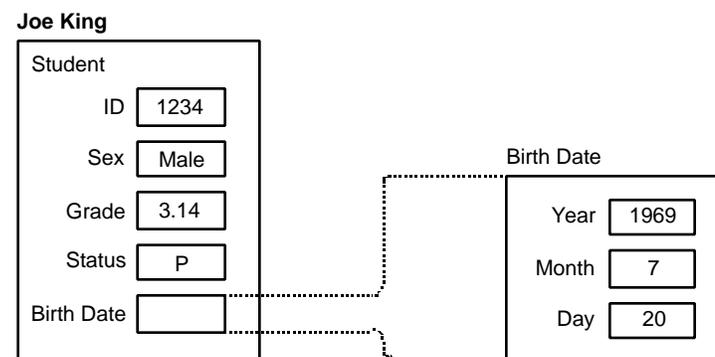
Figure 8.24 A Complex Number record



The Complex Number record of Figure 8.24, often used in engineering and mathematics, consists of two parts, a Real Part and an Imaginary Part.

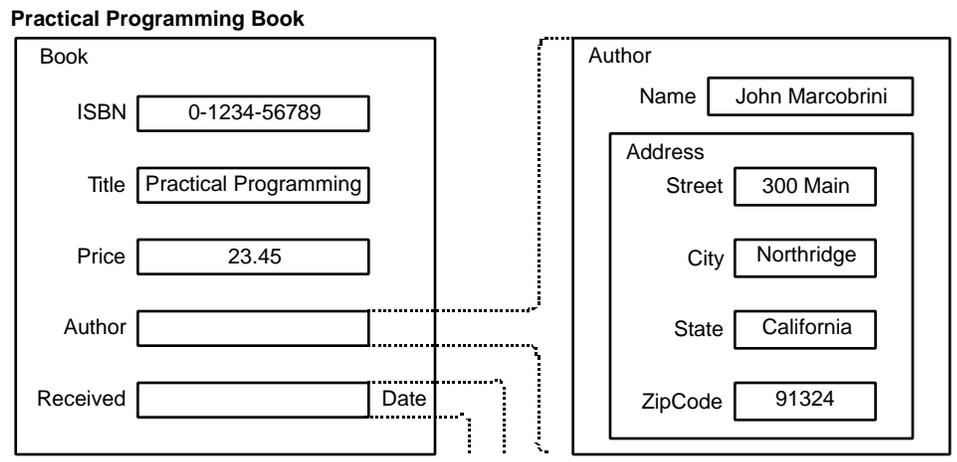
Figure 8.25 A Part record

The Part record of Figure 8.25 describes a typical part (like the parts of a chair we introduced earlier). It consists of an identification Number, a Price, a Quantity, and a Size, which is itself a record with two components (Length and Diameter). We could say that Size is a subrecord of Part.

Figure 8.26 Joe King's student record

The Student record of Figure 8.26 consists of an Identification number, a Sex (Male or Female), a Grade average, and a Status (Full time or Part time). A student also has a Birth Date, which is a subrecord within this record. This subrecord is described by the Date record, defined earlier.

The Book record of Figure 8.27 shows a more complex record. It comprises a number (ISBN), a Title (character string), a Price (dollars), an Author and a Date received. The author is described by another subrecord consisting of one field called Name, and another subrecord called Address. More of such complex records will be considered later.

Figure 8.27 Practical Programming Book record

Records can be used to describe most anything from cars to clothes, catalog items, employees, loans, reservations, customers, bank accounts, schedules, patients, and many others.

Accessing Record Components the Dot Notation

Record components are accessed by using a special dot notation. For instance, the value in the *Title* component of the *Book* record in Figure 8.27 is indicated by the following:

Pseudocode 8.19 Accessing a record's components

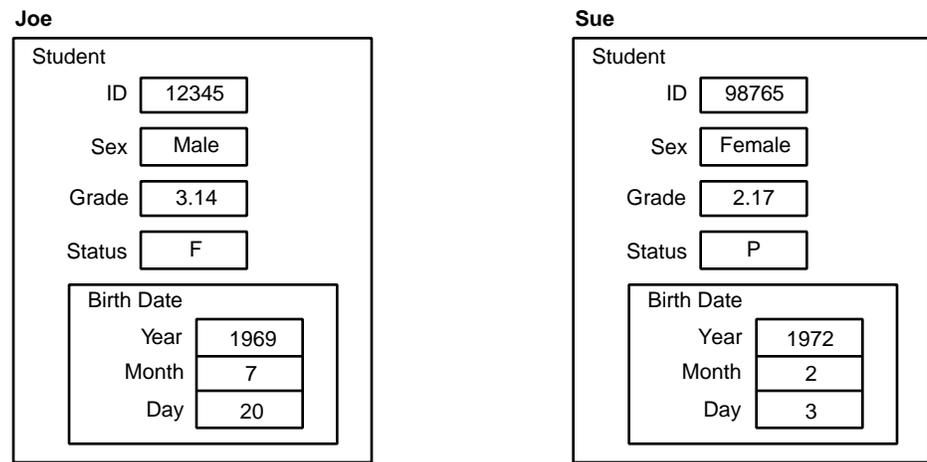
record name Book.Title record component

For example, consider the two students named Joe and X described in Figure 8.28. Joe's identification number is written *Joe.ID* and X's is written *X.ID*. Record components are treated as any other variables, and can be assigned, input, output, or compared, as shown in Pseudocode 8.20.

Pseudocode 8.20 Operations on a record's components

```
Set Joe.ID to 12345
If X.Grade < 2.5
  Output X.ID
While X.Status = P
  Set Sum to Sum + X.Grade
```

Figure 8.28 Accessing nested records



The fields of nested records (or subrecords), such as Birth Date, are also easily accessed by extending the dot notation. For example, since *Joe.BirthDate* is a record, the dot notation can be used to access its components, as in:

Pseudocode 8.21 Accessing components of records within records

Set Joe.BirthDate.Year to 1969
Output Joe.BirthDate.Day

We could have made our student record more detailed and used deeper nests of subrecords. The deeper the components, the longer the dot notations. For example, we could have had:

Pseudocode 8.22 Longer dot notations; deeper components

Joe.Spouse.BirthDate.Year
Joe.Address.Street.Number

An entire subrecord like *Joe.Birth Date* can be assigned values in one shot, without assigning values to each component individually, by a simple statement as shown in Figure 8.29.

Figure 8.29 Assigning values to entire subrecords

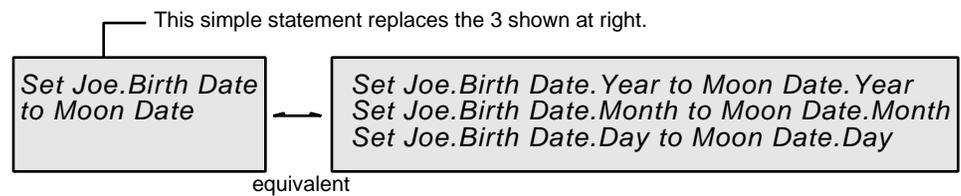
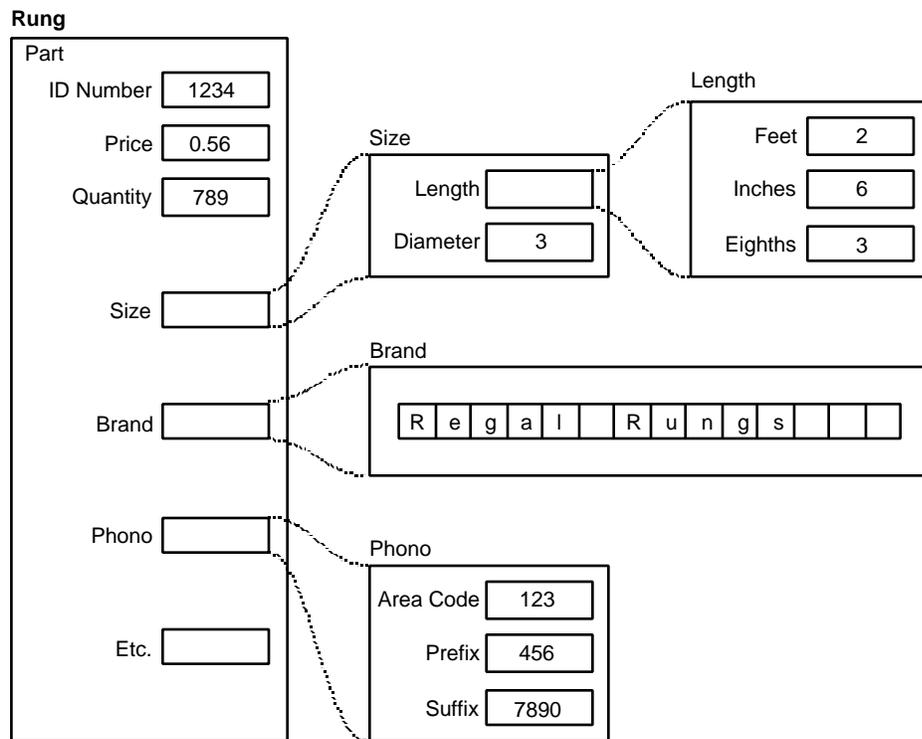


Figure 8.30 shows how one of the components of a chair can be described for inventory purposes.

Figure 8.30 Accessing nested records

Notice that the Size of a Rung has two parts, a Length and a Diameter, and that the Length itself is described by a number of Feet, a number of Inches and a number of Eighths of an inch. Accessing a deeply nested record component proceeds naturally top-down from the trunk of the main record. For example the Rung supplier's phone number Area Code is denoted:

Rung.Phono.Area Code

Similarly the statement:

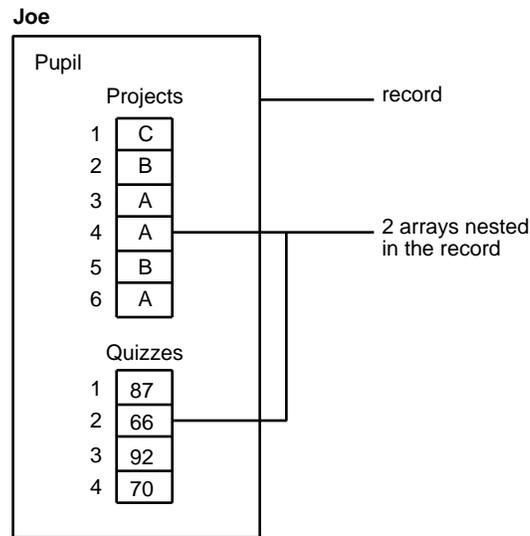
Set Rung.Size.Length.Eighths to 5

indicates that the part of the Rung's length expressed in eighths of an inch is assigned the value 5. In this way, the concept of top-down applies to items as well as actions.

Combining Arrays and Records

Arrays and records can be combined in various ways and may be nested within one another. We will show here some of the more common combinations.

Figure 8.31 Record of arrays



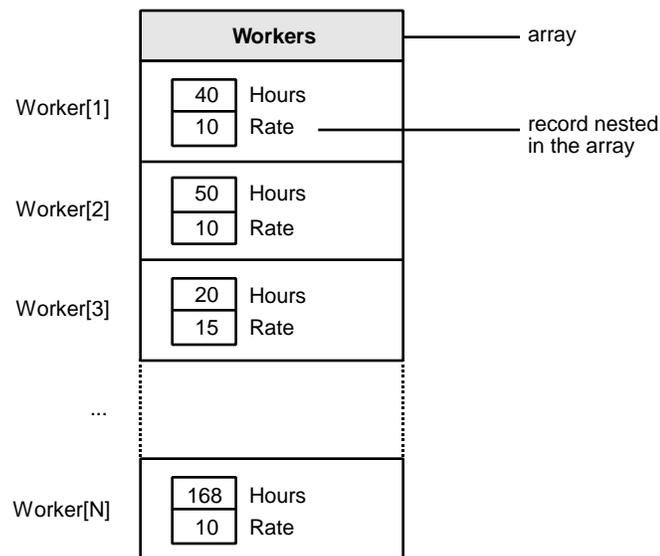
- Records of arrays are simply records whose components are arrays. For example, Figure 8.31 shows a Pupil's record consisting of two arrays. The first one contains all the grades received on Projects, and the second contains all grades received on quizzes. The score of Pupil Joe on Quiz 2 is represented by using the mixed dot and bracket notation as

Joe.Quizzes[2]

or more clearly, if the second Quiz is the midterm, as

Joe.Quizzes[Midterm]

Figure 8.32 Array of records



- Arrays of records are arrays whose elements are records. For example, Figure 8.32 shows an array of workers where the number of Hours worked and an hourly Rate of pay is specified for each Worker. The time worked by the third person can be selected by the notation:

Worker[3].Hours

More complex structures can be created from the above two structures. For example, the array of workers could actually be a component of a Department

record. In such a case, the hours worked by the third person in the research department are denoted by:

Research.Worker[3].Hours

We can keep on combining records and arrays, nesting them deeper and deeper within each other. We can even construct arrays of arrays which are convenient for structuring data. A two-dimensional array is in fact a one-dimensional array whose components are themselves one-dimensional arrays. Similarly, a three-dimensional array is a one-dimensional array whose components are themselves two-dimensional arrays, and so on.

8.4 Sets

What are Sets?

In mathematics, sets are collections of items where each item appears only once. The order of the items is immaterial and the size of the sets may be infinite. In computer science, we use a similar definition except for the fact that set sizes cannot be infinite. They also are implementation dependent. Sets are usually represented using braces as in {2, 4, 6, 8}, or {} for the empty set.

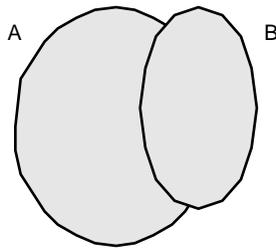
Operations on sets are the usual mathematical set operations: *membership*, *union*, *intersection*, and *difference*.

- Set *membership* is denoted by the symbol \in as in:

$$1 \in \{1, 2, 3, 4, 5\} \text{ and } 1 \notin \{2, 4, 6, 8\}$$

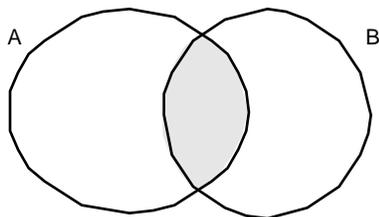
- The *union* of two sets A and B is the set comprising all elements in A and B, and is denoted by the symbol $A \cup B$, and is illustrated by Figure 8.33.

Figure 8.33 Set union

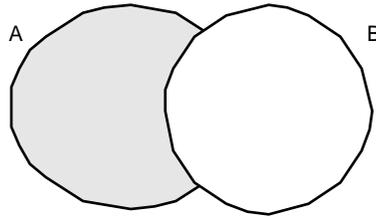


- The *intersection* of two sets A and B is the set comprising the elements common to both A and B, is denoted by the symbol $A \cap B$, and is illustrated by Figure 8.34.

Figure 8.34 Sets intersection



- The *difference* of two sets A and B is the set comprising the elements of A that are not in B, is denoted by the symbol “-” as in $A - B$, and is illustrated by Figure 8.35.

Figure 8.35 Difference of sets

Let's look at a few examples to illustrate the various set operations. First, let's define some sets:

Digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Evens = {0, 2, 4, 6, 8}

Primes = {2, 3, 5, 7}

Lucky = {7}

X = {1, 8}

Y = {1, 5, 9}

PowersOfTwo = {0, 2, 4, 8}

We then have the following identities:

X ∩ Evens = {0, 1, 2, 4, 6, 8}

Primes ∩ Evens = {2}

Lucky ∩ X = {}

Y - X = {5, 9}

1 ∩ Evens ∩ Primes

Digits = Evens ∪ Primes ∪ Lucky ∪ X ∪ Y

A Difficult Sets Example (optional)

As an example of the use of sets, we will consider a simplified version of the problem of constructing a school timetable. Suppose that:

- The students are identified by student numbers in the range 1 to *Number of Students*.
- The courses from which the students choose are numbered from 1 to *Number of Courses*.

During registration, each student makes a selection of courses. The problem is to construct a timetable where certain courses are scheduled concurrently but will not pose a conflict for any of the students. For instance, if Student 1 has chosen courses 1 and 2, these cannot be scheduled at the same time.

In practice, the construction of a timetable is a most difficult problem since the number of potential timetables is very large and a choice must be made under many constraints and with many factors to be considered, like faculty availability. In this example, we will simplify the problem drastically, and will not attempt a realistic solution of the timetable problem.

The data obtained from registration, *Registration*, can be represented as a vector of sets of courses. This vector will have as many elements as the *Number of Students*. To help understand this, Figure 8.36 shows a *Registration* array that might come from five students choosing from six courses.

Figure 8.36 Sample of Registration data for timetable problem

Student	Set of chosen courses
1	{1, 2}
2	{2, 3}
3	{2, 3, 4}
4	{1, 5, 6}
5	{3, 6}

Registration vector

By inspecting these data, we can try to find courses that can be offered concurrently. For instance, it is obvious that courses 2, 5 and 6 cannot be given at the same time as course 1 because of students 1 and 4. However, there is no conflict between course 1 and courses 3 and 4, so course 1 could be given at the same time as courses 3 and 4. But courses 3 and 4 are in conflict because of student 3 and cannot be given together, so it appears that course 1 can be given at the same time as either course 3 *or* course 4.

We could continue and determine our timetable this way. However, if we want to be able to solve timetable problems that are realistic, we must find a systematic way of doing this. You must be aware that even in a small college the number of students and the number of courses are much larger than our example, and would not be as easy to manage.

Based on our first try above, our solution will have the following structure:

Pseudocode 8.23 Structure of timetable solution

```

Build Timetable
  Get registration data
  Find the Conflicting Courses
  Build sets of Non-conflicting Courses
End Build Timetable

```

The first part of the solution will be easy to define, as we want to input data and end up with one set of courses per student as seen in Figure 8.36.

From this registration data, we want to *Find the Conflicting Courses*. To do that, we construct a new vector of sets of courses, *Conflicts*, with one element for each course. The *i*th element of *Conflicts* is the set of courses with which course *i* conflicts. A course conflicts with course *i* because one student (or more) has selected both of the courses. *Conflicts[i]* is then the set of courses that cannot be scheduled concurrently with course *i*. Thus, returning to our example data in Figure 8.36, courses 1, 2, 5 and 6 all conflict and cannot be run concurrently—because Student 1 has chosen courses 1 and 2, and Student 4 has chosen courses 1, 5 and 6. Pseudocode 8.24 constructs the *Conflicts* vector.

Pseudocode 8.24 Algorithm to construct Conflicts vector

```

Find Conflicting Courses
  For Course Num = 1 to Number of Courses by 1
    Set Conflicts[Course Num] to { }
  For Student ID = 1 to Number of Students by 1
    For Course Num = 1 to Number of Courses by 1
      If Course Num = Registration[Student ID]
        Set Conflicts[Course Num] to Conflicts[Course Num]
        Registration[Student ID]
    End For Course Num
  End For Student ID
End Find Conflicting Courses
    
```

Here we set up a vector of empty sets called Conflicts.

Loop through all courses, one student at a time.

Figure 8.37 Trace of Conflicts vector

After Course Number	Conflict Set Values					
	Initial	After Student 1	After Student 2	After Student 3	After Student 4	After Student 5
1	{ }	{1, 2}	{1, 2}	{1, 2}	{1, 2, 5, 6}	{1, 2, 5, 6}
2	{ }	{1, 2}	{1, 2, 3}	{1, 2, 3, 4}	{1, 2, 3, 4}	{1, 2, 3, 4}
3	{ }	{ }	{2, 3}	{2, 3, 4}	{2, 3, 4}	{2, 3, 4, 6}
4	{ }	{ }	{ }	{2, 3, 4}	{2, 3, 4}	{2, 3, 4}
5	{ }	{ }	{ }	{ }	{1, 5, 6}	{1, 5, 6}
6	{ }	{ }	{ }	{ }	{1, 5, 6}	{1, 3, 5, 6}

Based on the sample data in the *Registration* vector, Figure 8.37 shows a trace of how the *Conflicts* array is built, using the *Find Conflicting Courses* algorithm. The final *Conflicts* array is found in the last column at the right of the figure. This shows, for example, that courses 2, 4 and 6 all conflict with course 3 and cannot be run concurrently—because Student 2 has chosen courses 2 and 3, Student 3 has chosen courses 2, 3 and 4, and Student 5 has chosen courses 3 and 6.

Note In Figure 8.37, course 3 conflicts with course 3 - this is an unwanted but harmless by-product of our *Find Conflicting Courses* algorithm. Also, the shaded elements represent those that have changed during the loop.

Now that we have a list of the conflicting courses, we must construct a timetable. The timetable will be a vector, *Schedule*, with sets of non-conflicting courses as its elements, indexed by the session number. These elements are constructed one by one by picking from the complete set of courses a suitable non-conflicting subset of courses, *Session*. We then subtract the suitable course subset from the set of *Unscheduled* courses until *Unscheduled* is empty. *Unscheduled* is initialized to the set of all courses at the beginning.

Pseudocode 8.25 Algorithm to build Sets of Non-Conflicting Courses

```

Build Sets of Non-conflicting Courses
Set Unscheduled to {1..Number of Courses} — Initializing Unscheduled.
Set Session Num to zero
While Unscheduled { }
    | Find Next Possible Session — Still to be defined.
    | Set Unscheduled to Unscheduled - Session
    | Increment Session Num
    | Set Schedule[Session Num] to Session
End Build Sets of Non-conflicting Courses
    
```

How do we choose *Next Possible Session*? We start by selecting any course from *Unscheduled* and putting it into *Trial Set*. We then add to *Trial Set* other courses from *Unscheduled* that do not conflict with any of the courses already in *Trial Set* until no more courses can be added. The condition for a course to be added to *Trial Set* is that the intersection of the conflict set of the candidate course and *Trial Set* must be empty. We can therefore expand *Find Next Possible Session* as shown in Pseudocode 8.26.

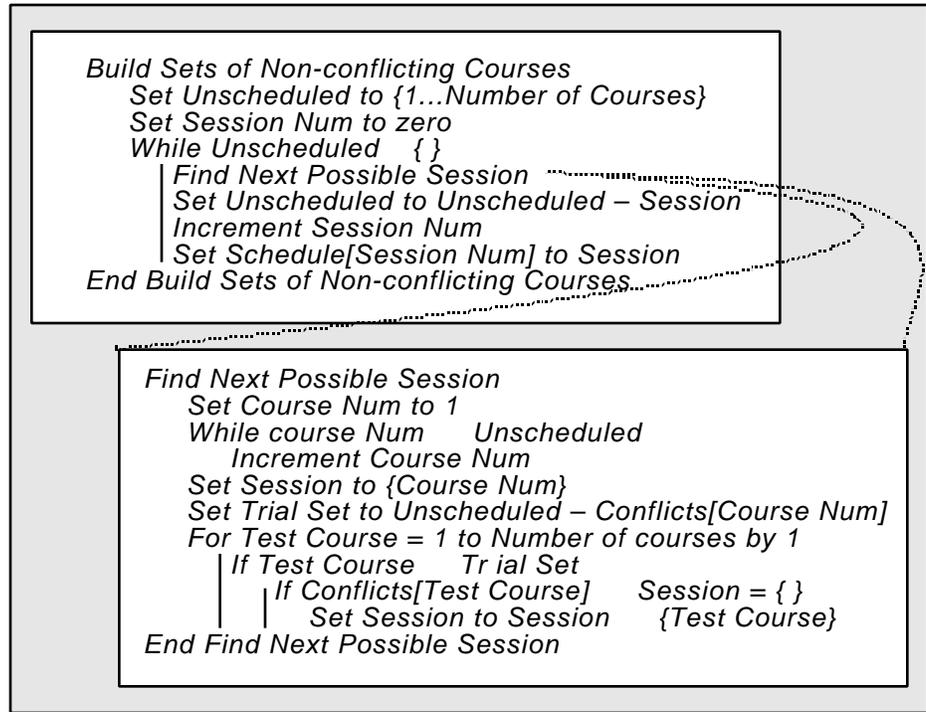
Pseudocode 8.26 Expanded Find Next Possible Session

```

Find Next Possible Session
Set Course Num to 1
While Course Num Unscheduled — finding lowest numbered
    | Increment Course Num — unscheduled course
Set Session to {Course Num}
Set Trial Set to Unscheduled - Conflicts[Course Num] — The trial set
For Test Course = 1 to Number of Courses by 1 — consists of all of
    | If Test Course Trial Set — the courses that
    | | If Conflicts[Test Course] Session = { } — don't conflict with
    | | Set Session to Session {Test Course} — the Course
End Find Next Possible Session — Number in
                                — question.
    
```

The first three lines of this algorithm find the lowest numbered course in *Unscheduled* and set *Course Num* to that value. Let's put these two pieces of pseudocode together (Figure 8.38).

Figure 8.38 Combined Algorithms from Pseudocode 8.25 and 8.26



We will now trace the *Build Sets of Non-conflicting Courses* algorithm with the same data we have used so far for the first three sessions.

Figure 8.39 Trace of Session 1 generation

Session 1		Unscheduled	{1, 2, 3, 4, 5, 6}	All of the courses have yet to be scheduled.			
Set Session to {Course Num}		Course Num	1				
Set Trial Set to Unscheduled – Conflicts[Course Num]		Session	{1}				
For Test Course from 1 to Number of Courses		Trial Set	{1, 2, 3, 4, 5, 6} - {1, 2, 5, 6} = {3, 4}				
If Test Course Trial Set		Test Course	1, 2	3	4	5, 6	
If Conflicts[Test Course] Session = { }			1, 2 {3, 4}? False	3 {3, 4}? True	4 {3, 4}? True	5, 6 {3, 4}? False	
Set Session to Session {Test Course}				{2, 3, 4, 6} {1} = { }? True	{2, 3, 4} {1, 3} = { }? False		
		Session	{1}	{1, 3}	{1, 3}	{1, 3}	

Figure 8.40 Trace of Session 2 generation

Session 2		Unscheduled	{2, 4, 5, 6}	We've already scheduled courses 1 and 3, as shown in Figure 8.35.		
Set Session to {Course Num}		Course Num	2			
Set Trial Set to Unscheduled – Conflicts[Course Num]		Session	{2}			
For Test Course from 1 to Number of Courses		Trial Set	{2, 4, 5, 6} - {1, 2, 3, 4} = {5, 6}			
If Test Course Trial Set		Test Course	1, 2, 3, 4	5	6	
If Conflicts[Test Course] Session = { }			1, 2, 3, 4 {5, 6}? False	5 {5, 6}? True	6 {5, 6}? True	
Set Session to Session {Test Course}				{1, 5, 6} {2} = { }? True	{1, 3, 5, 6} {2, 5} = { }? False	
		Session	{2}	{2, 5}	{2, 5}	

Figure 8.41 Trace of Session 3 generation

Session 3		Unscheduled	{4, 6}	We only have 2 courses left to schedule.
Set Session to {Course Num}	Course Num		4	
Set Trial Set to Unscheduled - Conflicts[Course Num]	Session		{4}	
For Test Course from 1 to Number of Courses	Trial Set		{4, 6} - {2, 3, 4} = {6}	
If Test Course Trial Set	Test Course	1, 2, 3, 4, 5	6	
		False	6 ∈ {6}? True	
If Conflicts[Test Course] Session = { }			{1, 3, 5, 6} ∩ {4} = { }? True	
	Set Session to Session {Test Course}	Session	{4}	{4, 6}

Figures 8.39, 8.40 and 8.41 trace the execution of the algorithm for each of the three sessions. Notice that it only took us 3 sessions to schedule the 6 courses. Let's summarize the course lists for each session:

- Session 1 Courses 1 and 3
- Session 2 Courses 2 and 5
- Session 3 Courses 4 and 6

Obviously, each session has 2 courses taught at the same time. However, please note that this algorithm for selecting "suitable" sessions will not generate an optimal *Schedule*. In unfortunate circumstances, the number of sessions in the *Schedule* may be as large as the number of courses, even if simultaneous scheduling were feasible.

If you have been able to follow this entire example, congratulations! This was a difficult example, and you will find that algorithms that need to use sets are usually complex. Check your understanding by using the algorithm to generate a timetable from the following registration data:

Student	Set of Chosen Courses
1	{1, 4, 6}
2	{2, 4}
3	{3, 4, 5}
4	{1, 4, 5}
5	{2, 5}

8.5 Data Structure Building Tools

Dynamic Variables

The problem with the data structures we have seen so far, is that their structure is static. That is, their size and layout are fixed when the program is written. This is a problem if we want to be able to use a broad range of data. *Dynamic variables* can solve this problem. Dynamic variable

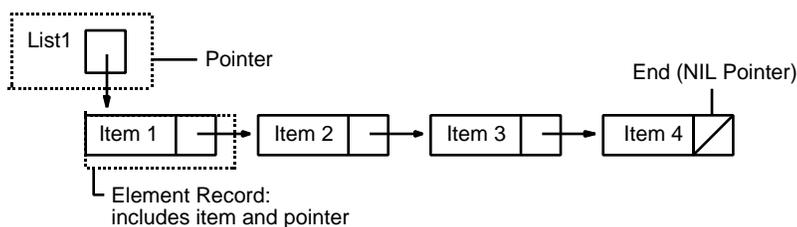
Dynamic variables are variables of a special nature. Like regular variables, they are used to store data, but unlike regular variables, their number is not known during algorithm development. They are instead created and used during the execution of the algorithm when needed. The advantage of using dynamic variables is that we do not have to guess at the number needed beforehand: we will use exactly what is needed and nothing more.

Pointers

Since we do not know how many dynamic variables we will need (that is the whole point of having dynamic variables), we cannot give them names, like we did with other variables. To represent and name dynamic variables, we need another kind of variable: *pointervariables*. *Pointers* can store only one kind of value: the location of a dynamic variable. When you point your finger at something, you answer the question “Where?” Same thing with pointers: they answer “Where is the dynamic variable?”

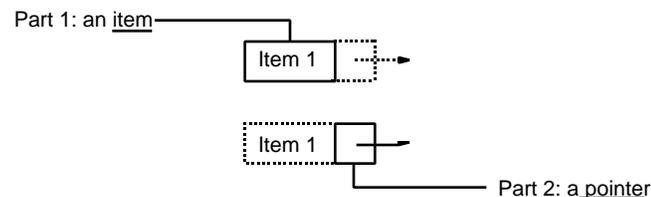
Let’s take a look at how pointers and dynamic variables are used together. In Figure 8.42, we use them together to create a list whose length is not known until the program is run.

Figure 8.42 Pointers in a dynamic list



You will notice that this list contains 4 element records. On top, we have a pointer called *List 1* whose function is to tell us “Where?” the list begins. The end of our list is marked with a pointer that has a special value, known as NIL, that points to nothing. A NIL pointer is shown as a slashed box. Each of the element records in Figure 8.42 contains 2 parts, as shown in Figure 8.43.

Figure 8.43 Two parts of an element record

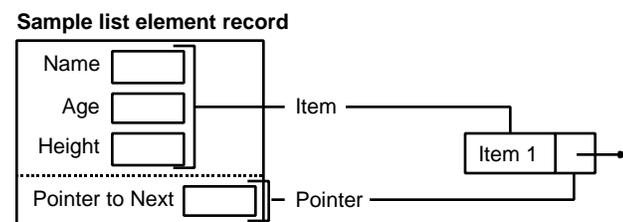


An element record in general contains a minimum of two fields (like ours above):

- one for the element information, and
- one for the pointer to the next element record.

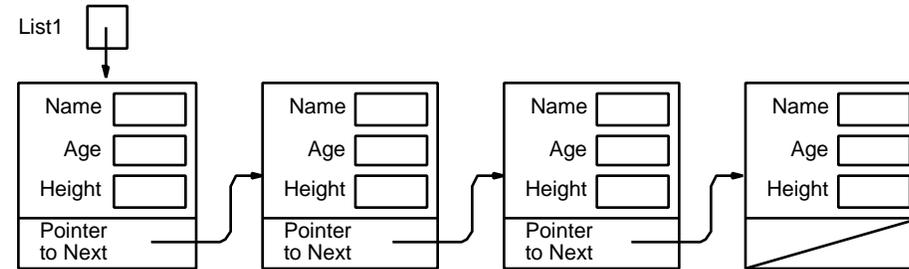
It is not unusual that a record contains more fields than that: it could have, say, three more information fields and a couple of other pointers. Data structures built in this way are known as “linked lists” because the pointers act as links between the elements. Here, we are going to keep it simple and say that we have one pointer per element record. Our element record could look like the one in Figure 8.44.

Figure 8.44 A sample list element record



The element record in Figure 8.44 has three “information” fields: *Name*, *Age*, and *Height*. Our last field is the pointer to the next element. Using this sample element record, we could redraw Figure 8.42, now specifying the content of the “Item” parts (Figure 8.45).

Figure 8.45 A simple dynamic list of linked element records



We have made the arrows bend here so that you remember this:

Note: Pointers point to the whole element record (both the item and the pointer), not to one part in particular. Usually, pointers are drawn as straight lines.

Now that we have seen what element records can contain, let’s return to our original generic example of Figure 8.42. There are certain algorithm instructions to remember when using dynamic lists. Remember that the whole of the dynamic list is only created *while the program is run*, not when it is being written by the programmer. So we have to give our algorithm instructions on how to create this list.

We will begin at the beginning: a dynamic list begins with a pointer. In our case, it is called *List1*. This pointer variable already exists in the algorithm. However, it has no value: it is empty. If we wanted to create a list, we would have to give *List1* a value to define “Where?” the list begins. The instruction to do this is:

Set List1 to New(Element)

There are two parts to this instruction:

- *New(Element)* uses a special function called *New* to: (i) allocate the memory space for the first element record; and (ii) return a value: the location of this new memory space.
- *Set List1 to New(Element)* takes this location value and puts it into the *List1* variable.

Now that *List1* is defined, we can use it to refer to either of the two parts of an element record (the item or the pointer). To do so, we write *List1* followed by an arrow like this: *List1->*.

List1->blabla can be read as *blabla* pointed to by *List1*.

- To refer to Item 1 (the first part of the first element record) in Figure 8.42, we would write:

List1->Information fields

- To refer to the pointer of the first record, we would write:

List1->Next

- Now to refer to Item 2, it gets a little trickier:

List1->Next->Information fields

- To refer to the pointer of the second element:

List1->Next->Next

In other words, we use the pointers to point the way towards the part we want. We refer to the element records as follows: the information in list elements is called *Information fields* and pointers are called *Next*. To remember how to reference dynamic lists, think of everything written before the last *->* as arrows pointing the direction of the path to follow. The name after the last *->* is the name of the part wanted. So when we are looking for Item 2, we need to write down the names of the pointers before it, followed by Item 2's name, hence:

List1->Next->Information fields.

Let's take a look at some pseudocode written to create the list of Figure 8.42.

Pseudocode 8.27 Algorithm to create the list from Figure 8.42

List Creation

Set List1 to New(Element)

Input Item 1

Set List1 Information fields to Item 1

Set List1 Next to New(Element)

Input Item 2

Set List1 Next Information fields to Item 2

Set List1 Next Next to New(Element)

Input Item 3

Set List1 Next Next Information fields to Item 3

Set List1 Next Next Next to New(Element)

Input Item 4

Set List1 Next Next Next Information fields to Item 4

Set List1 Next Next Next Next to NIL

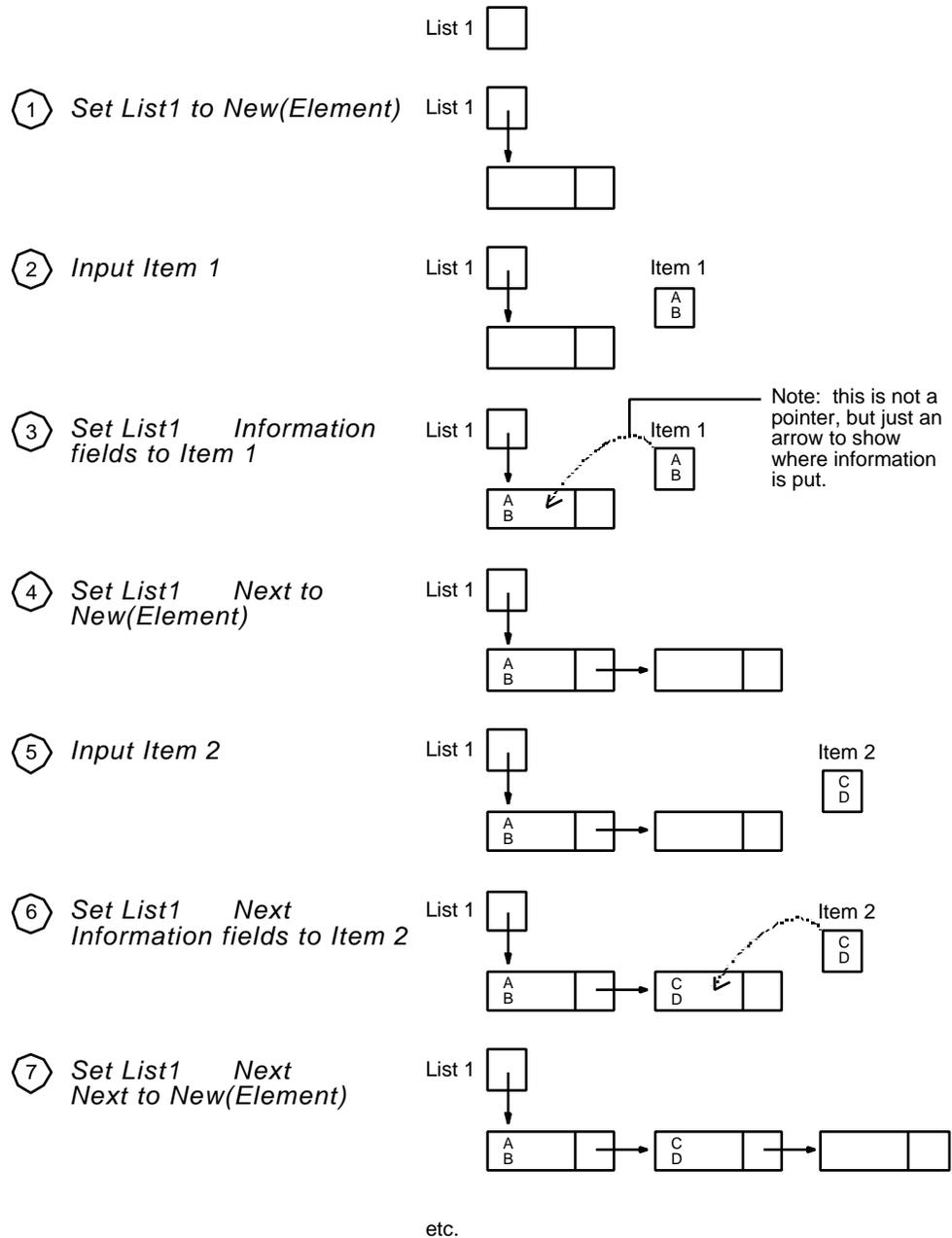
End List Creation

Everything should look relatively familiar here, in light of what we have just covered, except for the last statement:

Set List1->Next->Next->Next->Next to NIL

This makes sure the last pointer does not point to anything. We should take a look at what each instruction accomplishes.

Figure 8.46 The step-by-step creation of a dynamic list



Obviously creating a list in this manner will soon be too difficult to follow as we will have long lists of pointers. Worse than that, the length of such chains of pointers must be specified when the program is created, thus falling back into the very problem we were trying to solve.

Luckily, it is possible to proceed in a more general manner when developing such an algorithm. By using an extra pointer, *Current*, to refer to the list element being added, it is possible to rewrite our algorithm in the following manner without changing the order of the creation.

Pseudocode 8.28 Refined List Creation Algorithm using Current

```

List Creation
Input N
Set List1 to New(Element)
Set Current to List1
Input Item
Set List1 Information fields to Item
For Index = 2 to N by 1
    Set Current Next to New(Element)
    Input Item
    Set Current Next Information fields to Item
    Set Current to Current Next
Set Current Next to NIL
End List Creation

```

Shifting the pointer
so that it always refers
to element being added.

We now have an algorithm that can create a list with 4 elements or with 4,000 elements depending on the value we give to N. However, we still have not resolved the problem of creating a list without knowing its size (i.e. we still need to input N in the above pseudocode). That is left as an exercise for you to do (hint: use pieces from the *List Creation* pseudocode).

8.6 Abstract Data Types

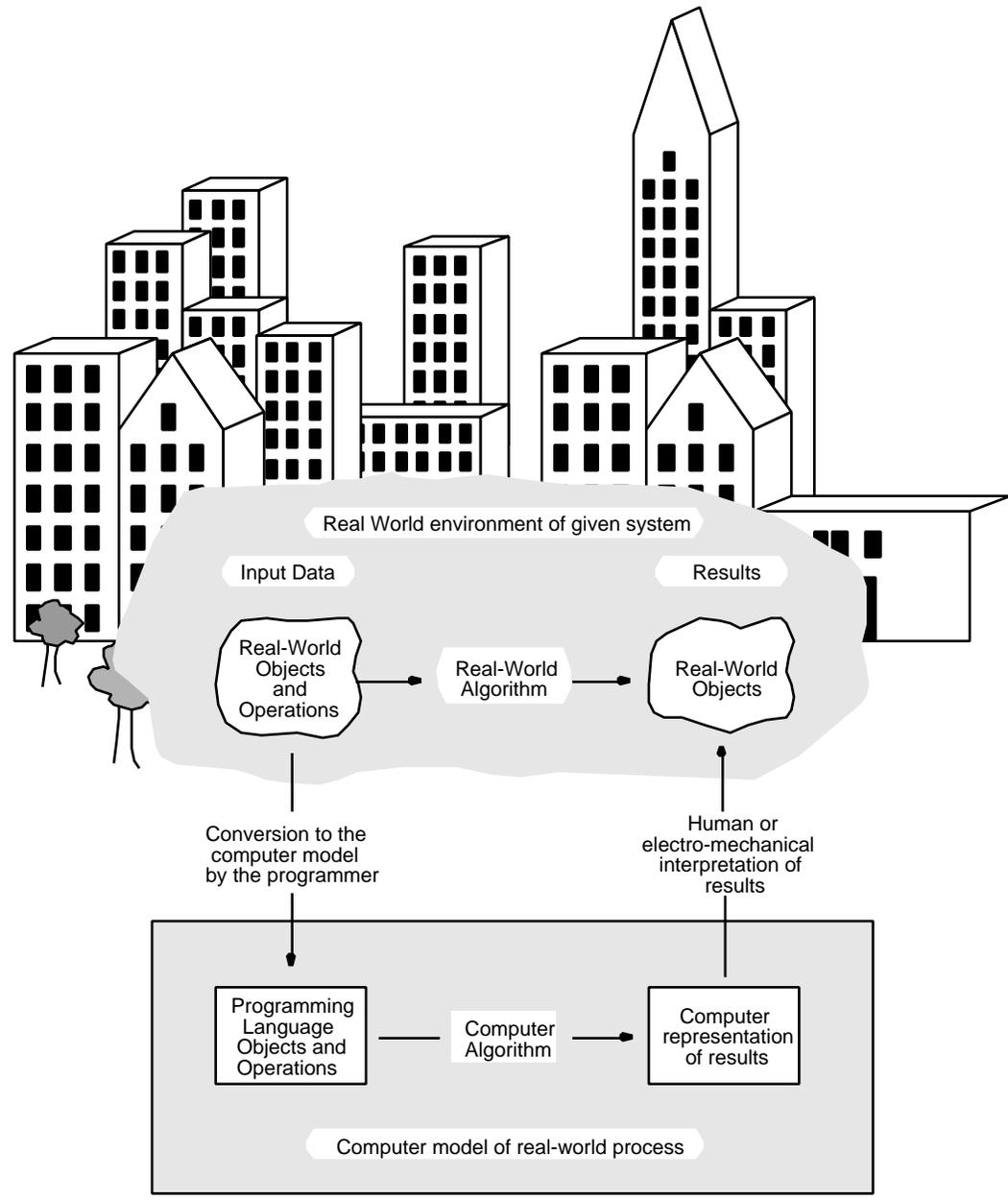
We know that, through programs, computers can model “real-world” processes like company payrolls (see Chapter 2), warehouse inventory control, and weather predictions. However, “real-world” processes are too complicated to be exactly mimicked by a computer. Using computers to model these “real-world” processes is only productive when the program accurately represents a problem. Figure 8.47 shows how a “real-life” process is transformed into a computer program. Abstract data type

Remember the company payroll example? To calculate the payroll, we needed to know the number of hours each employee worked. In the real world, an employee’s hours could be recorded by punching into a time clock, which records time in the form 8:33 a.m., or 4:59 p.m. But getting the number of hours worked is not as simple as subtracting the two numbers, as if we had 2 Integers: 459 - 833. We would get -374 hours? What does this mean?

If we converted 8:33 a.m. into the total number of minutes elapsed since midnight, when the day started, we could then represent it as an Integer. For example, 8:33 a.m. would become $8 \times 60 + 33 = 480 + 33 = 513$ minutes.

Time is one of those “real-world” objects that is hard for a computer to manipulate. Other “real-world” objects include names, dates, money, and meteorological data. Luckily, objects like money are easily represented by Integers. However, as we have seen above, Time is not easily represented by either Integers or Real Numbers. We need a data type that more closely models time as it is used in the real world.

For a computer to do this conversion, we would need an algorithm like the one called Time difference in Chapter 5, Figure 5.43.

Figure 8.47 A program is a model of a “real-world” process

You should remember from Chapter 3 that a data type (like Integers, Real Numbers, and Characters) is made up of 2 parts:

- the value
- the operations possible on the value

For example, an Integer can have any whole number value and it can be added, subtracted, multiplied, divided to produce a quotient and remainder, and compared.

All programming languages (like Pascal, Modula-2, and so on) have a number of predefined types. We saw a number of them in Chapter 5, including Integers, Real Numbers, and Characters. However, when we come across some “real-world” objects that cannot easily be modeled with these types, we need to define new types ourselves, called *Abstract Data Types*.

Abstract Data Types are not predefined: they must be defined by the programmer when needed. As the case with types in general, an Abstract Data Type must be defined in terms of its:

- possible values
- operations possible on the values.

For example, if we had to simulate a line of people waiting in line for tickets at a rock concert, we could do so quite easily by using an Abstract Data Type called a “queue”. The rest of the chapter is devoted to introducing four Abstract Data Types: strings, stacks, queues, and trees.

Strings

A character string is a sequence of characters that is viewed as one single item. A character is any element of the character set used on a particular computer. Character sets include:

- letters (upper case and lower case)
- digits
- punctuation signs, and
- other special signs including spaces.

The treatment of character Strings varies greatly from one programming language to another. Some programming languages do not include Strings, while others allow Strings of a fixed length, and still others allow all kinds of Strings of any variable length. Some programming languages, such as Snobol4, are specialized languages for string processing. Although general purpose programming languages treat Strings in a variety of ways, we can define here an Abstract Data Type String, which will illustrate the characteristics of character Strings.

Abstract data type Strings

Values

- Sequences of characters

Here are a few examples (we will enclose strings in double quotation marks).

“February 29, 1984”

“F”

“For a view of your results, press Enter”

Operations

<u>Operation</u>	<u>Example</u>	<u>Explanation</u>
• Input	<i>Input New Name</i>	---
• Output	<i>Output “Please enter quantity.”</i>	---
• Assign	<i>Set Customer Name to New name</i>	Assign one string to another

- | | | |
|---------------|---|--|
| • Concatenate | <i>Set StringC to
Concat(StringA,
StringB)</i> | Append one string to another to create a new string. If the value of <i>StringA</i> were “ABC” and <i>StringB</i> were “DEFG”, then this concatenation would produce “ABCDEFGG” for <i>StringC</i> . |
| • Count | <i>Set Character Count to
Length(StringC)</i> | Count the number of characters in the string. Here <i>Character Count</i> would be 7. |
| • Search | <i>Set Pattern Position to
Search(StringC,
“CDE”)</i> | Search a string for a particular pattern. If <i>StringC</i> had the value assigned in the Concatenate example, this instruction would set the value of <i>Pattern Position</i> to 3. Note that the way in which the position of a pattern in a string is defined varies from language to language, depending upon whether counting starts at 0 or 1. In this example, we started counting at 1. |
| • Insert | <i>Set StringD to
Insert(StringC,
“XYZ”, 3)</i> | Insert one string into another after a given position. This instruction would set the value of <i>StringD</i> to “ABCXYZDEFG” |
| • Extract | <i>Set StringE to
Substring(StringD, 2,
3)</i> | Extract a substring from a string. This instruction would set the value of <i>StringE</i> to “BCX” (3 characters starting at position 2) and leave <i>StringD</i> unchanged. |
| • Delete | <i>Set StringF to
DeleteString(StringD,
2, 3)</i> | Delete a substring from a string. This instruction would remove “BCX” (3 characters starting at position 2) from <i>StringF</i> , changing its value to “AYZDEFG”. |
| • Compare | <i>If Less(Month1, Month2)</i> | Compare two strings. This comparison is done alphabetically, i.e. “April” is less than “March”. |

Stacks

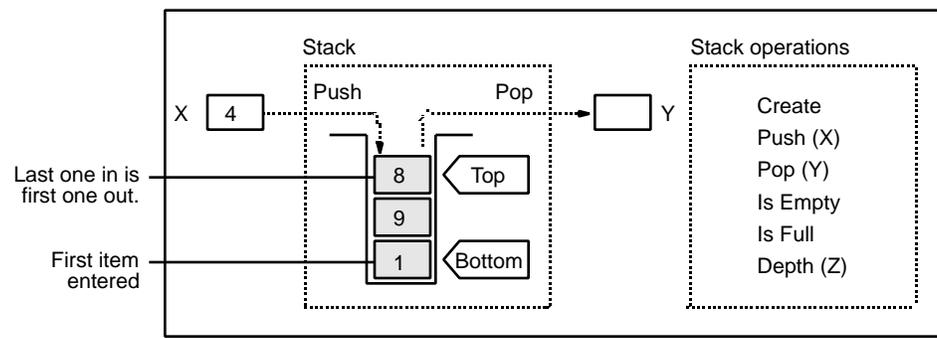
Stacks are used to implement a number of things, in particular subprogram calls. It is therefore quite natural to define an Abstract Data Type *Stack* that can be used in a number of applications. A *Stack* is an ordered collection of items, where only one item is accessible: the last one entered in the stack.

- A Stack is a homogeneous structure in which all the elements are of the same type, so the values in a Stack are all of a given type. We can have Stacks of Integers, Stacks of Real Numbers, Stacks of Characters, Stacks of a given record type, and so on.

Figure 8.48 shows a small stack of Integers, as well as the operations of the Stack Abstract Data Type.

- The top item in a Stack is the last one that was entered into the Stack.
- The first item entered into the Stack is at the bottom of the Stack.
- The behavior of a Stack is often described as “LIFO”: Last-In First-Out.

Figure 8.48 Stacks



Abstract data type Stacks

Values

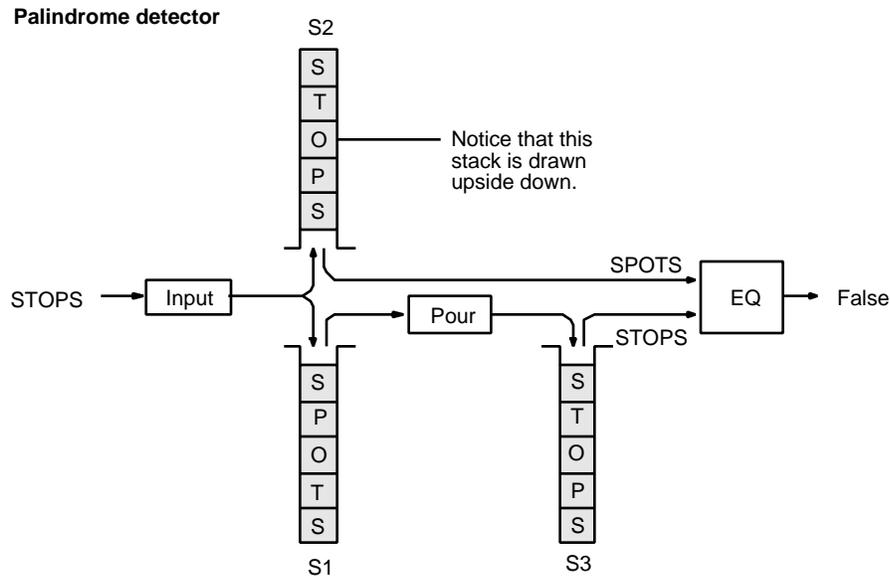
- All of the values of the stack element type

Operations

- Create a stack.
- Push an item onto a stack: the new stack top contains the pushed item.
- Pop an item from a stack: the top stack item is deleted from the stack and returned as result.
- Check if a stack is empty.
- Check if a stack is full.
- Count the number of elements in a stack.

As we have already mentioned, stacks are very useful in computing. Let's use the Abstract Data Type we have just defined to detect palindromes.

Figure 8.49 Using Stacks to detect palindromes



We need to develop an algorithm able to detect whether a given String is a palindrome. *Palindromes* are sequences of characters that read the same forwards as backwards (usually the spaces are not taken into account). For example, the sequence “RADAR” is a palindrome, while “RADIO” is not. The sentence “EVIL DID LIVE” is a palindrome. Some other well known palindromes are “ABLE WAS I ERE I SAW ELBA”, and “A MAN A PLAN A CANAL PANAMA”.

Figure 8.49 shows how we can test a String for this property. A string “STOPS” is input character by character into both Stacks S1 and S2. This could be done by a sequence of pushes, as shown in Pseudocode 8.29.

Pseudocode 8.29 Sequence of Pushes onto a stack

```

Input Character
While Character Terminator
  | If Character ''
  |   | Push Character onto S1
  |   | Push Character onto S2
  | Input Character
    
```

The Pour operation transfers the contents of Stack S1 into Stack S3. Notice that doing this reverses the order of the contents of the Stack: the top item of S1 becomes the bottom item of S3. This “reversing transfer” is done by the algorithm in Pseudocode 8.30.

Pseudocode 8.30 Reversing Stack S1 onto Stack S2

```

While NOT IsEmpty(S1)
  | Pop Character from S1
  | Push Character onto S3
    
```

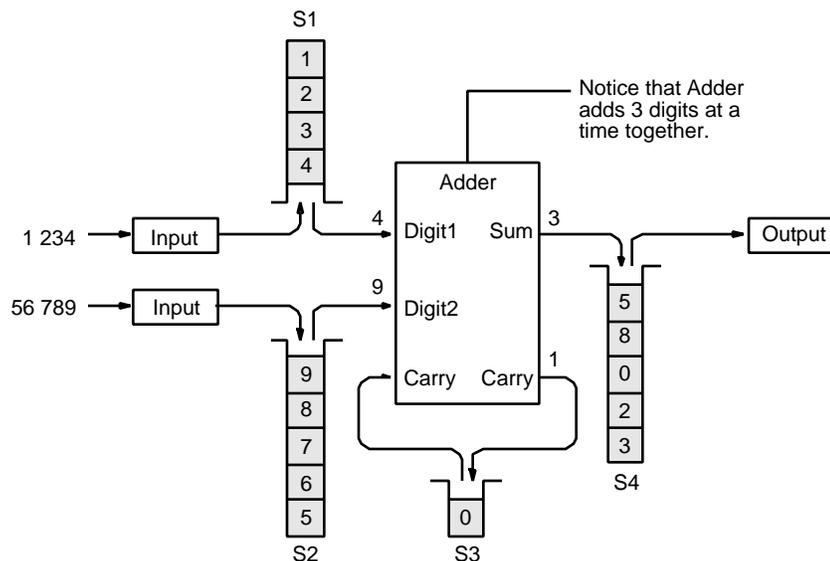
Finally the character sequence from Stack S2 can be compared to the reversed sequence in Stack S3 by the following pseudocode.

Pseudocode 8.31 Comparing Stack S3 to Stack S1

```

Set Palindrome to True
While NOT IsEmpty(S2)
  Pop Character 1 from S2
  Pop Character 2 from S3
  If Character 1 ≠ Character 2
    Set Palindrome to False

```

Figure 8.50 Big Adder

As another example, the Big Adder of Figure 8.50 shows how two Integers of any length can be added digit by digit.

1. The two numbers are input with most significant digits first, and put into stacks S1 and S2 with the least significant digits on top. The two numbers are 1 and 5 in the given example.
2. The Adder begins with a Carry of 0 on a one-digit-Stack S3.
3. The digit in S3 is added to the sum of the top values of Stacks S1 and S2.
4. The sum digit is placed onto another stack S4, with the new carry replacing the previous carry digit in S3.
5. The output stack S4 is finally output, with the most significant digits leading.

The algorithm for Big Adder is shown in Pseudocode 8.32.

Pseudocode 8.32 Algorithm for Big Adder

```

Big Adder
  Push 0 onto S3
  While NOT (IsEmpty(S1) AND IsEmpty(S2))
    If NOT IsEmpty(S1)
      Pop Digit1 from S1
    Else
      Set Digit1 to 0
    If NOT IsEmpty(S2)
      Pop Digit2 from S2
    Else
      Set Digit2 to 0
    Pop Carry from S3
    Set Sum to Digit1 + Digit2 + Carry
    If Sum > 10
      Push 1 onto S3
      Set Sum to Sum - 10
    Else
      Push 0 onto S3
    Push Sum onto S4
  Pop Carry from S3
  If Carry = 1
    Push 1 onto S4
End Big Adder

```

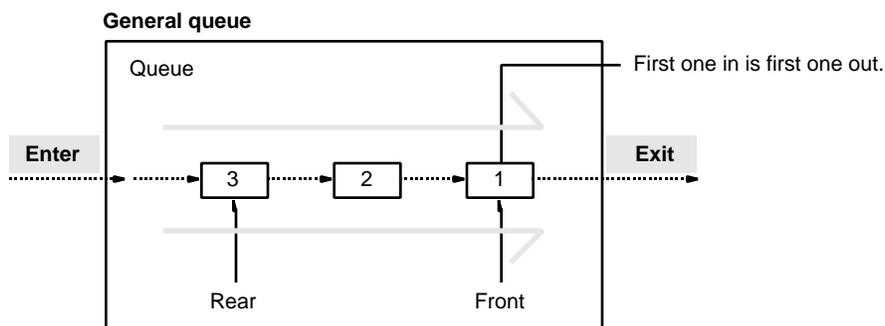
Queues

Queues occur much too often in everyday life. Queues, as we are familiar with, are the lines of people usually waiting to be served one at a time (bank teller, bus line, supermarket cashier). Examples of Queues are not restricted to people, but also include vehicles stopped at an intersection, tasks waiting to be done, and so on. Queue

Queues are used as a way of providing service according to the order in which it is requested. For example:

All our agents are busy helping other customers. Please hold. Your call will be answered in the order in which it was received.

Queues are “first-come, first-served” or “First-In-First-Out” (FIFO) structures, in contrast to Stacks, which are Last-In-First-Out structures. Queues are similar to stacks in some ways, but items are accessed from both ends: a front and a rear (unlike the single top access of stacks). Many of the Stack concepts extend to Queues. In particular, Queues are homogeneous and contain only values of the same type: queues of Integers, queues of vehicles, queues of people, etc.

Figure 8.51 A general queue

In general, a Queue consists of a channel where items enter at one end (called the Rear) and ultimately exit at the other end (the Front) as shown in Figure 8.51. There is no entry or exit in the middle of the queue. The first item into a queue is the first one out: this is why a queue is called a FIFO structure.

Abstract data type Queues

Values

- All of the values of the Queue element type

Operations

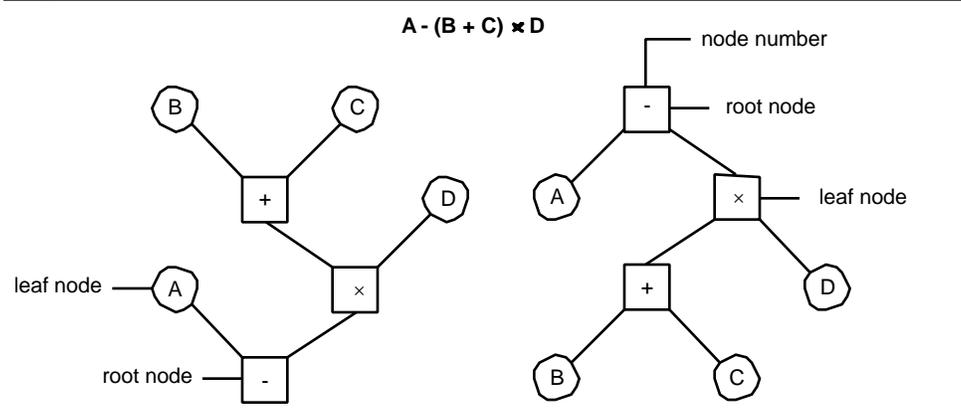
- Create a queue.
- Check if a queue is empty.
- Check if a queue is full.
- Count the elements in a queue.
- Enter an element into a queue.
- Remove an element from a queue.

Queues are used in a great number of computer science applications. They are needed in all operating systems, to keep track of the various processes that are active at a given time, and to help allocate the computer resources in an optimal manner. Queues are also used in all simulation applications. Simulation is an important field of computer science in which models of various systems of the physical world are developed. These models help us to better understand the behavior of complex systems, without having to develop or alter the “real thing”.

Trees

So far, we have seen that Stacks and Queues are useful ways to represent items that are arranged in lines (like the line of people at the bank). However, if you wanted to represent your family tree you would find that it cannot be represented by a stack or queue because a family tree contains lines with branches. This is where Abstract Data Structures called Trees can be extremely useful.

Figure 8.52 A tree



A tree consists of elements that are called nodes. Each node is a record that contains a number of information fields and a number of pointers. Each node in a tree may have zero, one or several descendants that are connected to the nodes by branches as shown in Figure 8.52.

In this figure, we have represented the same tree in two ways: one starts at the bottom and the other starts at the top. The nodes usually have values associated with them. The root is the main node and is the ancestor of all others. On the right, branches from a node lead downwards to children, or upwards to a parent. The nodes that have no children (A, B, C, D in Figure 8.52) are called leaf nodes, leaves, or terminal nodes.

Binary trees, where every node have at most two children nodes, are the most common in computing. Figure 8.52 shows a binary tree used to represent an arithmetic expression, $A - (B + C) \times D$. This tree is also called an expression tree.

8.7 Review Top Ten Things to Remember

1. The *data structures* we have considered here make it possible to create large groupings of smaller items. Arrays represent homogeneous groups; records represent heterogeneous groups. Sets represent homogeneous groups of items, but offer different operations than arrays.
2. *Arrays* give a common name to the items they group, and make it possible to access the different components via indices, and a bracket notation. *One-dimensional arrays* are quite common and easy to use.
3. *Two-dimensional and multi-dimensional arrays* are less common but sometimes better adapted to particular applications.
4. *Records* group heterogeneous items by their individual names, and make it possible to access their components by a dot notation.
5. Records may consist of other records or arrays, and in turn may be part of other records or arrays. This combination of structures is very powerful and leads to very useful structures like arrays of records or records of arrays, and so on.
6. Two-dimensional arrays are used to represent matrices and the algorithms for mathematical matrix operations.
7. In computer science, *sets* are a form of the familiar and useful concept found in mathematics. The elements of a set are homogeneous, like in arrays, but the operations on sets are very different. The set operations include Membership, Union, Intersection and Difference.
8. *Dynamic variables* are created during the execution of an algorithm and are referred to by *pointers*. *Linear lists*, a series of records connected by pointers, were also briefly presented as an introduction to dynamic data structures.
9. *Abstract Data Types* are defined by the values they represent and the operations that can be applied to these values. In this chapter, we defined the following three Abstract Data Types:
 - *Strings*,: a sequence of characters that is viewed as one single item,
 - *Stacks*,: an ordered collection of items, where the last item placed in a stack is the first one out (LIFO), and
 - *Queues*,: an ordered collection of items, where the first item placed in a queue is the first one out (FIFO).
10. *Trees* were also briefly introduced as an example of a nonlinear Abstract Data Type. Data structures are so important in computer science that, often, a whole course is devoted to them and their many implementations and uses.

8.8 Glossary

Abstract data type: A data type, generally specified by the programmer, that is defined purely in terms of its values and the operations that may be performed on them.

Dimension: The number of dimensions of an array is the number of subscripts that are needed to reference a single element of the array.

Dynamic variable: A variable that is created during execution of a program and is referenced through a pointer.

Field: An individual component of a record.

Leaf node: In a tree data structure, a node that points to other nodes.

Linked list: A data structure where the elements are linked by pointers.

Matrix: A two dimensional array that is used as a computer representation of a mathematical matrix.

N-dimensional list: An n-dimensional array.

N-tuple: A group of n numbers, a vector of n elements.

Node: A location in a tree where branches connect.

Pointer: A data item whose value is either the address of another data item or is the value NIL, in which case it is specifically not pointing to another item.

Queue: A data structure from which items are removed in the order in which they were inserted into the structure.

Root node: The first node of a tree data structure.

Set: A group of data values, all of the same type, stored without ordering or duplicates.

Stacks: A data structure in which data items are removed in the reverse order of their insertion.

String: A sequence of characters that is viewed as a single data item.

Subscripted variable: An array variable where the individual data items are referenced by subscripts.

Terminal node: The leaf node of a tree data structure.

Tree: A data structure that is similar to a linked list, except that each element, known as a “node”, has links, known as “branches”, to two or more elements instead of just one. The first node in a tree is known as the “root node” and the nodes that are not pointing to any other nodes are known as “leaf nodes” or “terminal nodes”.

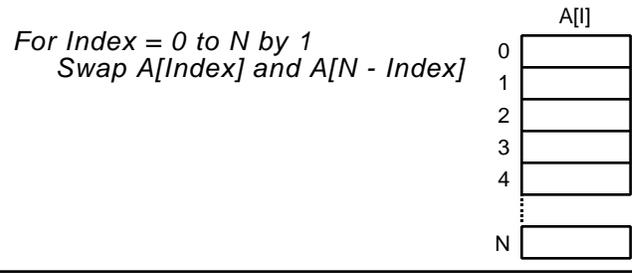
Vector: A one-dimensional array.

8.9 Problems

1. Mystery

What does the following algorithm do?

Problem 1



2. Side Bar Plot

Create an algorithm (top-down) to draw a bar plot as follows, given any number of bars N with values in array A for any given thickness of bars T , and space between bars S . Here $N = 4$, $T = 2$, and $S = 2$.



3. Upright Bar Plot

Create an algorithm to draw an upright bar plot as shown, having M bars and a maximum of N height. Here $M = 5$ and $N = 8$.

```
*
*  *
*  *
* * *
* * * *
* * * *
* * * * *
* * * * *
```

More Problems: Manipulations of Linear Lists

4. Reverse

Create an algorithm to reverse the values in an array A (with and without using recursion).

5. Normalize

Create an algorithm to convert an array of numbers (frequencies) into an array of probabilities, by summing all the values and then dividing each by this sum.

6. Weed

Create an algorithm to “weed out” or eliminate duplicate entries from an array V , creating a second array W .

7. K-Big

Create an algorithm to find the k -th largest value of an array A of N different values.

8. Intersection

Create an algorithm to compare the items of two sets (arrays each having no repeated items) and to output those items that are common to both sets.

Create also the union, those items in either set or in both.

9. Hilo-Grade

Create an algorithm that computes an average of grades, and then indicates for each grade whether it is above, at, or below the average.

10. Median

Create some algorithms to find the *median* value of an array; this is the middle value (for an odd-sized array), or the average of the two mid values (for an even-sized array).

11. A La Mode

The mode M of an array A of N values is the value which occurs most often. Show two general algorithms to determine the mode. Anticipate exceptional cases and handle them in any way you wish.

Problems On Tables**12. Scale Time**

Given a table indicating the time $T[I, J]$ to travel between points I and J in hours, create an algorithm to convert this time to minutes.

13. More-Charge

Recall the previous Charge algorithm of Chapter 3, when drawn as a table (where the charge $C[\text{Adult, Baby}]$ is three dollars for each Adult, and two dollars for each Baby). Create an algorithm to draw such a table for any given number of adults M and babies N .

14. Flip-N-Flop

Trace the algorithm given below when operating on the given table. Then verbally indicate its behavior in general.

Problem 14

```

For I = 1 to N by 1
  | For J = 1 to N by 1
  |   Swap A[I, J] and A[J, I]

```

15. Transpose

Create an algorithm to transpose a rectangular array (i.e., to convert all values $A[I, J]$ to $A[J, I]$). This essentially “rotates” the array values about a diagonal.

16. MiniMax

Given an array of M rows and N columns, construct a flow block diagram that finds the smallest entry in the row having the greatest sum.

17. Tic-Tac-Win

The game of Tic-tac-toe (or Xs and Os, or naughts and crosses) is played on a table as shown in Figure 8.11. A game is won if there are three identical symbols in a row, in a column, or along a diagonal. Show two different algorithms to detect a win.

18. Normalize Frequencies

Two dice are rolled N times, with outcomes $D1$ and $D2$ noted and put into a table $F[D1, D2]$ of frequencies of occurrence. Create an algorithm to convert (normalize) this table of frequencies F into a table of probabilities P , by dividing each entry by N . Then use this P table to find the probability of all sums S from 2 to 11. For example, the probability of $(S=11)$ is $P[5,6] + P[6,5]$.

More Problems on Data Structures

19. University Record

Create a record of a university. It is described by a name, enrollment, age, ZIP code, and phone number (having three parts: Area code, Prefix, and Suffix). The university is also classified as being Private or Non private.

Write a program to search an array of such universities and output the names of all those within a given telephone area code.

20. Play Record

Create a record describing a card game, such as Bridge or Poker, and indicate various actions which could be performed on such an item.

Still More Problems

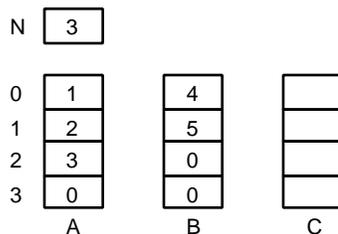
21. Convolution

Trace the following algorithm when operating on the given arrays.

Problem 21

```

Convolution
  For Index 1 = 1 to N by 1
    Set Sum to zero
    For Index 2 = 0 to Index 1
      Set Sum to Sum + A[Index 2] × B[Index 1 - 1]
    Set C[Index 1] to Sum
  End Convolution
    
```



22. Bar Plot

Create an algorithm that draws an histogram (bar plot) where values are plotted in proportion to some values (given in an array). Do this

first with the bars plotted horizontally, then again with the bars plotted vertically.

23. Weighted Forgiving Average

The Weighted Average computed for Figure 8.16 can be modified to forgive the lowest value. To do that, it is necessary to find the lowest value and its position in the array, and then to subtract this minimum from the sum and to modify the weighting that it applies to all of the other values. For example, if for some student the lowest grade were on the first project (having weight of 0.2), then the remaining projects (having a combined weight of 0.8) would be weighted by the new values 0.375 and 0.625 (computed from $0.3/0.8$ and $0.5/0.8$). Create the algorithm that will apply this forgiving method when computing a weighted average.

Chapter 9 Algorithms To Run With

The primary purpose of this chapter is to provide more extensive examples of how to use the data structures introduced in Chapter 8. We also want to introduce you to a number of standard algorithms that are used frequently in the course of programming applications.

Chapter Outline

9.1	Preview.....	416
9.2	Sorting.....	416
	General Sorting Strategies.....	416
	The Four Methods of Sorting.....	418
	Count Sort.....	419
	Count Sort Analysis.....	421
	Swap Sort.....	422
	Swap Sort Analysis.....	424
	Select Sort.....	425
	Select Sort Analysis.....	427
	Insert Sort.....	428
9.3	More Complex Sorting.....	428
	Improving Sorts.....	428
	Recursive Sorts.....	432
9.4	Searching.....	435
	Linear Searching.....	436
	Binary Searching.....	437
	Pattern Matching.....	438
9.5	Implementing Abstract Data Types.....	441
	Stacks.....	441
	Implementing Stacks with Arrays.....	441
	Implementing Stacks with Pointers.....	444
	Queues.....	445
	Implementing Queues of Arrays.....	446
	Implementing Queues with Pointers.....	448
	Trees.....	449
9.6	Review Top Ten Things to Remember.....	451
9.7	Glossary.....	453
9.8	Problems.....	454

9.1 Preview

The first two tasks that are studied, sorting and searching, are applications of the array data structures. The objective of sorting is to rearrange the data into a specific sequence, e.g. alphabetic or numeric order, while the objective of searching is information retrieval.

As is common with many tasks, there are several ways to perform sorting and searching. For example, although there are only four basic ways to sort an array, there are dozens of variations of each of these ways. Only a few of the variations are considered here. Their individual advantages and disadvantages are discussed.

A particular area of interest in the study of algorithms is their analysis. This is an investigation of the way in which their execution time increases with the amount of data being worked with. *Algorithm analysis* is presented and applied to the various algorithms that are introduced in this chapter.

As another example of the use of data structures, methods of implementing stacks, queues and trees are also introduced and discussed. Some of these techniques use arrays as the underlying data structure, while others use dynamic variables and pointers.

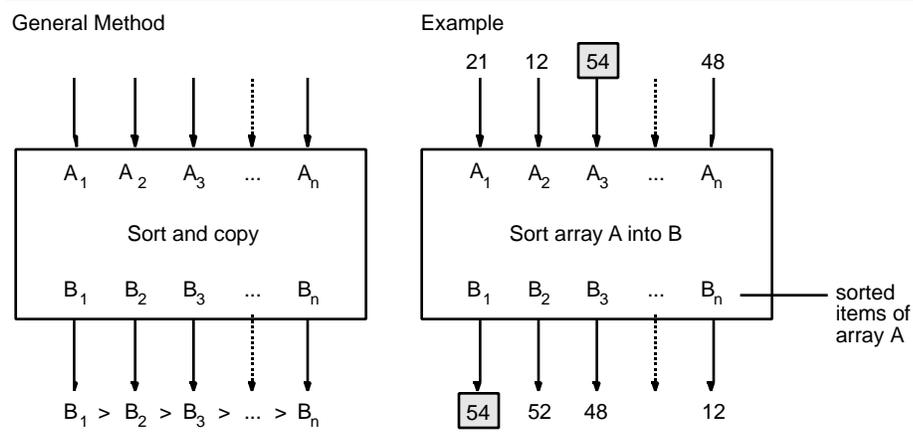
9.2 Sorting

General Sorting Strategies

Sorting is an extremely common operation in computer applications. Sorting is the process of putting items in sequence according to some criterion (e.g. numerical order, or alphabetical order, etc.). The sorted items can be small, like numbers, letters or character strings, or large records, like a library catalog where each record contains a lot of information about a book. The items of data are sorted in increasing or decreasing order of a specific field called the sort key (numerical or alphabetical). The small sorts that we have seen in Chapter 7, *Sort2* and *Sort3*, worked with a fixed number of items. Now, we want to be able to sort *any* number of items stored in an array.

The sorting process can operate according to three different strategies, as illustrated in Figures 9.1, 9.2 and 9.3. We will assume that we want to sort numbers in decreasing order here.

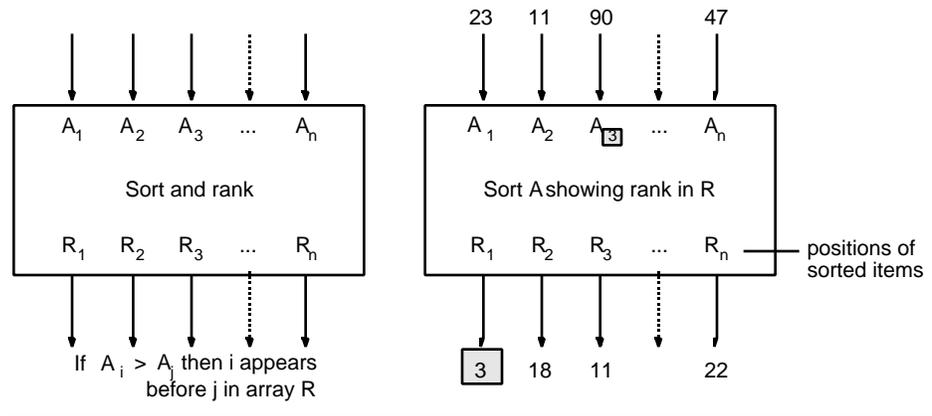
Figure 9.1 Sort and copy



- *Sort and copy*(Figure 9.1)

This method sorts the items of array A and copies them in their proper order into another array, B. This method may waste space, since it needs a second array as big as the original array to store the result. The important thing to note is that the original array is preserved throughout the sorting process.

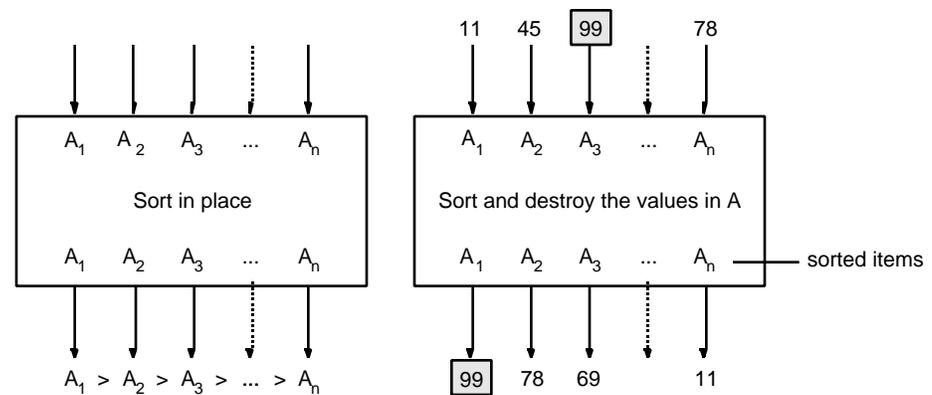
Figure 9.2 Sort and rank



- *Sort and rank*(Figure 9.2)

Here, using A once again, we generate a second array R which contains the *rank* of each sorted item. The rank is expressed as the position of the specific item in the *original* array A. For instance, in Figure 9.2, the highest value in A is 90. So R₁ refers to 90 by containing the position that 90 occupied in the original array, A: position 3. As for sort and copy, this method preserves the original array but requires a second array. That second array is probably not as big as the original one since its elements are only Integers instead of complete records.

Figure 9.3 Sort in place



- *Sort in place*(Figure 9.3)

This technique is also known as sort *in situ* or sort and destroy, since it uses only one array A. The final ordered values are placed back into A, thus destroying the original values in A. On the up side, this sorting in place is economical of space, as it requires only one array.

The Four Methods of Sorting

For each of these three fundamental strategies (Figures 9.1 to 9.3), there are many different algorithms for sorting. All of them fall into one of the following four categories:

1. Count Sort, where enumerating and comparing occur;
2. Swap Sort, where pairs of items are swapped;
3. Select Sort, where extreme values are selected;
4. Insert Sort, where moving and inserting of values occur.

We will illustrate these basic sorting methods with specific algorithms in the following sections. In order to simplify the algorithms, we assume that we will sort positive, non equal integer values into decreasing order. These algorithms can be modified very easily to sort character strings in alphabetical order. To illustrate the workings of our algorithms, we will use the first nine decimal digits as the sequence of numbers to be sorted. The original sequence is as follows:

```
8, 5, 4, 9, 1, 7, 6, 3, 2
```

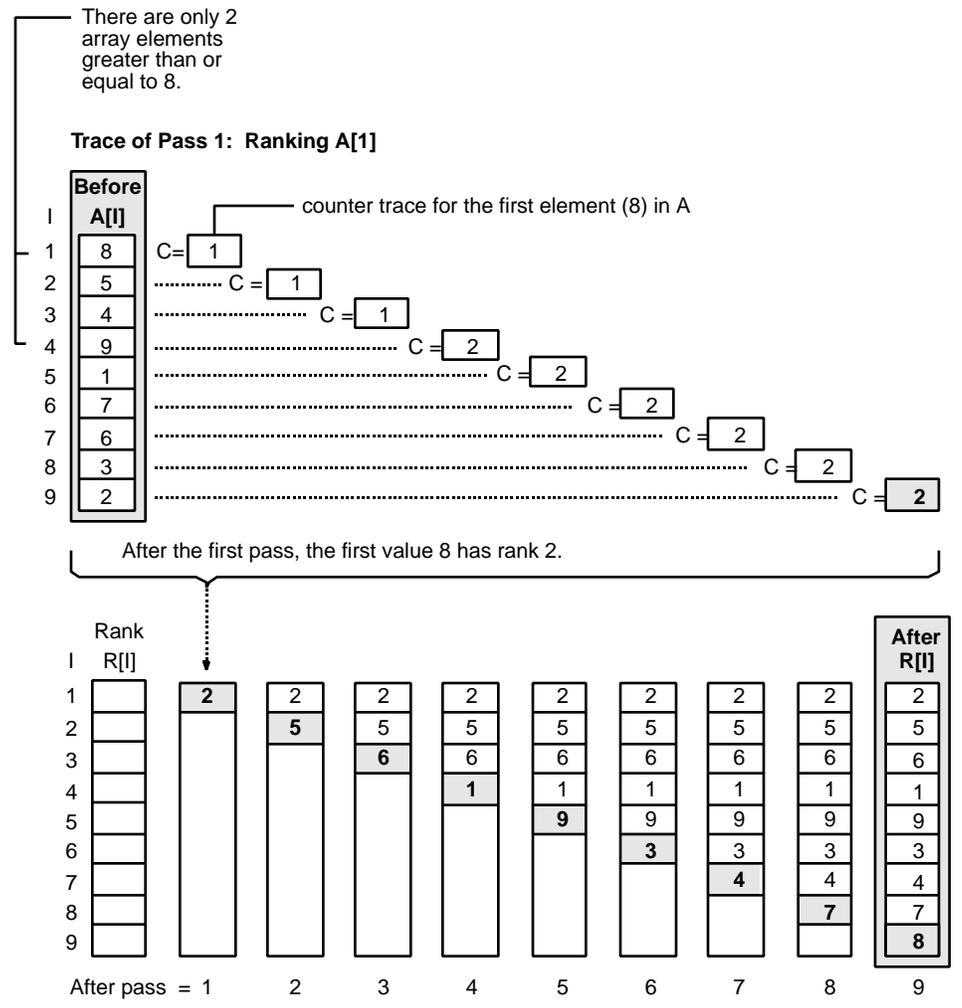
You might have noticed that these values are sorted in alphabetic order (eight, five, four, nine, etc.). However, we wish to sort them into numerical order.

Count Sort

One of the simplest sorting methods is *Count Sort* although it tends to be forgotten. This method finds the rank of all values in an array: the largest value has a rank of 1, the smallest has a rank of N (if all values are different).

The top left part of Figure 9.4 shows the original array A before it is sorted. The rank of a value X is found by comparing it to *all* values in the array of N elements, and counting the number of values that are *greater than or equal to* that value X. The largest value has only one value (itself) equal to it, and so has rank 1. The second largest has two values greater than or equal to it, and so has rank 2. The smallest has all N values greater than or equal to it, and so has rank N.

Figure 9.4 Count Sort



The first pass of the sort is shown in the top half of Figure 9.4. It takes the first value, 8, and compares it to all of the values in the array (including itself). It increments a counter each time it finds a value that is greater than or equal to 8.

1. In our case, it takes 8, compares it to the first value, 8, and puts the counter at 1.
2. Then, when it encounters the 9, the counter goes to 2.

This determines that the first value, 8, has a rank of two because only two values are larger or equal to it. The bottom half of Figure 9.4 shows how the results of each pass are used to build the Rank array. Nine passes are done in all, one for each value (N=9).

Now that we have seen how the sorting process operates, we can develop an algorithm. We will proceed top-down, first defining a rough outline for the *Count Sort* algorithm.

Pseudocode 9.1 Rough outline of Count Sort algorithm

```

Count Sort(A, Rank)
  For Pass = 1 to Number of Items by 1
    [ Count values larger than or equal to A[Pass]
      Set Rank[Pass] to count ] Actions for each pass
  End Count Sort
    
```

We now refine this algorithm by giving the detail of the loop body.

Pseudocode 9.2 Refined version of Count Sort

```

Count Sort(A, Rank)
  For Pass = 1 to Number of items by 1
    Set Value to A[Pass]
    Set Count to 0
    For Index = 1 to Number of items by 1
      If A[Index] > Value
        Increment Count
    Set Rank[Pass] to Count
  End Count Sort

```

} — Actions for each pass

- During each pass, a *Value* is selected.
- That *Value* is then compared to all the items in the array and a *Count* is kept of the number of elements that are greater than or equal to the *Value*.
- The resulting *Count* is stored into the *Rank* array.

Notice that all of the values used in this example were different, producing all different ranks from 1 to N . If some values were repeated, then some ranks would be repeated, and some ranks would not correspond to any values. For example, if the value 8 were changed to 9, then there would be two items having rank 2 and none having rank 1.

Count Sort Analysis

Whenever you develop an algorithm, you should do two things:

- (i) Test it to make sure it works, and
- (ii) Analyze it to see how well it performs.

An algorithm's performance can be measured in several ways. It can be measured through its execution time or through the storage space needed for the data. This algorithm analysis can become quite complex, and a thorough treatment of this subject is beyond our scope here. However, a brief introduction to measuring algorithm performance, often called *efficiency*, is important.

The time it takes for an algorithm to run is called its *execution time*. It is directly related to the number of operations it must perform. For example, in *Count Sort*, for an array of N values, the outer loop repeats N times and, for each of these N times, the inner loop also repeats N times. Hence, the total number of repetitions is $N \times N$ or N^2 . In our example, there were 81 repetitions because the array had 9 elements. Since each repetition means one comparison, we can say that there were 81 comparisons.

You can see that doubling the size of the array (say from 9 to 18) does not simply double the number of comparisons, but quadruples it (from 81 to 324). This quadratic growth results in a very slow sorting algorithm, especially for large size arrays.

Some computers are better than others for doing comparisons. In order to measure the algorithm's performance in a manner which is independent from the computers that run them, we must rate algorithms by assigning them *orders of complexity*. We call *time complexity* an expression that gives us an estimate of the running time of the algorithm. For example, *Count Sort* requires about n^2 comparisons for sorting an array of n items. Such an algorithm is said to have a complexity of order n -squared, denoted $O(n^2)$ —read as “big-oh of n -squared”. This big- O notation defines an upper bound for the complexity of the algorithms. More formally the big- O notation is defined as follows:

We say that $g(n)$ is $O(f(n))$ if there exist two constants, C and k , such that:
 $|g(n)| < C |f(n)|$ for all $n > k$.

If a sort has a complexity of $O(n^3)$ it is definitely slower than *Count Sort*. The best general sorts have a complexity of $O(n \log_2 n)$, which is far better than $O(n^2)$. If the sorts are very specific and limited in some way, then their computing times could be shorter, possibly linear (of order $O(n)$) or even logarithmic (of order $O(\log n)$).

Space efficiency is related to the amount of memory required. In the *Count Sort* algorithm, a second array R is required for storing the ranks, so this *Count Sort* algorithm is not the most economical of space. However, although the extra array has N elements, if the original array were an array of records, the extra array is likely to be much smaller since it only has to store Integers. The space efficiency of the algorithm might be acceptable after all. Of course, in our specific example, the rank array is as big as the original array (they both contain Integers), and the space efficiency might be considered to be somewhat weak.

Swap Sort

This family of sorts comprises a great number of variations on a common theme. We will introduce the Swap Sort theme on a specific and very well known example, the *Bubble Sort*.

Bubble Sort involves the comparison of adjacent values in the array being sorted, and swapping them if they are not in order. The top half of Figure 9.5 shows the first pass over all the items of an original array A , where all pairs of adjacent items are compared. After this pass we can say that the array is *slightly* more sorted. Notice that the largest value (originally in position 4) has finished in the first position, which is where it should be.

The results after each pass are summarized in the bottom half of Figure 9.5. As we see, after pass 1, the largest value (which is 9) has bubbled up to its final position at the top of the array, and everything else has shifted past it. After pass 2, the second largest value (which is 8) has bubbled down to the second position. This continues for each pass as shown by the shaded values. This bubbling action of each value leads to the name of this sorting method, “Bubble Sort”.

Note that after pass 4 all the values are already sorted, but the algorithm continues and does all 8 passes. Why? Because in a worst case scenario, the algorithm actually requires $N-1$ passes for all N values to arrive at their final positions. To check this out, try sorting some values that are already sorted in reverse order (like $A[9] = 9$, $A[8] = 8$, etc.). Let’s develop the algorithm top-down as usual, starting with a first draft of the algorithm.

Pseudocode 9.3 First draft of our Bubble Sort algorithm

```
Bubble Sort( $A$ , Size)
  For Pass = 1 to Size - 1 by 1
    Bubble largest value up
  End Bubble Sort
```

We now need to refine the bubbling action, where we must pass through all pairs of adjacent elements, comparing and swapping them as we go along.

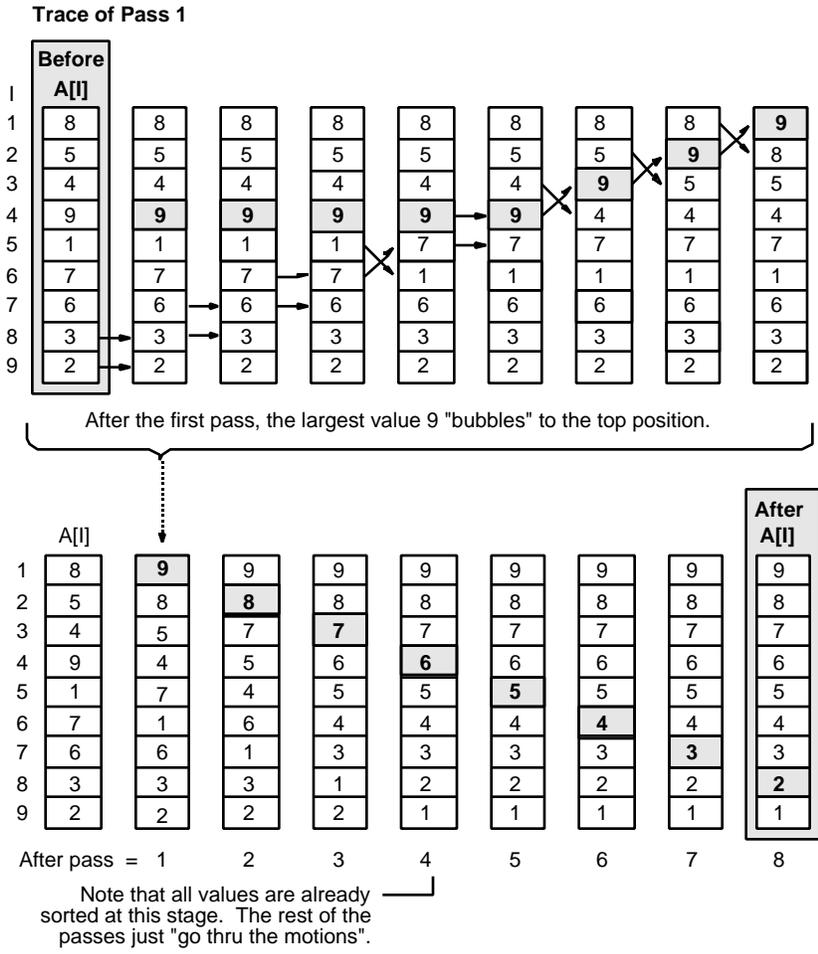
Pseudocode 9.4 Second draft of our Bubble Sort algorithm

```

Bubble Sort(A, Size)
  For Pass = 1 to Size - 1 by 1
    For Index = Size - 1 to 1 by -1
      Compare adjacent values and swap them if increasing
    End Bubble Sort

```

Figure 9.5 Bubble Sort



We reach the final and complete algorithm by specifying the details of the comparison.

Pseudocode 9.5 Final version of our Bubble Sort algorithm

```

Bubble Sort(A, Size)
  For Pass = 1 to Size - 1 by 1
    For Index = Size - 1 to 1 by -1
      If A[Index] < A[Index + 1]
        Swap A[Index] and A[Index + 1]
    End Bubble Sort

```

The Swap sub-algorithm was introduced in Chapter 5, Figure 5.12.

Swap Sort Analysis

As the pseudocode shows, this *Bubble Sort* algorithm requires (N-1) passes (in our case, 8 of them). Furthermore, each of these passes requires (N-1) comparisons, for a total of (N-1)×(N-1) comparisons. In our example of Figure 9.5, we have 9 elements to sort, so we need 64 comparisons. This algorithm is

already better than the *Count Sort* (which required 81 comparisons), and could be improved still further.

There are many ways to improve this *Bubble Sort*. Perhaps the easiest improvement comes from noticing that each pass can be one comparison shorter than the previous pass (because at the end of each pass one item has bubbled to its final position and should not be considered any more). So on pass 1 we still make 8 comparisons, but on pass 2 we need make only 7 comparisons since the 1 is already in its final position. On pass 3, we need only 6 comparisons, and so on until the eighth pass where we need make only 1 comparison. The total number of comparisons for our specific example is then

$$(8 + 7 + 6 + 5 + 4 + 3 + 2 + 1) = 36$$

which is less than half of the 81 comparisons of *Count Sort*.

We can generalize this result for an array of N values. In this case, the number of comparisons necessary for an improved *Bubble Sort* is

$$C = (N - 1) + (N - 2) + (N - 3) + \dots + 3 + 2 + 1$$

or

$$C = \frac{N(N - 1)}{2}$$

Let's compare the new and improved *Bubble Sort* to the first sort, *Count Sort*. In *Count Sort*, N^2 comparisons were needed. Here, using the new and improved *Bubble Sort*, $N(N-1)/2$ comparisons are needed.

If we were to sort big arrays, where N is very large, we notice two things:

- *Count Sort* would need N^2 comparisons
- Improved *Bubble Sort* would only need half as many comparisons: $N^2/2$. This is because when N is large, $N(N-1) \approx N^2$.

So we can say that improved *Bubble Sort* is almost twice as fast as *Count Sort*, but both sorts have complexity $O(N^2)$. If you do not know the above formula, its derivation is shown below.

The formula

$$C = \frac{N(N - 1)}{2}$$

for the sum of the first $N-1$ natural numbers is easy to derive by noting that the sum C can be expressed in two ways:

$$C = (N - 1) + (N - 2) + (N - 3) + \dots + 3 + 2 + 1$$

and

$$C = 1 + 2 + 3 + \dots + (N - 3) + (N - 2) + (N - 1)$$

If we add both formulas, forming the sum term by term, we get the following:

$$2C = ((N - 1) + 1) + ((N - 2) + 2) + \dots + (2 + (N - 2)) + (1 + (N - 1))$$

Notice that each of these $N-1$ terms has a value of N , so that

$$2C = N(N - 1).$$

The formula below is equivalent to the one above as an expression for the sum of the first N natural numbers:

$$1 + 2 + 3 + \dots + (N - 3) + (N - 2) + (N - 1) + N = \frac{N(N + 1)}{2}$$

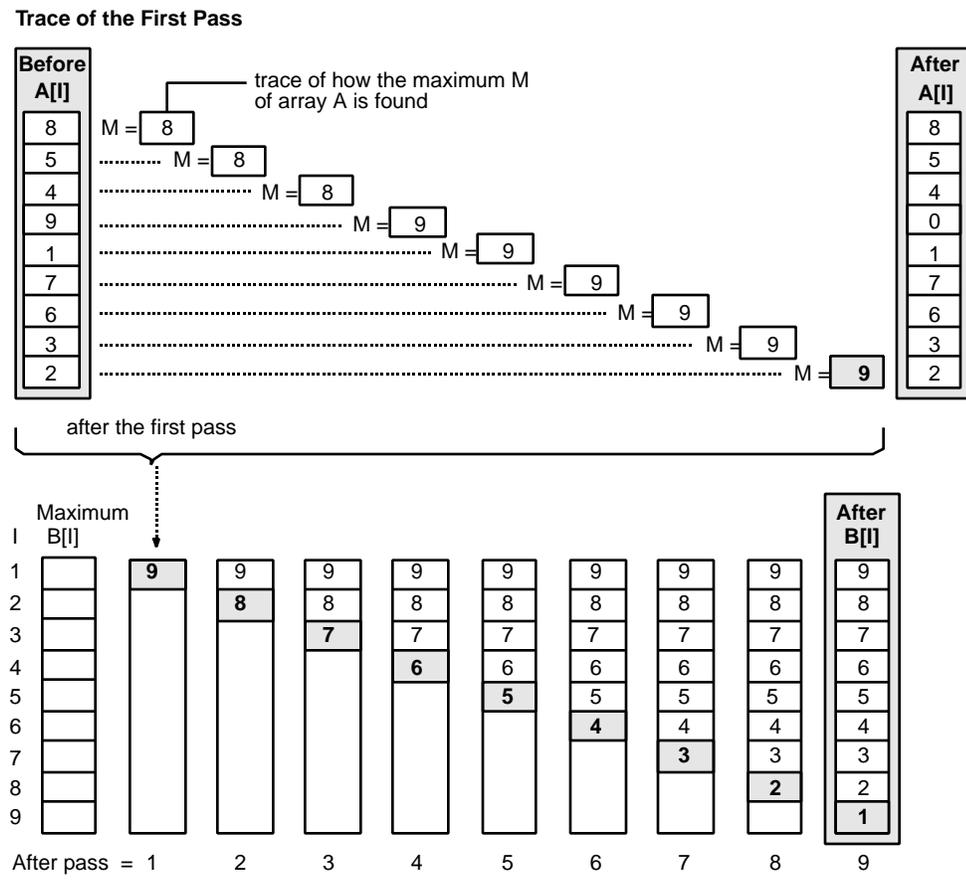
Another way to prove this is to use induction:

1. Show that it holds for $N = 1$.
2. Suppose it holds for N , show that it also holds for $N+1$.

Select Sort

The *Select Sort* algorithm is based on the selection of extreme values, either the maximum value or the minimum value. For example, the algorithm could start by finding the maximum value of the array to be sorted. This value is then noted, put into another array and eliminated from the original array. This process is repeated on the remaining original array: the maximum of the remaining values is selected, recorded and eliminated. So, as the sorted array is filled, piece by piece, the original one is emptied, piece by piece. This cycle continues until the N values in the array have been “recorded”.

Figure 9.6 Select Sort



The top half of Figure 9.6 shows the first pass over our array of positive integers. The first maximum value is found to be 9, is output (or stored in another array), and its value is replaced by zero.

The bottom half of Figure 9.6 shows snapshots of the resulting array B after each pass. Notice that the original array A is slowly destroyed in the process, ultimately becoming an array of zeros. Let's develop our *Select Sort* algorithm using our usual top-down approach. We loop N times, one pass for each value in the array to be sorted.

Pseudocode 9.6 Rough draft of Select Sort algorithm

```

Select Sort(Table, Size, Result)
  For Pass = 1 to Size by 1
    | Set Maximum value
    | Output Maximum
    | Eliminate Maximum
  End Select Sort

```

Let's refine the various parts of the algorithm. During each pass, the maximum value is found, stored in the resulting array and eliminated from the original array.

Pseudocode 9.7 Refined version of Select Sort algorithm

```

Select Sort(Table, Size, Result)
  For Pass = 1 to Size by 1
    | Set Maximum to 0
    | For Index = 1 to Size by 1
    |   | Compare Maximum to Table[Index]
    |   | Update Maximum and Position if necessary
    |   └─── finding the Maximum value in A
    | Set Result[Pass] to Maximum
    | Set Table[Position] to 0
  End Select Sort

```

Let's refine further the algorithm into its final form. The *Maximum* is compared in turn to each value in the array and updated together with its position in the original array.

Pseudocode 9.8 Final form of Select Sort algorithm

```

Select Sort(Table, Size, Result)
  For Pass = 1 to Size by 1
    | Set Maximum to 0
    | For Index = 1 to Size by 1
    |   | If Maximum < Table[Index]
    |   |   | Set Maximum to Table[Index]
    |   |   | Set Position to Index
    |   └─── finding the Maximum value in A
    | Set Result[Pass] to Maximum
    | Set Table[Position] to 0
  End Select Sort

```

The *Select Sort* method, as we have presented it, only works with positive integers, since eliminated values were replaced by zero, the smallest positive integer. If the data had included negative values, we would have to modify our algorithm to mark deleted items with the lowest possible negative value available.

Select Sort Analysis

The analysis of this sorting algorithm shows that there are N passes, and that each pass examines the N elements of the table. Consequently there are N^2 comparisons, which is similar to what we found for *Count Sort*. This version of *Select Sort* uses a second array to store the result, which, as we have already seen, is not the most efficient use of space.

We can improve this algorithm in several ways:

- We could first take the initial value of *Maximum* to be the first value in the table, and thus reduce the number of comparisons to $N-1$.
- We could also combine the selection with swapping, as follows. Instead of storing the first *Maximum* found into a second array, we swap it

with the first element of the table. Then the *Maximum* of the rest of the array is found and is swapped with the second element of the table. This selection and swapping continues until the entire array is sorted. This *Select Swap Sort* involves only one array, the original array, which is considered during the sort to be split into two parts, a sorted part initially empty, and the rest of the array which is unsorted. The sorted part will grow until it occupies the entire array.

Insert Sort

The fourth sorting method, *Insert Sort*, involves entering one item at a time from the original array A into a sorted array B by moving the existing items in B to make room for the inserted item. It is rather like a card player sorting a hand. Initially, the sorted array B is empty, and the items are inserted into it one at a time. B grows as the sort proceeds. This algorithm has a form very similar to all the sorting algorithms we have seen and, for that reason, we leave it as an exercise.

9.3 More Complex Sorting

Improving Sorts

The four sorts we considered are the simplest forms of the four basic methods. As we have already glimpsed, many variations and improvements are possible on each of these sorts. Here we will consider some ways of improving the *Bubble Sort* algorithm.

The first improvement was already discussed when we noticed that each pass could be one shorter than the previous pass, because at the end of each pass one item had bubbled to its final position. So, on Pass 1, we must make $Size - 1$ comparisons, on Pass 2 we must make $Size - 2$ comparisons and for a given Pass we only have to make $Size - Pass$ comparisons. This leads us to the algorithm in Pseudocode 9.9.

Pseudocode 9.9 Improvement to Bubble Sort algorithm

```

Bubble Sort(A, Size)
  For Pass = 1 to Size - 1 by 1
    For Index = 1 to Size - Pass by -1
      If A[Index] < A[Index + 1]
        Swap A[Index] and A[Index + 1]
    End Bubble Sort

```

Only one change is necessary: 1 was changed to Pass.

In the pseudocode describing the algorithm, this improvement is made by simply changing the inner loop terminal value from 1 to *Pass* as shown above.

In the presentation of the example of Figure 9.5, we have already noted that after Pass 4 the array was already completely sorted. This shows that the number of passes the outer loop does are not always necessary. Hence, this number, $Size - 1$ is a worst case *maximum*. In fact, in many cases like in our example, the array is sorted before all $Size - 1$ passes are done.

In order to not be wasteful, we can try to detect when the array is fully sorted and stop the loop right away. When a full pass is made without any swapping (like pass 4 in the example of Figure 9.5), this means that the array is ordered. To check this, we will use a logical variable, *Finished*, that will be set to True

before each pass, and set to False whenever we actually do a swap. This way, if during a pass we do not have a swap we can stop the process because the values are all in order.

Pseudocode 9.10 More efficient version of Bubble Sort algorithm

```

Bubble Sort(A, Size)
  Set Finished to False
  Set Pass to 1
  While (Pass < Size) AND NOT Finished
    Set Finished to True
    For Index = Size - 1 to Pass by -1
      If A[Index] < A[Index + 1]
        Set Finished to False
        Swap A[Index] and A[Index + 1]
      Increment Pass
  End Bubble Sort

```

If a swap is done,
Finished is set to False.

The above pseudocode implements this solution by changing slightly our original pseudocode. With this new version, if the original array is already sorted, only one pass will be made!

You may have wondered why the inner loop of our *Bubble Sort* algorithm works backward. We must admit that this was done so that you could see the largest value bubbling up to the top of the list. If the inner loop had worked forward, you would have seen the smallest value sink to the bottom. This would have been fine if the algorithm had been called “Sink Sort”. Either way, the result is the same. Try it for yourself and see. In the following examples, the inner loop works forward.

We can further improve our example of *Bubble Sort* in a major way by only comparing those values that are situated at a given distance apart instead of comparing adjacent values. The idea is that when two values that are a *Distance* apart are swapped, this possibly saves *Distance* individual swaps.

Such a method of sorting is called *Shell Sort*. It starts by comparing values that are a distance *Size/2* apart, then on the next cycle it compares values that are *Size/4* apart, then values that are *Size/8* apart, and so on. It continues to reduce this distance until finally it compares adjacent values. This last pass resembles our original version of the *Bubble Sort* algorithm, but as the values have already been largely pre-sorted, the number of actual swaps is low.

Figure 9.7 illustrates the *Shell Sort* method. During the first pass we consider the sequences (8, 1, 2), (5, 7), (4, 6), (9, 3). They are sorted and in the second cycle we consider the sequences (8, 6, 2, 4, 1) and (7, 9, 5, 3). The last cycle considers the entire array. In the figure, the comparisons that actually lead to a swap are marked with crossed arrows. Pseudocode 9.11 illustrates this Shell Sort algorithm.

Pseudocode 9.11 The Shell Sort algorithm

```

Shell Sort(Table, Size)
  Set distance to Size/2
  While Distance > 1
    Distant Bubble Sort (Table, Distance, Size)
    Divide Distance by 2
  End Shell Sort

```

Shell Sort uses a modified version of *Bubble Sort* called *Distant Bubble Sort*. This version compares subsets of values from the original array, values that are *Distance* apart. To do that, we have to modify the part of our latest version of

Bubble Sort shown below so that the comparisons are made between values *Distance* apart.

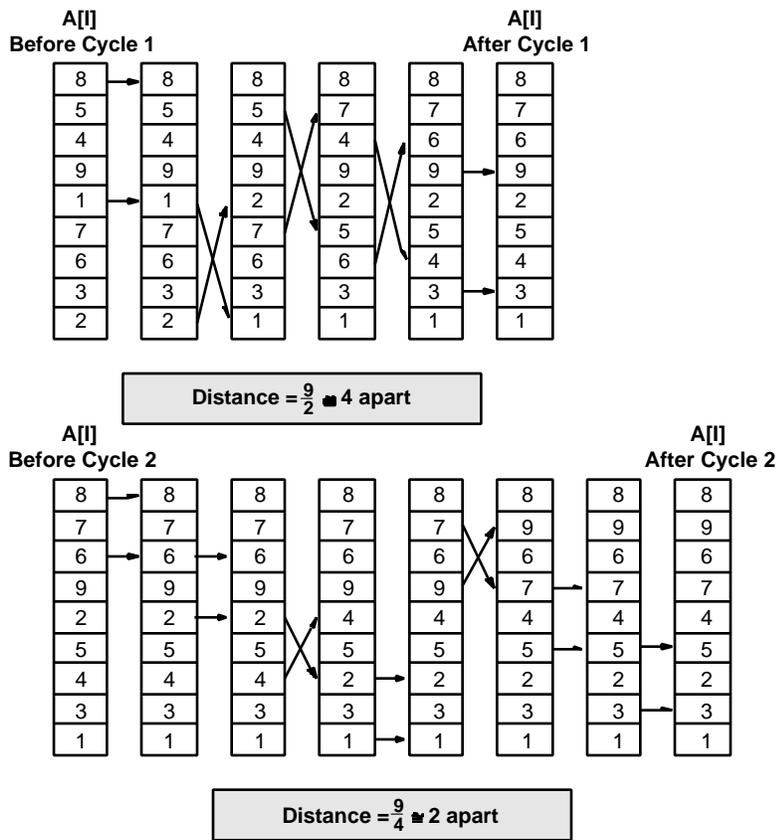
Pseudocode 9.12 Section of Bubble Sort algorithm to be changed

```

For Index = 1 to Size - Pass by 1
  | If A[Index] < A[Index + 1]
  | | Set Finished to False
  | | Swap A[Index] and A[Index + 1]
  
```

The inner For loop above must be changed to a While loop, and the comparison and swap must be changed so that elements *Distance* apart (not adjacent anymore) are considered. Another necessary modification was to make certain all sequences of spaced values were processed. To do that, we had to repeat this inner loop *Distance* times, as the number of such sequences is equal to *Distance*. These modifications, the addition of three nested loops, lead to a more complicated algorithm, *Distant Bubble Sort*, which makes it harder to follow. The algorithm is shown in Pseudocode 9.13.

Figure 9.7 The Shell Sort process



Pseudocode 9.14 The recursive Sum Up algorithm

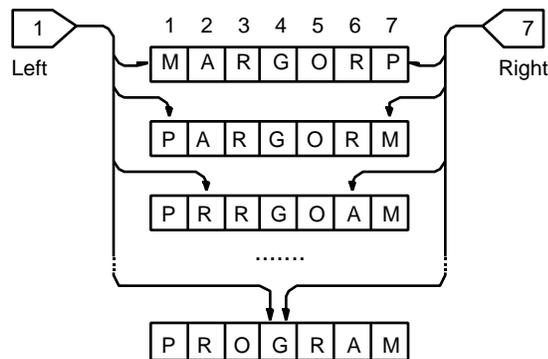
```

Sum Up(Table, Size) function
  If Size is 0
    Return 0 _____ base case
  Else
    Return Table[Size] + Sum Up(Table, Size - 1)
End Sum Up

```

As another example of the use of recursion with arrays, let's look at how to reverse the elements of an array. Among the many ways this reversal can be done, we'll choose to do it recursively by following two steps:

- 1 Swap the end points of the array, and
- 2 Apply the same *Reverse* algorithm to the remaining items, as shown in Figure 9.8. The end points *Left* and *Right* correspond to the positions of the array element. They continue to move inward, and the recursion stops when they meet (base case).

Figure 9.8 Recursive Reverse

The following pseudocode defines this recursive reversal.

Pseudocode 9.15 Recursive Reverse algorithm

```

Reverse(Left, Right, Table)
  If Left < Right
    Swap Table[Left] with Table[Right]
    Reverse(Left + 1, Right - 1, Table)
  End Reverse

```

Let's take now a more complete example of the use of recursion with arrays. Another *Swap Sort* method for an array of *N* elements, *Merge Sort*, can be defined as a recursive algorithm by finding the array midpoint, and splitting the array into two arrays starting at indices *First* and *Middle + 1*. Then these two arrays are sorted (by calling this same *Merge Sort* algorithm!). Once sorted, the two parts are finally merged together.

Pseudocode 9.16 The Merge Sort

```

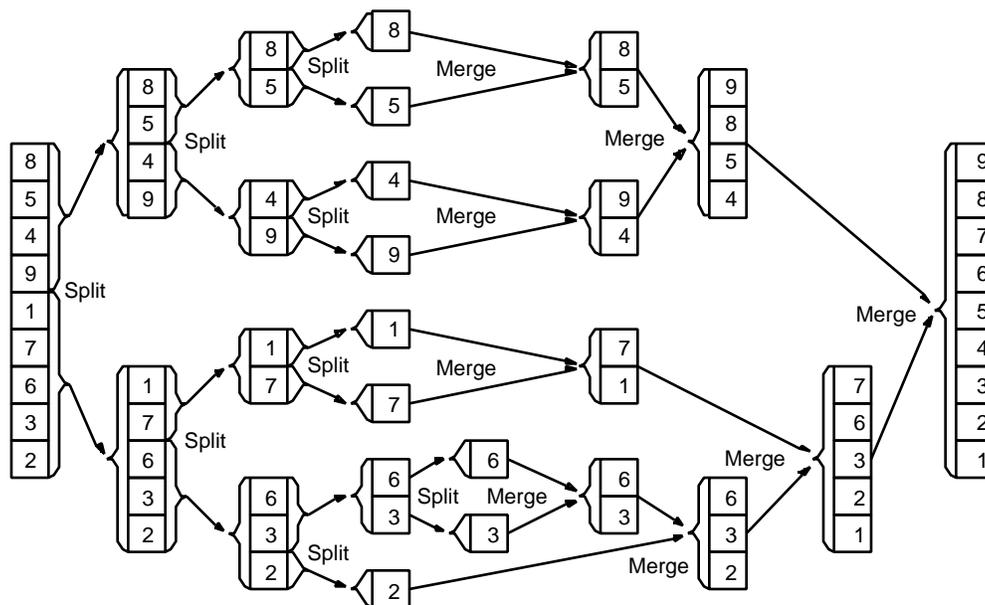
Merge Sort(First, Last, Table)
  If First = Last
    Set Middle to (First + Last) / 2
    Merge Sort(First, Middle, Table)
    Merge Sort(Middle + 1, Last, Table)
    Merge(First, Last, Table)
  End Merge Sort

```

Figure 9.9 illustrates the *Merge Sort* method applied to the set of data that we have been using in the other sorting algorithms.

The left half of the figure shows a succession of array splitting operations, corresponding to the recursive calls. The right half of the figure shows the reconstruction of the array through invocations of the *Merge* subprogram. Although this *Merge Sort* mechanism might look complex, this sort is considerably faster than all the sorting methods considered thus far.

Figure 9.9 Merge Sort execution



The merging of the two sub-arrays is done simply with an index “sliding down” each array, with the maximum value of the two indexed values put into the array *Result*, as shown by the following pseudocode:

Pseudocode 9.17 The Merge Sub-algorithm

```

Merge(First, Last, Table)
  Set Index to 1
  Set Middle to (First + Last)/2
  Set Top to First
  Set Bottom to Middle + 1
  While (Top < Middle) AND (Bottom < Last)
    If Table[Top] > Table[Bottom]
      Set Result[Index] to Table[Top]
      Increment Top
    Else
      Set Result[Index] to Table[Bottom]
      Increment Bottom
    Increment Index
  Copy remaining elements to Result
  Copy Result back to Table
End Merge
    
```

The method of Bisection described in Chapter 6 is similar in behavior to the *Merge Sort* algorithm. In both methods the array is repeatedly split until we are left with one element sub-arrays. The number of times we split a sub-array is easily found if we note that the size of the sub-arrays is divided by two at each pass:

$$N \quad N/2 \quad N/4 \quad N/8 \quad \dots \quad N/2^P$$

The last size is equal to 1 which gives us the following:

$$N/2^p = 1 \quad \text{or} \quad 2^p = N \quad \text{or} \quad p = \log_2 N$$

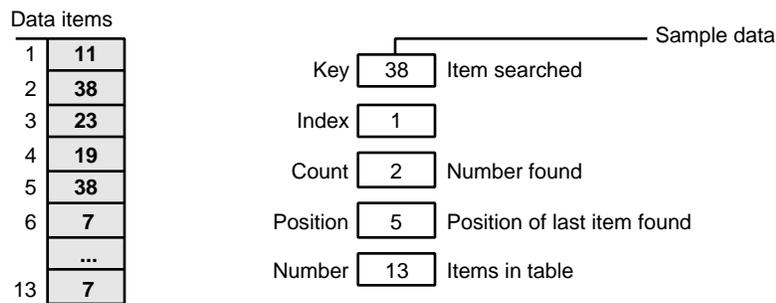
The number of passes, or the number of times the array is split, is thus $\log_2 N$. *Merge Sort* must also perform as many merges as there were splits, and each merge operation is $O(N)$. The performance of *Merge Sort* is therefore proportional to the product $N \times \log_2 N$, and its time complexity is $O(N \log N)$. This complexity is considerably better than the $O(N^2)$ performance of all the previous sorting algorithms.

To obtain a better perspective on algorithm complexity, let's look at the time complexities of the other recursive algorithms we have seen so far. Algorithms *Sum up* and *Reverse* both have a linear complexity of $O(N)$, while algorithm *Merge Sort* has a larger time complexity of $O(N \log N)$.

9.4 Searching

Searching for a particular item in a table is a very common operation. The item searched is sometimes called the search key or the target. If the item occurs in the table, we would like to know its position. If it is not in the table, we would like to receive a special value that can be tested for, or some message to indicate its absence. In some cases, we might even want to count the number of times the required item occurs in the table.

Figure 9.10 Searching for data items



The data structure of Figure 9.10 shows an array of 13 items. We are to search the array for a given item *Key*, to count in the variable *Count* the number of times it is found and to record in the variable *Position* the index of the last item found with the given key.

Of the many ways to search, we will consider the two most important ways:

- Linear search; and
- Binary search.

Linear Searching

The simplest search, a linear search, involves one pass over the elements of an entire array. The following pseudocode illustrates a simple sequential or linear search algorithm.

Pseudocode 9.18 The Linear Search algorithm

```

Linear Search
  Input Array(Table, Number)
  Input Key
  Set Position to 0
  Set Count to 0
  For Index = 1 to Number by 1
    If Table[Index] = Key
      Increment Count
      Set Position to Index
    If Count = 0
      Output "Not found"
    Else
      Output "Found" Count "at position" Position
  End Linear Search

```

increments counter
each time a key is found

This algorithm examines each element in the array once, incrementing the counter of found items when it finds an array element equal to the *Key*. *Position* indicates the position of the last occurrence found. If only the first occurrence is required, then the algorithm can be easily modified, replacing the For loop by a While loop as shown in Pseudocode 9.19. If the item is not found, the value of *Count* remains zero.

In our example in Figure 9.10, if the key were 38, the following would be output.

Found 2 at position 5

If we require only the first occurrence, we can define a faster linear search algorithm, where the loop stops with *Position* indicating the first occurrence. Pseudocode 9.19 illustrates this change.

Pseudocode 9.19 The Fast Linear Search algorithm

```

Fast Linear Search
  Input Array(Table, Number of Items)
  Input Key
  Set Position to 1
  While (Position <= Number of Items) AND (Table[Position] <= Key)
    Increment Position
  If Position > Number of Items
    Output "Not found"
  Else
    Output "Found at position" Position
  End Fast Linear Search

```

only
searching
for first
occurrence
of key

With this faster search, the item might be found after only a few comparisons or after inspecting almost all the elements in the table. It depends on where the item is located. On average, half of the elements of the table must be inspected before finding the desired element, which gives the *Fast Linear Search* a complexity of $O(N)$.

Binary Searching

The previous two versions of the linear search were general, and made no assumptions about the data. Sometimes the data values in the table are sorted, in increasing or decreasing order, and we can use this fact to speed up the search.

We used the Bisection method as early as in Chapter 4 for our guessing game.

We can use the method of Bisection described in Chapter 6 to keep reducing the size of the part of the array to be searched until either we find the item, or we know that it is not present. The binary search algorithm proceeds by halving the search range at each stage. You may wish to create this Binary search algorithm on your own by modifying the Bisection algorithm described in Chapter 6 (Pseudocode 6.26).

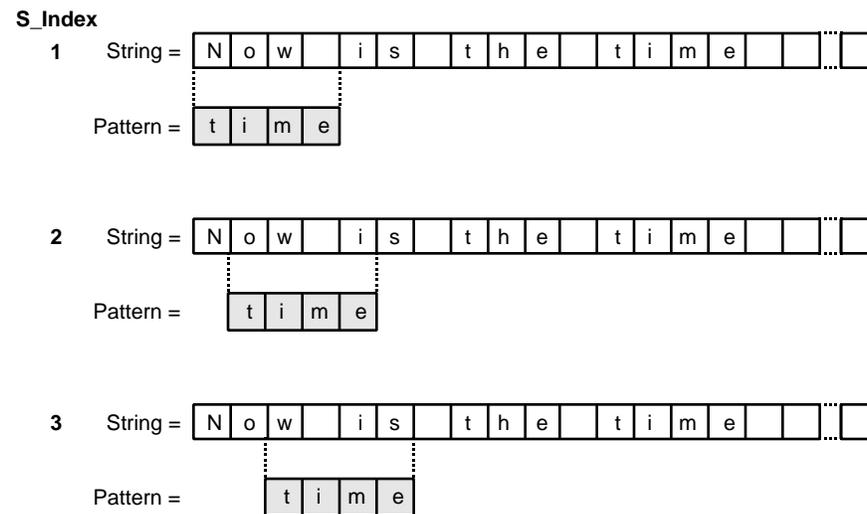
The Binary search method is considerably faster than the previous linear search algorithms. For example, searching an array of N items requires $N/2$ comparisons, on average, using a fast linear search algorithm. On the other hand, using a binary search method only requires $\log_2 N$ comparisons. This means that, for an array of 1000 items, the fast linear search makes on the average 500 comparisons, whereas the binary search makes only 10 comparisons. For arrays with very large sizes the contrast is even more significant; for a million items the linear search will make 500 000 comparisons whereas the binary search makes only 20 comparisons!

Of course, remember that nothing is free, and to achieve this speed, the binary search requires the initial array to be sorted. If a sort has to be done before the binary search, this will add quite a number of additional comparisons. But if the array, once sorted, is searched often, this binary search method is extremely efficient.

Pattern Matching

Sometimes, we need to do a different kind of searching called *Pattern Matching*. Pattern Matching is necessary when we are searching a string of characters for a particular sequence of characters. It can be accomplished by trying to match the particular pattern as we move it along the string, like in Figure 9.11.

Figure 9.11 Pattern matching



Let's look at 2 different pattern matching algorithms called *Search and Count* and *Find First Match*. The first pattern matching algorithm, *Search and Count*, is a rather simple and plodding algorithm. It examines the entire string and counts the number of times the given pattern is found. At each character in the string (except for the last characters for which the pattern would go beyond the string end) we check to see if the pattern fits.

This algorithm skeleton (Pseudocode 9.20) is extremely simple but we have yet to give the details of how the pattern comparison is done: it is performed one

character at a time. Each character of the pattern is compared to the corresponding character in the string; if a difference is discovered, the logical indicator *Found* is set to false, as there is no match. Note in Figure 9.11 how the pattern is compared to parts of the string, 4 characters at a time.

Pseudocode 9.20 Skeleton of Search and Count algorithm

```

Search and Count
  Set Count to 0
  For S_Index = 1 to (String length - Pattern length + 1) by 1
    | If Pattern fits after position S_Index
    |   Increment Count
  Output Count
End Search and Count

```

The expanded algorithm (Pseudocode 9.21) is not very efficient because it keeps comparing the characters of the pattern and the string even after setting *Found* to False.

Pseudocode 9.21 The Search and Count algorithm

```

Search and Count
  Set Count to 0
  For S_Index = 1 to (String length - Pattern length + 1) by 1
    | Set Found to True
    | For P_Index = 1 to Pattern length by 1
    |   | If Pattern[P_Index] = String[S_Index + P_Index - 1]
    |   |   Set Found to False
    |   If Found
    |     Increment Count
  Output Count
End Search and Count

```

Our second example of a pattern matching algorithm, *Find First Match*, will be faster, because it stops after finding the first match, and compares the characters of the pattern to the characters of the string only until it finds a mismatch. Once a mismatch is found, the pattern is moved forward in the string for another try. In order for this algorithm to stop when finding the first match, we will use a logical flag to indicate this condition as soon as it happens. Pseudocode 9.22 defines this new pattern matching algorithm.

Pseudocode 9.22 The Find First Match algorithm

```

Find First Match
  Set Found to False
  Set S_Index to 1
  While S_Index < (String length - Pattern length + 1) AND NOT Found
    | Check if Pattern fits
    | Increment S_Index
  If Found
    Output "Found at " S_Index
  Else
    Output "Not present"
End Find First Match

```

We still have to define the details of the actual pattern matching. We are using a solution similar to our previous algorithm, but modifying it so that the process stops as soon as a mismatch is detected. To do that, we use another logical indicator, *Equal*, to transmit the result of the pattern matching to the enclosing loop (Pseudocode 9.23).

Pseudocode 9.23 The Find First Match algorithm with detailed pattern matching

```

Find First Match
  Set Found to False
  Set S_Index to 1
  While S_Index (String length - Pattern length + 1) AND NOT Found
    Set Equal to True
    Set P_Index to 1
    While P_Index Pattern length AND Equal
      If Pattern[P_Index] String[S_Index + P_Index - 1]
        Set Equal to False
        Increment P_Index
      Set Found to Equal
      Increment S_Index
    If Found
      Output S_Index
    Else
      Output "Not present"
  End Find First Match

```

As soon as equal is false (one pair of characters does not match), we move on to next string chunk.

These two pattern matching algorithms are rather simple, but are *not* the most efficient pattern matching algorithms. There are other much faster pattern matching algorithms, such as the Boyer-Moore algorithm, the Knuth-Morris-Pratt algorithm and the Rabin-Karp algorithm. These algorithms go beyond our scope and will not be presented here.

9.5 Implementing Abstract Data Types

In Chapter 8, we defined and used Abstract Data Types without giving any information about their implementation. In order to run programs involving ADTs, we need to learn how to implement them. The pattern matching algorithms we just saw could be part of the implementation of the String ADT, which will not be covered here.

Stacks

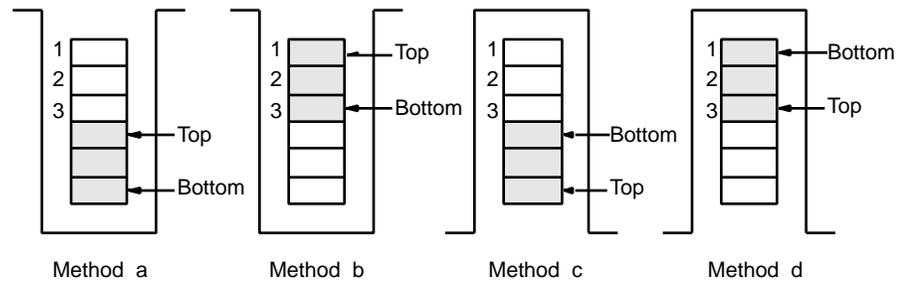
Stacks were introduced in Chapter 8 as an abstract data type with the following operations:

- Create a stack.
- Push an item onto a stack: the new stack top contains the pushed item.
- Pop an item from a stack: the result is the top stack item, which is deleted from the stack.
- Check if a stack is empty.
- Check if a stack is full.
- Count the number of elements in a stack.

Stacks can be implemented in many ways. Here we will look at two implementations, one using arrays, and the other using pointers.

Implementing Stacks with Arrays

Figure 9.12 Four ways of implementing stacks as arrays

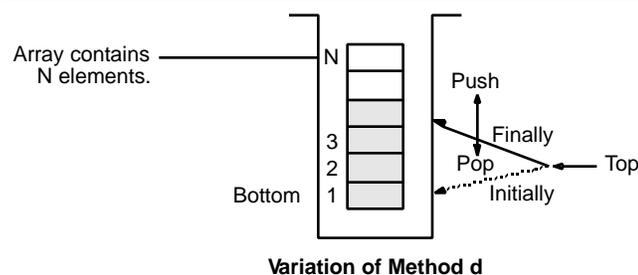


We can use arrays to implement stacks and this can be done in a number of different ways as shown in Figure 9.12, where the bottom and top of the stacks are shown as well as the indices of the array. Notice how the indexing changes from method to method. Each of the following four methods is analogous to a physical situation

- *Method a* corresponds to books in a box; the first book put in goes to the bottom and the top changes as books are placed on the stack.
- *Method b* corresponds to stacks of plates in some restaurants, where the top plate is at the counter level, and the bottom plate is moved up by a spring whenever the top plate is taken.
- *Method c* corresponds to a number of drinking cups in a dispenser where the “top” cup is at the bottom, and as it is taken all the cups move down one.
- *Method d* corresponds to lighter than air balloons in a vertical tube, with the “bottom” balloon floating at the ceiling.

Selection of one of these stack implementations is made easy if you realize that Methods b and c require all the items in a stack to be moved when the top is pushed or popped. Methods a and d are better candidates for our implementation; they do not involve such inefficient moving of the stack elements. We might choose the first way, Method a, to implement our stacks, for it seems more “natural” in that the stack Top can be represented at the top of the page. On the other hand, it would seem more natural to number the array elements from the bottom of the stack as Method d does. So we’ll choose *Method d* but we will reverse it in the diagram as shown in Figure 9.13.

Figure 9.13 One stack implementation arrays



Our choice consists of an array with N elements, and a variable, *Top*, which indicates the stack top. Because of the structure we chose, the implementation of the various stack operations is simple. First, we create an empty stack as shown by the following pseudocode.

Pseudocode 9.24 Create Stack using arrays

```

Create Stack
  Set Top to 0
End Create Stack

```

Checking whether or not a stack is full is easily done by checking if the stack top is at the last element in the array.

Pseudocode 9.25 Stack Full function using arrays

```

Stack Full function
  If Top = N
    Return True
  Else
    Return False
End Stack Full

```

To check if the stack is empty, we check to see if the stack *Top* is zero.

Pseudocode 9.26 Stack Empty function using arrays

```

Stack Empty function
  If Top = 0
    Return True
  Else
    Return False
End Stack Empty

```

We cannot push an item onto a full stack. If the stack is not full, the *Top* of the stack moves up one position and the item is copied into the next position.

Pseudocode 9.27 Push on Stack algorithm using arrays

```

Push on Stack(Item)
  If Stack Full
    Output "The stack is full"
  Else
    Increment Top
    Set Stack[Top] to Item
End Push on Stack

```

We cannot pop an item from an empty stack. If the stack is not empty, the *Top* of the stack is copied into *Item*, and the *Top* is changed to point to the next item on the stack.

Pseudocode 9.28 Pop from Stack algorithm using arrays

```

Pop from Stack(Item)
  If Stack Empty
    Output "The stack is empty"
  Else
    Set Item to Stack[Top]
    Decrement Top
End Pop from Stack

```

As always, remember that there are many different ways of doing things and the implementation of stacks is no different. The implementation we have just seen has the disadvantage that the stack size must be fixed once and for all, when we choose the array to represent our stack. Each time an application needs more space in its stack, we must redefine the array used in the implementation.

Implementing Stacks with Pointers

To solve the problem of the fixed stack size we just mentioned, we can abandon the array implementation of stacks and use dynamic variables instead. In fact, we can use the dynamic list representation that was introduced in Chapter 8 (Figure 8.42) to implement our stack abstract data type. In that case, a stack will be represented as shown in Figure 9.14

The creation of a stack will be done easily by giving the stack pointer a NIL pointer value (pointer pointing nowhere), indicating that there are no elements in the stack. Its algorithm is shown in Pseudocode 9.29.

Pseudocode 9.29 Algorithm for creating a stack

```

Create Stack
  Set Stack to NIL
End Create Stack

```

"Stack" is the name of the pointer to the top of the list

Notice that we are creating a stack in the opposite direction from the creation of a dynamic list in Chapter 8, (see Figure 8.46).

Checking whether a stack is empty is easy, as shown in Pseudocode 9.30.

Pseudocode 9.30 Stack Empty function using pointers

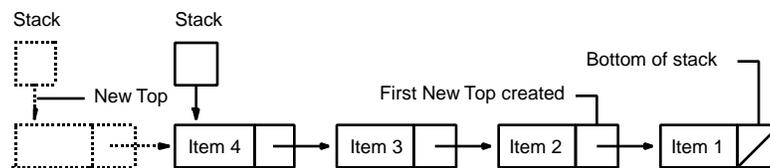
```

Stack Empty function
  If Stack = NIL
    Return True
  Else
    Return False
End Stack Empty

```

Checking whether a stack is full cannot be done as it was done with the array implementation. If a call to *New*, which creates a new element, fails, then we will know all the memory has been used up. This is equivalent to having a full stack.

Figure 9.14 A dynamic stack



The pointer variable *Stack* will always point to the top element of the stack. Our Push and Pop operations will be redefined by Pseudocode 9.31 and 9.32.

Pseudocode 9.31 Redefinition of Push on Stack algorithm

```

Push on Stack(Item)
  Set New Top to New(Element)
  If memory allocated
    Set New Top Information to Item
    Set New Top Next to Stack
    Set Stack to New Top
  Else
    Output "No more memory"
End Push on Stack

```

Before pushing a new element onto the stack, we need to create that element. This is done by a call to *New*. If this memory allocation is successful, we copy the information into the new element. Remember we use *New Top*-> to refer to

the dynamic variable indicated by pointer *New Top*. We connect the new element to the top element of the stack through its *Next* pointer field, and we change the value of *Stack* to indicate this new element as the top element. Try to follow this algorithm as you add Item 5 to the stack of Figure 9.14.

Note: The dashed items in Figure 9.14 were added to help you visualize the “Push” operation.

Pseudocode 9.32 Redefinition of Pop from Stack algorithm

```

Pop from Stack(Item)
  If Stack Empty
    Output "The stack is empty"
  Else
    Set Item to Stack      Information
    Set Old Top to Stack
    Set Stack to Stack     Next
    Dispose(Old Top)
  End Pop from Stack

```

We cannot pop an item from an empty stack, so we must check for that condition. If the stack is not empty, the top element of the stack is copied into *Item*, and *Stack* is changed to point to the next element on the stack, while the old top element is freed (*Dispose* is the inverse operation of *New*). Try to follow this algorithm as you delete Item 4 from the stack in Figure 9.14.

Queues

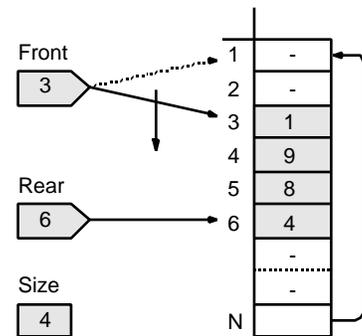
Queues were introduced in Chapter 8 as an ADT with the following operations:

- Create a queue.
- Check if a queue is empty.
- Check if a queue is full.
- Count the elements in a queue.
- Enter an element into a queue.
- Remove an element from a queue.

As was the case with the stack, there are many ways of implementing queues. We will show only two here: one based on the use of an array, and another one based on dynamic elements.

Implementing Queues of Arrays

Figure 9.15 shows a queue implementation using an array to store the queue elements. Two variables *Front* and *Rear* indicate the array position of the front and rear elements. Another variable *Size* indicates the number of elements in the queue

Figure 9.15 A queue implementation

As items are entered in the queue, the rear indicator advances, and as items are removed from the queue the front indicator also advances, leaving behind some “used” values. So the queue “snakes” through the array. As each indicator passes the last item of the array it continues to the first item, thus creating a “circular array”.

The algorithm to create a queue initially sets both *Front* and *Rear* variables to the first index, and sets the *Size* of the queue to zero, as shown below.

Pseudocode 9.33 Create Queue algorithm using arrays

```

Creat Queue
  Set Front to 1
  Set Rear to 1
  Set Size to 0
End Create Queue

```

The functions to check whether the queue is empty or the queue is full are obvious, as illustrated by Pseudocode 9.34.

Pseudocode 9.34 Queue Empty and Queue Full using arrays

<pre> Queue Empty function If Size = 0 Return True Else Return False End Queue Empty </pre>	<pre> Queue Full function If Size = N Return True Else Return False End Queue Full </pre>
---	---

The function to count the elements in a queue is trivial since the *Size* variable tells us how many there are.

Pseudocode 9.35 Count Queue function using arrays

```

Count Queue function
  Return Size
End Count Queue

```

To implement the *Enter* and *Remove* operations, we need a small utility algorithm to advance the front and rear indicators in a cyclic manner in the array. *Advance* index increases the index by 1, but if the new index value goes beyond the end of the array, it is reset to one. The pseudocode for *Enter Queue* is self-explanatory and is shown below.

Pseudocode 9.36 Enter Queue algorithm using arrays

```

Enter Queue(Item)
  If Queue is full
    Output "The queue is full"
  Else
    Advance Rear
    Set Queue[Rear] to Item
    Increment Size
  End Enter Queue

```

The pseudocode for *Remove from Queue* is also self-explanatory.

Pseudocode 9.37 Remove Queue algorithm using arrays

```

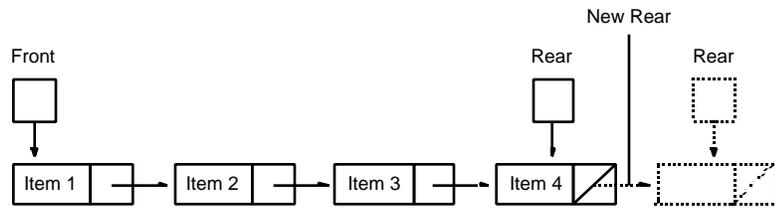
Remove from Queue(Item)
  If Queue is empty
    Output "The queue is empty"
  Else
    Set Item to Queue[Front]
    Advance Front
    Decrement Size
  End Remove from Queue

```

Implementing Queues with Pointers

The second queue implementation we will briefly present here is based on dynamic variables. We will use a data structure based on a dynamic list, as shown in Figure 9.16

Figure 9.16 A dynamic queue



The variables *Front* and *Rear* are pointer variables and the elements in the queue are dynamic variables. With this representation for queues, the creation operation is simple and just sets the *Front* and *Rear* pointers to NIL pointers. The *Enter Queue* algorithm is defined by Pseudocode 9.38. Look at the dashed right-hand-side of Figure 9.16 to get an idea of the operations involved.

Pseudocode 9.38 Enter Queue algorithm using pointers

```

Enter Queue(Item)
  Set New Rear to New(Element)
  If memory allocated
    Set Rear Next to New Rear
    Set New Rear Information to Item
    Set New Rear Next to NIL
    Set Rear to New Rear
    Increment Size
  Else
    Output "No more memory"
  End Enter Queue

```

Follow this algorithm while adding Item 5 to the queue of Figure 9.16.

The *Remove from Queue* algorithm is very similar, and is shown below. Apply this algorithm to the queue in Figure 9.16.

Pseudocode 9.39 Remove from Queue algorithm using pointers

```

Remove from Queue(Item)
  If Queue Empty
    Output "The queue is empty"
  Else
    Set Item to Front      Information
    Set Old Front to Front
    Set Front to Front      Next
    Dispose(Old Front)
    Decrement Size
  End Remove from Queue

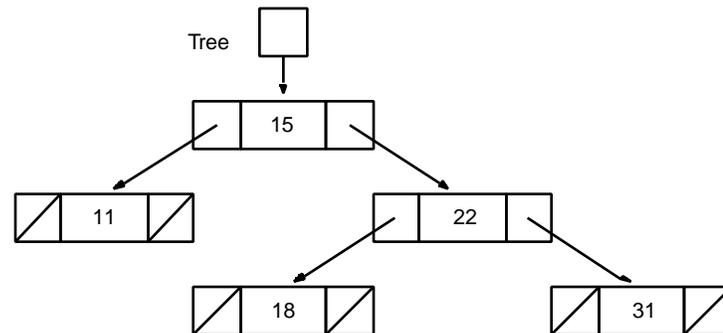
```

The implementation of the other operations is easily done as it was for the stacks, and we leave it as an exercise.

Trees

We have seen an example of the nonlinear data structure Tree in Chapter 8. Although it is possible to represent trees using arrays, a more natural implementation of trees uses pointers. Such a representation is illustrated in Figure 9.17, which presents a *binary search tree*

Figure 9.17 A binary search tree



In this type of representation, a binary tree element is made of three parts: an Information part, and two pointer parts indicating the Left and Right descendants. Figure 9.17 represents a special kind of binary tree: a binary search tree. At each node in a binary search tree, all the elements in the left sub-tree have values that are less than the value of the node, while all the elements in the right sub-tree have values greater than the node value. Binary search trees are very useful in a number of specific applications, where searching operations are used very often.

The operations that can be applied to trees include inserting a node, deleting a node, searching a tree for a given value, and traversing a tree to process all its nodes.

Let's illustrate such operations with a tree traversal operation. Trees can be defined recursively (a tree is either empty or a node with a number of sub-trees) and most tree algorithms are naturally expressed using recursion. The *Tree Traversal* algorithm can be defined by Pseudocode 9.40.

Pseudocode 9.40 The Tree Traversal algorithm

```
Tree Traversal(Tree)
  If Tree is not empty
    | Tree Traversal(Tree Left)
    | Output Node information
    | Tree Traversal(Tree Right)
  End Tree Traversal
```

Let's trace this algorithm using the tree in Figure 9.17.

We start at the root and process its Left sub-tree (*Tree*->*Left*). This will produce the value 11. Next, we output the value of the root, 15. Then we process the Right sub-tree of the root (*Tree*->*Right*) by recursively invoking the *Tree Traversal* algorithm, starting at the node with value 22. There we again process that node's left sub-tree, giving the value 18. The root value of the sub-tree, 22, is then output and, finally, we output the value of the right sub-node of the sub-tree's root, 31. Thus, the values were output in ascending order, 11, 15, 18, 22, 31.

The *Search Tree* algorithm will have a similar pattern, where we start inspecting the root, and if it is not what we are looking for, we invoke *Search Tree* recursively, first on the root's Left sub-tree, and then on its Right sub-tree. The base case remains the empty tree. Recursive algorithms are very natural for tree structures, and make the manipulation of trees quite easy.

9.6 Review Top Ten Things to Remember

1. The algorithms considered in this chapter show the many ways in which the data structures described in Chapter 8 can be manipulated and give some idea of their applications.
2. One of the most common operations on arrays is sorting and searching. There are three fundamental *strategies* for sorting:
 - *sort and copy*: this strategy sorts an array while copying it to another array.
 - *sort and rank*: this strategy sorts an array and stores the rank of its elements in another array.
 - *sort in place*: this strategy sorts an array without copying it to another array (the best as far as space complexity goes).
3. Within the three fundamental strategies, there are many different algorithms for sorting. These sorting methods can be divided into four *categories*:
 - Count Sort, where comparing and counting occur;
 - Swap Sort, where pairs of items are swapped;
 - Select Sort, where extreme values are selected;
 - Insert Sort, where moving and inserting of values occur.
4. Recursion can be used to sort arrays as shown by the *Merge Sort* algorithm (recursive subprograms were introduced in Chapter 7).
5. Another very common operation on arrays is searching. Searching arrays is usually done in two basic ways:
 - a *Linear Search*; examines elements sequentially (one at a time)
 - a *Binary Search* reduces the number of searched items by half at each comparison (but the search array must be sorted).
6. Pattern Matching is another search method which is used to search a string of characters for a particular pattern of characters. Two variations of a simple pattern matching algorithm were introduced in this chapter:

7. The measure of the performance of various algorithms has been introduced briefly. The *time complexity* of an algorithm is usually indicated by the “big-O” notation. Some common complexities and examples are shown below, beginning with the higher complexities (slower algorithms).

Order	Order Type	Example algorithms
$O(2^n)$	Exponential	Traveling salesman problem ¹
$O(n^2)$	Quadratic	Select Sort
$O(n \log n)$	$N \log N$ (Entropic)	Merge Sort
$O(n)$	Linear	Sum, Reverse, Linear search
$O(\log n)$	Logarithmic	Binary search
$O(1)$	Constant	Factorial

8. Another way of measuring algorithms is by considering *space efficiency*: the amount of memory required when executing an algorithm.
9. The implementation of abstract data types has been illustrated first based on arrays, and then on dynamic variables for Stacks and Queues.
10. The implementation of tree data structures is best done with dynamic variables.

¹ The problem is to find the shortest route by which a salesman can visit a particular set of cities (in any order) given the distance between all pairs of cities.

9.7 Glossary

Execution time: The time that it takes an algorithm to execute.

In situ: In place, applied to array manipulations that place the results in the original array, without the need for duplicating the array..

Order of complexity: A measure of the performance of an algorithm, which shows how the number of operations required to execute the algorithm is related to the size of the problem being solved by the algorithm.

Rank: In an array, the rank of an element is the number of array elements that are of a greater than or equal to it.

Space efficiency: A measure of the amount of space required by an algorithm during execution.

Time complexity: An expression that gives an estimate of the execution time of an algorithm

9.8 Problems

1. Speedy Sort of Binary Values

Given an array B of N binary values (0s and 1s only), create an algorithm to sort this array using the least number of passes.

Do this four different ways corresponding to the four methods of sorting.

2. Median: MidArray

The middle or median value of an array of an odd number of values, N, is that value which has as many values less than it as are greater than it. Create an algorithm to find this Mid value.

Do it in a least two other ways.

More Problems: Manipulations of Linear Lists

3. Slow-Sort

Given an array A of N integers (not necessarily different) ranging from 0 to 1000, create an algorithm to sort the values by checking (in increased order) if each of the 1000 integers is in the array and if so, by outputting the value. Compute the number of comparisons required for this sort, and compare it to some of the other sorts.

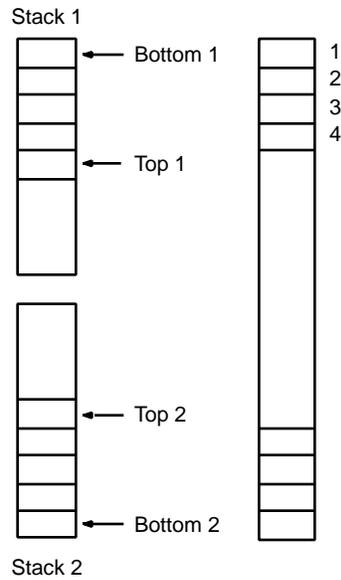
4. Quick Queue

Create a queue and its operations EnterQ and ExitQ, using the already created Stack and its operations.

5. Double Stacks

Two stacks can be implemented by a single array, with the stacks “growing” toward each other as shown below. Create a general algorithm for pushing (Push1 and Push2) and popping (Pop1 and Pop2) the appropriate stacks.

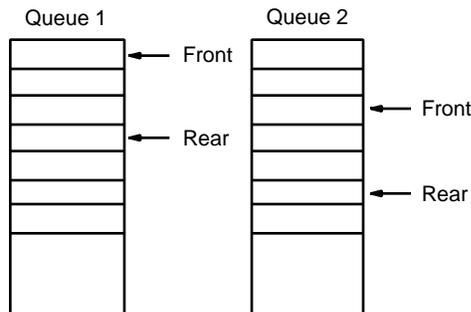
Problem 5



6. More Queues

The given diagrams show two more implementations of a queue, using arrays. The first queue Q1 always has its Head at position one of the array, and on ExitQ all values are moved up. The second queue Q2 moves “snake wise” down the queue (like the circular queue), but when the Rear hits the bottom of the array, then all entries are shifted up so that the Front is at the top again. Create programs to Enter and Exit these queues. Compare the two queue implementations briefly.

Problem 6



7. Insert Sort

Create a sorting algorithm which enters new values into an array by moving existing values to make space for the new value.

8. Merge

Create an algorithm to merge two already sorted arrays into one larger sorted array. Do this in two ways.

9. Recursive Searches

Given an array of N items, create a recursive algorithm to search the array. Do this as a linear search and also as a binary search.

10. Other Tree Traversals

Using the following recursive algorithms, traverse the tree of Figure 9.17.

Problem 10

```
Tree Traversal 1(Tree)
  If Tree is not empty
    | Tree Traversal 1(Tree  Right)
    | Tree Traversal 1(Tree  Left)
    | Output Node information
  End Tree Traversal 1
```

```
Tree Traversal 2(Tree)
  If Tree is not empty
    | Tree Traversal 2(Tree  Right)
    | Output Node information
    | Tree Traversal 2(Tree  Left)
  End Tree Traversal 2
```

```
Tree Traversal 3(Tree)
  If Tree is not empty
    | Output Node information
    | Tree Traversal 3(Tree  Left)
    | Tree Traversal 3(Tree  Right)
  End Tree Traversal 3
```

Chapter 10 The Seven-Step Method

This chapter returns to the seven step problem solving method and reviews it, using the concepts that you learned in the previous nine chapters. The chapter then develops a complete case study, a program to create a text index, which integrates all the concepts covered previously.

Chapter Outline

10.1	Preview.....	458
10.2	The Seven-Step Method and Applications.....	458
	Step 1 Problem Definition.....	458
	Problem Definition Application.....	460
	Step 2 Solution Design.....	461
	Solution Design Application.....	461
	Step 3 Solution Refinement.....	464
	Solution Refinement Application.....	464
	Step 4 Testing Strategy Development.....	466
	Development of Testing Strategy Application....	468
	Step 5 Program Coding and Testing.....	471
	Step 6 Documentation Completion.....	471
	Documentation Completion Application.....	472
	Step 7 Program Maintenance.....	474
	Program Maintenance Application.....	474
10.3	An Advanced Case Study Building a Text Index.....	476
	Step 1 Problem Definition.....	476
	Step 2 Solution Design.....	476
	Step 3 Solution Refinement.....	478
	Binary Search Tree Unit.....	483
	Queues Unit.....	486
	Step 4 Development of Testing Strategy.....	486
	Step 5 Program Coding and Testing.....	488
	Step 6 Documentation Completion.....	488
	Step 7 Program Maintenance.....	489
10.4	Review Top Ten Things to Remember.....	490
10.5	Glossary.....	491
10.6	Problems.....	492
	Level 1 — Getting Started.....	492
	Level 2 — Getting Organized with Subprograms.....	495
	Level 3 — Getting Fancier with Parameters.....	498
	Level 4 — Getting Your Wings with Units.....	501

10.1 Preview

Up until now, we have shown you that programming involves the following:

- A problem-solving method that can be described by the seven steps introduced in Chapter 2. These steps are:
 1. Problem Definition
 2. Solution Design
 3. Solution Refinement
 4. Testing Strategy Development
 5. Program Coding and Testing
 6. Documentation Completion
 7. Program Maintenance

Steps 1 to 4 are usually part of the design stage, and steps 5 to 7 are part of the implementation stage. This problem-solving method is closely related to the software life cycle (the various stages in the life of a software package.)

- Algorithms that can be represented in many forms. They all consist of a precise set of instructions to follow to solve a problem.
- Data structures that are the means of representing the data used in the algorithms. They must be developed at the same time as the algorithm.

In this chapter, we will explain in greater detail what each of these seven steps involves. At the same time, we will present a complete case study of a payroll system, to show you how to use each of these steps to develop algorithms and data structures.

10.2 The Seven-Step Method and Applications

Step 1 Problem Definition

Programs are written so that computers can solve problems posed by humans. When faced with having to write a program, you have one thing: a description of the problem to solve. This description may be very precise or vague, but nevertheless is present.

The first thing to do, is to make sure that you understand the problem. As Albert Einstein once said, “If you can’t explain something to a six-year-old, you really don’t understand it yourself.” If you are in this state, you must examine more closely the imprecise parts of the problem that you do not understand.

For instance, if your teacher asks you to write a program to “Find the average of five grades for each of my students”, it might at first sound simple. But, you should ask yourself:

- “What does average mean, exactly?”
- “Are the grades in percentages or letters?”
- “Do some grades count more than others so that we have to do a weighted average?”

Asking yourself such questions forces you to define the problem very precisely.

Once you are sure of what the problem entails, you must jot down a list of specifications. *Specifications* are precise definitions of what the program must do. At a minimum, they must include the following:

- Input: what data must be input and in what form
- Output: what data must the program produce and in what form (in order to solve the problem)

Virtually all computerized solutions have the structure shown in Figure 10.1.

Figure 10.1 A typical computerized process



Let's take the problem of averaging student grades described above. We could generate this list of specifications:

- Input: student number, followed by student name, followed on the *second* line by five Real Numbers representing the grades in percentages
 Sample: 666 Lucifer El Diablo
 20.5 66.6 75.0 70.9 100.0
- Output: student number, followed on the next line by one Real Number for the average and one Character for the corresponding letter grade
 Sample: 666
 66.6 % C
- Process: for each student, the grades will be summed and averaged, and this average converted to a letter according to a predetermined scale.

Note: At the end of the Problem Definition step, you should have a list of specifications.

Problem Definition Application

To better understand this seven-step method, we will show how each step applies to the following real-life problem.

“Computerize the payroll of the ACME Company.”

This is understandably too vague and our first task is to be more precise.

Up until now, the ACME Company had an archaic system using index cards and hand calculators for handling the payroll. We gather information on the existing system, after consulting with the payroll department, and we establish the following:

- We must compute the pay of a sequence of hourly paid employees based on a line of input data for each employee.
- The input data will be kept in a separate file.
- Each line of data will consist of three integers followed by a character string. These data correspond respectively to the number of hours worked (stored in 1/100ths of an hour), the hourly rate (in cents), the number of dependents (for tax purposes), and the employee name, and will have the format:
 3050 1025 8 Gabrotil Michael
- The end of the data will be indicated by the end-of-file marker provided by the system.

We have already seen some simple pay algorithms in chapters 2, 3, 4, 6 and 7.

- The computation of the gross pay is done by multiplying the hours worked by the hourly rate.
- Hours above 40 are to be paid one and a half times the normal hourly rate.
- For each dependent, the employee gets an exemption of \$20 for tax withholding computation.
- Federal and state withholdings are computed by applying rates of 18% and 3%, respectively, to the taxable amount.
- Social security withholdings are computed by applying a 5% rate to the gross pay.
- Net pay is computed by subtracting the various withholdings from the gross pay.
- Results must be displayed on one line per employee: name, followed by gross pay, federal withholdings, state withholdings, social security withholdings, and net pay in dollars and cents, using the format:

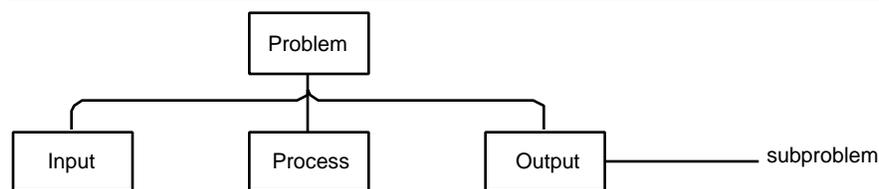
```
Gabrotil Michael 312.63 27.47 4.58 15.63 264.95
```
- After processing all of the employees, a summary line with the totals of the various withholdings and pay categories should be displayed, using a similar format, so that the results are aligned with the preceding individual columns.
- Input data must be validated before they are used. The following validity ranges should be used:
 - The number of hours worked cannot be negative or greater than 55.
 - The hourly rate cannot be less than \$3.50 or more than \$16.50.
 - The number of dependents cannot be negative or greater than 12.

Step 2 Solution Design

In this step, we break our problem down into a number of smaller, more manageable parts. We must analyze the original problem, and divide it into a number of sub-problems. Because these sub-problems are necessarily smaller than the original problem, they are easier to solve and their solutions will be the components of our final program. Each sub-problem is itself divided into smaller sub-problems, and this decomposition is carried on until we have sub-problems whose solutions are simple.

If we use a solution structure like the one in Figure 10.1, we can readily decompose the problem into three sub-problems: input, process, and output. We can represent this decomposition by a structure chart. As we have seen in Chapter 4, such a method is called top-down design, and leads to an outline of the solution, as in Figure 10.2. Note that you can use break-out diagrams instead, if you are more comfortable with them.

Figure 10.2 Structure chart



This outline will help us write the algorithm since each of the boxes in the structure chart will typically be implemented as a sub-algorithm.

Note: At the end of the Solution Design step, you should have a structure chart describing the hierarchy of your algorithm.

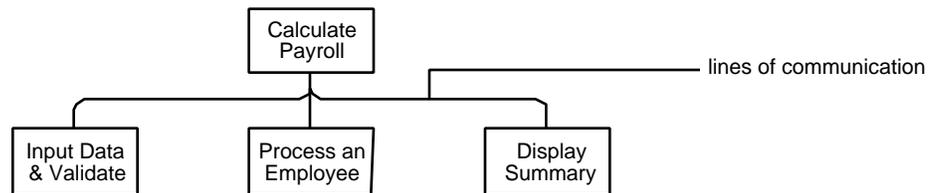
Solution Design Application

To illustrate this design process a little more, let's continue to develop the payroll system for the ACME Company. We must define all the tasks that have to be done in order to produce the payroll. This first level of decomposition is simple. The system must be able to do the following:

- Read in and validate pay data for all the employees,
- Process the data for each employee, and
- Display a summary of the payroll operation.

This leads us to the structure chart of Figure 10.3.

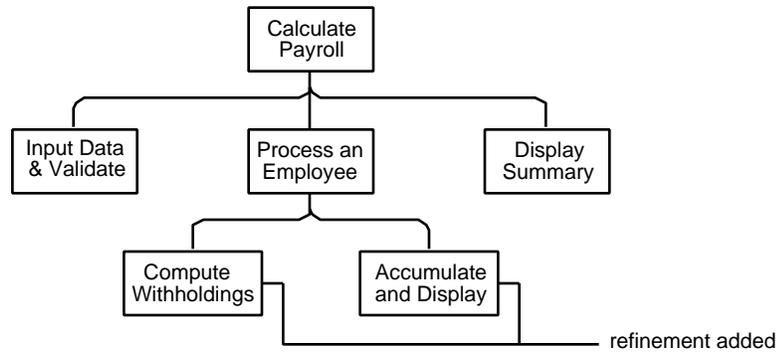
Figure 10.3 Structure chart for payroll system



In this structure chart, the left-to-right order indicates a probable order of execution, but this is not always true because of unseen repetitions and condition testing. There will be some communication between the components along the lines in the structure chart, so we must also develop interfaces for the data that will be transmitted between these components, as we have done in Chapter 7. The design of Figure 10.3 can still be refined by subdividing each of the lower-level components into their major components, if this is possible. For instance, "Process an Employee" might be subdivided further into two tasks, as shown in Figure 10.4:

- "Compute Withholdings" calculates the federal and state taxes and the social security tax.
- "Accumulate and Display" adds the various withholdings to the running totals for all employees, and then displays the results for the current employee.

You might recall that this way of doing things is sometimes called stepwise refinement.

Figure 10.4 Three-level structure chart for payroll system

See Chapter 7 for more information on developing interfaces via parameter use.

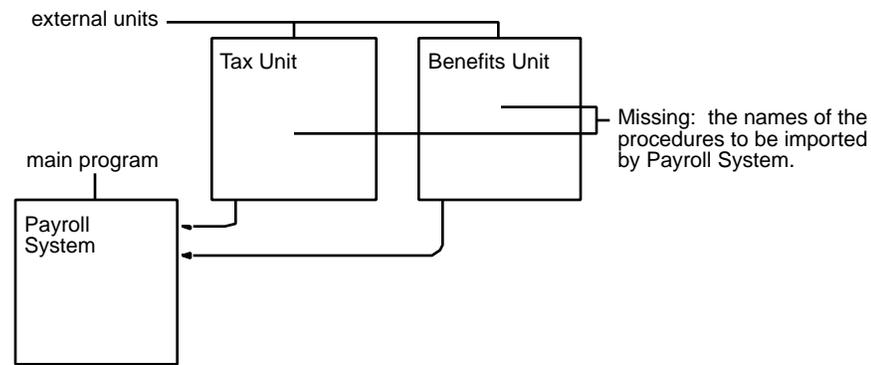
A structure chart is actually a skeleton of the structure of the final program. Each box in the structure chart will be implemented as a *procedure* or a *function*. For example, the box “Input Data & Validate” will be implemented as a procedure later on. This should not be surprising, because the structure chart is the result of breaking down the solution function by function.

In addition to decomposing the problem into its functional components, you should also try to identify *groups* of related operations that could be used throughout the program to make implementation easier. For instance, if a group of operations all dealt with a certain kind of data structure, then it might be desirable to include them in a separate unit—also referred to as an *external unit*.

For example, a program that deals with complex numbers could use a separate unit that defines a complex abstract data type. Such separate units are called “external” because they are physically external to the main application. When the main program needs to use part of the external unit, this must be specified in the main program. Units are sometimes called *Libraries*, *Modules*, or *Packages*.

Our payroll problem has been greatly simplified and we do not anticipate the need for external units. If the payroll application were more realistic and were to process thousands of employees, then we could envision a need for separate units to process taxes and to deal with all the company benefits (health insurance, pension plan, etc.).

These units could then be documented by a modular design chart like the one in Figure 10.5. This chart shows the various interconnections of the units we intend to use for this solution. The arrows show what elements are *imported* from one unit for use in another unit. In Figure 10.5, “Payroll System” imports elements from units “Tax” and “Benefits”. The figure should be completed by indicating which procedures from the “Tax” unit or the “Benefits” unit will be used by the “Payroll System”. If we were designing a complete payroll system, we would be able to complete the chart once the solution has been refined in the next step.

Figure 10.5 Modular design chart for a more realistic payroll system

In this structural design step, various alternatives for doing the decomposition must be considered, and the relative advantages and disadvantages of each alternative must be weighed. Initially, if you are a beginner, you might find it hard to judge the advantages and disadvantages of a given solution, but this will become easier with experience.

Step 3 Solution Refinement

Now that we have the skeleton of our solution, we can start putting some meat on its bones. As previously stated, each box in the structure chart corresponds to a sub-problem. We are now ready to develop one algorithm to solve each sub-problem as well as the main problem.

As you have seen so far, algorithms can be represented in many different ways: by flowblocks, flowcharts, data-flow diagrams, etc. Pseudocode is the most common type of algorithm representation developed. The advantage of using pseudocode is that it is programming language independent. It can easily be translated into most programming languages, particularly imperative languages such as Pascal, Modula-2 or C.

Since we are now faced with the task of designing the actual algorithms, we must at the same time decide which types of data structures to use. It is important to realize that data structure selection and algorithm design are directly related. If you change your type of data structure, you must accordingly change your algorithm. This is why both must be developed concurrently.

Note: At the end of the Solution Refinement step, you should have developed pseudocode for each box of the structure chart. As well, all of the data structures and variables used must be defined.

Solution Refinement Application

At this stage of design we need only define data types for the variables used for (i) input, (ii) major processing, and (iii) output. Temporary variables and counters need not be specified at this time.

For our payroll program, we will define a record variable, *Employee Data*, with fields *Name*, of type String, and *Hours*, *Rate* and *Dependents* of type Integer. We will also specify simple variables like *Gross Pay*, *State Tax*, *Federal Tax*, *Soc Sec Tax*, *Net Pay*, *Gross Total*, *Soc Sec Total*, *Federal Total*, *State Total*, and *Net Total*. All these simple variables are Integers, because computations involving money must be exact, and only integer

arithmetic gives exact results. We will also need a Boolean variable *Valid Data* to indicate the result of the data validation.

Using these simple variables, we can *begin* to refine the high-level solution defined by the structure chart of Figure 10.4. Using pseudocode, we develop the algorithms for each functional part of the solution, starting with the main program: *Calculate Payroll*.

Pseudocode 10.1 The Calculate Payroll algorithm

```

Calculate Payroll
  Display title and column headings
  Set all totals to zero
  While there are data to read
    Input Employee Data.Hours
    Input Employee Data.Rate
    Input Employee Data.Dependents
    Input Employee Data.Name
    Input Data And Validate(Employee Data, Valid Data)
    If Valid Data
      Process Employee(Employee Data, Totals)
    Display Summary(Totals)
  End Calculate Payroll

```

In order to simplify the pseudocode, we will use collective names to represent groups of variables, like *Totals* to represent the following five variables: *Gross Total*, *Soc Sec Total*, *Federal Total*, *State Total*, and *Net Total*. This makes it possible to have shorter argument lists in our invocations. The subprograms *Input Data & Validate* and *Process Employee* are shown in Pseudocode 10.2 and 10.3.

Pseudocode 10.2 The Input Data & Validate sub-algorithm

```

Input Data And Validate(Employee Data, Valid Data)
  Set Valid Data to True
  If (Employee Data.Hours < 0) OR (Employee Data.Hours > 55)
    Set Valid Data to False
    Output "Invalid hours", Employee Data.Name
  If (Employee Data.Rate < 3.50) OR Employee Data.Rate > 16.50)
    Set Valid Data to False
    Output "Invalid rate", Employee Data.Name
  If (Employee Data.Dependents < 0) OR
    (Employee Data.Dependents > 12)
    Set Valid Data to False
    Output "Invalid dependents", Employee Data.Name
  End Input Data And Validate

```

Pseudocode 10.3 The Process Employee sub-algorithm

```

Process Employee(Employee Data, Totals)
  Set Gross Pay to Hours × Rate
  If Hours > 40
    Add Overtime Bonus to Gross Pay
  Compute Withholdings(Gross Pay, Employee Data, Withholdings)
  Set Net Pay to Gross Pay – Withholdings
  Accumulate and Display(Employee Data, Pay Data, Totals)
  End Process Employee

```

Here, the collective name *Withholdings* has been used to represent the three variables *Federal Tax*, *State Tax*, *Soc Sec Tax*.

Pseudocode 10.4 The Compute Withholdings sub-algorithm

```

Compute Withholdings(Gross Pay, Employee Data, Withholdings)
  Set Taxable to Gross Pay – Employee Data.Dependents ×
                                                    Exemption
  If Taxable > 0
    Compute Federal Tax and State Tax
  Else
    Set Federal Tax and State Tax to 0
  Compute Soc Sec Tax
End Compute Withholdings

```

Here we have used the collective name *Pay Data* to represent the four variables *Net Pay*, *Federal Tax*, *State Tax*, *Soc Sec Tax*.

Pseudocode 10.5 The Accumulate and Display sub-algorithm

```

Accumulate and Display(Employee Data, Pay Data, Totals)
  Update all payroll totals
  Output Employee Data.Name, Gross Pay, Federal Tax, State Tax,
    Soc Sec Tax, Net Pay
End Accumulate and Display

Display Summary(Totals)
  Output "Totals", Gross Total, Federal Total, State Total,
    Soc Sec Total, Net Total
End Display Summary

```

Step 4 Testing Strategy Development

We have now completed the pseudocode for each part of our program. You might think that the next step consists of translating this pseudocode into code. Wrong!

At this point, you need to check your design to see if it will produce the expected results. It is much easier to check the design now, when we can easily make the necessary corrections, than later, when the entire code is written. To check your design, you need to think up a *testing strategy*. This strategy will be used for two things: (i) to check your design now, and (ii) once you decide upon a final design and write the code, to check if the final program works correctly.

The advantage of deciding on a testing strategy now, as opposed to after coding is completed, is that you are still at the design stage where the specifications are fresh in your mind. The testing can then be planned with those specifications in mind, from a relatively objective viewpoint. If changes need to be done, it will not be as painful as if the whole program were already written. If the entire program were written, you would be more prone to trying little fixes to solve the problems rather than thinking about changing the underlying design.

- For smaller programs, it is usually enough to define a variety of test cases, each case including the input data and the corresponding expected results. These test cases must include *extreme* cases and *erroneous* cases. For instance, test cases for a grades program should include a negative grade, an erroneous value, to make sure that the appropriate error message is produced. A procedure involving a number of data elements should be tested with zero or the maximum allowed number. Test data should be included to make sure that all the program statements, without exception, are executed at least once.

- For large programs, the testing and the coding must be planned together, piece by piece. The approach chosen may be top-down, bottom-up, or a combination of both.
 - *Top-down testing* means to start developing the program component at the top of the structure chart and working down component by component, thus the name “top-down.” In order to be able to do this, we must use program *stubs* for the lower level components in the structure chart during the initial steps of development. A program stub is a small piece of program that substitutes for a larger piece that will be written later. It may simply leave a trace of its execution by printing a message or it may also supply predefined results.

For example, the structure chart given in Figure 10.4 shows that the main program component, *Calculate Payroll*, will have three major sub-components: *Input Data & Validate*, *Process Employee*, and *Display Summary*. We could start a top-down development by writing the main program and *one* of these three major procedures, but the other two procedures and the lower level procedures could merely be stubs.

In pseudocode, a stub for the *Display Summary* procedure might be

Pseudocode 10.6 Preliminary Display Summary sub-algorithm

```
Display Summary(Totals)
  Output "Display summary"
End Display Summary
```

A message indicating that the procedure has been called is displayed. This stub would later be replaced by a procedure that actually does the desired processing, but meanwhile the program can be run to make sure that other procedures are called in the right order, and that those procedure calls are correct.

- *Bottom-up testing* means that the components at the bottom of the structure chart are coded and tested first, and then the next higher level is developed and tested, and this is repeated all the way up the structure chart. Since the bottom-level components usually cannot operate alone, it is necessary to write special programs called *drivers* to test the components.

For instance, let's assume that we have just developed the *Compute Withholdings* subprogram. Before incorporating our subprogram into a larger program, we should write a small driver program to check it. A testing driver might include the following pseudocode fragment:

Pseudocode 10.7 Fragment of code for a testing driver

```
For Gross Pay = 100 to 1000 by 100
  For Dependents = 0 to 12 by 1
    Compute Withholdings(Gross Pay, Dependents,
                        Federal, State, Social)
    Output Gross Pay, Dependents, Federal, State, Social
```

This small driver would allow you to test subprogram *Compute Withholdings* with a variety of arguments. You could even write a

more sophisticated testing program, which generated random test arguments. Obviously, some of the results must still be checked manually.

- Often a combination of both bottom-up and top-down approaches is used. With the bottom-up approach, frequently used utility procedures might be coded and tested independently. Then development and testing could switch to a top-down approach with stubs being used for major program components. When utility procedures are used as development progresses, there is a good degree of confidence in them and this allows us to concentrate on the testing of major components.

Note: At the end of the Testing Strategy Development step, you should have your testing strategy developed, including the input and expected output data for all test cases, as well as the pseudocode for any necessary stubs or drivers.

Development of Testing Strategy Application

Before we start translating the algorithms we developed earlier into code, we must plan our program testing. Experience has shown that when testing is not well planned, it takes an enormous amount of time. Early testing can uncover forgotten cases or unexpected combinations of data. Modifications to correct these design defects are much easier to make at this stage, since no actual programming has been done. Therefore, the testing strategy should be defined before coding begins and should lead to the *definition of a set of test data* that is both comprehensive and practical.

So, here we will define our testing strategy before we begin to code. We define some test data to be the following:

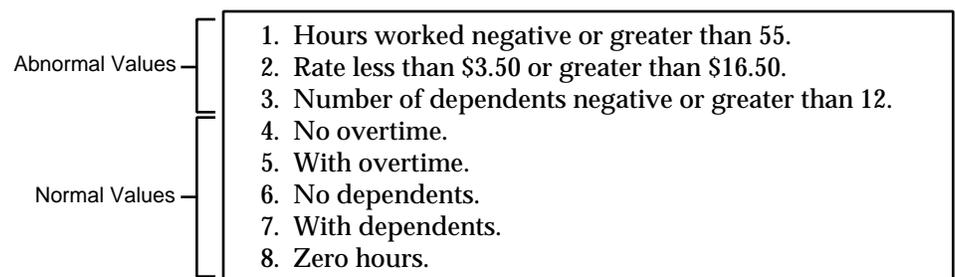
- Comprehensive
- Practical
- Certain to execute every part of the program at least once.

Once this is done, some subtle errors will undoubtedly remain. When those errors pop up later (probably when someone else is using your program), they should be corrected by the programmer who is responsible for the maintenance of your program (usually you). This is the basis for one of the most well-known sayings in the computer science field:

*Testing can show the presence of bugs,
but cannot guarantee their absence.*

For our payroll program, we need to identify the various cases the test data must cover as shown in Figure 10.6.

Figure 10.6 Various cases for Test Data



We will use these cases and combine them to generate test data as follows:

- We will use boundary value data, such as \$3.50 for a normal value and \$3.49 for a wage less than \$3.50.
- For erroneous test data we should have cases 1, 2, and 3 individually, then cases 1 and 2 together, 2 and 3 together, 1 and 3 together, and 1, 2, and 3 together.
- Cases 4 and 5 can be combined with cases 6 and 7 to produce several normal cases.
- Case 8 should be tried alone once to make sure the results are zero.

Using this approach, we define the *test data* shown below, where we have used artificial employee names suggestive of the errors.

Figure 10.7 Test Data Definition

	Hours	Rate	Dep.	Name	
Abnormal values	-10	400	2	hours-negative	
	5510	400	2	hours-too-large	
	1050	349	2	rate-too-low	
	1050	1651	2	rate-too-high	
	1050	400	-1	dependents-negative	
	1050	400	13	dependents-too-large	
	6000	100	3	hours-and-rate	
	6000	400	14	hours-and-dependents	Bogus employee name
	1050	100	14	rate-and-dependents	
	6000	100	14	hours-rate-dependnts	
Normal values	4000	400	0	no-overtime-no-dep	
	4000	400	2	no-overtime-with-dep	
	5500	400	0	max-overtime-no-dep	
	5500	600	12	maximum-hrs-and-dep	
	4000	1650	12	maximum-rate-and-dep	
	0	500	2	zero-hour-and-dep	

The expected results for these test data have to be computed manually by following the pseudocode and given here. These computations should give us some insight into the way our algorithms operate. Here are the *expected* results for the test data given above:

Figure 10.8 Expected Results for Test Data

Computation of Weekly Pay					
	Gross	Fed	State	Soc.	Net
Invalid hours for hours-negative					
Invalid hours for hours-too-large					
Invalid rate for rate-too-low					
Invalid rate for rate-too-high					
Invalid dependents for dependents-negative					
Invalid dependents for dependents-too-large					
Invalid hours for hours-and-rate					
Invalid rate for hours-and-rate					
Invalid hours for hours-and-dependents					
Invalid dependents for hours-and-dependents					
Invalid rate for rate-and-dependents					
Invalid dependents for rate-and-dependents					
Invalid hours for hours-rate-dependents					
Invalid rate for hours-rate-dependents					
Invalid dependents for hours-rate-dependents					
no-overtime-no-dep	160.00	28.80	4.80	8.00	118.40
no-overtime-with-dep	160.00	21.60	3.60	8.00	126.80
max-overtime-no-dep	250.00	45.00	7.50	12.50	185.00
maximum-hrs-and-dep	375.00	24.30	4.05	18.75	327.90
maximum-rate-and-dep	660.00	75.60	12.60	33.00	538.80
zero-hour-and-dep	0.00	0.00	0.00	0.00	0.00
Totals	1605.00	195.30	32.55	80.25	1296.90

Step 5 Program Coding and Testing

Starting with our pseudocode algorithms, this step consists of coding the actual computer program. Then, using our testing strategy, we test the program and compare our test results with the expected ones. For large programs, the coding will be done progressively, the various components being coded with the necessary stubs and drivers, so that they can be tested systematically.

Errors in design or peculiarities of the programming languages used may cause difficulties in coding, and prevent the implementation of some parts of the design characteristics. In this case, we must go back to the design step and modify our design. The programming and coding step is finished when all the coding has been done and when all the test data have been correctly processed.

Step 6 Documentation Completion

Whether or not a program is to be used by others, it must be documented. Both experienced and naive users must be given instructions for running the program. Furthermore, someone wishing to modify the program must be given information about what design decisions were made, and what implementation and testing problems were encountered. Even the original programmer needs documentation, to be able to modify the program long after it has been developed. It is always extremely surprising to realize how quickly one forgets design decisions along with the reasons why they were made.

Documentation does not begin in this sixth step of our problem-solving method. It begins with the first step of program development and continues throughout the lifetime of the program. The program itself, with its comments, is called *internal* documentation, whereas the testing information is part of the *external* documentation (all documentation other than the program listing). At each step of the program development process, some documentation is produced, as shown in Figure 10.9.

Figure 10.9 Documentation during the Seven Steps

Steps	Documentation produced
1. Problem definition	The problem specifications, including a precise description of the input/output
2. Solution design	A description of the design, with structure charts and modular design charts
3. Solution refinement	Data specifications, pseudocode, and unit interfaces
4. Testing strategy development	An outline of the testing strategy, test data, and expected results, as well as pseudocode for drivers and stubs
5. Program coding and testing	The program code (internal documentation), test data, and results
7. Program maintenance	Change log and, if the changes are important, all the documentation normally produced for a complete program (as in the preceding steps)

All of this documentation must be collected, and must be kept current (with perhaps the exception of the pseudocode) throughout the lifetime of the software. In addition to the above documentation produced at each step, some additional user documentation may be needed. This documentation must provide the user with enough information to be able to use the program and its functions fully, without providing the mental burden of implementation details. This type of documentation is often referred to as a “user’s manual”.

It is usually helpful to develop a preliminary version of the user documentation during the problem definition step and then to refine it once coding and testing are done. This preliminary version of the user documentation can be given to the users, to verify that you have correctly understood the problem and are producing a program that will satisfy the users.

Documentation Completion Application

The documentation will include the problem definition, the design documents, a description of the testing performed, a history of the program development and its different versions, and a user’s manual. Such a manual is designed for a naive user and illustrates how to prepare input data, how to run the program, and how to obtain and interpret results. The following is a sample ACME Payroll User’s Manual:

ACME Payroll User's Manual

Payroll is a program to compute and display the weekly pay of hourly paid employees. For each employee, it will read a series of four data items separated by at least one blank: number of hours worked during the week (0-55), hourly rate of pay (\$3.50-\$16.50), number of dependents (0-12), and name of employee (20 characters). If the data are valid, the program will compute and display federal (18% of taxable income), state (3% of taxable income), and social security

(5% of gross pay) withholdings, as well as gross and net pay for the employee.

- The program will read data for each employee from the file `Payroll.data` until it reaches the end of the file. It will produce results on the screen. Input data format is such that three integer values precede a character string, as in
4500 600 3 Allan Mackenzie
for 45.00 hours worked, \$6.00 per hour, and 3 dependents.
- Output data will be written one line at a time, each line corresponding to an employee. A normal output line will consist of a 20-character string followed by five values. The last line will contain the word "Totals" followed by five values and will be separated from the previous output line by a blank line. The output will be preceded by the title lines:

```

Computation of Weekly Pay
      Gross      Fed   State Soc.      Net

```

- A normal output line will look like
Allan Mackenzie 285.00 40.50 6.75 14.25 223.50
- Erroneous data will produce error messages of the form
Invalid hours for Robert A. Verner
Invalid rate for Simon J. C. W. Surry
Invalid dependents for T. Guy Rimmer
- These messages will appear if the values read are not within the given limits. A single employee data line with erroneous data can generate from one to three error messages. The erroneous data have to be corrected and resubmitted.

To run the program, enter the payroll data in file `Payroll.data` and execute `Payroll`.

Step 7 Program Maintenance

Program maintenance is not directly part of the original implementation process, but needs special emphasis. All activities that occur *after a program first becomes operational* are part of the *program maintenance*. Many large programs have long lifetimes that often exceed the lifetime of the hardware they run on. Usually, the cost of program maintenance over the lifetime of a program will be greater than the total program development costs.

Program maintenance includes the following:

- Finding and eliminating previously undetected program errors;
- Modifying the current program, often to improve its performance, or to adapt it to new laws or government regulations, or to adapt it to new hardware, or to a new operating system;
- Adding new features or a better user interface, or new capabilities to the program; and
- Updating the documentation.

Maintenance is an important part of the life cycle of a program. It is also important from a documentation point of view, since changes to a program will require updating the existing internal as well as external documentation.

Maintenance documentation will include many of the results of the program development steps: design documents, program code, and information about testing.

The discovery and elimination of previously undetected errors inevitably involves new test data for showing that the bugs have indeed been fixed. Similarly, modifications to the program will entail new test data to prove the correctness of the modifications. In both cases, once the modifications are completed, all the old test data—perhaps changed because of the modifications—must still be run correctly to demonstrate that the modifications have not introduced new bugs.

Note: The new test data must be added to the old test data to build a bigger “test suite” to be used in future maintenance.

Since most program maintenance is done by someone not involved in the original design, it is imperative that the programs be well designed and well structured, as well as readable, and that the documentation be complete and accurate.

Program Maintenance Application

Improvements to the payroll program are numerous. Here are a few possible ones:

- Having the program actually produce the paychecks for the employees;
- Adding a unit to compute tax withholdings in a more flexible way;
- Adding an interactive way of fixing the erroneous data.

10.3 An Advanced Case Study Building a Text Index

This section presents a larger and more complex case study, one that exercises everything that we have learned in the previous chapters. It will also show some of the advantages that can be obtained from a modular construction, which is a concept introduced in Chapter 7.

Step 1 Problem Definition

We want a program capable of the following:

1. Reading in some text stored in a given file;
2. Collecting all the *significant* words of that text together with the page numbers where the words occur; and
3. Displaying an alphabetical index of the words with their page numbers.

A word will be defined as a sequence of letter or digit characters, starting with a letter. To find whether or not a word might be significant, we will use a dictionary of trivial words that must be ignored (words like “the”, “a”, “at,” etc.) To simplify, we will use the backslash character (“\”) to indicate the end of a page.

The program will prompt the user for the name of the dictionary of trivial words, for the name of the text file, and the name of the new index file, using the following format for the output messages.

```
Give name of trivial words file:
Give name of text file:
Give name of output file:
Index complete
```

The format of the index will be the following:

```
June      1  8
Karine    1  2  3  4  5  6  8  9  10
Kludge    5  9
```

Step 2 Solution Design

The program will use functions for the following purposes:

- Getting a word from the input file,
- Inserting a word in the index if it is deemed to be significant,
- Comparing two words to alphabetize,
- Displaying a word from the index, and
- Displaying an item from the index (a word and its associated page numbers).

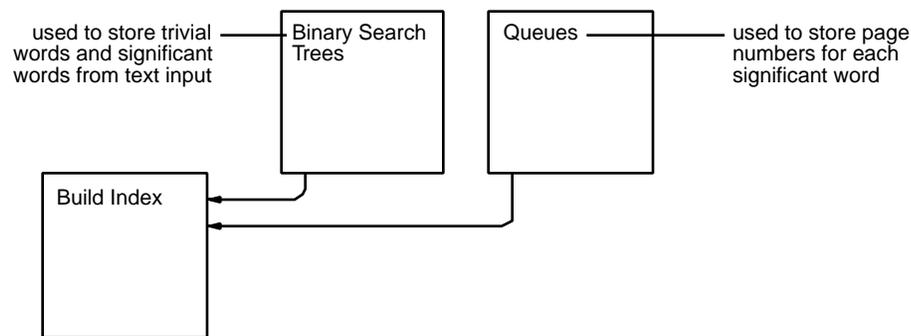
The program will keep the index in a *binary search tree* (introduced in Chapter 9), a data structure most suitable and efficient for cases where operations like searches and insertions are numerous. This structure also has the property of keeping its elements ordered, which makes it possible to display its contents easily in order. We will define an ADT binary search tree, and will keep it in a separate *module* or external unit consisting of the binary search tree definitions and operations.

Like a binary tree, a *binary search* tree is made of nodes, and each node has a left and a right descendant (either or both of which can be absent). As we have seen in the last section of Chapter 9, the main property of a binary search tree is that all the left descendants of a node have a value that is less than the node value, and all the right descendants have a value greater than the node value.

Likewise, the page numbers associated with a word will be kept in a *queue*, where each page number is unique and will be output in the order of insertion, that is from lowest number to highest number. Here again we will define a separate unit for the ADT queue, to include the definitions and operations of queues. Remember that queues are first-in-first-out structures: the elements in a queue are output in the order in which they entered the queue.

Our solution will be illustrated by the modular design chart of Figure 10.10. This chart illustrates the relationships between the various units of our solution: “Build Index” will use types and operations from ADT Binary Search Trees and ADT Queues.

Figure 10.10 Modular design chart for building a text index



We will use two Binary Search Trees for two purposes:

- Storing all the trivial words, and
- Storing all the significant words from the text.

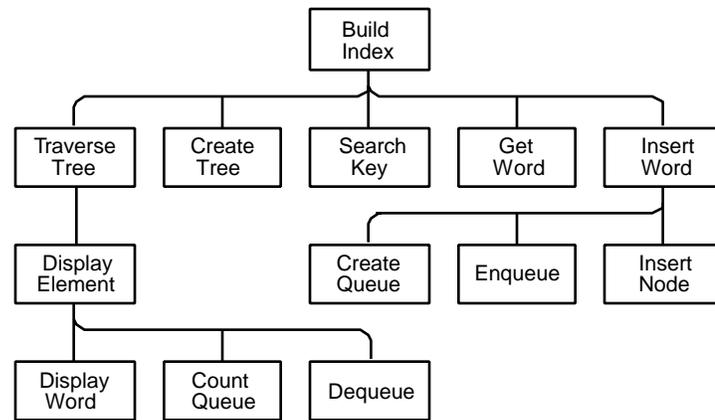
Our solution will include the functions we have mentioned earlier as well as functions from the binary search trees unit and the queues unit. We will use the following binary search tree operations:

- Initialization of a tree,
- Searching for a word in a tree,
- Inserting an element in a tree, and
- Traversing a tree to display its contents (elements and words).

We will use a queue structure to store all the page numbers associated with a word. We will use the following queue operations:

- Initializing a queue,
- Inserting an element in a queue,
- Eliminating an element from a queue, and
- Counting the elements in a queue.

We can now proceed to draw the structure chart for our solution as shown in Figure 10.11.

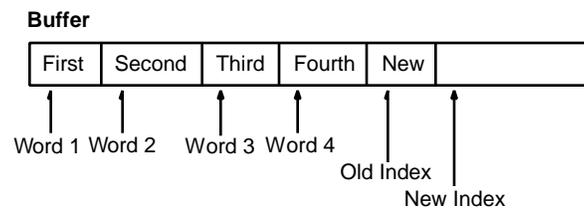
Figure 10.11 Structure chart for building a text index

Step 3 Solution Refinement

To refine our solution we will need to define the functional elements of “Build Index”, as well as the functional elements of units “Binary Search Trees” and “Queues”. If we are lucky, someone might already have developed such useful units, and all we will have to do will be to use them, or, more likely, adapt them before using them. We can also give you a hint. If you have access to computer science books, you will find such units already defined; use them!

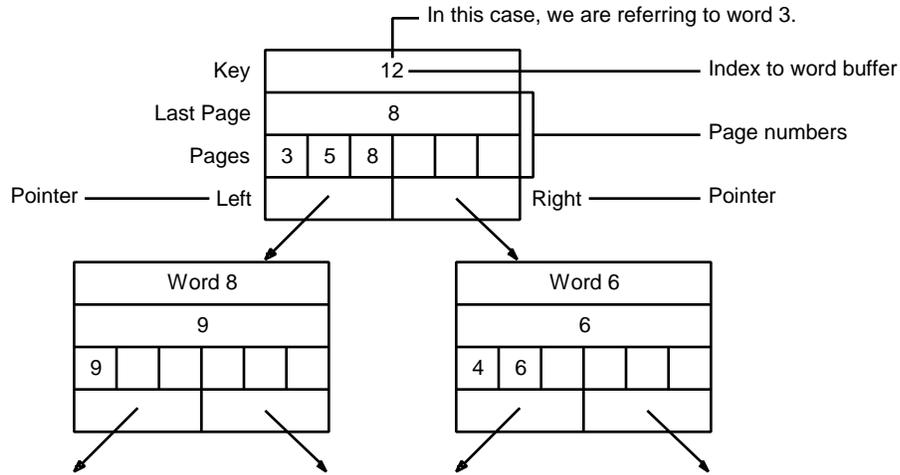
We will also need to define the data structures that will be used in our program. Obviously, the elements of our index will be kept in a binary search tree. Each element record will be comprised of (i) the corresponding word and (ii) its list of page numbers. If we keep the words in the tree nodes, we will have to reserve enough space for the longest word, and this is not necessarily the most efficient way to use memory space, as much of this space will be empty.

Instead, we’ll use a well-known technique of storing the words in a long *buffer* (storage area), to keep only nontrivial words, and no duplicates. The buffer will be a big array of characters, and each word in it will be identified by the index of its first character, as shown in Figure 10.12. Note that every character has a corresponding index number.

Figure 10.12 Word buffer organization

With the buffer, we will keep two special indices, *Old Index* and *New Index*. *Old Index* always points to the next free space in the buffer. *New Index* is used to enter a new word in the buffer, before deciding whether to keep it or not. To keep a new word in the buffer, *Old Index* is updated to the value of *New Index*.

Figure 10.13 Binary search tree organization

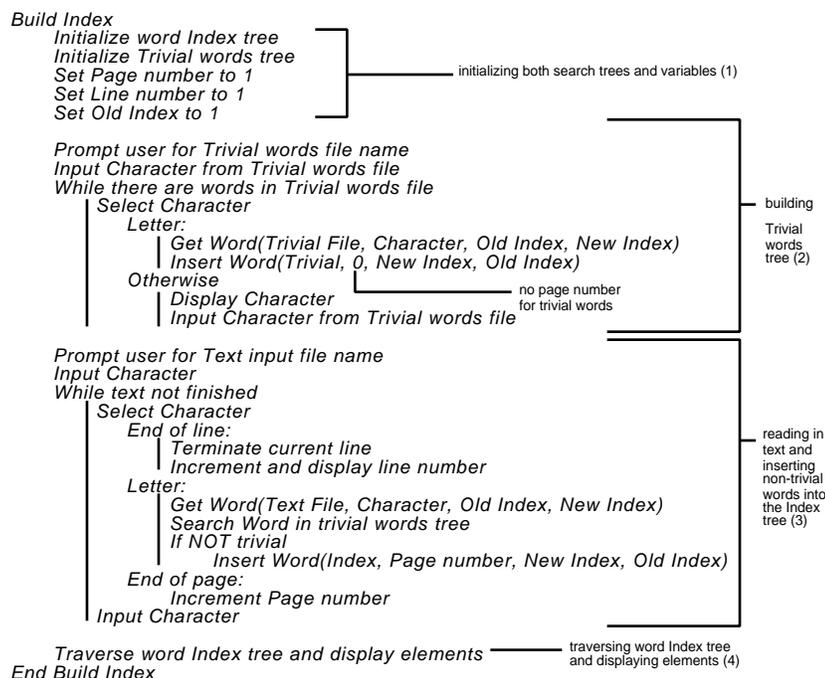


The elements kept in the binary search tree will have the structure illustrated by Figure 10.13. Each element will have five parts:

- an Index to the word buffer, called Key,
- a Last Page number (to avoid keeping duplicate page numbers),
- a Pages queue,
- a Left pointer for the tree structure, and
- a Right pointer for the tree structure.

With these data structures established, we can develop the pseudocode for *Build Index* as follows.

1. First we initialize various variables.
2. Then, we read in the trivial words file and build the trivial words tree.
3. Next, we read the text and insert the nontrivial words in the word index tree.
4. Finally, we traverse the word index tree, displaying the elements as we encounter them.

Pseudocode 10.8 The Build Index algorithm

Besides some output subprograms, this program uses two important subprograms, *Get Word*, to read a word, and *Insert Word*, to insert a word in a binary search tree. These two subprograms are repeatedly invoked in the two loops that build the *Trivial* and *Index* trees. The two loops read the text between words character by character until a letter, denoting the start of a word, is detected.

The loop that builds the *Trivial* tree ignores all inter-word characters. When a letter is encountered, the *Get Word* subprogram is invoked to read the word. Since all words read from the *Trivial File* are, by definition, to be inserted into the *Trivial* words tree, *Insert Word* is invoked for every word.

The action of the loop that builds the *Index* tree is somewhat different. In that loop, two of the inter-word characters are detected and lead to special actions:

- The end-of-line character causes the *Line number* to be incremented, and
- The end-of-page character causes the *Page number* to be incremented.

As in the first loop, when a letter is encountered, the second loop invokes the *Get Word* sub-algorithm to read the word. However, in the second loop, the word is not inserted into the *Index* tree until a search of the *Trivial* words tree has shown that the word is not trivial.

Get Word reads in a word, character by character, and stores it in the character buffer. This storage will only be temporary if it is discovered that the word is trivial or has already been encountered. *Get Word* also leaves a special inter-word marker in the buffer to separate the words. This marker could either be the size of the word or a special end of word marker. *Get Word* only modifies *New Index*, it does not modify *Old Index*. Thus, several successive calls to *Get Word* will simply overwrite the last word in the buffer and leave no permanent record of the words that were read.

Pseudocode 10.9 The Get Word sub-algorithm

```

Get Word(File, Character, Old, New)
  Set New to Old
  Repeat
    Copy Character into word buffer
    Increment New
    Input Character
  Until(Character NOT(letter or digit)) OR end of file
  Set inter-word marker in buffer
End Get Word

```

reading in one word and putting it in buffer

In *Insert Word*, we first check to see whether or not the word is already in the binary search tree. If it is, we only need to update its element in the tree by adding a new page number to its associated queue (if that page number is already there, we do nothing, since we do not want duplicate page numbers). If the word is already in the tree, we do not want to keep a duplicate in the buffer and, for this reason, we do not update *Old Index*.

If this is a new word, *Insert Word* creates a new element for the tree and initializes its associated queue. Since we will also use *Insert Word* to build the *Trivial* tree and since trivial words do not need a page queue, we will use a zero page number for trivial words. Once the element is ready, we insert it into the tree and we update the *Old Index* so that, this time, the word is kept in the buffer. The pseudocode for the *Insert Word* sub-algorithm is shown below.

Pseudocode 10.10 The Insert Word sub-algorithm

```

Insert Word(Tree, Page, Old Index, New Index)
  Search for Word in tree
  If Word already in tree
    If Page Word.Last page
      Enqueue(Word.Pages, Page)
      Set Word.Last page to Page
    Else
      Create Word element
      Set Word.key to Old Index
      If Page 0
        Create Queue(Word.Pages)
        Enqueue(Word.Pages, Page)
        Set Word.Last page to Page
      Insert Node(Tree, Word)
      Set Old Index to New Index
  End Insert Word

```

to keep the new word in the buffer

To display an element, we first display the associated word followed by a number of spaces to align all index entries. Then we output the page numbers after removing them one by one from the *Pages* queue. As we output them, we count them and, if they take more than one line, we continue on the next line after skipping the space underneath the word.

Pseudocode 10.11 The Display Element sub-algorithm

```

Display Element(Element)
  Display Word(Element.Key)
  Set Counter to 0
  While Count Queue(Element.Pages) > 0
    If Counter = Maximum
      Terminate line
      Set Counter to 0
      Display Word length blanks
    Dequeue(Element.Pages, Item)
    Display Item
    Increment Counter
  Terminate line
End Display Element

```

To display a word we only have to display all its characters from the buffer, and then to pad the rest of the word space with blanks.

Pseudocode 10.12 The Display Word sub-algorithm

```

Display Word(Index)
  For all characters in Word
    Display character from buffer
  For remaining length
    Display blank
End Display Word

```

Binary Search Tree Unit

Although this unit is not directly part of our problem, we need it to implement our solution. A tree will be defined as a pointer to a tree node, and a tree node will include an element and two pointers (left and right), as shown in Figure 10.9. To illustrate once more a binary search tree (see Figure 9.17 in Chapter 9), we can show the tree structure corresponding to the following list of trivial words.

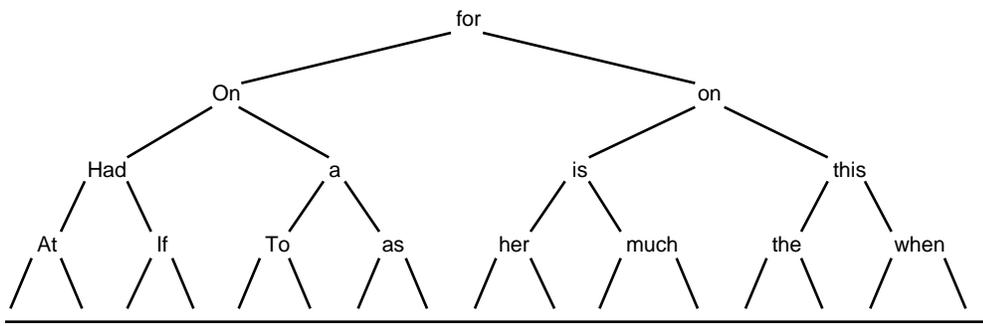
```

for On on Had a is this At If To as her much the when
up was we Am Did He Its That With an by do did be but
has his like nor ours there will you with whom A An
But For Do Has His I In My Much She The We You all and
at from have hers if it me my of or so their these
those to us whose All As By From Have No Our So This
Will Your am are etc had he him in its mine no not off
our she that them they who your yours

```

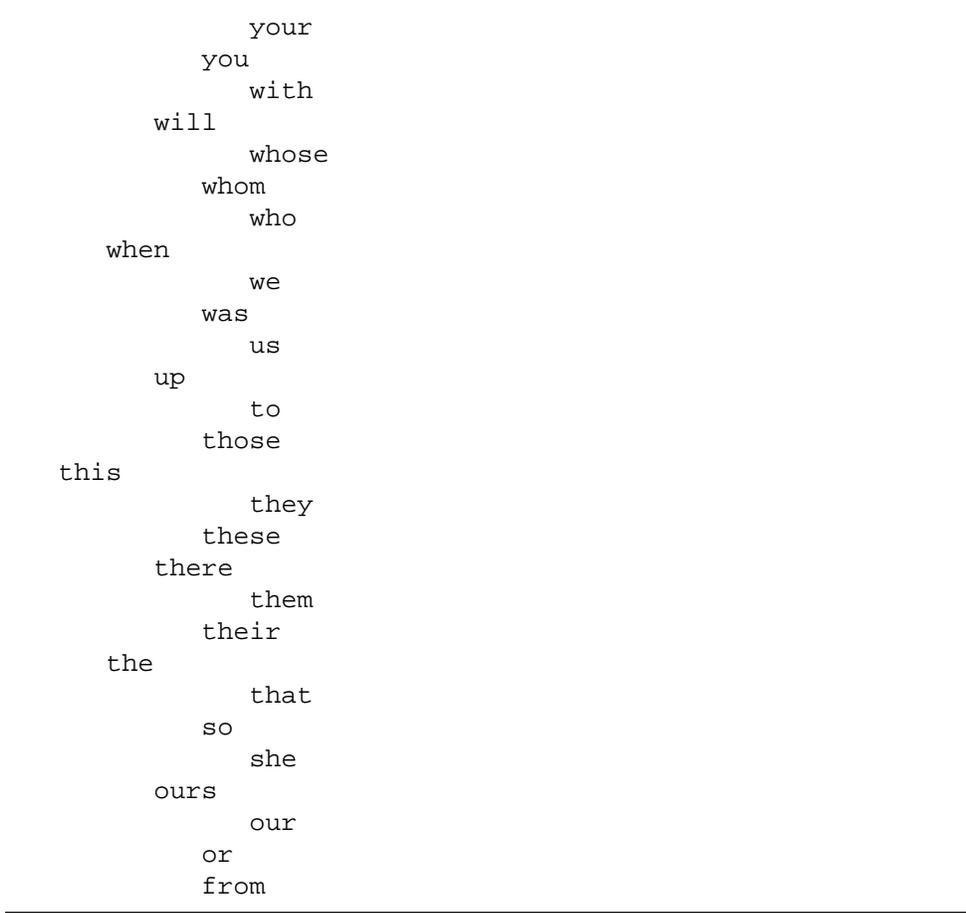
Note that the root of the tree is the first word of the list, “for”, its left descendant is the second word of the list, “On” (upper case letters have a smaller code than lower case letters), its right descendant is the third word of the list, “on” and so on. We have been careful to set the trivial words above in an order that gives the best tree structure (most nodes actually have two descendants). The top part of the corresponding tree is shown in Figure 10.14.

Figure 10.14 Trivial words tree



In Figure 10.15, we give a partial tree of trivial words, as printed by utility subprogram *Display Tree*, which shows the tree structure. See if you can redraw this figure with its connecting lines to get a complete picture of the tree, as in Figure 10.14.

Figure 10.15 Partial tree of trivial words output by *Display Tree*



The Binary Search Tree external unit implements the ADT binary search tree. It includes the following operations.

- *Initialize Tree(Tree)*
This is the first operation to call, in order to initialize *Tree*.
- *Delete Tree(Tree)*
This operation deletes all information about the *Tree* and its data.

- *Insert Node*(Tree, *Element*)

This operation inserts *Element* at the proper position into the *Tree*, according to the key for the *Element*. If a node already existed with the same key, it is updated with the value of *Element*. The tree insertion operation is defined simply as a recursive subprogram (it calls itself on the left and right sub-trees).

Pseudocode 10.13 **Insert Node recursive sub-algorithm**

```

Insert Node(Tree, Element)
  If Tree is NIL
    Attach new node with NIL left and right pointers
  Else
    If Element.Key < Tree key
      Insert Node(Tree Left, Element)
    Else
      If Element.Key > Tree key
        Insert Node(Tree Right, Element)
      Else
        Update Tree element
  End Insert Node

```

- *Delete Node*(Tree, *Key*)

This operation finds the element with this *Key* and deletes it from the *Tree*. If no node has this *Key*, *Tree* is unchanged.

- *Traverse Tree*(Tree, *Process*)

This operation applies *Process* to each node of *Tree* in order. Note that *Tree* is underlined, which makes it an in-out parameter (called by reference), as *Process* might modify the *Tree* contents. This tree traversal operation is a variation of the traversal algorithm we saw at the end of Chapter 9; it is recursive and very simple.

Pseudocode 10.14 **Variation of Traverse Tree algorithm**

```

Traverse Tree(Tree, Process)
  If Tree NIL
    Traverse Tree(Tree Left, Process)
    Apply Process to element
    Traverse Tree(Tree Right, Process)
  End Traverse Tree

```

- *Search Key*(*Tree*, *Key*, Element, *Success*)

This operation searches for an element identified by *Key* in the *Tree*. If it finds it, it returns the complete *Element*, and *Success* is set to True, otherwise *Success* is set to False. The tree search operation is also recursive and very simple.

- *Display Tree*(*Tree*, *Indentation*, *Display Key*)

This operation displays *Tree* keys with indentations to show the *Tree* structure (we showed it above).

- *Display Key*(*Element*)

This operation displays a key value.

Pseudocode 10.15 The Search Key algorithm

```

Search Key(Tree, Key, Element, Success)
  If Tree = NIL
    Return False
  Else
    If Tree Key = Key
      Set Element to Tree element
      Set Success to True
    Else
      If Tree Key < Key
        Search Key(Tree Left, Key, Element, Success)
      Else
        Search Key(Tree Right, Key, Element, Success)
  End Search Key

```

Queues Unit

The external unit Queues implements the ADT Queues whose operations are described below. A queue will be defined as a record with two indices, a counter, and an array of elements, as shown in Figure 9.15 in Chapter 9.

- *Initialize Queue(Queue)*
Creates an empty *Queue*.
- *Enqueue(Queue, Item)*
Inserts element *Item* at the end of *Queue*.
- *Dequeue(Queue, Item)*
Deletes first element of *Queue*, returned in *Item*.
- *Count Queue(Queue) function*
Counts current number of elements in *Queue*.
- *Queue Head(Queue, Item)*
Returns value of first element of *Queue* in *Item*.

These operations have already been described in pseudocode in Chapter 9.

Step 4 Development of Testing Strategy

To test our program we need a trivial words file and a text file. These are various cases we can envision:

1. Empty trivial words file: the index will include *all* the words in the text.
2. Only one trivial word: the only trivial word will not appear in the index.
3. One-page text file: the page numbers are all the same, and appear only once for each word.
4. Text file with several pages: general case.
5. Text file with only trivial words: the index will be empty.
6. Text file with no trivial words: the index will include all the words in the text.
7. Word found on more pages than fit on a single line: necessary to test the splitting of the page numbers over several lines.

To test all cases, we will need at least three trivial words files:

- Trivial 1: an empty file for case 1
- Trivial 2: a file with the single word “kernel” for case 2
- Trivial 3: a general file including the list we have given earlier.

We will also need at least three special text files:

- Text 1: a one-page text file with words in alphabetical order

Sample: albatross beauty cartography demon earl
 fugitive gross helicopter indeed joy
 kernel lullaby mammoth nerd opera possible
 quintessence refrigeration subtlety ton
 utilitarian vampire wapiti xylophone yak
 zero

- Text 2: the same file as Text 1 but with an end-of-page mark between every word

Sample: albatross\beauty\ cartography\ demon\
 earl\ fugitive\ gross\ helicopter\ indeed\
 joy\ kernel\ lullaby\ mammoth\ nerd\
 opera\ possible\ quintessence\
 refrigeration\ subtlety\ ton\ utilitarian\
 vampire\ wapiti\ xylophone\ yak\ zero

- Text 3: a file with only two words repeated thirty times with an end-of-page after each occurrence.

Sample: albatross beauty\ albatross beauty\
 albatross beauty\ albatross beauty

For the various cases we will use the following combinations:

- Cases 1 and 3: Text 1 and Trivial 1. The index contains all the words in the same order with only a page 1 reference.
- Cases 2 and 3: Text 1 and Trivial 2. The index contains all the words except “kernel” in the same order with only a page 1 reference.
- Cases 4 and 6: Text 2 and Trivial 3. The index contains all the words in the same order each with a different page reference.
- Case 5: Trivial 3 and Trivial 3 (same text). The index is empty.
- Case 7: Text 3 and Trivial 2. The index contains two words each with thirty page references.

You should also add words that are longer than the word length to make sure they are treated correctly, and of course a long and realistic text.

Step 5 Program Coding and Testing

The main program follows from our pseudocode. The details of this stage are covered in the Practice book.

Step 6 Documentation Completion

All the results of our previous steps should be part of our documentation: problem specifications, solution design and refinement, testing strategy, program code, and testing results. We must also add a user's manual, as shown below:

User's Manual for the Build Index Program

The Build Index program builds the index of a text. The program prompts you for a file of trivial words (words of the text that must not be part of the index). The program then prompts you for the name of the text file, and the name of the output index file. To run it, select *Execute* in the menu and double-click on Build Index. The program will start executing and will start prompting you. When the execution is over, the program will display the message:

```
Index complete
```

You can examine the index and print it if need be. The index file has the input text with added line numbers, and an alphabetical list of all the words in the text (except for the trivial words) followed by a list of page numbers as in the following:

```
June          1      8
Karine        1      2      3      4      5      6      7      8
              9      10
Kludge        5      9
```

Step 7 Program Maintenance

A non-negligible part of the software life cycle, program maintenance, starts as soon as the program is released to its user. The user might find some bugs: this program is not simple enough to be able to feel sure that there are no bugs left.

Maintenance includes bug removal, but it also includes making improvements to a program or a system. In the case of *Build Index*, we can suggest three areas of improvement:

- We can improve the program by giving the user the option to list the trivial words used as part of a better documentation of the index.
- We can also make the program recognize true end-of-pages instead of an arbitrary sign.
- We can also modify the program so that it displays page *and line numbers* as part of the index.

10.4 Review Top Ten Things to Remember

1. In this chapter, we have reviewed the seven-step problem-solving method that was introduced in Chapter 2, and illustrated in various other chapters. This has made it possible to link a number of the concepts that were introduced in the previous chapters.
2. We have illustrated here mostly the steps related to design, as the implementation stage is programming-language dependent, and cannot be discussed in total abstraction from the actual programming context.
3. At the end of Problem Definition step, you should have a list of *specifications*. These are precise definitions of what the program must do. At a minimum, they must include input (what data must be input and in what form) and output (what data must the program produce and in what form in order to solve the problem).
4. At the end of the Solution Design step, you should have a structure chart describing the hierarchy of your algorithm. Developing a structure chart, using top-down design, will help us write the algorithm to solve our problem since each of the boxes in the structure chart will typically be implemented as a sub-algorithm.
5. At the end of the Solution Refinement step, you should have developed pseudocode for each box of the structure chart. As well, all of the data structures and variables used must be defined.
6. At the end of the Testing Strategy Development step, you should have your testing strategy developed, including the input and expected output data for all test cases, as well as the pseudocode for any necessary stubs or drivers.
7. The implementation stage, steps 5 through 7, are thoroughly illustrated in the Practice book. It is important for you to remember that a lack of method will be disastrous for you when you start developing larger, more complex programs.
8. The complexity of a given application grows quickly, as the number of interactions between the various parts of a system increases. This was illustrated by the examples presented in this chapter.
9. The complete example to build a text index is relatively small, but still big enough to show that a larger system is harder to understand, even under good conditions.
10. Using what you have learned in this book and in the Practice book, you will have to write and run real programs using a given programming language. This is where you will notice that a lack of method is extremely costly in time.

10.5 Glossary

Buffer: A holding area in memory for data.

Driver: A program that is used to test a subprogram by simulating the context from which the subprogram will be invoked when the full program is completed.

External documentation: A program description that is designed to serve the needs of the users of the program. It thus describes the program in terms of the way in which it is used and its observable actions.

External unit: A collection of subroutines that are compiled separately from the main program and are concerned with the manipulation of a particular type of data, for example, an abstract data type is frequently implemented as an external unit.

Internal documentation: The internal structure and workings of a program described in a form to assist programmers who will have to make corrections or other modifications to the program in the future.

Software life cycle: The sequence of stages that a program passes through during its useful life, these stages essentially correspond to the seven steps of software development described in this book.

Stub: A temporary version of a subprogram that provides a very simplified simulation of the action of the subprogram until the subprogram itself is written. Stubs allow early testing of the routines that will call the subprogram. A typical action of a stub program is to output a message that it has been entered so that the logic of the calling program can be checked.

Testing strategy: A plan for the orderly verification of the execution of a program during development.

10.6 Problems

The following is a series of problems and projects to be solved using the seven-step method presented in this chapter. As we have done in the case studies, all of the steps, except for the actual implementation, have to be followed. These problems are presented in order of increasing difficulty and complexity.

Level 1 — Getting Started

The first-level problems require the development of your first programs.

1. A Guessing Game

You are on vacation at home and planning to enjoy your free time. Alas! your parents ask you to take care of your little sister, and she is a real pest. In order not to see your vacation time slowly wasted, you decide to have the computer entertain your little sister. To do this, you want to develop a simple game program that will pick randomly an integer number between 1 and 1,000. The program will ask your little sister to guess that number in a maximum of ten tries, and will produce an appropriate message when the end of the game is reached.

Obviously, the program needs to be interactive. It will display a message at the start of the game and prompt your sister for a guess. Each time she makes a guess, it will have to indicate whether the guess was between the limits or was high or low, or detect that the guess was right. At the end of a game, the program will allow your little sister to decide to continue to play or to stop.

The input format is simply that of an integer number, or a character for yes or no. The output formats are mostly messages.

Start-of-game message:

```
Let's play a guessing game.  
I pick a number between 0 and 1,000. You have to  
guess it. But you have only 10 tries to guess my  
number.
```

End-of-game messages:

```
Congratulations! 999 is right.  
You lose! My number was 999.  
Do you want another game? (Y/N)
```

Game messages:

```
Make a guess:  
Wait a minute! My number is greater than 0!  
You wasted a guess! My number is 1,000 or less.  
Well ... your number is too small.  
Sorry, but your number is too big.
```

Remember! We have already seen this example in Chapter 4.

2. Computing a Customer's Change

Your cousin just opened a small store and does not have the funds to buy one of those sophisticated cash registers that compute the change to return to a customer. Since he still possesses his old personal computer,

he asks you to develop a program that, given an amount due and a payment, computes the change. This way he will be able to make sure that whoever he hires will not make a mistake on the change to give back to the customer. The program will compute the change repeatedly until a zero value is given to indicate termination.

The change must be computed in dollar bills, quarters, dimes, nickels, and pennies, with the smallest number of coins possible. The clerk will enter the amount due in cents, the payment also in cents, and the program will return the number of dollars, quarters, dimes, nickels, and pennies to give back. The clerk will be prompted to enter the amount due and the payment by the following messages:

```
Enter amount due in cents (negative or zero to stop):
```

```
Enter payment in cents:
```

The change will be indicated in the following way:

```
Dollars  1
Quarters 1
Dimes    1
Nickels  1
Pennies  1
```

Remember! We have seen examples of change-making in Chapters 3, 4, 5, 7 and 8.

3. A Bouncing Ball

While waiting for your date to show up, you idly bounced a tennis ball on the sidewalk. This gave you the idea to develop a program to compute and display some data on the bounces a ball will make when dropped from a given height. Forgetting your late date, you went home to solve this interesting problem.

To simplify the problem, you assume the ball bounces in place; that is, it remains bouncing on the same spot and does not have any forward motion. The program will prompt the user for the initial height of the ball, the number of bounces to consider, and the ball's elasticity (which must lie between 0 and 1). It will compute the height of each bounce based on the initial height and the elasticity of the ball, and display it. The program will also compute the total distance traveled, which is the sum of the up and down bounces, for the given number of bounces, and display it. The program will repeat this process until the user tells it to stop.

If the ball is at height h , when it bounces, it reaches new height h' , which is computed by the formula

$$h' = h \times \text{resilience}$$

where resilience is expressed as the elasticity coefficient raised to the n th power, if n is the bounce number:

$$\text{resilience} = \text{elasticity}^n$$

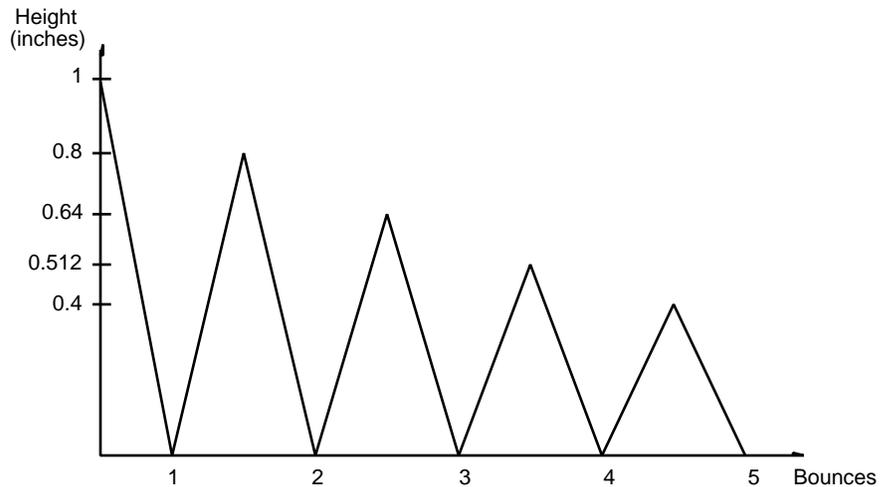
If the original height is H , then the first bounce height will be

$$h_1 = H \times \text{elasticity}$$

With an elasticity in the range between 0 and 1, each bounce will be smaller than the preceding one, but never become 0. We will therefore stop the program after a specified number of bounces.

The ball will travel the distance shown in Figure 10.11 (where the elasticity is 0.8) which we have drawn to help you. We have represented a forward motion for the sake of clarity (the ball bounces on the same spot).

Problem 3 Distance traveled by ball as it bounces



Each time the ball bounces, it travels twice the height of the bounce, so the total distance traveled is

$$H + 2h_1 + 2h_2 + 2h_3 + 2h_4 + \dots$$

The user will be prompted for the initial height in this way:

Give the height in inches from which the ball is dropped:

Then the user will be asked the number of bounces:

Give the number of times the ball will bounce:

And finally the user will be asked the elasticity of the ball:

Give the elasticity of the ball (between 0 and 1):

For each bounce the program will display the following:

On bounce 9 the ball rises to a height of 99.9 inches.

After the last bounce the program will display the following:

The total distance traveled by a ball with an elasticity of 0.999 when dropped from a height of 99.9 inches and after bouncing 9 times is 999 inches.

The user will be asked if the program is to continue:

Another try?

Level 2 — Getting Organized with Subprograms

The second level of problems requires the use of subprograms to manage larger programs.

4. General Application: Your Age in Days

Your problem is to write a program that will read in a person's birth date as well as the current date, compute and display the person's age in days. The program will have to be interactive, to validate the dates it reads and to display results in a clear manner. When the dates given are incorrect, precise messages should be displayed. The program should be set up in such a way that it can compute repeatedly a number of ages and let the user decide when to terminate execution.

The various output formats are mostly user prompts and error messages.

```
Enter birth date in the form DD MM YY:
Enter current date in the form DD MM YY:
Today you are 2059 days old
Want to compute another age?
Incorrect value for month.
Incorrect value for day.
Incorrect value for year.
Incorrect values for day and month.
Incorrect values for month and year.
Incorrect values for day and year.
Incorrect values for day, month and year.
You are not born yet!
```

5. Business Application: What's the Cost of My Mortgage?

The loan department of your bank still uses silly tables to determine what the monthly payment of a mortgage loan is going to be. The consumer association wants you to design a program to compute interactively monthly mortgage payments as well as the cost of such a annual loan over the first 5 years of the loan, and the total cost of that loan. In order to validate the input data, the annual loan amounts must be between \$1,000 and \$500,000, the loan interest must be between 3 and 20%, and the loan length must always be between 1 and 35 years.

First, we must compute the monthly payment. To see how this can be done, let's start with a simple example and build up to the general case.

Suppose a loan of \$1000 made at an interest rate of 10%. If there is to be a single payment P at the end of the load period then

$$1000 = \frac{P}{1.1} \text{ or } P = 1000 \times 1.1 = \$1100$$

If the repayment were to be made in two equal payments of P spaced at equal intervals and the interest rate for each of the periods were 5%, then P must be such that

$$1000 = \frac{P}{1.05} + \frac{P}{1.05^2} = P \left(\frac{1}{1.05} + \frac{1}{1.05^2} \right) = 1.86P$$

whence

$$P = \$537.63$$

More generally, if we make a loan of L to be repaid in n payments of P and an interest rate of r per repayment period, then the n payments P must be such that

$$L = \frac{P}{1+r} + \frac{P}{(1+r)^2} + \frac{P}{(1+r)^3} + \dots + \frac{P}{(1+r)^n}$$

If we multiply both sides of this identity by $1 + r$, we obtain

$$L(1 + r) = P + \frac{P}{1 + r} + \frac{P}{(1 + r)^2} + \frac{P}{(1 + r)^3} + \dots + \frac{P}{(1 + r)^{n-1}}$$

If we subtract the first identity from the second, many terms cancel out and we are left with

$$L(1 + r) - L = P - \frac{P}{(1 + r)^n}$$

whence

$$P = rL \frac{(1 + r)^n}{(1 + r)^n - 1}$$

In the case of our mortgage, where the payments are made every month over a period of y years, we can express this as

$$\text{MonthlyPayment} = \text{MonthlyRate} \times \text{Loan} \times \frac{(1 + \text{MonthlyRate})^{12y}}{(1 + \text{MonthlyRate})^{12y} - 1}$$

Now, our problem calls for the interest rate to be expressed as an annual rate and the monthly rate must be derived from this. Although banks are a little secretive about the formulas they use to compute mortgages, we know that to take account of the effects of compounding, they use a monthly rate that, if compounded twice per year, will yield the stated annual rate of interest. Loan rate tables show you that

$$(1 + \text{MonthlyRate})^{12y} = 1 + \frac{\text{AnnualRate}}{2}^{2y}$$

which leads to

$$(1 + \text{MonthlyRate})^6 = 1 + \frac{\text{AnnualRate}}{2}$$

or

$$6 \times \log(1 + \text{MonthlyRate}) = \log \left(1 + \frac{\text{AnnualRate}}{2} \right)$$

and finally

$$\text{MonthlyRate} = e^{\frac{\log(1 + \text{AnnualRate}/2)}{6}} - 1$$

The program will use these formulas to compute and display the monthly payment of a given mortgage loan and to produce a detailed report of the payments over the first 5 years if needed.

The output format used will be the following:

```
Amount of mortgage loan:
Annual interest rate:
Length of mortgage loan (in years):
Monthly payment: 999.99
Do you want a detailed report?
Interest paid in 5 years: 12345.67
5 year balance: 23456.78
Total cost of mortgage loan: 34567.89
```

The detailed report format will be the following:

Payment#	Interest	Capital	Acc. Int.	Balance
1	97.59	44.21	97.59	9955.79
2	97.16	44.65	194.75	9911.14

6. Scientific Application: Solving the Quadratic Equation

The problem you have to solve now is the well-known quadratic equation. Remember its form?

$$ax^2 + bx + c = 0$$

Your program will accept the values of the three coefficients a , b , and c and then compute the roots of the equation. The program will prompt the user for the three coefficients repeatedly, and stop when the user enters three zeroes for the coefficients. The program will distinguish between the various solutions and display the results with an appropriate message: one root, double root, real roots, complex roots, as well as an error message if coefficients a and b are zero while c is not.

The output formats for prompts and results will be the following:

```
Give values of three coefficients:
Contradiction: 2.0 = 0
One root = -25.0
Double root = 120.0
Root 1 = -2.0
Root 2 = -1.0
Complex roots = -18.0 +/- 12.4i
```

Level 3 — Getting Fancier with Parameters

The third level of problems again requires the use of subprograms to manage larger programs. It provides practice in how to parameterize those programs.

7. General Application: Count the Word Occurrences in a Text

We want a program to read a text, to extract the various words from the text, and to count the number of occurrences of each word. The program will output a list of all the words in the text in alphabetical order (ascending or descending) with their number of occurrences. A word is defined to be a sequence of characters between two separators, and the separators will include all punctuation signs, as well as all available special characters. In fact, a separator will be any character other than a letter (upper case or lower case) or a digit.

The program will prompt the user for the name of a text file to use as input file. It will read the entire file, separate the words and count their occurrences, and finally display the words in alphabetical order, one per line, with their corresponding number of occurrences.

The input and output formats can be summed up by the following messages and examples of output.

```
Give the name of your text file:
In what order do you want the word list? (A/D):
Number of occurrences for the 96 words found
in fffff.ttt
```

```
Word: Occurrences
```

```
A          18
```

```
At         1
```

```
Words table is full, no room for xxxxxx
```

Hint: Although this problem can easily be solved with arrays, if you are smart and have understood the Build Index case study, it might be easier for you to re-use and adapt that case study!

8. Business Application: Processing Personnel Data

Your chum Arnie, from the personnel department of the good old municipal services, has just given you a frantic call. He needs, right now, all sorts of employee lists, and his information systems department just told him it would take three to six months for them to produce a *feasibility study* for a needed program to read, sort, and display data on municipal employees. Obviously, he will get nowhere with his own services, and has the OK to contract out to you the writing of this program.

Further prodding on your part elicits a little more information on the program. It must be interactive; it must run on a personal computer; it will be used by his boss in the personnel department; it should be able to read various employee files; and it should be able to sort employee records extremely quickly by name, by age, or by seniority, and to display these records in four different simple formats. The employee files all have the same format: one line per employee with family name, followed by first name, employee number, hiring date, birth year, and various information including the social security number.

The program will display a short menu to the user, and read and validate the user's choice. The menu format will be the following:

1. Read data from file
2. Sort by age
3. Sort by name
4. Sort by seniority
5. Display name and birth year
6. Display name and first name
7. Display name and hired date
8. Display all information
9. Exit program

Please enter your selection and press return:

The program will have to make sure that operations 2 to 8 are not usable until operation 1 has been used at least once. The program will display the following error message:

```
No data has been read yet
```

After each sort, one of these message will be displayed:

```
Employees sorted by age
Employees sorted by name
Employees sorted by seniority
```

The output formats for operations 5, 6, 7, and 8 are quite simple. The employee name will be followed by a single value, or all the values:

```
Berger 1956
Berger Antonia
Berger 710221
Berger Antonia BERA0 710221 1956 198-39-9739
Middle-management
```

The personnel department foresees a new format for employee records that would double or even triple their size. The sorting of employee records must be designed so that the change in size of the employee records does not unduly affect the sorting time (Hint: Use the sort and rank method introduced in Chapter 9).

In order to read in the employee data, the program will prompt the user in the following manner:

```
Please give name of employees file:
```

Once the data have been read, a message indicating the number of records read will be displayed:

```
Number of employee records read: 99
```

In the case of a large file, the program will check that the data can be stored in the employees table. If not, it will display the following message and skip the rest of the file.

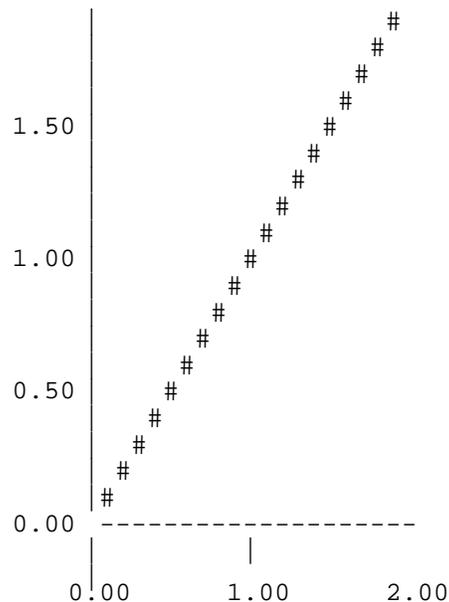
```
File too large, input truncated
```

9. Scientific Application: Plotting a Function

In this graphic era, we want to be able to plot a given function $y = f(x)$ between two values of x . The plot of such a function will be graphical and will appear in the usual manner, that is, assuming a vertical y axis and a horizontal x axis. We want the plot to show parallels to the x and y axes for the minimum values of x and y , with some indication of the x and y values. We also want the user to be able to plot any function of one variable, and to define the range of x values for the plot, as well as the size of the plot.

A typical output would look as shown below.

Problem 9 Plot for $y = x$



Level 4 — Getting Your Wings with Units

The fourth level of problems requires the creation and use of separately compilable units to construct a complete program.

10. General Application: The KWIC Index

Suppose we were interested in programming languages and looking for books or papers on the subject. It will be easy to find promising titles in a listing provided that the authors have been considerate enough to put the words *Programming Language* at the beginning of the title. Thus the book *Programming Language Concepts* would be where we expect to find it in the alphabetical ordering. However, the paper *A Comparative Study of Programming Languages* will be in another part of the listing and, unless we think of looking under *Comparative*, we would be unlikely to find it without a sequential scan through the catalogue. This scan would rapidly exceed our attention span, making us very likely to miss items.

The Key Word in Context (KWIC) index tries to solve this problem by listing each title several times, once for each of its keywords (“noise words” such as *a*, *the*, *and*, *of*, and so on, are not counted as keywords). We might define a KWIC index as being produced by taking each title, generating *circularly shifted* copies, each with a different keyword at the beginning and then sorting the newly generated list alphabetically. A circularly shifted copy is formed by moving one or more words from the beginning of the title to the end. The title *A Comparative Study of Programming Languages* would appear four times as:

```
Comparative Study of Programming Languages. A
Study of Programming Languages. A Comparative
Programming Languages. A Comparative Study of
Languages. A Comparative Study of Programming
```

This is not very easy to read and so in the final form of the index, part of the listing might be rearranged as:

```
A Comparative Study of Programming Languages
Programming Language Concepts
Programming Language Landscape:...
Programming Language Structures
A Comparison of Programming Languages for Softw...
Programming Languages: Design a...
Principles of Programming Languages: Design, ...
...ion to the Study of Programming Languages
Concepts of Programming Languages
Concurrency and Programming Languages
Fundamentals of Programming Languages
...cture and Design of Programming Languages
```

where the titles have been aligned on the word that is being used for the alphabetical ordering and sufficient other words are provided to give some context. Also appearing would be a citation to allow the reader to find the work.

The KWIC program accepts an ordered set of lines, each line being an ordered set of words, and each word being an ordered set of characters. Each line consists of two parts: a Title Part and a Reference Part. The Reference Part is enclosed in brackets. It is assumed that brackets never occur in either the Title Part or the Reference Part other than as identification of the Reference Part. An example of input data is:

```
Software Engineering with Ada [Booch 1983]
The Mythical Man Month [Brooks 1975]
An Overview of JSD [Cameron 1986]
Nesting in Ada is for the Birds [Clark et al.1980]
Object Oriented Programming [Cox 1986]
```

Social Processes and Proofs of Theorems and Programs [DeMillo et al. 1979]
 Programming Considered as a Human Activity [Dijkstra 1965]

A Title Part may be *circularly shifted* by removing the first word and appending it at the end of the line to form a new Title Part. From each line in the input data, new lines are constructed by appending a copy of the Reference Part from the original line to all *distinct* circularly shifted Title Parts. The first word of each such line is the *keyword*. Those Title Parts that begin with the keywords: a, an, and, as, be, by, for, from, in, is, not, of, on, or, that, the, to, with, and without are ignored.

The output of the KWIC program is a listing of all the titles constructed in this way and presented so that the title is shown with its original word ordering and all keywords lined up vertically in the center of the page. Where a line must be truncated at the beginning or end in order to fit on the printed line with the keyword aligned, the truncation is shown with an ellipsis, (...), as shown above.

11. Business Application: Information Retrieval

The board of directors of the Piranha Club, of which you are a member, has commissioned you to implement a small database system for the members of the club. Members will give information that will be stored in the system and will be retrieved for the benefit of other members.

The data for the members will be kept in a permanent file and the system will allow the addition of new members, as well as the deletion of departing members, and also the updating of member information. Some security checks will be implemented in the system by keeping a password with each member's data, and requiring that password for certain operations. The password will be kept in the system in a ciphered form in order to improve system security.

The system will display a menu of the possible operations to the user, who will then choose the desired operation. On program exit, the members table will be automatically stored in a new file.

After discussions with the board of directors, the following formats are agreed upon. The format of the menu will be the following:

1. Add a new member
2. Check membership
3. Get a member's name
4. Get a member's address
5. Get a member's phone number
6. Get information on a member
7. Change member's password
8. Remove member
9. Show member list
10. Exit program

Please enter your selection and press return:

The various messages of the query system will be the following:

- Initialization from a data file

Initializing members table. Give file name:

- Addition of a new member

Give ID of member to add:

Give member name:
 Give member address:
 Give member phone number:
 Give member password:
 Member added

- **Checking a membership**

Give ID of member to look for:
 XXXXXXXX is a member.
 XXXXXXXX is not a member.

- **Information query on a member**

Give ID of member:
 The member's name is:
 The member's address is:
 The member's phone number is:
 Sorry but this ID does not belong to a member.

- **Changing a member's password**

Give ID whose password must be changed:
 Give new password:
 Give your old password:
 Give new password again:
 Password has been changed.
 Wrong password.
 You don't seem to be sure.
 Password has not been changed.

- **Removal of a member**

Give ID of member to eliminate:
 Give Password of member you want to eliminate:
 Wrong password, member could not be removed.
 Member removed.

- **Display of the member table**

Give administrative password:
 Sorry but you do not have access to the list.

- **Copy of members table into a data file at end of execution**

Saving members table. Give file name:

12. Scientific Application: Complex Algebra

Several methods are known to find the roots of an equation. One of the most efficient of these methods is the Newton-Raphson method, developed by Isaac Newton around 1685, and refined by Joseph Raphson in 1690. We can design a program to apply the method without much difficulty. However, now that we have reached level 4, we can be a little more ambitious. We will design and implement a program to apply the Newton-Raphson method to find the root of a polynomial equation in x of a given degree *whose coefficients are complex numbers* (Hint: Create a Complex numbers unit).

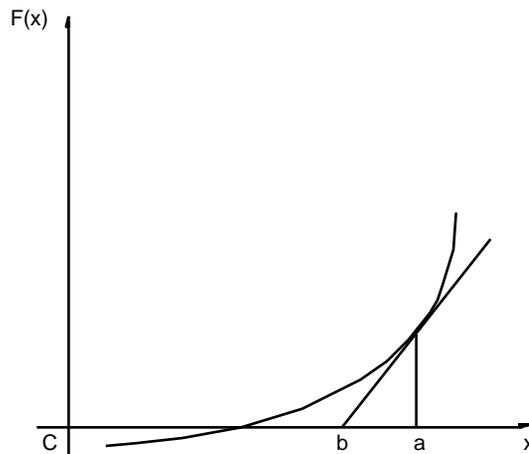
The program will read in the degree of the equation, the corresponding coefficients, the requested accuracy, and the starting point for the Newton-Raphson method. It will apply the method and return a result for the root or an explicative message when the root cannot be computed. The program will repeatedly prompt the user for data, and stop when the user wants to stop.

The following is the format of the dialogue messages that will be used by the program.

Give the degree of the polynomial:
 Give the polynomial coefficients defining the function:
 Coefficient of degree 9; real part:
 Coefficient of degree 9; imaginary part:
 Indicate the precision you wish to achieve:
 Indicate the maximum number of iterations:
 Now give the starting value for x(real part):
 Now give the starting value for x(imaginary part):
 The root is 9.9999999999 + 9.9999999999i
 Newton-Raphson took 9 iterations.
 For a precision of 9.999999E-09
 More?

Let's review briefly the Newton-Raphson method, considering first only polynomial functions with real variables and real coefficients. Given a function of variable x, F(x), we want to find the value of the root in the neighborhood of a. Figure 10.13 illustrates a function and its tangent at x = a.

Problem12 Newton-Raphson method



The linear tangent to F(x) at point (a, F(a)) intersects the x axis at point b, which may be closer to the root than a. Point b is then used as a new approximation of the root value, and the same process is repeated. It is possible to compute the value of b if we know F'(x), the first derivative of F(x).

$$\text{Slope} = F(a)/(a - b) = F'(a)$$

gives: $b = a - F(a)/F'(a)$

The method will fail if the slope is zero, as the tangent is parallel to the x axis: in that case there is no root in the vicinity of the point. We must protect ourselves against failure by not allowing a division by zero, and by limiting the number of iterations. This can be directly transposed to complex numbers.

Appendix Solutions

This appendix contains the solutions to the first few problems at the end of each chapter. Oftentimes, there may be many correct answer for the same question; however, only one answer to each problem is provided.

Appendix Overview

Chapter 1 Solutions 1-3.....	508
Chapter 2 Solutions 1-3.....	508
Chapter 3 Solutions 1-7.....	510
Chapter 4 Solutions 1-3.....	511
Chapter 5 Solutions 1-3.....	512
Chapter 6 Solutions 1-3.....	514
Chapter 7 Solutions 1-3.....	515
Chapter 8 Solutions 1-3.....	516
Chapter 9 Solutions 1-3.....	517

Chapter 1 Solutions 1-3

Solution to Problem 1

- a. Keyboard — human environment.
- b. Analogy-Digital Converter — physical environment.
- c. Scanner — human environment.
- d. Mouse — human environment.
- e. Clock — physical environment.

Solution to Problem 2

- a. Display — human environment.
- b. Sensor — physical environment.
- c. Digital-Analog Converter — physical environment
- d. Scanner — human environment
- e. Clock — physical environment

Solution to Problem 3

- a. **Alan Turing** developed a machine used to define the limits on what is computable.
- b. **Al-Khwarizimi** studied sets of rules which are now called algorithms.
- c. It is not clear who designed and built the first electronic digital computer. The contributor is not in the list.
- d. **George Boole** discovered that symbolism of mathematics could be applied to logic.
- e. **Charles Babbage** designed the first general-purpose “analytical engine”.
- f. **John Von Neumann** introduced the concept of a stored program where instructions and data are both stored in memory.
- g. **Blaise Pascal** designed the first mechanical adding machine.

Chapter 2 Solutions 1-3

Solutions to problems 1 through 3 (next page)

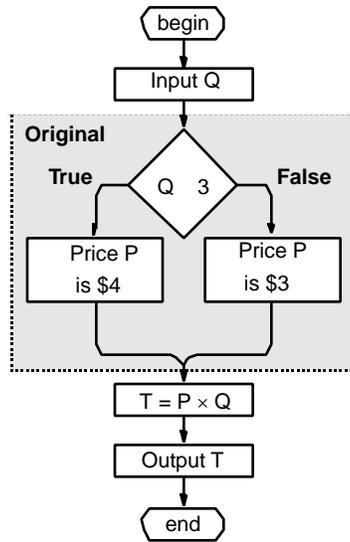
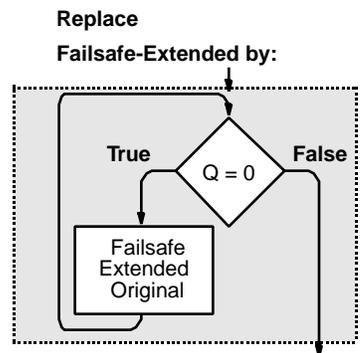
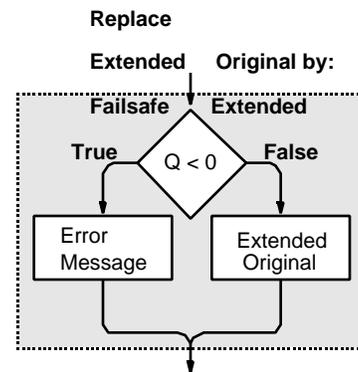
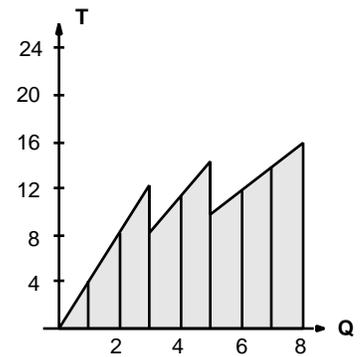
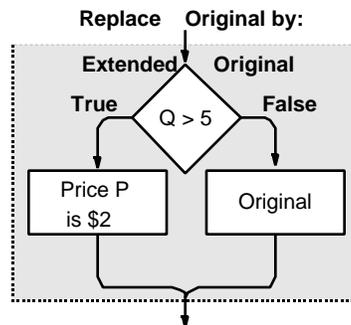
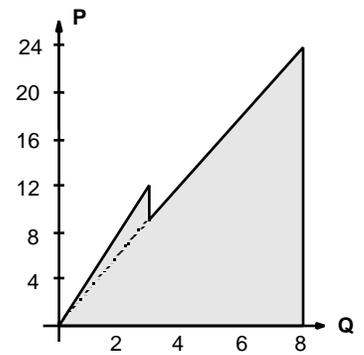
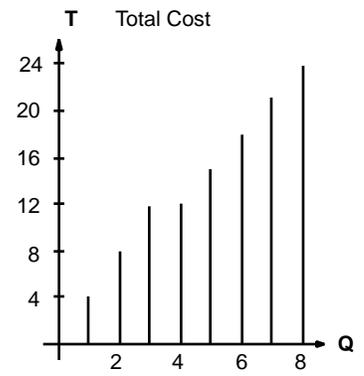


Table of Values

Q	P	T
0	4	0
1	4	4
2	4	8
3	4	12
4	3	12
5	3	15
6	3	18
7	3	21
8	3	24



Chapter 3 Solutions 1-7

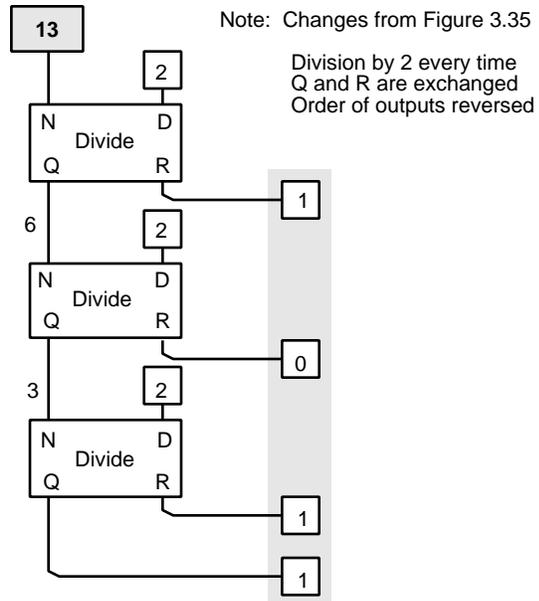
Solutions to problems 1 through 7

1. Binary Number Drill
 - a. $10 = 2$ in decimal
 - b. $1010 = 8 + 2 = 10$ decimal
 - c. $101010 = 32 + 8 + 2 = 42$ decimal
 - d. $7 = 111$ binary
 - e. $17 = 16 + 1 = 10001$ binary
 - f. $170 = 128 + 32 + 8 + 2 = 10101010$ binary

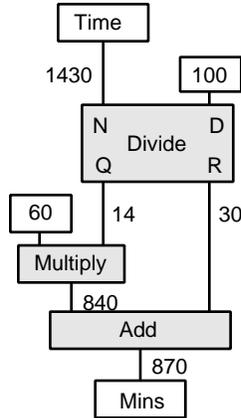
2. Small Binary Numbers

Dec	Bin
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

4. Decimal to Binary: Another way



3. TimeBase

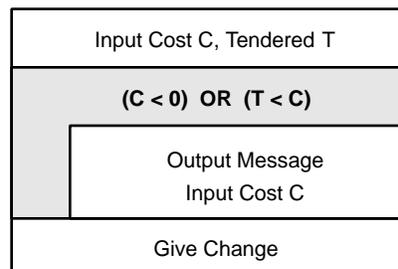
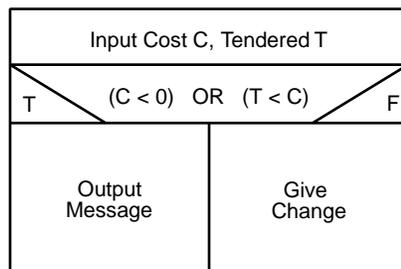


5. Octal: Base 8
 - a. $11 = 8 + 1 = 9$ decimal
 - b. $23 = 2 \times 8 + 3 = 19$
 - c. $132 = 1 \times 64 + 3 \times 8 + 2 = 90$
6. Hex: Base 16
 - a. $11 = 16 + 1 = 17$
 - b. $CC = 12 \times 16 + 12 = 204$
 - c. $F00 = 15 \times 16 \times 16 = 3840$
7. Other Bases

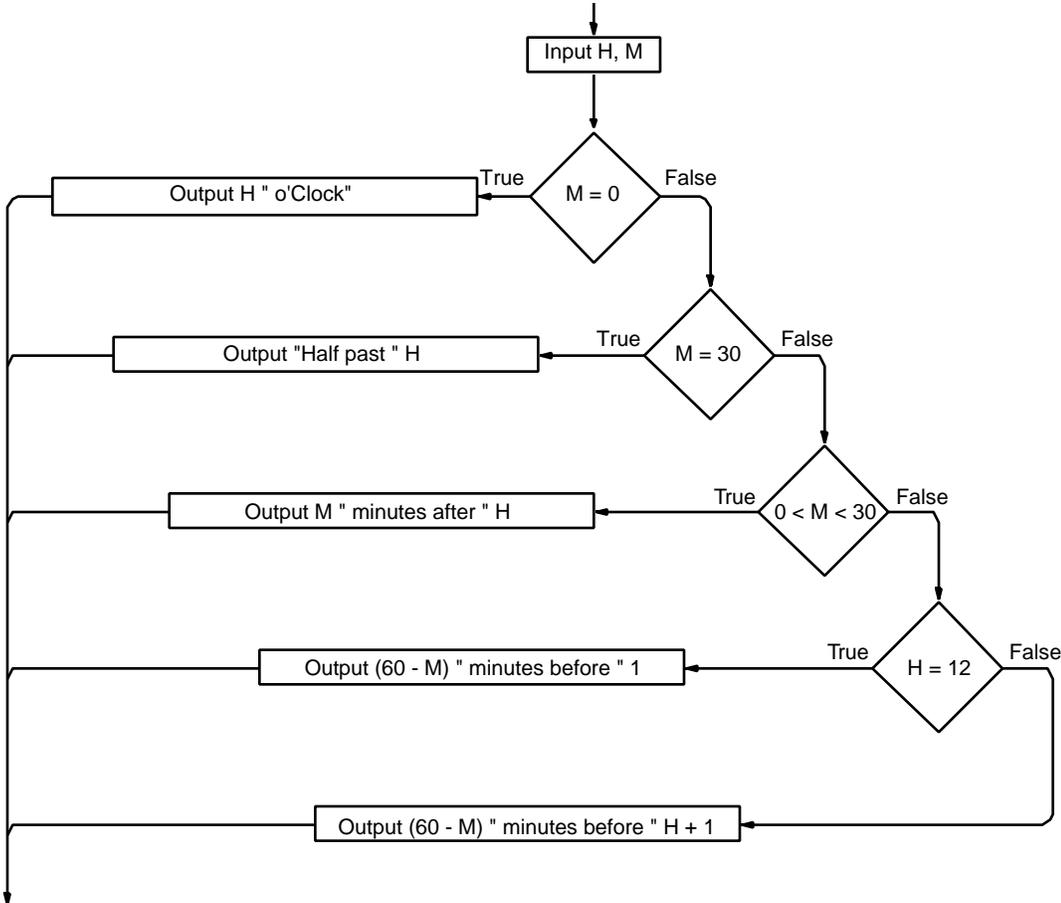
5, 10, 20 are from anatomy
12, 20, 60 have many divisors
(used for time and arc measure)

Chapter 4 Solutions 1-3

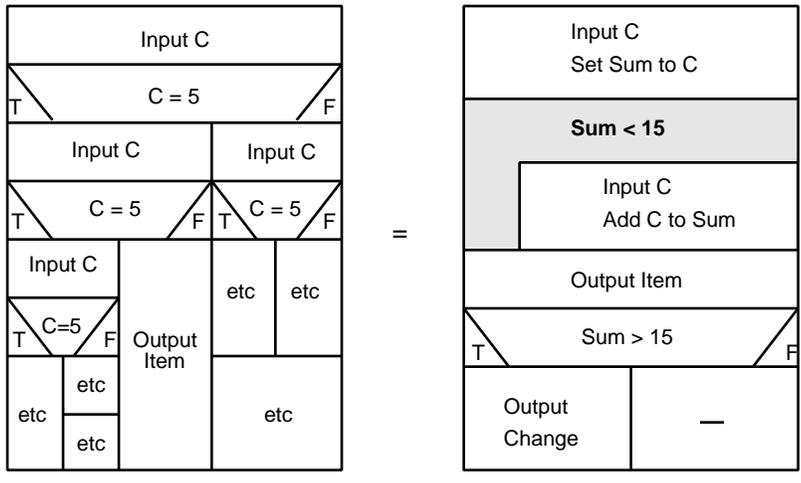
Solution to problem 1



Solution to problem 2

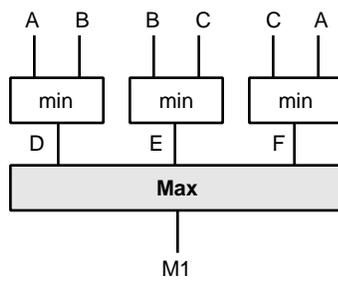


Solution to problem 3



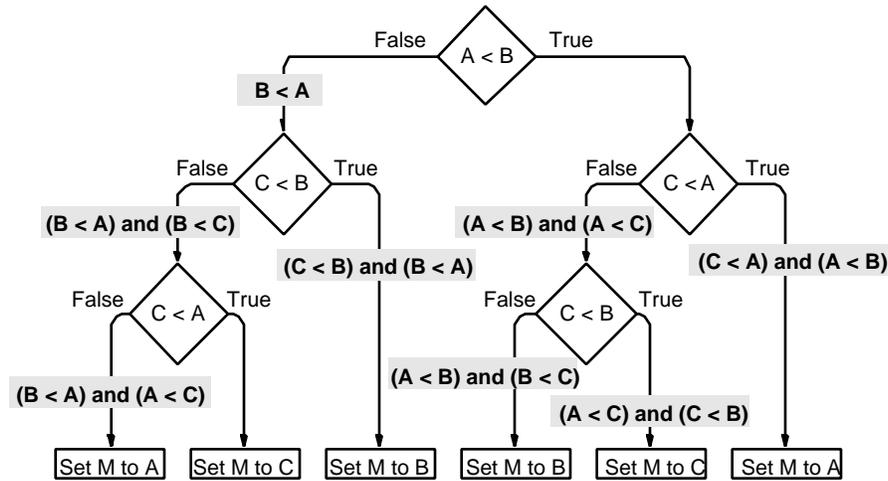
Chapter 5 Solutions 1-3

Solution to problem 1



Conditions	Input	Values	Output
	A B C	D E F	M1
1	A < B < C	1 2 3	1 2 1
2	A < C < B	1 3 2	1 2 1
2	B < A < C	2 1 3	1 1 2
2	B < C < A	2 3 1	2 1 2
2	C < A < B	3 1 2	1 1 2
2	C < B < A	3 2 1	2 1 1
2			

Solution to problem 2



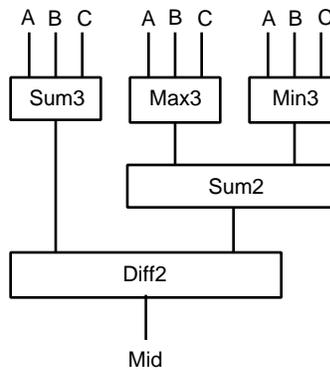
Solution to problem 3

More Mid

```

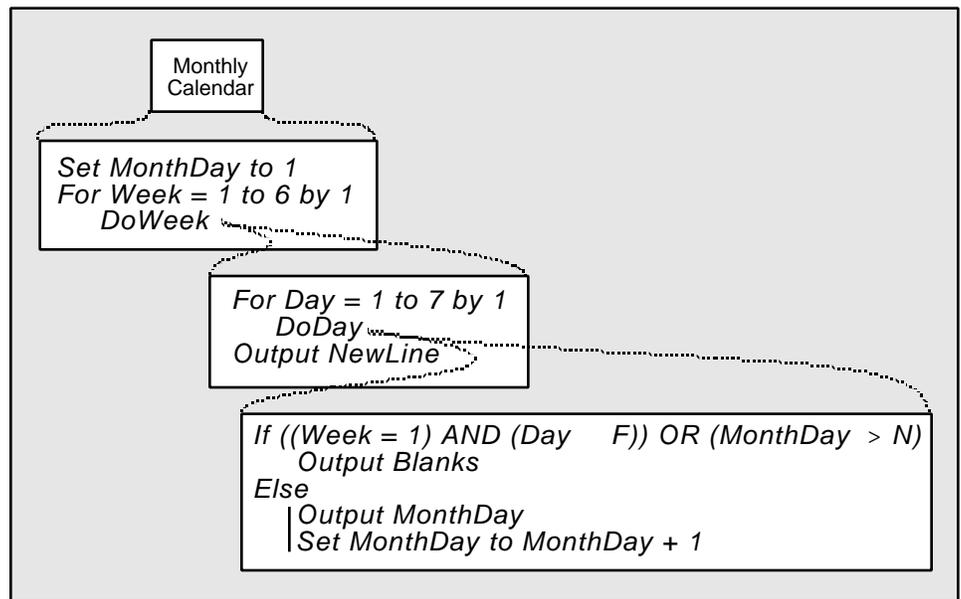
    If (B < A) AND (A < C) OR
      (C < A) AND (A < B)
      Set Mid to A
    Else
      If (A < B) AND (B < C) OR
        (C < B) AND (B < A)
        Set Mid to B
      Else
        Set Mid to C
  
```

Another Mid with data flow

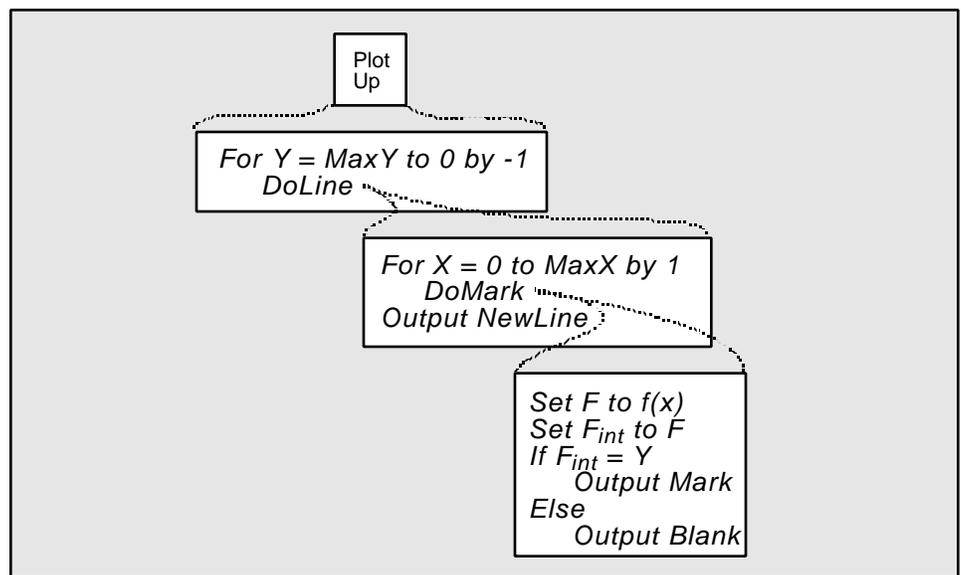


Chapter 6 Solutions 1-3

Solution to problem 1



Solution to problem 2

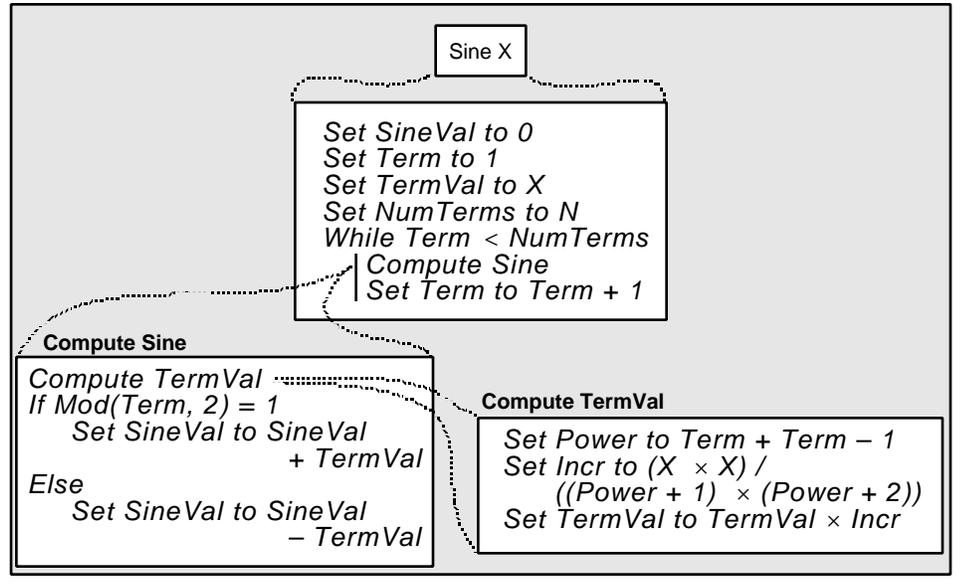


Solution to Problem 3

Sine S =	X	-	$\frac{X^3}{3!}$	+	$\frac{X^5}{5!}$	-	$\frac{X^7}{7!}$	+...-	$\frac{X^P}{P!}$
Term T =	1		2		3		4		N
Power P =	1		3		5		7		P

Vertical View: Loop invariant is $P = 2 \times T - 1 = T + T - 1$.

Horizontal View: Even terms are subtracted from the sum S.



Chapter 7 Solutions 1-3

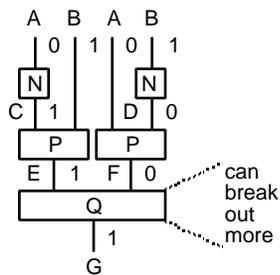
Solution to Problem 1

Here is a definition of divide that involves multiplication: $Num = Den \times Quot + Rem$. Try values of Quot until Remainder is less than the Denominator, where $Rem = Num - Den \times Quot$.

```

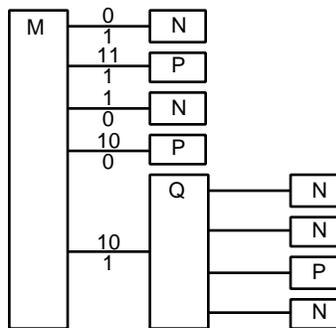
    Divide(Num, Denom, Quot, Rem)
      Set Quot to 0
      Set Prod to 0
      While (Num - Prod) >= Denom
        Set Quot to Quot + 1
        Set Prod to Denom * Quot
      Set Rem to Num - Prod
    End Divide
  
```

Solution to Problem 2

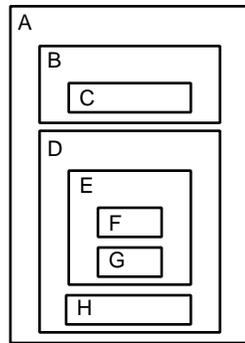


Note:
If variables A, B are binary then N is a NOT, P is an AND, Q is an OR and M is an eXclusive-OR.

Note:
Tree of Calls:
Values above each line are input to the right.
Values below each line are output to the left.



Call Order is N P N P Q N N P N.
For A=0, B=1, the output G is 1.

Solution to Problem 3

Block A can call B and D.
 Block B can call B recursively, D and C.
 Block C can call C.
 Block D can call D, B, E and H.
 Block E can call E, H, F and G.
 Block F can call F and G.
 Block G can call G and F.
 Block H can call H and E.
 X in E can be used in E, F and G.
 If Y is defined in B, D, F then

- Y of B is seen in B and C.
- Y of D is seen in D, E, G and H.
- Y of F is seen in F.
- A does not see any Y.

Chapter 8 Solutions 1-3**Solution to Problem 1**

This algorithm does nothing, but it does it the hard way. First it swaps $A[0]$ with $A[N]$, then $A[1]$ with $A[N-1]$, etc., so it seems to reverse the array. But after reaching the midpoint (when all entries are reversed), it continues to reverse the reversed values (ultimately swapping $A[N]$ with $A[0]$ again). To get a true reverse, the loop should end at $N/2$.

Solution to Problem 2

```

Side Bar Plot
Input Array(A, N)
Input Array(T, S)
For Bars = 1 to N by 1
  Set Length to A[Bars]
  For Thickness = 1 to T by 1
    For Star = 1 to Length by 1
      Output Star
    For Spaces = 1 to S by 1
      Output New Line
  End Side Bar Plot

```

Solution to Problem 3

```

Up Bar
Input A, M, N
For Row = N to zero by -1
  For Bar = 1 to M by 1
    If A[Bar] > Row
      Output Mark
    Else
      Output Space
  Output New Line
End Up Bar

```

Chapter 9 Solutions 1-3**Solution to Problem 1**

There are four ways to do this “Simple Sort”, in about one pass! Such efficiency is not possible in general but here the binary values make it possible. We could use a second array B for the result or do it “in place”.

- a. Count: Loop through A counting the zeros, then loop again putting in that many zeros and then filling the rest with ones.

- b. Select: Loop through A, for each zero in A, put a zero into array B, when done finish B with ones.
- c. Swap: (in place): Position “pointers” at each end of A, advancing them inward and while they do not meet, compare values at these points, swapping when necessary.
- d. Insert: Position “insertion points” in array B at the ends. Loop through A, inserting each value at the appropriate point in B.

Solution to Problem 2

- a. The easy way: Sort the values first and then select the one at the middle index.
- b. Like Count Sort: Loop through values of the array stopping when half the values are less than the given value.
- c. Keep selecting the Max and Min values and delete them until only one value remains.

Index

A

- abstract data type 358, 397, 409
- abstract data types
 - array queues 446
 - array stacks 441
 - implementing 441
 - pointer queues 448
 - pointer stacks 444
- Abstraction 134, 164, 166, 234
- Abstraction Form 130
- Action 65, 119, 128, 165, 166, 176
 - in programs 180
 - on data 180
- actions
 - groups of 302
- Al-Khwarizmi 23
- algorithm 30, 65, 74, 75, 118, 119, 458
 - algebraic 74, 82, 118
 - alternate 52, 164
 - alternative 137
 - analyze 421
 - assertions 146
 - communicating 56
 - comparing 54
 - Complete 76, 119
 - Data-flow diagram 47, 74, 118
 - decomposing 111
 - Deterministic 76, 119
 - dynamic behavior 234
 - dynamic view 176
 - Ease of Use 77
 - Efficiency 77, 198, 232
 - Elegance 77, 119
 - Embedding 51
 - equivalence 53, 135, 164, 187
 - execute 30, 49
 - Execution time 453
 - Extending 48, 131, 186, 257
 - Finite 76
 - flowblock 74, 118
 - Flowchart 45, 74, 118
 - Foolproofing 50
 - Generality 77, 119
 - Generalizing 47
 - Good Structure 77
 - Graphs 46
 - hierarchical 74, 119
 - high-level 31
 - low-level 31
 - modifying 47
 - notations 78
 - Order of complexity 453
 - pseudocode 47, 75, 118
 - refinement 256
 - representations 74, 78, 118
 - reuse 51
 - Robustness 120
 - selecting 79
 - Space efficiency 453
 - static structure 234
 - static view 176
 - tabular 74, 86, 87, 118
 - test 421
 - trace 49, 66, 162, 182, 266
 - Unambiguous 76, 120
 - Verbal 45, 74, 80, 118
 - well-structured 120
- Alias 350
- analysis
 - Count Sort 421
 - Select Sort 427
 - Swap Sort 424
- analytical engine 23
- argument 307, 350
- array 86, 119, 166
 - Binary search 436, 437
 - Bubble Sort 422
 - Count Sort 419
 - deck of cards 158
 - homogeneous items 358
 - indices 358
 - Linear search 436
 - Rank 453
 - Recursive Sort 432
 - searching 435
 - Select Sort 425
 - Sort and copy 417
 - Sort and rank 417
 - Sort in place 418
 - sorting 416
 - Swap Sort 422
- Arrays 358
 - homogeneous 360

artificial intelligence 59, 65
Assertion 166
assertions 119, 157, 215
 Data-flow diagram 157
 pseudocode 157
assignment 180
 equivalence 231

B

Babbage, Charles 23, 59
Backus, John 57
balloon payment 210
Base case 350
Bell Laboratories 58
Bigger algorithms 290
Bigger Blocks 251, 290
bigger data 290
bigger mixtures of forms 290
Binary
 Search 166
binary conversion 159
Binary Counter 268
binary search 437, 477
Binary to Decimal 106, 160
binding 335, 350
 operands 92
bird's eye view 89, 141, 305
Bisection 281, 282, 283
 Search 132, 166
Bisection algorithm 290
black box 75, 89, 119, 305
 "passed in" 91
 "passed out" 91
block structure 300, 348, 350
BOD 41, 65
Boole, George 23, 98
Boolean 179
Bottom-up design 65
Bottom-up testing 468
break-out diagram 39, 65
 alternate forms 44
 Cohesive 43
 complete 143
 Consistent 41
 important questions 143
 Orderly 41
 Refined 42
Bubble Sort 422
 improving 428
buffer 479, 491

C

Case form 259, 262, 290, 291
Character 178, 235
 assignment 196
 comparison 276
Character data type 275, 290
coding 65
cohesion 339, 350
compiler 25
compound conjunction 198
computer 14, 65
 a look at 18
 A to D converter 25
 ALU, arithmetic and logic unit 19, 25
 analog value 25
 basic capabilities 18
 bits 153
 bus 20, 25
 compiler 25
 CPU, central processing unit 20, 25
 D to A converter 25
 execute 14, 25
 four units 18
 hardware 19, 21, 24, 25
 history 21
 I/O devices 20
 inaccuracy 93
 input unit 19, 24
 Input/Output 25
 instruction register 25
 instruction register 19
 Memory 24, 25
 memory bus 21
 memory unit 18, 24
 modem 25
 operations 14
 output unit 19, 24
 problem solving 30
 processing unit 19, 24
 program register 19
 register 25
 software 19, 21, 24, 66
 system 20
concurrency 105
Conjunction 198
Contour diagram 333
 essentials 340
control variable 269, 291
conversational mode 271

Count Sort 418, 419
Coupling 338, 350

D

data 128, 165, 166, 176
 actions 234
 break-out diagram 40, 41
 external 252
 global 301
 heterogeneous 358
 hiding and sharing 332
 homogeneous 358
 structured 153
Data space diagram 309, 322, 324, 348
 conventions 310
 memory locations inside 310
 memory locations outside 310
 passed-in parameter 310
 passed-out parameter 310
Data Structure 166, 458
 array 158
data transfers
 program and subprogram 300
data type 91, 119, 177
 Boolean 179
 character 178
 integer 94, 119
 logical 179, 279
 Real Number 91, 120
 string 179
data validation 35, 65
Data-flow diagram 65, 89, 103, 119
 bigger blocks 105
 black box 47, 101
 concurrency 106
Data-flow diagrams 165
Decimal to Binary 107, 159
decision tables 88, 119
Deep Nesting 133
DeMorgan's laws
 first law 100
 second law 101
DeMorgan's Rules 199
 Data-flow diagram 199
 first rule 199
 second rule 199
Descartes, René 152
design 65
 bottom-up 65, 141
 modular 464

 top-down 66, 128, 140
Dimension 409
Discourse on Method 152
Disjunction 199
divide and conquer 33, 39, 65, 164, 166,
281, 343
documentation 65
 design decisions 37
 method 37
 program listing 37, 66
 seven steps 471
 testing problems 37
 user's manual 37, 472, 488
driver 468, 491
dynamic variable 358, 392, 409
dynamic view 176

E

embedding
 algorithm 65
end-of-file marker 255
environments
 human 20
 physical 21
EOF 255, 291
error
 in algorithm 36
 in coding 36
 in design 36
 in test results 36
execute 65
execution time 416, 421
expression
 arithmetic 91
 logical 99, 119, 280
 polynomial 93
 Time complexity 453
External Data 252, 254, 290, 332
external documentation 471, 491
external unit 463, 491

F

field 359, 409
file 254, 291
 EOF 255
flow of control 64, 65, 74, 89, 119, 303
 pseudocode 115
flowblock 113, 119
 vs. flowchart 135

- flowchart 65, 107, 119
 - Actions 107
 - Assertion 108
 - Decisions 107
 - diamond 45
 - flow of control 46
 - oval box 45
 - square box 45
 - vs. flowblock 135
- foolproofing 65
- For Form 262
 - limitations 264
 - nesting 265
- For loop 252, 262, 290
 - creating plots 273
 - limitations 264
 - vs. While loop 263
- Four Fundamental Forms 108, 128, 166, 176
 - deep nesting 133
- function 300, 350, 463
 - logical 99
- function s 345
- functions
 - conventions 346
 - vs. subprograms 346

G

- glass box 75, 119, 305
- global data 301, 350
- Global variable 336

H

- hardware 19
- heterogeneous 377, 408
- hiding and sharing of data 348
- Hierarchical programming 166
- Hollerith, Herman 23
- homogeneous 377, 408
- Horizontal views 213

I

- I/O devices
 - transducers 21
- identifier 177, 235
- if-then-else 129
- implementation 36, 66

- in situ 418, 453
- infinite loop 207, 235
- input parameter 306, 312
- Insert Sort 418, 428
- Instruction 235
- instruction register 19
- integer 94, 177
- interconnecting forms
 - Nesting 131, 164
 - selections 131
 - Serial 131, 164
- interface specification 306
- internal documentation 471, 491
- International Standard Book Number 278
- Invocation 166
- Invocation Form 109, 130, 164, 176
 - flowblock 114
 - flowchart 114
 - group of actions 130
- invoked 306
- iteration 110, 213, 235

J

- Jacquard, Joseph 23
- Julian date 342, 350
- Julius Caesar 87

K

- Kemeny, John 56
- kludge 208, 235

L

- leaf node 407, 409
- least significant digit 106
- Libraries, Modules 463
- linear search 436
- linked list 393, 409
- Local variables 310
- Logical conditions 198
- Logical data type 279, 290
- logical expression 100, 119
 - equivalency 203
- logical operation
 - Conjunction (AND) 198
 - Disjunction (OR) 199, 279
 - Negation (NOT) 199, 280
- Logical type 179

loop
 initialization of values 259
 loop control variable 264, 267, 291
 loop invariant 176, 214, 234, 235
 complete body of the loop 231
 using 229
 Loop-and-Count form 262
 Loosely coupled 350
 loosely-coupled 338, 339, 349
 Lord Byron 59
 Lovelace, Ada 59

M**N**

n-dimensional list 358, 409
 n-tuple 360, 409
 Nassi-Shneiderman diagrams 113
 Negation 199
 nested selections 257
 Nesting 166
 Nesting Selections
 in Fors 269
 nests of loops 290
 node 407, 409

O

one-dimensional arrays 360
 operand 119
 operating programs 21
 operation
 assignment 180
 comparing values 181
 incrementing 181
 modulus 205
 operations
 on Characters 276
 operator 91, 119
 precedence 92
 operators
 AND 99
 logical 100
 NOT 100
 OR 100
 Order of complexity 453
 output parameter 306, 312

P

Packages 463
 paradigm 186, 235
 parameter 307, 350
 Parameter crossing 336
 Parameter passing 323, 348
 by reference 312, 315
 by value 312, 315
 parameterized block structure 304
 parameters 305
 Pascal, Blaise 23
 Pass by reference 350
 Pass by value 350
 Pattern Matching 438
 Search and Count 438
 Pattern MatchingFind First Match 438
 pointer 392, 409
 Polya, George 31
 precedence 119
 precision 283, 291
 predicate 99, 119
 problem solving 30
 coding 35
 detailed design 38
 development of testing strategy 486
 development of testing strategy
 application 469
 divide and conquer 33, 39
 documentation completion 37, 471,
 488
 documentation completion
 application 472
 general algorithms 281
 Polya's four steps 31
 problem definition 32, 458, 476
 problem definition application 460
 program coding and testing 35, 471,
 488
 program maintenance 37, 474, 489
 program maintenance application 474
 seven step method 32, 63, 458
 solution design 33, 140, 461, 476
 solution design application 461
 solution refinement 74, 464, 478
 solution refinement application 464
 systematic method 32
 task 66
 test 66
 testing algorithms 197

- testing strategy development 35, 466
- using, adapting 38
- procedure 463
- program 35, 66, 177
 - “stored program” 18
 - editor 25
 - errors 36
 - execution 61
 - external data files 254
 - improving 224
 - large 467
 - large programs 62
 - listing 66
 - loop invariant 214
 - medium-sized programs 61
 - modifications 273
 - register 19, 25
 - reuseability 256
 - run 61
 - small 467
 - small programs 60
 - test 66
 - testing 36
 - utility 26
 - variable 66
- program maintenance
 - adding features 38, 474
 - cost 38
 - documentation 38, 474
 - implementation 37
 - large programs 62
 - program errors 38, 474
 - program modification 38, 474
- program register 19
- program structure 318, 348
- programming 30
 - hierarchical 141
 - personalities 61
 - structured 128
- programming language 56, 66
 - Ada 59
 - Algol 60 60
 - APL 60
 - BASIC 56
 - C 58
 - COBOL 60
 - Fortran 57
 - GPSS 60
 - high-level 31, 65
 - Lisp 59
 - Logo 60
 - low-level 31, 66

- machine language 21, 25
 - Modula-2 58, 156
 - Pascal 36, 57, 156
 - PL/I 60
 - Prolog 60
 - Simscript 60
 - Simula 67 60
 - Smalltalk 60
 - Snobol 4 60
 - variable 36
- programming languages
 - Algol 60 333
 - C 333
 - Modula-2 333
 - Pascal 333
- proposition 99, 119
- Pseudo-random number 291
- pseudocode 34, 47, 66, 115, 120
 - assertions 136
 - Extending 257
 - set verb 34

Q

- queue 399, 405, 406, 409, 445, 446
- Queues Unit 486

R

- random value 287, 291
- rank 417
- Real Number 178
 - inaccuracy 93
 - operations on 91
- Records 166, 358
 - heterogeneous elements 358
 - indices 358
- Recursion 300, 343, 349, 350, 432
- recursive definition 217
- Recursive Sort 432
- register
 - instruction 25
 - program 25
- Repeat-Until
 - condition 204
 - loop 204
- Repetition 166
- Repetition Form 109, 120, 130, 164, 176, 234
 - body 130
 - flowblock 115

- flowchart 115
 - For Form 262
 - infinite loop 207
 - initialization 212
 - initialization of values 259
 - initializing loop 110
 - iteration 110, 205
 - loop 109
 - loop body 109, 205
 - nested 132
 - nesting 265
 - Repeat-Until 203
 - side effect 208
 - terminating value 258
 - termination condition 109
 - trace 205, 266
 - While loop 205
 - repetitions 264, 290
 - Roman numerals 78
 - Root node 409
- S**
- scientific notation 91, 178, 235
 - Scope 350
 - Select Form 259, 260, 261, 290, 291
 - Select Sort 418, 425
 - Selection 120, 166
 - Selection Form 108, 120, 129, 164, 176, 185, 234, 280
 - flowblock 114
 - flowchart 114
 - larger 189
 - nested 131, 189, 192, 194
 - proof of equivalence 187, 192
 - selections 257, 290
 - sentinel value 255, 291
 - Sequence 166
 - Sequence Form 108, 120, 129, 164, 176, 182, 234
 - flowblock 114
 - flowchart 114
 - set 409
 - difference 385
 - intersection 385
 - membership 385
 - union 385
 - Sets 358
 - homogeneous 358
 - side effect 208, 235, 350
 - software 19
 - software life cycle 32, 60, 66, 458, 489, 491
 - solution design
 - task 34
 - solution refinement
 - algorithm 34
 - sorting
 - pass 428
 - sorting strategies 416
 - Space efficiency 421
 - specification 194
 - specifications 32, 66, 459
 - speed of operation 198
 - stacks 399, 401, 409
 - State trace 211, 235
 - statement
 - logical 99
 - static view 176
 - stepwise refinement 34, 66
 - strategies 416
 - Strength 350
 - String 179, 361, 399, 409
 - pattern matching 438
 - structure chart 33, 66
 - structured data 166
 - break-out diagrams 153
 - linear form 154
 - structured programming 128, 164, 166
 - stub 467, 491
 - structure of actions 165
 - sub-algorithm 86, 109, 120, 219, 234
 - subproblem 33
 - subprogram 300, 350
 - "parameterless" 316
 - arguments 315
 - binding 335
 - errors 326
 - groups of actions 302
 - input parameter 306
 - interface specification 306
 - invocation 311
 - invoked 219
 - no local variables 316
 - no parameters 316
 - one-way communication 315
 - output parameter 306
 - parameter 300
 - parameter crossing 336
 - parameters only 319
 - recursion 343
 - rules for variable use 318, 334, 348
 - simply large programs 300
 - structure charts 329

- transmitting values 306
- two-way communication 315
- using 110
- variable scope 335
- vs. functions 346

subprogram invocation 303, 307, 309, 348
subscripted variable 358, 360, 409
Swap Sort 418, 422
Symbolic logic 99, 120

T

table 86, 120

- indice 88
- subscript 88
- two-dimensional 88

temporary variable 184
terminal node 407, 409
terminating “sandwich” 256, 290
terminating value 255, 258, 290, 291, 363
test case 35, 66
test data 469
testing strategy 466, 469, 491

- test cases 55

testing strategy development

- test case 35

tightly-coupled 339, 349
time complexity 421, 453
Top-down 166

- design 128, 164
- development 152
- refinement 252
- testing 467

Trace 66, 235

- assertions 214
- Four Fundamental Forms 224
- state 212
- two-dimensional 176, 205, 218

Trace tables 206
tree 399, 407, 409
Trees 449, 477
truth tables 88, 99, 187
Turing, Alan 23
Two-dimensional arrays 367, 408
Two-dimensional plots 273
Two-dimensional tracing 213

U

utility programs 21

V

value

- analog 21, 25
- binary 106
- continuous 21, 65
- counting 178
- decimal 106
- digital 21, 65
- discrete 21, 65
- Integer 119
- measuring 178
- Real Number 120

variable 36, 66, 234

- actions 234
- assigning values 180
- components 177
- data type 177
- global 336
- identifier 177
- incrementing 181
- initialization 212
- logical 279
- scope 335
- temporary 184
- using in subprograms 318, 334, 348

variable access rules 318, 334, 348
vector 360, 362, 409
Vertical views 214
von Leibniz, Gottfried Wilhelm 23
Von Neumann, John 23

W

well-structured algorithm 77
While loop 205, 209
Wirth, Niklaus 57, 58
worm’s eye view 89, 141