How to Think like a Programmer: Problem Solving for the Bewildered

Book · March 2008

CITATION
1

READS
22,363

1 author:

Paul Vickers
Northumbria University
121 PUBLICATIONS **889** CITATIONS

SEE PROFILE

Paul Vickers

How to THINK Like a Programmer

Problem Solving for the Bewildered

Visit the website at: **www.cengage.co.uk/vickers**

COURSE TECHNOLOGY
CENGAGE Learning™

How to Think Like a Programmer: Problem Solving for the Bewildered
Paul Vickers

Publishing Director: John Yates
Publisher: Gaynor Redvers-Mutton
Editorial Assistant: Matthew Lane
Production Manager: Alissa Chappell
Senior Production Controller: Maeve Healy
Manufacturing Manager: Helen Mason
Marketing Manager: Jason Bennett
Typesetter: Integra, India
Cover design: Jackie Wrout

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

ISBN: 978-1-84480-900-4

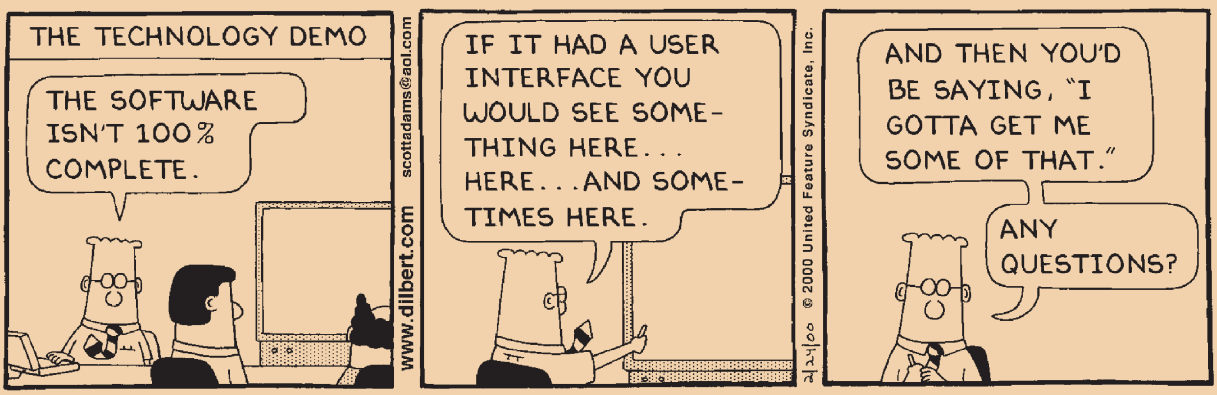
Cengage Learning EMEA
High Holborn House,
50-51 Bedford Row
London WC1R 4LR

Cengage Learning products are represented in Canada by Nelson Education Ltd.

For your lifelong learning solutions, visit **www.cengage.co.uk** and **www.course.cengage.com**

Printed by C&C Offset Printing, Hong Kong
1 2 3 4 5 6 7 8 9 10 – 10 09 08

1 Introduction: Starting to Think Like a Programmer



Learning Objectives

- Understand how to use the book and its special features
- Understand how programs are structured recipes (algorithms) to calculate/compute the answer to a given problem
- See how using abstraction is necessary for solving problems
- Understand the difference between solving problems and writing computer programming language code

This chapter serves two purposes. First, it describes for whom the book is intended, how the book is structured, and how to use it. Secondly, you will learn what a computer program is, why programmers write programs, and what they use to write those programs. You will discover the difference between writing program code and solving problems and you will learn why good programmers are, primarily, good problem solvers.

1.1 Who Is This Book For?

This book has two main audiences. The first audience is those who are taking an introductory programming or computer science course in which the first four to six weeks are spent developing the skills necessary to think like a programmer (algorithmically) and the course follows a more traditional programming language text.[1] The second audience is the "bewildered" programmer. If you can identify yourself in one or more of the following descriptions, then this book is also intended for you:

- You have just started to learn a programming language at a university or college. You are only a little way into the course, but already you are starting to feel lost and panicky and may even be falling behind.
- You have tried to learn programming and have come away feeling that it is terribly difficult. You have either fallen at the first hurdle or have finished an introductory course, but in either case you are left feeling bewildered with a sense that you never really understood it. If someone were to ask you which aspects of the subject you were having particular difficulty with, you would reply "all of it".
- You have not learned any programming before, but it is a mandatory part of your university or college degree (which may not even be in Computer Science or Engineering). You feel anxious about it.
- You are taking, or are about to take, an introductory programming course. You have had a look at the set textbook and even the first few chapters seem too advanced for you.
- You are a secondary/high school student and you need an introductory book to get you started with the basics.
- You are a mature reader. You are not necessarily on a formal course of study (though you may be) and you would like to find out what computer programming is about. Perhaps you have had a look at some other books and they all seem too advanced, too technical (even the introductory ones). You do not think you are a dummy or even an idiot, but you would like to see if you can get a foothold on what appears to be an interesting subject.

If you have identified yourself in the above list then continue reading. If you have not, continue reading anyway so that you can recommend this book to people you know who need to read it (and who can then stop bugging you for help!).

1.2 Conventions and Learning Aids Used in This Book

This book uses a number of techniques that have been designed to help you get the most out of its content. These are explained below.

Think Spots

The Think Spot is a point in the text where a question (or a number of questions) is raised for you to think about. To get the most benefit you should take

[1] If this book is being used at pre college/university level then it might be used over an entire term or semester.

a little time to think about the questions rather than just reading them and moving straight on. The famous Kodak Picture Spot signs at Disney World are sited at places where a great photograph can be taken; similarly, the Think Spot is located at points where you will really benefit from some reflective thinking and so develop the bigger picture.

In-text Exercises

Throughout the book you will see a picture of a pencil in the margins alongside some text in a shaded box denoting a short exercise:

This is a short exercise.

As these exercises are intended to help you understand a particular concept, you should not really proceed beyond an exercise until you have had a reasonable attempt at solving it. The exercises are like the Think Spots inasmuch as you need to stop and think, but unlike the Think Spots, you also need to physically do something such as sketching a solution, walking through a problem, or discussing something with a friend.

If an in-text exercise looks like this with a key in the margin, then a solution is also available in Appendix C. Just go to the In-text exercises section for the corresponding chapter in Appendix C and then look for the number that appears next to the key.

This is a short exercise with a solution available in Appendix C.

Key Terms and Important Points

Key terms ▶

Key terms and important points appear in the margin next to the paragraph or section in which they are introduced or defined. Together with the index this feature should make it easier for you to find what you are looking for.

Brian Wildebeest FAQ

Meet Mr B. Wildebeest. Brian is a wildebeest (also known as a 'gnu' and pronounced 'will-dur-beast') and usually he is happy.

However, sometimes Brian is a bewildered wildebeest at which times he looks like this:

Throughout the book you will see pictures of Brian looking bewildered in the margin. He appears at points where beginning programmers often have trouble understanding the point being made. Further down the page (or possibly on the next page) after seeing Brian you will find a box like this:

I don't understand why you said . . .

This is a common misunderstanding which arises from . . .

In the box is a question Brian the bewildered programmer is asking about the material next to which his picture appeared. Brian's queries are the kinds of frequently-asked questions (FAQ) I have been asked over the years by beginning programmers. The answer to the FAQ usually appears in the box below the question, though sometimes you are required to try to answer Brian's FAQ yourself.

The book's accompanying website (at www.cengage.co.uk/vickers) contains a section where readers can submit their own FAQs seeking answers to issues that still cause puzzlement. I would like to encourage readers to submit their own FAQs and I will then provide answers to these on the website as appropriate.

End-of-chapter Exercises

Each chapter has exercises at the end. The exercises are designed to help you reflect on what has been covered in that chapter. Any time you cannot complete an exercise suggests that you would benefit from going over the material again. Some exercises may be based on a single section (identified by the heading number), others may require ideas from several sections, and a few bring together the whole chapter. Solutions to selected exercises are given in Appendix C.

Projects

After the normal end-of-chapter exercises you will find some longer project-style exercises. These longer exercises are themed and will give you practice in incrementally building larger and larger solutions to more complex problems. The initial themes are introduced in the exercises at the end of this chapter and are then developed with each subsequent chapter. In these projects, you will be working toward developing complete programs that make use of most of the programming techniques discussed in this book. The projects cover a range of different problem scenarios, such as constructing a vending machine algorithm, decoding hazchem

signs, working with Roman numerals and dates, playing with sentences that use all the letters of the alphabet, and working with ISBNs for an online bookstore.

Layout

All programming language code, pseudo-code (Chapter 3), examples of text that would appear on a computer screen or in a file on the computer's hard disc (or removable diskette), and examples of text that would be entered into a computer via the keyboard will appear in `this monospaced typewriter-style font`.

Reflections

Sometimes you will see a word or phrase set in SMALL CAPITAL LETTERS. Such words and phrases are the titles of short reflective opinion pieces that appear in the Reflections chapter (immediately following Chapter 8). These *Reflections* are designed to introduce some more complicated ideas that the interested reader can use to deepen their understanding of some of the problems and issues faced by programmers today.

1.3 Why Do We Write Programs and What *Are* They?

People write programs for a number of reasons. Authors like to use computers to help with the jobs of typing in and laying out text, checking their spelling, and so forth. Scientists and mathematicians use computers to calculate the answers to complex problems. Musicians and composers use the signal processing capabilities of computers to manipulate sound. In all cases, before the computer can carry out these tasks we must first tell it how to perform them. We do this by first solving the problem of deciding how the task can be carried out, and then expressing that solution in a form that can be turned into something the computer can interpret. The first stage (deciding how the task can be carried out) is the really important part as without doing this well we cannot progress to feeding the information into the computer.

Many beginners approach computer programming with a sense of awe, as if a computer program is some mystical artefact that only those initiated into the secret and black arts of computer science can produce. It is true that some programs are phenomenally complex comprising tens of millions of lines of programming language code.[2] But, at its heart a computer program is nothing more than a sequence of instructions to tell a machine (the computer) to do something specific.[3] The word processor I used to write this book is a program running on my home computer. When I use an ATM (cash machine) I am interacting with a program that decides whether there is enough money in my account to meet my

[2] It has been estimated that some of today's very large computer systems are the most complex artefacts ever built.

[3] Actually, the program may itself be considered a machine, only one built from logic rather than metal and plastic. For a clear and concise discussion of the program as machine, see *Software Requirements and Specifications* (Jackson, 1995).

withdrawal request. When I select a hot wash to get my cotton shirts really clean I am telling my washing machine to use the program (set of instructions) that will draw and heat enough water, release the detergent at the right time, rinse with cool water, and so on.

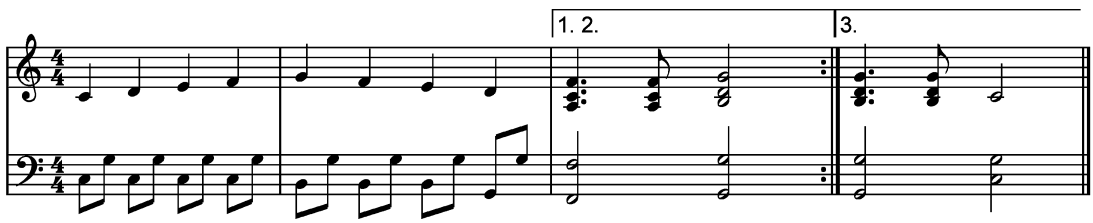
If you think about it, a computer program is in many ways just like a recipe. If you have ever cooked a meal then you will recall having to carry out particular tasks in a certain order. Even something as simple as making buttered toast or muffins requires that you spread the butter *after* the bread/muffin has come out of the toaster. You successfully make buttered toast or muffins (and keep your toaster in good working order) when you carry out the steps in the right order.

If you have never done so, find a cookery book and look at some of the recipes. You will see all the things you have to do in order to prepare various meals.

A good recipe is one that clearly sets out all that you have to do and when, that gives precise quantities for the ingredients, and that tells you what temperature to use in the oven and for how long to cook the dish. If the instructions are well set out then it should be possible for anyone to follow them and produce the desired result.

A musical score is a bit like a program too. Over hundreds of years musicians and composers have developed formalized languages of notation that allow musical instructions to be communicated on paper. A music score indicates all the notes to be played including their durations and volumes. Sections of the score can be marked for repetition including alternate endings to repeated phrases. Other marks tell the musician to speed up, slow down, pause, play louder, play more softly, etc. As long as a musician knows how to read and interpret a score then he or she can play music written by somebody else. The score in Figure 1.1 is presented in Western musical notation and shows a simple piece of piano music. It has a repeated section and alternative endings for the repeated section.

FIGURE 1.1 A simple musical score: In this piece of music the first two bars are played through three times. The first two times they are followed by the music in Bar 3, and by the fourth bar after the third play through. The dots before the third bar line indicate that the previous section is to be repeated. The brackets with numbers indicate what should be played and how many times.



1.4 Teaching Approach

What is needed by the novice programmer before all else is an understanding of the processes involved in examining and analyzing problems, in understanding the component parts of a problem, and in understanding what is required. You, the learner, then need to be able to solve the problem and check your solution for mistakes, inconsistencies, and limitations. Then you must be able to write down your plan for solving the problem in such a way that your solution can be repeatedly applied to the problem even when some of its values are altered.[4] Once this ability is acquired then, and only then, should you concern yourself with the details of programming languages. There is little use trying to write programs in programming language code if you do not first know how to solve the underlying problem, or worse, even understand the problem you have been asked to write a program to solve.

The philosophy behind this book is that before we can write programs on a computer we must first learn to think like a programmer. This can be approached in three stages:

1. Problem understanding and problem solving
2. Writing solutions in a structured form (algorithms)
3. Writing algorithms in a programming language

This book deals with Stages 1 and 2 for it is only after achieving competence in these stages that one should approach a real programming language. Before starting to think about writing any programming code, we must first focus on the problem statement, that is, on the *real-world domain*. First we must *understand* the problem and then we try *solving* it. Once we think we have solved it we *systematize* our solution by writing it out in a more formal way as a series of steps (an *algorithm*) that can be followed by another person.

Algorithms

Algorithm is a common word in programming circles. An algorithm is a rule, or a finite set of steps, for solving a mathematical problem. In computing it means a set of procedures for solving a problem or computing a result. The word *algorithm* is a derivation of Al-Khwarizmi (native of Khwarizm), the name given to the ninth-century mathematician Abu Ja'far Mohammed ben Musa who came from Khwarizm (modern day Khiva in the south of Uzbekistan). Thus, this book is about learning how to understand problems and design algorithms that are solutions to those problems.

Abstraction: Taking a Higher View

One thing all programmers do, whether they realize it or not, is use something called ABSTRACTION. Abstraction happens when we view something in general

[4] For example, imagine if the problem were to calculate the monthly payments on a loan. Your solution should work for all the possible values of loan amount, loan duration, interest rate, and so on, and not just for the example numbers given in the problem specification.

FIGURE 1.2 **Abu Ja'far Mohammed, aka Al-Khwarizmi after whom the algorithm is named.**

terms without focusing on its concrete details. For example, if you talk about driving in your car, the word *car* is really an abstraction for the specific individual car you drive. Your car will be different from my car. Even two cars of the same make, model, year, and specification are still different from each other inasmuch as they are both *individuals. Person* is an abstraction, as are *man, woman, child, boy, girl,* and so on. We all use abstraction in our everyday lives; indeed, without it we would not be able to function for it enables us to ignore all the fine details that would otherwise overwhelm us. The money in my pocket (how much, what currency, how many coins, what year were they minted, how many notes, their serial numbers), the people in the shop (how many men, women, boys, girls, what are their names, nationalities, ethnic groups, ages, heights, educational qualifications, first languages, etc.), and the stars in the sky are all abstract ways of managing an otherwise unmanageable amount of information.

Programmers use abstraction as a way of simplifying and managing detail. However, unlike most of our everyday abstractions, programmers do not actually ignore the detail, instead they *defer* its consideration. At some point the detail will need to be considered. Part of being a programmer is learning how to juggle abstractions, ignoring the fine detail when it is appropriate to do so.

This book follows the practice of dealing with **control abstraction** and **data abstraction** separately. The algorithms you will learn to build in this book **control** and manipulate **data** in order to produce desired results. When you withdraw money from your bank you are performing control (the sequence of actions necessary to withdraw the money) to manipulate data (the amount of money in your account, the date it was withdrawn, and so forth). Regarding data, this book takes a highly abstract view treating all the data in the problems it presents simply as values. Our control abstractions take a fairly general form in the beginning but as the book progresses the level of abstraction is lowered as we consider more specialized ways of performing actions.

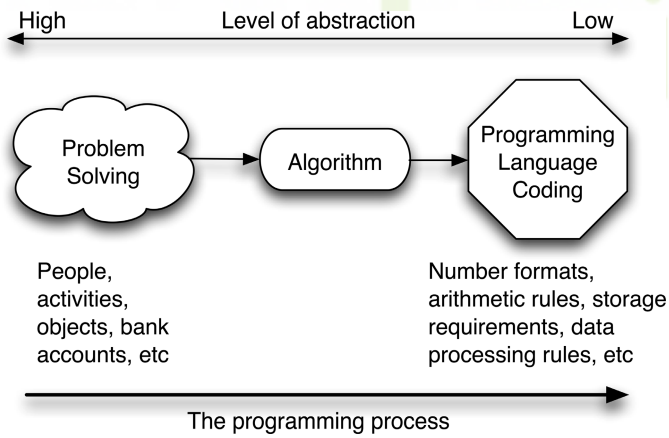
Heuristic

This book takes a HEURISTIC approach to solving problems and expressing those solutions as algorithms. Having solved the problem and produced a corresponding algorithm, the programmer would then translate the algorithm into programming language code (a lower level of abstraction). When presented with an algorithm in a computer programming language such as Java or BASIC, the computer can carry out the instructions in the algorithm many millions of times faster (and reliably and accurately) than any person could.

Most books jump right away into the specific requirements of a given programming language and the learner will, through no fault of his or her own, associate the art of programming with writing instructions in a programming language. But using a programming language is one of the final steps in the process of writing a program. This book focuses on teaching you to concentrate on the most important stages: understanding the problem at hand and solving it algorithmically.

Too many beginning programmers blend problem solving with coding and treat them as one activity and then (reasonably) see programming as hard. Problem solving requires thinking about the problem at a high level of abstraction while writing programming language code requires a very low level of abstraction. Inevitably, the learner starts trying to apply the very low level of abstractions in their thinking about the problem. In fact, if problem solving and coding are separated we discover that the coding aspects are reasonably straightforward while it is really problem solving where the difficulty lies.

FIGURE 1.3 Abstraction in the programming process



One skill you will develop as a programmer is being able to think in terms of high-level abstractions (understanding and thinking about the problem at hand) and in terms of low level abstractions (individual data items, their format, and their status) simultaneously. However, having witnessed the confusion that can arise when a beginner is asked to do this from the very beginning, I decided in this book to make a clear separation between the high-level problem-solving skills and the low-level language-coding skills. Once you have become comfortable in approaching problems and producing algorithmic solutions, it is then time to think about translating the algorithms into a programming language. This book

unashamedly deals with the high-level abstraction and leaves the translation exercise to other books that deal with specific programming languages. Eventually, after learning how to solve problems and then how to translate algorithms into programming language code you will find yourself able to mix the low-level abstractions with the high-level ones and the boundary between problem solving and writing in the chosen programming language will become more fluid.

Getting the Most from This Book

To get the most out of the book (and especially the exercises) you will find it helpful to get a willing friend or relative who can try out your solutions. If your friend can follow your instructions without seeking clarification from you and, using your instructions, can successfully complete the task or calculate the right answers, then you have begun to grasp problem solving and solution description. What you will have done is solve the general problem and create a set of steps and instructions that, when followed exactly, will allow you (or anybody else using the instructions) to solve any specific problem of the same type. In fact, the instructions are the solution to the problem and anyone using them no longer has to solve the problem, they simply have to follow some steps to calculate (or compute) the required answer.

1.5 Structure of the Book

The book is set out in the following way. In Chapter 2 we will look at what is meant by problem solving and how learning to solve problems will help us to become computer programmers. We will learn how to apply a structured strategy for problem solving. This strategy has steps dealing with understanding and describing the problem, planning how to solve it, and testing the solution.

Chapter 3 introduces different ways to think about problems and provides a form of structured English called *pseudo-code* that we will use to write down our algorithms (solutions to problems).

Chapters 4 and 5 are concerned with a few real-world problems that are used to develop skills in problem analysis and solution. The complexity of the problems gradually increases during the course of the two chapters to allow the introduction of basic programming concepts and techniques that you need in order to be able to think like a programmer, to solve problems, and to write down systematic solutions.

Chapter 6 takes our basic problem-solving skills and adds some more specialized vocabulary with which we will develop a repertoire of standard structures for implementing solutions to common programming problems.

Chapter 7 takes a small excursion into a different way of approaching programming and problem solving. Today object-oriented programming is very common and this chapter introduces some of the very basic concepts of this approach and shows how we can begin to think in object-oriented terms.

Solving the problem is the process of deriving a correct set of instructions that would enable us (or our friend) to calculate the right answer every time. By writing the instructions down in such a way that they are unambiguous and can be easily followed by anybody reading them, you have created an effective solution. If your solution does not involve unnecessary steps and is easily followed, then you have a good solution. A bad solution would be one that is hard to follow or

is clumsy, or both (though this is a general rule that does not always apply – sometimes speed of execution is more important than elegance or comprehensibility). Therefore, in Chapter 8 we will look at some of the techniques available to you for designing good programs that are, after all, just solutions to problems.

Pseudo-code: An Algorithmic Language

As you probably have observed, people tend to use natural language (especially spoken language) imprecisely and meanings are often ambiguous.[5] Imagine the friend you have chosen to follow the instructions you will be writing down as you work through this book is unspeakably stupid and speaks only one language that has about thirty words in it. He is unable to interpret vague or woolly instructions and will reject any instructions that do not conform to the exact grammatical rules of his own, small language. Furthermore, he will slavishly obey everything you say to the letter. Imagine how careful you would have to be to write down your solution exactly right so that your friend could understand it and carry out your wishes. That is precisely how it is with a computer that cannot think for itself. For this reason, although this book does not deal with an actual programming language, it does use a form of structured English (called *pseudo-code*) for expressing solutions to problems. The pseudo-code used in this book is first introduced in Chapter 3. Learning to use pseudo-code provides a solid foundation for making the move to a computer programming language later on.

1.6 Coding Versus Problem Solving

We can say that a recipe is a solution to the problem of preparing a meal. Likewise, a computer program is a description of a solution to a computational or logic problem that is carried out by a computer rather than a person. The problem may be to work out the amount of tax we owe or to calculate the average mark of a university student. The problem may even be something as general as allowing a person to enter text, amend it, move it around, apply various formatting to it, save it, and print it out (just what a word processor does). However, in all these cases the program is the series of steps that, if followed, will lead to the desired outcome (assuming the program was correctly written). However, notice that the program does not actually solve the problem at hand,[6] rather it *calculates* (or computes) the *answer* to

[5] Take the following anonymous book dedication: "to my parents, George Pólya and God." What does that mean? Surely the author is not claiming that God is one of his parents? The sentence is syntactically (grammatically) correct though its meaning can be misunderstood. The addition of an extra comma makes things much clearer: "to my parents, George Pólya, and God." Most people are taught not to put a comma after the element that precedes the 'and' in a list, yet in this case adding the *serial comma* really helps to make the author's meaning crystal clear. You may be interested to know that the serial comma is also known as the Oxford comma as it is a stylistic practice of the Oxford University Press (OUP). The OUP uses it precisely because it removes ambiguity from lists. Most people do not use it, but I do. If you look closely you will see that it is used throughout this book. You hadn't noticed? Shame on you, programmers need to have an eye for detail you know.

[6] Certain problems in mathematics and engineering excepted.

the problem. That is, it calculates the correct tax or it correctly stores the text entered by the user of the word processor. The solution to the problem is provided by **you**, the programmer. It is you who solves the problem by deciding the correct series of instructions that, when followed, result in the desired outcome. The process of solving the problem is really the essence of computer programming. Many people are fooled into thinking that writing programming language code is what defines programming. Not so. Writing the code is merely the stage of expressing the solution to the problem in a way that it can be communicated to the computer. Once we have the solution, correctly expressing it in the chosen programming language does take skill and experience, but to be able to write the program code we must first solve the problem. Moreover, before we can solve the problem we must first understand it. It is one thing to try to write the program code for a problem we understand but have not completely solved yet (though this is still bad practice); it is quite another thing to try to write the code for a problem we do not even understand. Explaining tasks to a person and to a computer is essentially the same; the difference is the required level of precision, or un-ambiguity (abstraction), in the language used.

1.7 Chapter Summary

In addition to learning how this book is organized and the various ways it can be read, we also discussed what computer programs are, why we write them, and how good problem-solving skills are essential for successful programming. The programming process can be thought of as a two-stage activity: in the first stage, we work on understanding and solving the underlying problem. When that is accomplished we proceed to Stage 2, which is translating the problem solution into programming language code ready for compilation and execution on the computer.

Many beginners go astray because they start at Stage 2, that is, they read the problem definition and immediately try writing programming code. This is a major cause of bewilderment, hence the emphasis in this book is on doing Stage 1 properly. Before moving on to discussing the nature of problem solving in the next chapter, try the exercises below to make sure you have grasped the ideas presented in this chapter.

1.8 Exercises

1. What is an algorithm?
2. What is a program? Try giving an answer in no more than thirty words. Do you know someone who is very poor at understanding technology (usually they cannot program their video recorder or set the stations on their car radio)? If so, how would you explain to him or her what a computer program is?
3. Describe your wallet using three different levels of abstraction: low (as many details as you can think of), medium (the main points), and high (identifying characteristics only).
4. *Put on my shoes* is a highly abstract description of a common task. To describe the task at a lower level of abstraction requires some details to be known. Jot down some of the

main pieces of information that are needed to be able to describe step-by-step the process of putting on a pair of shoes.

5. Learning to write Java/C/Visual Basic/programming language of your choice is not the same thing as learning to program. Why not?

6. What were the causes of the First World War?

1.9 Projects

Below are the four themed projects that will be used to develop your programming skills throughout the rest of the book. After each set of end-of-chapter exercises you will find additional exercises related to one or more of these projects. As you progress through the book you will find yourself extending your solutions until you have outline algorithms for complete programs. Your task for this chapter is to read through the project descriptions and familiarize yourself with their contents. Try to identify what problems might exist.

StockSnackz Vending Machine

The University of Stocksfield has installed a StockSnackz brand vending machine in its staff common room for the benefit of the faculty. The University places a high value on its staff so the machine dispenses free snacks including chocolate, muesli bars, apples, popcorn, and cheese puffs. Drawing upon your own experiences of using vending machines, think about the problems associated with maintaining the StockSnackz machine: How will the user select an item? Which items should be dispensed? What happens when the machine runs out of an item? What information does the machine owner need to know about the number of dispensed snacks? If the snacks were not free, how is money taken and change given?

Stocksfield Fire Service: Hazchem Signs

Attached to the back of trucks transporting chemicals in many countries you will find a hazchem sign (Figure 1.4).The three-character code at the top is the EAC, or Emergency Action Code, which tells firefighters how to deal with a chemical spillage and fire.

FIGURE 1.4 **Hazchem sign**

The first character of the EAC is a number identifying the method to be used for fighting any fire. The second character is a letter identifying the safety precautions to be taken by firefighters, whether a violent or explosive reaction is possible, and whether to dilute or contain any spill. The third character is either blank or an E indicating the existence of a public safety hazard. The four-digit code is the United Nations substance identification number that is used to find out the exact name of the chemical. The hazard warning diamond gives specific information about the nature of the hazard. Table 1.1 shows how to decode the EAC.

Table 1.1 Emergency Action Code: required firefighting methods and precautions

1	Coarse spray	**3**	Foam
2	Fine spray	**4**	Dry agent

P	**V**	LTS(CPC)	Dilute spillage
R			
S	**V**	BA & fire kit	
T			
W	**V**	LTS(CPC)	Contain spillage
X			
Y	**V**	BA & fire kit	
Z			
E	Public safety hazard		

Key
V = Can be violently or explosively reactive
BA = Breathing apparatus
LTS = Liquid tight Suit/Chemical Protection Suit and BA required
DILUTE = Spillage may be washed away when greatly diluted with large quantities of water.
CONTAIN = Spillage must not enter water courses or drains.
DRY AGENT = Water must not be allowed to contact substance.

What patterns can you see in Table 1.1? How might these patterns help in solving problems related to the decoding of an Emergency Action Code?

Puzzle world: Roman Numerals and Chronograms

We express numbers in base 10 using digits derived from a Hindu-Arabic system. Arithmetic is straightforward in this system. However, consider the Roman

Empire that had an altogether different numbering system. In the Roman system, numbers are represented by combinations of the primitives given in Table 1.2 below. The number 51 is written as LI, the number 1,500 is written as MD, and so on. Further, the numbers 4, 9, 40, 90, 400, and 900 are written as IV, IX, XL, XC, CD, and CM respectively. Thus, 14 is XIV, 99 is XCIX, etc. (What is common to the numbers 9, 40, 90, and 900?). In this system, the year 1999 would be written as MCMXCIX and the year 2007 as MMVII.

Table 1.2 The basic Roman Numerals

Roman Numeral	Decimal Equivalent
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

As you can imagine, arithmetic is not so simple using such numbers. For example, consider the simple sum 1,999 + 2,007 using Roman numerals:

MCMCXIX + MMVII = ?

The answer is MMMMVI. Why do we need to know about Roman numerals today? The media industry still uses them. TV shows have the year of production expressed in Roman numerals, as do some movies, books, and so on. The pages in the front matter of books (before the first chapter) are numbered using Roman numerals with Arabic digits being reserved for the main body (look at the page numbers for the preface in this book).

Imagine you are writing software for a media production company that needs a reliable way of dealing with translation of dates into Roman numerals. What sorts of problems might you face in converting between decimal numbers and Roman numerals?

Pangrams: Holoalphabetic Sentences

Pangrams are holoalphabetic, that is words or sentences that contain every letter in the alphabet. Here is a famous one used by computers to show how text looks in different fonts: *The quick brown fox jumps over the lazy dog*. Pangrams are useful in digital typography because they demonstrate all the letters in a font within a more meaningful context than just writing the alphabet – the interactions between the letters are also easier to see.

The "perfect" pangram is *isogrammatic*, that is, it uses each letter only once. It is extremely hard to produce meaningful isogrammatic pangrams in English. For example, here is one that uses only 26 letters:

Quartz glyph job vex'd cwm finks.

It is not terribly meaningful even if they are all real words. Most pangrams are not isogrammatic, so the next goal is to make them as close to being isogrammatic as possible. Here are some more pangrams with their letter count shown in parentheses.

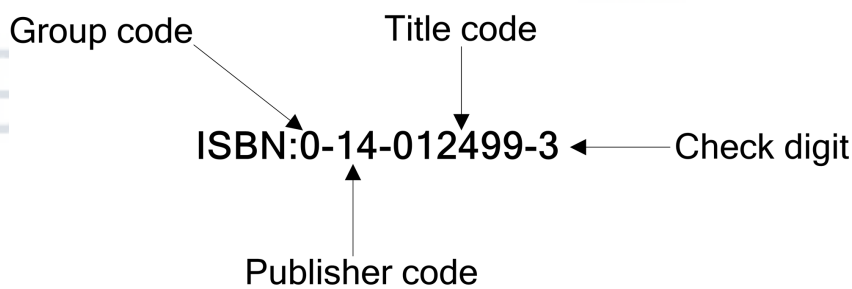
Pack my box with five dozen liquor jugs (32, *e, i,* and *o* repeated).
Waltz, bad nymph, for quick jigs vex (28, only *a* and *i* repeated – not as meaningful though).
Six plump boys guzzled cheap raw vodka quite joyfully (46).
Sympathizing would fix Quaker objectives (36).
Quick waxy bugs jump the frozen veldt (31).
Brick quiz whangs jumpy veldt fox (27).

Think about what problems exist in constructing a pangram and in determining whether a sentence is a pangram. If it is, determine if it is isogrammatic.

Online Bookstore: ISBNs

The International Standard Book Number (ISBN) is a unique 10- or 13-digit number used to identify books. The system was invented in 1966 (then simply called SBN – Standard Book Numbering) by W.H. Smith (the U.K. bookseller and stationer) and was adopted as an international standard (ISO 2108) in 1970. Beginning in January 2007, ISBNs will have 13 rather than 10 digits.

FIGURE 1.5 Anatomy of the International Standard Book Number (ISBN)



The number comprises four parts:

1. The country of origin or language code (called the group code)
2. The publisher's code
3. A number for the book title
4. A check digit

The different parts can have different lengths and usually are printed with hyphens separating the blocks (the hyphens are not part of the number). The check digit is introduced to ensure that the previous nine digits have been correctly transcribed. It can be a digit (0–9) or the character 'X' (representing the value 10 – it is not necessary yet to understand how the check digit is calculated).

Until January 2007 all ISBNs were 10 digits. A new 13-digit format was introduced in January 2007 (known as ISBN-13 or "Bookland EAN"). All 10-digit

ISBNs can be converted to ISBN-13 by adding a prefix of 978 and recalculating the check digit. The 10-digit ISBN 0-14-012499-3 becomes 978-0-14-012499-6 and 0-003-22371-X becomes 978-0-003-22371-2. In the following chapters, you will find exercises focusing on three specific problems related to ISBNs:

1. Validating an ISBN (checking it is correct)
2. Converting a 10-digit ISBN to ISBN-13 format
3. Displaying a raw ISBN such as 0140124993 with the correct hyphenation; Table 1.3 shows correct hyphenations for a few ISBNs

Table 1.3 Hyphenating an ISBN

Raw ISBN	Hyphenated ISBN	Book Title
0140124993	0-14-012499-3	How to Solve It
999361419X	99936-14-19-X	Gross National Happiness and Development.[7]
8466605037	84-666-0503-7	Los Simpson ¡Por Siempre!

Thinking of the three ISBN-related problems stated above, what sub-problems can you identify? That is, what things would you have to do to be able to solve the three problems for any ISBN?

[7] The book in question is Karma Ura and Karma Galay (eds), *Gross National Happiness and Development: Proceedings of the First International Seminar on Operationalization of Gross National Happiness*, The Centre for Bhutan Studies, Thimphu, Bhutan, 2004. However, you won't find it on Amazon. The group code is 99936 which is used for books published in the Kingdom of Bhutan.

This page contains answers for this chapter only.

Chapter 1

End of Chapter Exercises

1. There are some quite formal definitions, but in layman's terms an algorithm is a set of clear instructions to carry out a defined task.
2. A computer program is an algorithm expressed in a programming language to be carried out by a computer.

3. **High**: black leather. **Medium**: A black leather billfold with a single fastening. Two main currency pockets plus slots for cards. **Low**: A black leather billfold with a single fastening about 5 years old, contains £25 in cash: a £20 note (the old design with Edward Elgar on the back) and a £5 note; 4 debit card receipts, 1 ATM receipt, 1 debit card, 1 credit card, and my Engineering Council registration card ...
6. Well, you might be a history student taking programming as a compulsory course! The answer "the assassination in Sarajevo of Archduke Ferdinand by Gavrilo Princip" is too simplistic and will not receive credit.

Chapter 1 Projects

StockSnackz Vending Machine

No solutions for this chapter.

Stocksfield Fire Service: Hazchem Signs

No solutions for this chapter.

Puzzle World: Roman Numerals and Chronograms

No solutions for this chapter.

Pangrams: Holoalphabetic Sentences

No solutions for this chapter.

Online Bookstore: ISBNs

No solutions for this chapter.

This page contains answers for this chapter only.